A Hardware/Software Partitioning Algorithm for Processor Cores of Digital Signal Processing

Nozomu Togawa Takashi Sakurai Masao Yanagisawa Tatsuo Ohtsuki
Dept. of Electronics, Information and Communication Engineering, Waseda University
E-mail: togawa@ohtsuki.comm.waseda.ac.jp

Abstract—*A hardware/software cosynthesis system for processor cores of digital signal processing has been developed. This paper focuses on a hardware/software partitioning algorithm which is one of the key issues in the system. Given an input assembly code generated by the compiler in the system, the proposed hardware/software partitioning algorithm first determines the types and the numbers of required hardware units, such as multiple functional units, hardware loop units, and particular addressing units, for a processor core (initial resource allocation). Second, the hardware units determined at initial resource allocation are reduced one by one while the assembly code meets a given timing constraint (configuration of a processor core). The execution time of the assembly code becomes longer but the hardware costs for a processor core to execute it becomes smaller. Finally, it outputs an optimized assembly code and a processor configuration. Experimental results demonstrate that the system synthesizes processor cores effectively according to the features of an application program/data.*

1 Introduction

A general digital signal processor is mainly composed of a micro processor core and several hardware units for digital signal processing such as multiple memory banks, addressing units, and hardware loop units [6], [8]. However, if a particular application program is run on a general digital signal processor, some hardware units can be often used and other hardware units can never be used. We consider that an appropriate configuration for digital signal processors is required according to requirements for a given application program as well as hardware costs for required hardware units.

Hardware/software codesign is to design a hardware part and a software part of a processor and/or a system simultaneously according to a given application program. The researches on hardware/software codesign are classified into (1) design for an overall system including an existing processor core and its peripheral hardwares, (2) simultaneous simulation for a hardware part and a software part of a given system, and (3) design of a micro processor core or ASIP (Application-Specific Instruction set Processor). We can consider the problem to obtain a processor configuration for digital signal processing application to be one of ASIP design problems. With respect to ASIP design the systems called COACH [1], PEAS [2], [3], [10], and ASIA [5] have been presented. Several basic researches have been also reported as in [7], [11]. However, there have been very few researches on overall synthesis for digital signal processors including memory bank design and hardware unit design, though there have been several reports on individual hardware unit design for digital signal processors.

Based on the above discussion, we have developed a hardware/software cosynthesis system for processor cores of digital signal processing [12]. This paper focuses on a hardware/software partitioning algorithm which is one of the key issues in the system. Given an application program written in the C language and a set of application data, the system synthesizes a processor core ranging from a RISC core to a DSP core by selecting the required hardware units, such as multiple functional units, hardware loop units, and particular addressing units, based on the application program/data and the hardware costs. Given an input assembly code generated by the compiler in the system, the proposed hardware/software partitioning algorithm first determines the types and the numbers of required hardware units to execute it (initial resource allocation). Second, the hardware units determined at initial resource allocation are reduced one by one while the assembly code meets a given timing constraint (configuration of a processor core). The execution time of the assembly code becomes longer but the hardware costs for a processor core to execute it becomes smaller. Finally, it outputs an optimized assembly code and a processor configuration. The experimental results demonstrate that the system synthesizes processor cores effectively according to the features of an application program/data.

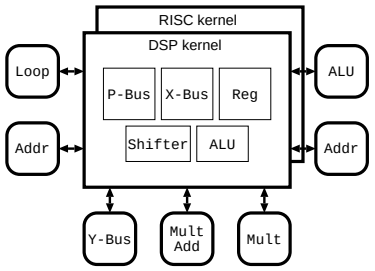

Figure 1. Processor kernels and hardware units added to them.

2 A Hardware/Software Cosynthesis System for Processor Cores of Digital Signal Processing

2.1 Processor Model for Digital Signal Processing

We consider a target architecture model ranging from a general-purpose RISC processor to a general digital signal processor. Based on [9], Fig. 1 shows processor kernels and a part of hardware units which will be added to them. A *processor core* is constructed by adding several hardware units to a processor kernel. Processor kernels and hardware units added to them are defined as follows:

A processor kernel is (i) a RISC-type kernel or (ii) a DSP-type kernel. A RISC-type kernel has the five pipeline stages composed of IF (instruction fetch), ID (instruction decode), EXE (execution), MEM (memory access), and WB (write back) stages based on [4]. A DSP-type kernel has the three pipeline stages composed of IF, ID, and EXE stages based on [6], [8], [9]. A processor kernel has a Harvard architecture and consists of (c-i) an instruction memory, (c-ii) a data memory (X data memory), (c-iii) a register file, and (c-iv) an ALU (Arithmetic Logic Unit) and a barrel shifter. In (c-i) and (c-ii), data bus width can be changed and address bus width is fixed to 16 bits. A RISC-type kernel has five pipeline stages and can run faster compared with a DSP-type kernel. On the other hand, it is hard to control addressing units and hardware loop units. A DSP-type kernel has three pipeline stages and can run slower compared with a RISC-type kernel. On the other hand, hardware units for digital signal processing can be added to it.

