A Hardware/Software Partitioning Algorithm for SIMD Processor Cores

Koichi Tachikake† Nozomu Togawa††,‡ Yuichiro Miyaoka† Jinku Choi†
Masao Yanagisawa† Tatsuo Ohtsuki†

†Dept. of Electronics, Information and Communication Engineering, Waseda University
††Dept. of Information and Media Sciences, The University of Kitakyushu
‡Advanced Research Institute for Science and Engineering, Waseda University
3-4-1 Okubo, Shinjuku, Tokyo 169-8555, Japan
Tel: +81-3-5286-3396 Fax: +81-3-3203-9184
E-mail: tatikake@ohtsuki.comm.waseda.ac.jp

Abstract

This paper proposes a new hardware/software partitioning algorithm for processor cores with SIMD instructions. Given a compiled assembly code including SIMD instructions, a timing constraint of execution time, and available hardware units, the proposed algorithm synthesizes an area-optimized processor core with a new assembly code. Firstly, we assume an initial processor core on which an input assembly code can run with the shortest execution time. Secondly we reduce a hardware unit added to a processor core one by one while the timing constraint is satisfied. At the same time, we update the assembly code so that it can run on the new processor configuration. By repeating this process, we finally obtain a processor core architecture with small area under the given timing constraint. We expect that we can obtain a processor core which has appropriate SIMD functional units for running the input application program. The promising experimental results are also shown.

1 Introduction

In image processing applications such as image synthesis and/or image corrections, each pixel in an image is composed of small bits of data. For example, a pixel can be represented by an 8-bit data. However, a general micro processor has a basic bit width of 32 bits or more. In image processing applications, how to deal with a short-word data with a long-word functional unit is a main problem. A *packed SIMD type operation*[5], [9], [10], [15] (or *a SIMD operation* in short) gives one of the most effective solutions for this problem. A SIMD operation is n-parallel b/n-bit sub-operations executed by a modified b-bit functional unit. An instruction corresponding to a SIMD operation is called a *SIMD instruction*. A functional unit executing SIMD operations is called a *SIMD functional unit* and a processor core with SIMD instructions is called a *SIMD processor core*. A SIMD processor core can be effectively applied to image processing applications since we can deal with n pixels concurrently by modifying normal b-bit functional units.

Generally, a SIMD operation has many options (see 2.3.2 in detail). Thus we can configure so many different SIMD operations. However, a particular image application program often uses very limited SIMD operations. We consider that appropriate configuration for a image processor core is required depending on application programs as well as hardware costs. Hardware/software cosynthsis must be a very powerful strategy to synthesize a SIMD processor core.

Hardware/software codesign is to design a hardware part and a software part of a processor and/or a system simultaneously depending on application programs. Particularly the hardware/software codesign systems such as in [1], [2], [4], [7], [12], [14], [16] synthesize micro processor cores for given application programs. All the systems proposed so far, however, focus on conventional micro processor cores and then they do not deal with SIMD operations/instructions.

We have been developing a hardware/software cosynthesis system for SIMD processor cores [11], [13], [17], [18]. For image processing applications, the system automatically synthesizes an optimal image processor architecture through compiling, hardware/software partitioning, and hardware/software generation. The basic system which automatically synthesizes a digital signal processor architecture was proposed in [17], [18]. A parallelizing compiler with SIMD instructions was proposed in [13]. The compiler generates an initially scheduled assembly code including SIMD instructions given to hardware/software partitioning. The functional unit generator for SIMD operations was proposed in [11]. The functional unit generator estimates area/delay values for each functional units used in hardware/software partitioning.

In this paper, we focus on hardware/software partitioning in our system and propose a new hardware/software partitioning algorithm for SIMD processor cores. Firstly, we determine the numbers and types of hardware units added to a processor core to execute an input assembly code. Then we reduce the number of the hardware units or we reduce a sub-function of the hardware units, one by one. At the same time, we reconfigure the processor core and update the assembly code. Finally, we obtain a processor core architecture with small area under the given timing constraint.

2 Architecture Model and Instruction Set

In this section, we define our processor architecture model and its instruction set [11], [13], [17], [18]. Fig. 1 shows our processor architecture model. Our processor architecture is based on a digital signal processor in [6] and composed of one of the two *processor kernels* and extra *hardware units*. A *processor core* is constructed by adding several hardware units to a processor kernel. In the following, processor kernels, hardware units, and an instruction set are defined.

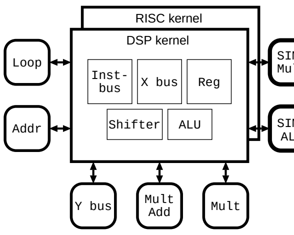


Figure 1. Processor kernels and hardware units.

Table 2. Basic Instructions (a minimum instruction is underlined).

Arithmetic and logic operation	$\underline{\texttt{ADD}}$, $\underline{\texttt{SUB}}$, $\underline{\texttt{SRA}}$, $\underline{\texttt{SRL}}$, $\underline{\texttt{SLL}}$, $\underline{\texttt{AND}}$, $\underline{\texttt{OR}}$, $\underline{\texttt{XOR}}$, `MUL`, `DIV`, `SLT`, `SEQ`, `SNE`, `COM2`, `MAC`, $\underline{\texttt{INC}}$, $\underline{\texttt{DEC}}$, $\underline{\texttt{ADDI}}$, $\underline{\texttt{SUBI}}$, $\underline{\texttt{SRAI}}$, $\underline{\texttt{SRLI}}$, $\underline{\texttt{SLLI}}$, $\underline{\texttt{ANDI}}$, $\underline{\texttt{ORI}}$, $\underline{\texttt{XORI}}$, `MULI`, `DIVI`
Load and store	$\underline{\texttt{LDX}}$, `LDY`, $\underline{\texttt{STX}}$, `STY`, $\underline{\texttt{LDRX}}$, `LDRY`, $\underline{\texttt{STRX}}$, `STRY`, $\underline{\texttt{LDXI}}$, `LDYI`, $\underline{\texttt{STXI}}$, `STYI`, `LDIX`, `LDIY`, `STIX`, `STIY`, $\underline{\texttt{MV}}$, $\underline{\texttt{IMM}}$
Jump	$\underline{\texttt{BEQ}}$, $\underline{\texttt{BNE}}$, $\underline{\texttt{BZ}}$, $\underline{\texttt{BNZ}}$, $\underline{\texttt{JP}}$, `LOOP`, `RPT`, $\underline{\texttt{CALL}}$, $\underline{\texttt{RET}}$, $\underline{\texttt{NOP}}$, $\underline{\texttt{HLT}}$
Parallel load and store	`LDPX`, `STPX`

2.1 Processor Kernels

A processor kernel is (i) a RISC-type kernel or (ii) a DSP-type kernel. A RISC-type kernel has the five pipeline stages (IF, ID, EXE, MEM, and WB) as in the micro processor of [3]. A DSP-type kernel has the three pipeline (IF, ID, and EXE) stages as in the DSP processors of [6], [8]. The number of pipeline stages and processes in each pipeline stage are fixed and cannot be changed. A processor core will become a general-purpose RISC core if a RISC-type kernel is selected. It will become a DSP core if a DSP-type kernel is selected. A hardware configuration of each processor kernel is determined in the same way as in [17].

2.2 Hardware Units

Our processor core can have extra hardware units: (1) a Y-bus for Y data memory, (2) functional units (shifters, ALUs, multipliers, MAC units, bit extenders/extractors, and data move units), (3) addressing units, and (4) hardware loop units (see [13], [17], [18] for detailed functions in each hardware units). A functional unit has a functional unit type t_{fu} (see Table 1).

All these hardware units can be added to the DSP kernel. The hardware units except addressing units and hardware loop units can be added to the RISC kernel.

2.3 Instruction Set

2.3.1 Basic Instructions and Parallel Instructions

Our synthesized processor core has *basic instructions* such as ADD and MUL and *parallel instructions* such as (ADD || ADD) and (ADD || MUL). The basic instructions correspond to the functions of our processor kernels and hardware units. A parallel instruction executes more than one basic instructions. All the combination of basic instructions cannot be a parallel instruction. Our hardware/software partitioner determines which basic instructions should be included in a processor core and which combination of basic instructions should be a parallel instruction.