Y data memory can be added to a processor kernel. If Y data memory is added to a processor kernel, we can load two data from the two data memories (X data memory and Y data memory) or store two data to the two data memories simultaneously.

The types and the numbers of functional units added to a processor kernel can be changed.

An addressing unit can execute the addressing operations of (i) no operation, (ii) post increment, (iii) post decrement, (iv) index add, (v) modulo add, and (vi) bit reverse.[1] The addressing operations are realized by address registers for (ii) and (iii), index registers for (ii)–(iv) and (vi), and modulo registers for (v). An addressing unit executes the operations required by an application program and data.

A hardware loop unit can execute a fixed number of loops without disturbing any pipeline streams. It is realized by loop registers.

The number of registers can be changed. Registers refer to (a) general-purpose registers in a register file, (b) address registers, index registers, and modulo registers in addressing units for X and Y data memories, and (c) loop registers in a hardware loop unit.

Data bus width of a data memory can be changed. It is the same as the bit width of the `int` type of an application program written in C. The bit width of the `int` type is set by

[1] See [6], [8] for detailed functions of the addressing operations (i)–(vi).

a designer in an application program. The bit width of general-purpose registers and functional units is also set to the bit width of the `int` type.

Data bus width of an instruction memory is determined based on a synthesized instruction set.

Each hardware unit has its hardware cost. Here a hardware cost refers to an area. A processor core with more hardware units requires more hardware costs and thus it increases a chip area. *Configuration of a processor core* is to determine a processor core by adding hardware units to a processor kernel.

2.2 Instruction Set

A processor core synthesized by our hardware/software cosynthesis system has *basic instructions* and *parallel instructions*. The basic instructions are those in a general digital signal processor [9], such as `ADD` and `MAC`, and correspond to processor kernels and/or hardware units. A parallel instruction executes more than one basic instructions. All the combination of basic instructions cannot be a parallel instruction but the hardware/software cosynthesis system determines which combination of basic instructions should be a parallel instruction based on an application program. The combination of basic instructions executed by distinct hardware units can be a parallel instruction.

A processor core synthesized by our hardware/software cosynthesis system requires minimum instructions so that it can function as a general processor. The minimum instructions in basic instructions are called *necessary instructions* and will be synthesized whatever application program is given. In hardware/software cosynthesis, required basic instructions are extracted and parallel instructions are synthesized in addition to necessary instructions.

2.3 The Hardware/Software Cosynthesis System

Fig. 2 shows the proposed system. The system synthesizes a processor core of digital signal processing in the following way:

Compile and application analysis: First the system considers a processor core to which all the hardware units are added and complies an application program on the processor core. In this process, an assembly code with the maximum number of parallel instructions is generated since there are no limitations of hardware units but only data dependency constraints in the application program. The execution time of a generated assembly code is short but the required area for the processor core to execute it becomes large. At the same time, the application data is given to the application program and how many times each basic blocks are executed in the application program is analyzed.

Hardware/software partitioning: Second, the system replaces a part of hardware with software by eliminating hardware units added to a processor kernel. The execution time of the assembly code becomes longer but the required area for the processor core to execute it becomes smaller. The system repeats this process while the execution time of the assembly code satisfies the timing constraint and obtains a processor core satisfying the timing constraint with a small hardware cost.

Hardware and software generation: Finally, the system generates a hardware description of the processor core, the object code of the application program run on the processor core, and software environment.

The proposed approach has the two advantages: First, functions of addressing units and hardware loops can be correspond to a part of the high-level codes of an application program. By much utilizing those functions in compiling, the system can obtain an assembly code whose execution time is the shortest. Second, a complicated hardware/software partitioning process can be simplified since it is considered independent of a compiling process of an application program,

3 A Hardware/Software Partitioning Algorithm

In our system, one of the key issues is hardware/software partitioning. In this section, we define a hardware/software partitioning problem and propose a hardware/software partitioning algorithm.