Table 3. Packed SIMD Type Instructions in basic instructions.

Arithmetic operation	`ADD`, `SUB`, `MUL`, `MAC`
Shift operation	`SRA`, `SLA`, `SLL`
Bit extend/extract operation	`EXTR`, `EXTD`
Data move operation	`EXCH`, `PERM`

A processor core requires *minimum instructions* so that it can function as a general processor. The minimum instructions will be included in a processor core whatever application program is given. Table 2 shows our basic instructions and minimum instructions.

Basic instruction has its instruction type t_{inst}. If a basic instruction i is executed by a functional unit with the type of t_{fu}, its instruction type, t_{inst}, is defined as t_{fu}.[1]

2.3.2 SIMD instructions

Several basic instructions can be SIMD instructions as shown in Table 3. For example, Fig. 2 shows SIMD multiplication. In this figure, we assume that a register has 32 bits and four 8-bit data is packed into a single register. In Fig. 2(a), two four-packed data are multiplied and the four results are packed into a single register. In this case, since each of the intermediate data has a maximum of 16 bits, we must shorten its bit length to 8 bit. Wrap around operation or saturation operation will be applied to the intermediate data and then we can obtain 8-bit data. In Fig. 2(b), the lower words of two four-packed data are multiplied and the two results are packed into a single register. In this case, bit-extend operation is applied to the intermediate data if it has the bit length of 15 bits or less and we can obtain a 16-bit data.

As discussed above, each of SIMD arithmetic operations and SIMD shift operations has the options of (1) a packing number n, (2) whether the data is signed or unsigned, (3) whether the saturation operation is applied to the resultant data or not, (4) whether the bit-extend operation is applied to the resultant data or not, and (5) how much the resultant data is shifted. For example, the instruction `MUL_4_sr2s` shows that four data are packed into one register, all the data are singed, bit-extend operation is not applied, and each of four resultant data is shifted to the right by two bits and saturation operation is applied to it ($\underline{\text{mul}}$tiplication, $\underline{4}$ packing data, $\underline{\text{s}}$igned, $\underline{\text{r}}$ight shift by $\underline{2}$ bits, and $\underline{\text{s}}$aturated).

A bit extend instruction constructs $n/2$-packed data from n-packed data. A bit extract instruction constructs $2 \times n$-packed data from n-packed data.

A data move instruction obtains new n-packed data by rearranging old n-packed data. The behavior of the EXCH instruction is fixed. The PERM instruction can rearrange n-packed data into any new n-packed data. The behavior of the PERM instruction is determined depending on an application program by our hardware/software cosynthesis system.

SIMD instructions also has its instruction type t_{inst}. If a SIMD instruction is executed by a SIMD functional unit with type t_{fu}, its instruction type t_{inst} is t_{fu}. For example, `MUL_4_sr2s` have the type of *mul*.

[1] The type of `ADD` and `SUB` is defined as ALU, although `ADD` and `SUB` can be executed by either an ALU or a MAC unit. The type of SIMD version of `ADD` and `SUB` is also defined as ALU.

Table 1. Functional unit type and its corresponding operations.

Function unit	FU type t_{fu}	Operations
Shifter	sft	Shift operation
ALU	alu	Arithmetic and logic operations
Multiplier	mul	Multiply
Divider	div	Divide
MAC unit	mac	multiply and addition
Bit extractor/extender	ext	Bit extend/extract
Data move unit	exh	Data exchange and permutation

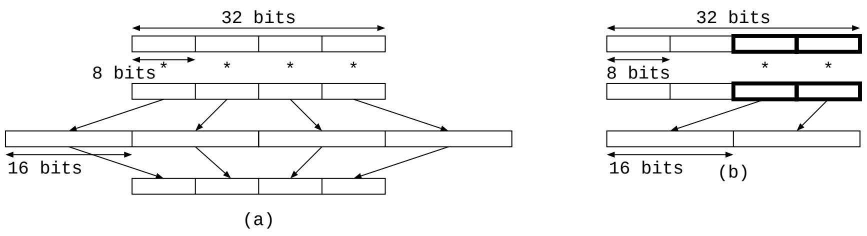


Figure 2. SIMD multiplications. (a) Four 8-bit multiplications. (b) Two 16-bit bit-extend multiplications.

3 A HW/SW Partitioning Algorithm for SIMD Processor Cores

3.1 Hardware/Software Cosynthesis System

We have been developing a hardware/software cosynthesis system for SIMD processor cores [11], [13], [17], [18]. We named the system SPADES (System for Processor Architecture Design with Estimation – type SIMD). In this subsection, we briefly review our basic idea of the system.

Given an application program in C and a set of its application data, our system synthesizes a hardware description of a processor core and generates an object code and a software environment (compiler, assembler and simulator) for the processor core under the constraint of the execution time to run the application program. The objective is to minimize the hardware cost of a processor core. The hardware cost of a processor core is given by the sum of hardware costs of a processor kernel and hardware units used in the processor core. The hardware cost refers to area in this paper. The execution time to run an application program is given by multiplying the clock period by the number of clock cycles to run the application program.

3.2 The HW/SW Partitioning Algorithm

In this subsection, we focus on a hardware/software partitioning algorithm for SIMD processor cores. We first define a hardware/software partitioning problem. Then we propose a hardware/software partitioning algorithm for SIMD processor cores.

3.2.1 Problem Definition

An assembly code is defined as a graph (*call graph*, *control-flow graph*, and *data-flow graph*)[17]. A *call graph* $G_c = (V_c, E_c)$ is defined as a graph representing function calls in an application program. A node $v \in V_c$ in G_c represents a function. Each node in a call graph has a control-flow graph. A *control-flow graph* $G_{cf} = (V_{cf}, E_{cf})$ is defined as a graph representing control flow in a function. A node $v \in V_{cf}$ in G_{cf} represents a basic block. Each node in a control-flow graph has a data-flow graph. A *data-flow graph* $G_{df} = (V_{df}, E_{df})$ is a graph representing data flow in a basic block. A node $v \in V_{df}$ in G_{df} represents a basic instruction.

Let $\mathcal{B}_{app}$ and $\mathcal{F}_{app}$ be a set of basic blocks and a set of functions, respectively, in an input assembly code. Consider that a basic block $B \in \mathcal{B}_{app}$ is executed N^B_{exe} times. N^B_{exe} is calculated 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 as

$$N_{cycle} = \sum_{B \in \mathcal{B}_{app}} N^B_{exe} \cdot N^B_{cycle}. \tag{1}$$

The execution time T_{app} of an assembly code is defined as

$$T_{app} = N_{cycle} \times T_{cycle}, \tag{2}$$

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 which is given by the designer. Then a *timing constraint* is given by

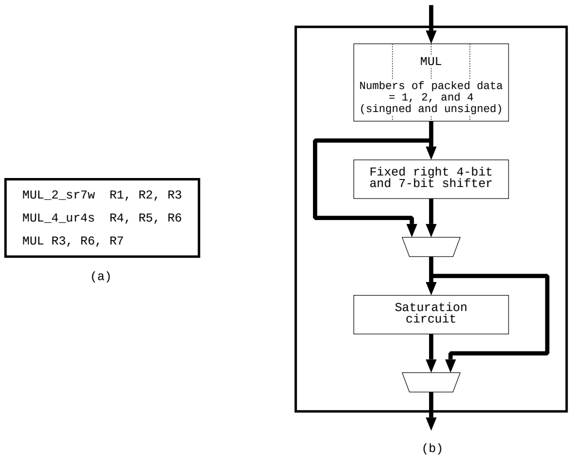
$$T_{app} \le T^{max}_{app}. \tag{3}$$

Then a hardware/software partitioning problem is defined.