3.1 Definitions

Consider that a basic block $B \in \mathcal{B}_{app}$ in an input assembly code is executed N^B_{exe} times. N^B_{exe} is computed by the application analyzer by our system. Let N^B_{cycle} be the number of clock cycles to execute B. The number of the total clock cycles N_{cycle} to execute an input assembly code can be computed by $N_{cycle} = \sum_{B \in \mathcal{B}_{app}} N^B_{exe} \cdot N^B_{cycle}$. Then the execution time T_{app} of an assembly code is defined as $T_{app} = N_{cycle} \times T_{cycle}$, where T_{cycle} is a clock period of a synthesized processor core. Let T^{max}_{app} be the maximum execution time of an application program. Then a *timing constraint* is given by $T_{app} \leq T^{max}_{app}$.

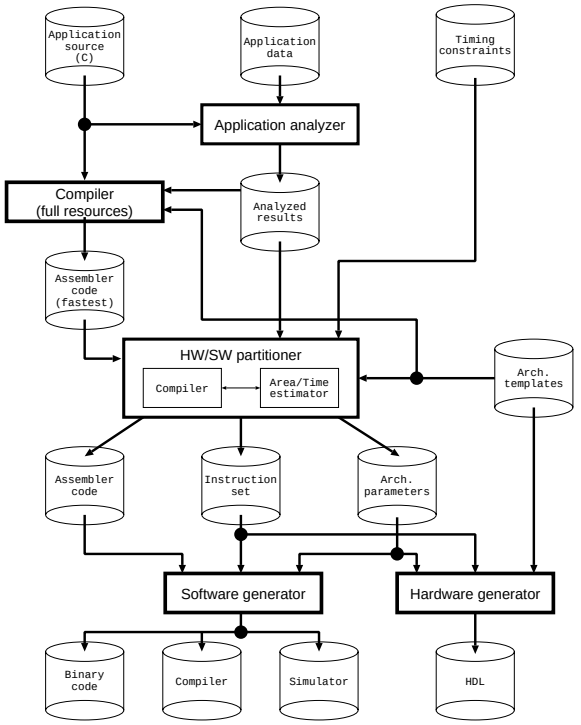

Figure 2. The hardware/software cosynthesis system for processor cores of digital signal processing.

Definition 1 *A hardware/software partitioning problem is, for given an assembly code generated by the compiler, N^B_{exe} for each basic block $B \in \mathcal{B}_{app}$, the timing constraint, and available hardware units for a processor core, to obtain a processor core, an assembly code run on the generated processor core, and an instruction set under the timing constraint so as to minimize hardware costs for a processor core.*

3.2 The Algorithm

Several hardware/software partitioning algorithms for processor cores has been proposed [2], [3], [5], [10]. Those algorithms are based on simulated annealing or the branch-and-bound method and obtain optimum solutions. However, if the size of a given problem is larger, they may require much computation time. Those algorithms can be applicable to only a simple and small problem. Since there are many available hardware units in processor cores of digital signal processing, we consider that those algorithms are not appropriate for our hardware/software partitioning problem.

Our hardware/software partitioning is realized by the following approach. First, the types and the numbers of required hardware units are determined to execute the input assembly code (initial resource allocation). Second, the hardware units determined at initial resource allocation are reduced one by one and the assembly code is rescheduled while the assembly code satisfies the timing constraint (configuration of a processor core). Since this approach is heuristic, we can obtain only a suboptimum solution. However, it optimizes the types and the numbers of all the hardware units simultaneously and then we expect that it can obtain better results in a practical computation time.

3.2.1 Phase 1. Initial Resource Allocation

We can determine whether Y data memory is used or not and which addressing operations are used in addressing units according

Input: Assembly code generated by the compiler, timing constraint, and execution profile
Output: Assembly code, configuration of a processor core, and instruction set

Step 0. Assume that a processor kernel is DSP-type or RISC-type and apply Steps 1–6 to each kernel.

Step 1. Insert `NOP` instructions so that they resolve the data hazard between instructions.

Step 2. For each hardware unit which is currently added to a processor kernel, try to eliminate it or reduce its function and check how much the execution time of an assembly code will be increased.

Step 3. For the hardware unit which gives the minimum hardware cost of a processor core without violating the timing constraint after eliminating it or reducing its function, eliminate it or reduce its function.

Step 4. Reschedule an assembly code.

Step 5. While there exists a hardware unit which meets the condition in Step 3, repeat Steps 1–4. Otherwise finish.

Figure 3. The configuration algorithm of a processor core.

to an input assembly code. If an input assembly code uses Y data memory, it is initially allocated. If an input assembly code uses some addressing operation, an addressing unit to execute the addressing operation is initially allocated. As described earlier, data bus width of data memories is set to the bit width of the `int` type in an application program. The number of registers are given by the maximum number of registers used in each functions in an input assembly code. The required types and the numbers of functional units are determined by the maximum number of instructions executed concurrently.

For example, if the assembly code below is given, six registers, one adder, and one multiply and add unit are allocated. In the first instruction, two additions are concurrently executed using the adder and the multiply and add unit.

```
   ADD    R1, R2, R3
|| ADD    R4, R5, R6  // R3 = R1 + R2; R6 = R4 + R5;
   ADD    R3, R6, R1
|| MAC    R3, R6, R2  // R1 = R3 + R6; R2 = R3 * R6 + R2;
```

3.2.2 Phase 2. Configuration of a Processor Core

Fig. 3 shows a procedure to configure a processor core. In the procedure, we first assume a DSP-type kernel or a RISC-type kernel in Step 0. Then Steps 1–5 are applied to each processor kernel. Step 1 and Step 2 are trivial. In Step 4 we execute a scheduling process under the constraint of current hardware unit configuration so as to minimize the number of clock cycles in each basic block.

In configuring a processor core, the key issues are Steps 2 and 3 where an assembly code is updated according to reconfiguration of hardware units and the hardware cost of a processor core and execution time of an assembly code are estimated.

Update of an assembly code according to reconfiguring hardware units: For each hardware unit which is currently added to a processor core, we eliminate it or reduce its function. At the same time, we update an assembly code. We consider (1) Y data memory, (2) functional units, (3) addressing units, and (4) registers as hardware units.

Here we describe here how to eliminate one addressing unit. Let us consider the address operations of (i) no operation, (ii) post increment, (iii) post decrement, (iv) index add, (v) modulo add, and (vi) bit reverse in addressing units. If one of those address operations is eliminated, it is executed using general-purpose registers. An assembly code is updated based on the following procedure for X data memory (the similar procedure can be applied to Y data memory).

1. Allocate general-purpose registers for address registers, index registers, and modulo registers.
2. Apply 3. and 4. to each instruction using an addressing unit.
3. Replace an instruction using an addressing unit with (1) preprocessing for the address operation, (2) a memory access instruction, and (3) postprocessing for the address operation, using general-purpose registers.

Table 1. Hardware unit libraries.

Hardware unit		Bit width	Clock cycles	Area [μm^2]	Delay [ns]
Kernel	DSP-type	32	—	798225*	—
	RISC-type	32	—	985668*	—
Functional Unit	ALU	32	1	434107	5.0
	Shifter	32	1	447567	5.0
	Multiplier	32	1	401450	12.0
	MAC	32	1	979717	15.0
Addressing unit	(i)	16	1	79624	5.0
	(i)–(iii)	16	1	100876	5.0
	(i)–(iv)	16	1	143876	5.0
Hardware loop		—	—	111696	—
Register	General (DSP-type)	32	—	178304	—
	General (RISC-type)	32	—	97644	—
	DP, DN and DMX	16	—	57250	—
	Loop	16	—	408695	—

Addressing unit (i): no operation, (ii): post increment, (iii): post decrement, and (iv): index add
*$n = 2$, $m = 1$ and there are an ALU, a shifter, and two registers in the processor core

Table 4. Comparison of the synthesized processor cores and existing processors.

Processor	Clock cycles to run app. programs DCT	Matrix	Huffman
Ours1	4357	2.11×10^6	1164
Ours2	3014	1.12×10^6	1164
MMX Pentium (133MHz)	12954	13.04×10^6	1702
Pentium II (300MHz)	4080	3.15×10^6	1377
Micro Sparc II (110MHz)	34760	34.54×10^6	1254
Turbo Sparc (170MHz)	19040	25.33×10^6	1336
Ultra Sparc (167MHz)	15865	15.50×10^6	982
PA8000 (gcc, 160MHz)	27360	20.48×10^6	821
PA8000 (HP C, 160MHz)	4544	3.98×10^6	904

Ours1 and Ours2: The maximum number of basic instructions executed concurrently is one and two, respectively
In our system, the clock cycles are shown when the tightest timing constraints are given. Note that the clock frequency of our processor core varies according to an application program. See Tables 2–3.