Definition 1 *Given an initially scheduled assembly code, N^B_{exe} for each basic block $B \in \mathcal{B}_{app}$, the timing constraint, and available hardware units for a processor core, a hardware/software partitioning problem is to find a processor core configuration, an assembly code executed on the processor core, and an instruction set for the processor core under the timing constraint and the hardware configuration conditions so as to minimize the hardware cost of the processor core.*

3.2.2 The Algorithm

The proposed algorithm is an extended version of the algorithm in [17] so that it can deal with SIMD instructions and SIMD functional units. Firstly, we determine the numbers and types of hardware units added to a processor core to execute an input assembly code (Phase 1). Phase 1 determines an *initial processor core*. An initial processor core includes full SIMD functional units where a functional unit with type t_{fu} can execute all the SIMD instructions in an input assembly code with the instruction type $t_{inst} = t_{fu}$. Then we reduce the number of the hardware units or we reduce a sub-function of the hardware units, one by one, while the timing constraint



```
MUL_2_sr7w  R1, R2, R3
MUL_4_ur4s  R4, R5, R6
MUL R3, R6, R7
```

(a)
MUL
Numbers of packed data = 1, 2, and 4 (singned and unsigned)
Fixed right 4-bit and 7-bit shifter
Saturation circuit
(b)

Figure 3. Instructions with the type of *mul* (a) and a multiplier configuration for them (b).

is satisfied. At the same time, we reconfigure the processor core and update the assembly code (Phase 2).

Our approach is heuristic but we expect that it can find a globally good solution in a practical time since it optimizes the numbers, types, and sub-functions of hardware units including SIMD functional units simultaneously.

Phase 1. Allocate an Initial Resource: In Phase 1, we configure an initial processor core.

Let us consider processor kernel parameters. A *processor kernel type*, RISC or DSP, is not determined in Phase 1 but this is determined in Phase 2. The basic bit width $b_{knl,fu}$ of a processor core is given as input and all the other parameters are determined in the same way as in [17]. The configuration of the ALU and shifter in a processor kernel will be discussed later together with other functional units.

Let us consider hardware unit parameters. If an input assembly code includes an instruction using the Y data memory, we add the Y data memory to a processor kernel. The number of loop registers, the number of address registers, and the type of addressing units are all determined by an input assembly code. Finally, we must determine the configuration of functional units including SIMD functional units.

Configuration of functional units: The configuration of each functional unit is determined in the following way. Let us consider a set I_t of the instructions whose instruction type of $t_{inst} = t$ in an input assembly code. We construct the functional unit with the type $t_{fu} = t$ so that it can execute all the instructions in I_t and minimum instructions with the type of t. For example, assume that an input assembly code includes the instructions of `MUL` , `MUL_4_ur4s` , and `MUL_2_sr7w` for multiplication. In this case, we construct a SIMD multiplier as shown in Fig. 3. The SIMD multiplier is composed of a multiplier for one, two and four data, a 4-bit and 7-bit right shifter, and a saturation unit.

The number of each functional unit is determined in the following way. If n_t-parallel instructions are executed for a set I_t of the instructions with $t_{inst} = t$ in an input assembly code, we add n_t functional units with

---

Inputs: Assembly code, initial processor core, and timing constraint.

Outputs: New processor core and its corresponding assembly code

Phase 2. For each of a DSP-type kernel and a RISC-type kernel, execute Steps 1–4.

Step 1. For each u in the hardware units, sub-functions of hardware units, and registers currently added to a processor kernel, try to eliminate u or try to replace u with the one which has the smaller hardware cost than u.

Step 2. Evaluate the $T_{rate}(u)$ value. For u_{min} which gives the minimum $T_{rate}(u_{min})$ value without violating the given timing constraint, eliminate u_{min} from a current processor kernel or replace u_{min} with the one which has the smaller hardware cost than u_{min}.

Step 3. Update the assembly code according to a new processor core configuration.

Step 4. While there exists a hardware unit, sub-function, or register which meets Step 2, repeat Steps 1–3. Otherwise finish.

Figure 4. The algorithm of Phase 2 (configuration of a processor core).

the type of $t_{fu} = t$ to a processor kernel. For example, assume that an input assembly code includes the parallel instruction as below:

```
MUL_4_ur4s R1,R2,R3 || MUL_2_sr7w R4,R5,R6
```

In this case, we add two multipliers whose configuration is shown in Fig. 3(b) to a processor kernel.

Phase 2: Determine a Processor Core Configuration: Phase 2 determines (1) a processor kernel type (RISC or DSP), (2) the number of general-purpose registers, (3) whether the Y data memory is added to a processor kernel or not, (4) the number of address registers and types of addressing units, (5) the number of loop registers in the hardware loop unit, and (6) functional unit configuration, depending on an input assembly code and timing constraint.