Estimation of the hardware cost of a processor core: The hardware cost of a processor core is estimated by adding the hardware cost of a processor kernel and the hardware costs of hardware units. Let U be a set of hardware units for current configuration of a processor core. Let t be a type of a processor kernel (t is RISC-type or DSP-type), b be bit width of a general-purpose register, n be the maximum number of basic instructions executed concurrently, m be the number of memory banks ($m = 1, 2$). The hardware cost of a processor kernel can be denoted as $c_k(t, b, m, n, U)$. Let $c(u)$ be the hardware cost of a hardware unit $u \in U$. Then the hardware cost of a processor core can be computed as $c = c_k(t, b, m, n, U, R) + \sum_{u \in U} c(u)$.

Estimation of the execution time of an assembly code: In each basic block $B \in \mathcal{B}_{app}$, the number of clock cycles N^B_{cycle} to execute B can be given by the number of steps when scheduling the instructions in B. Then the total number of clock cycles N_{cycle} can be obtained according to Section 3.1. A clock cycle T_{cycle} is estimated by the maximum delay in all the pipeline stages for a DSP-type kernel or a RISC-type kernel.

Based on N_{cycle} and T_{cycle}, we can obtain the execution time of a current assembly code.

4 Experimental Results

The system has been implemented in the C language on Sun Ultra 1. The system was applied to the two-dimensional discrete cosine transform (for 8×8 pixels), the matrix multiplication (for two 100×100 matrices), and the Huffman encoding (8×8 pixels). The maximum number of basic instructions executed concurrently was set to be 1 and 2. The timing constraint ranged from the minimum possible execution time to the maximum possible execution

Table 2. Experimental results 1 (The maximum number of basic instructions executed concurrently is one).

	Timing	Area	T	Hardware configuration					CPU
	consts [μs]	[μm^2]	[ns]	Kernel	Y mem	#FUs	#Regs	Addr unit	time [s]
DCT	135	8346460	31	DSP	Yes	(1,1,0,1)	(6,10,3)	(i)+(ii)+(iii)	4.8
	200	7729246	31	DSP	No	(1,1,0,1)	(6,12,3)	(i)+(ii)+(iii)	6.8
	300	5937686	31	DSP	No	(1,1,0,1)	(7,12,0)	(i)+(ii)+(iii)	9.2
	400	3823499	20	RISC	No	(1,1,0,1)	(10,0,0)	—	18.1
	500	3725855	20	RISC	No	(1,1,0,1)	(9,0,0)	—	27.1
	577	3725855	20	RISC	No	(1,1,0,1)	(9,0,0)	—	31.0
Matrix	0.65×10^5	7363570	31	DSP	Yes	(1,1,0,1)	(6,2,3)	(i)+(ii)+(iii)+(iv)	1.6
	1.01×10^5	6849050	31	DSP	No	(1,1,0,1)	(6,3,3)	(i)+(ii)+(iii)+(iv)	2.1
	2.02×10^5	5057490	31	DSP	No	(1,1,0,1)	(7,3,0)	(i)+(ii)+(iii)+(iv)	2.8
	3.03×10^5	3823499	20	RISC	No	(1,1,0,1)	(10,0,0)	—	5.7
	4.04×10^5	3628211	20	RISC	No	(1,1,0,1)	(8,0,0)	—	10.8
	5.05×10^5	3530567	20	RISC	No	(1,1,0,1)	(7,0,0)	—	15.2
	6.05×10^5	3432923	20	RISC	No	(1,1,0,1)	(6,0,0)	—	18.4
Huffman coding	11.6	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	68.2
	15	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	68.2
	20	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	69.8
	25	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	123.2
	28.6	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	87.4

Table 3. Experimental results 2 (The maximum number of basic instructions executed concurrently is two).

	Timing	Area	T	Hardware configuration					CPU
	consts [μs]	[μm^2]	[ns]	Kernel	Y mem	#FUs	#Regs	Addr unit	time [s]
DCT	93	7863540	31	DSP	Yes	(1,1,0,1)	(6,7,3)	(i)+(ii)+(iii)	4.8
	100	7863540	31	DSP	Yes	(1,1,0,1)	(6,7,3)	(i)+(ii)+(iii)	6.7
	200	6071980	31	DSP	Yes	(1,1,0,1)	(7,7,0)	(i)+(ii)+(iii)	6.9
	300	4216843	20	RISC	No	(1,1,0,1)	(11,0,0)	—	11.5
	400	3823499	20	RISC	No	(1,1,0,1)	(10,0,0)	—	20.1
	500	3725855	20	RISC	No	(1,1,0,1)	(9,0,0)	—	29.2
	577	3725855	20	RISC	No	(1,1,0,1)	(9,0,0)	—	33.8
Matrix	0.35×10^5	7410324	31	DSP	Yes	(1,1,0,1)	(6,2,3)	(i)+(ii)+(iii)+(iv)	1.6
	1.01×10^5	5618764	31	DSP	Yes	(1,1,0,1)	(7,2,0)	(i)+(ii)+(iii)+(iv)	2.2
	2.02×10^5	5517306	31	DSP	No	(1,1,0,1)	(7,3,0)	(i)+(ii)+(iii)+(iv)	2.9
	3.03×10^5	3725855	20	RISC	No	(1,1,0,1)	(9,0,0)	—	9.1
	4.04×10^5	3628211	20	RISC	No	(1,1,0,1)	(8,0,0)	—	12.8
	5.05×10^5	3530567	20	RISC	No	(1,1,0,1)	(7,0,0)	—	17.3
	6.05×10^5	3530567	20	RISC	No	(1,1,0,1)	(7,0,0)	—	17.4
Huffman coding	11.6	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	85.6
	15	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	85.6
	20	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	69.1
	25	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	122.7
	28.6	3039070	10	RISC	No	(1,1,0,0)	(12,0,0)	—	85.5

T [ns]: Clock period of the synthesized processor core
#FUs: (#ALUs, #Shifters, #Multipliers, #MACs)
#Regs: (#General_registers, #Data_pointers, #Loop_registers)
Addressing unit (i): no operation, (ii): post increment, (iii): post decrement, and (iv): index add

time for each application program. Table 1 shows a part of our hardware unit libraries. They are synthesized using the Synopsis Design Compiler with the VDEC libraries (CMOS and 0.5μm technology).

Tables 2–3 shows the synthesized processor cores for each application program. The tables indicate that, particularly when the tight timing constraint is given, processor cores with a DSP-type kernel are synthesized for the DCT and the matrix multiplication because those application programs have high parallelism for source codes and require many addressing operations and loops. On the other hand, processor cores with a RISC-type kernel are synthesized for the Huffman encoding because those application programs have low parallelism for source codes and do not have addressing operations.

In order to compare our results with the existing processors, Table 4 shows the results of the clock cycles to run each application program. In our system, the clock cycles are shown when the tightest timing constraints are given in Tables 2–3. The results show that processor cores synthesized by our system achieve the minimum number of clock cycles for the DCT and the matrix multiplication.

Acknowledgments

The authors would like to thank M. Hamabe, T. Kawasaki, A. Nose, T. Nakamura, Y. Kataoka, and D. Yoshizawa of Waseda University for their implementations and valuable discussions.

References

[1] H. Akaboshi and H. Yasuura, "COACH: A computer aided design tool for computer architects," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E76-A, no. 10, pp. 1760-1769, 1993.

[2] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint," in *Proc. ICCAD-93*, pp. 526–532, 1993.

[3] N. N. Bình, M. Imai, A, Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate count," in *Proc. 33rd DAC*, pp. 527–532, 1996.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufman, 1990.

[5] I.-J. Huang and A. M. Despain, "Synthesis of instruction sets for pipelined microprocessors," in *Proc. 31st DAC*, pp. 5–11, 1994.

[6] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Inc., 1994–1996.

[7] C. Liem, P. Paulin, and A. Jerraya, "Address calculation for retargetable compilation and exploration of instruction-set architectures," in *Proc. 33rd DAC*, pp. 597–600, 1996.

[8] V. K. Madisetti, *Digital Signal Processors*, IEEE Press, 1995.

[9] NEC, http://www.ic.nec.co.jp/micro/micro.html.

[10] J. Sato, A. Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai, "PEAS-I: A hardware/software codesign system for ASIP development," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E77-A, no. 3, pp. 483–491, 1994.

[11] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," in *Proc. ICCAD-95*, pp. 388–392, 1995.

[12] N. Togawa, T. Sakurai, M. Yanagisawa, and T. Ohtsuki, "A hardware/software cosynthesis system for processor cores of digital signal processing," *IEICE Tech. Rep.*, VLD97-115, 1998 (in Japanese).