Firstly, we assume that a processor core has a RISC-type kennel or a DSP-type kernel. For each of a kernel, we reduce the parameters in (1)–(6) one by one while the processor core satisfies the timing constraint. Finally, we can find an processor core architecture with small area satisfying the timing constraint.

Fig. 4 shows our proposed algorithm. In the algorithm, Step 1 and Step 3 are discussed later. Step 4 is trivial. In Step 2, $T_{rate}(u)$ for each hardware unit, each sub-function of hardware units,[2] or each register is defined as:

$$T_{rate}(u) = \frac{T_1(u) - T_0}{A_0 - A_1(u)}, \qquad (4)$$

where A_0 and T_0 refer to a hardware cost and execution time of the processor core before eliminating u, respectively, and $A_1(u)$ and $T_1(u)$ refer to a hardware cost and execution time of the processor core after eliminating u, respectively. Step 2 finds u_{min} which gives minimum

[2] An addressing unit and a SIMD functional unit have sub-functions. Sub-functions of an addressing unit refer to the addressing operations such as post increment, post decrement, index addition, and modulo operation. For sub-functions for a SIMD functional unit, see the discussion later.

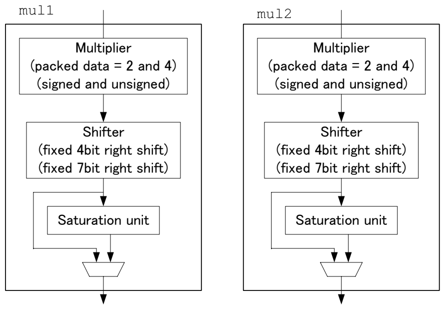


Figure 5. Original multiplier configuration.

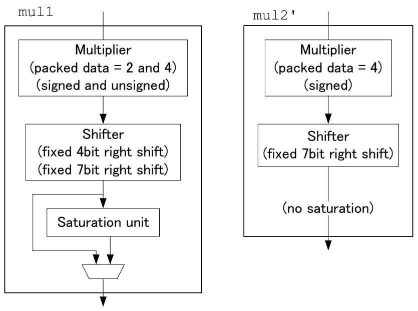


Figure 6. Multiplier configuration (after eliminating a sub-function in mul_2).

$T_{rate}(u_{min})$ and actually eliminates u_{min} from a current processor core. By using the $T_{rate}(u)$ value, we can effectively reduce a hardware cost of a processor core with satisfying a timing constraint.

In the following, we discuss Step 1 and Step 3.

SIMD functional unit reduction and assembly code update (Steps 1 and 3): In Step 1 and Step 3, we can deal with hardware units other than SIMD functional units in the same way as in [17]. Then we discuss here SIMD functional unit reduction and its corresponding assembly code update.

For any SIMD functional unit u added to a processor core, we consider to (a) replace u with a SIMD functional unit u' which has the same functions with u and has the smaller hardware cost than u or (b) eliminate some sub-function of u.

(a) is realized by calling our SIMD functional unit generator proposed in [11]. If u is replaced with u', assembly code update is unnecessary since the function of u' is just the same as that of u.

Now let us focus on the case of (b). The SIMD functional unit u can execute several SIMD instructions. Then we can consider a sub-function corresponding to each SIMD instruction and eliminate the sub-function from the SIMD functional unit. After eliminating the sub-function in a SIMD functional unit, we update an assembly code according to a new SIMD functional unit. Note that, we eliminate a sub-function in SIMD functional units only when the SIMD instruction is executed by another SIMD functional unit.

For example, we assume that a processor core has two SIMD multipliers, mul_1 and mul_2, each of which can execute the two SIMD instructions `MUL_2_ur4s` and `MUL_4_sr7w`. The SIMD multiplier configuration is shown in Fig. 5. Each SIMD multiplier, mul_1 or mul_2, is composed of a multiplier for two and four data, a 4-bit and 7-bit right shifter, and a saturation unit. Using these mul_1 and mul_2, we can execute the following two parallel instructions in two clock cycles.

```
MUL_2_ur4s R1,R2,R3 || MUL_2_ur4s R4, R5, R6
MUL_4_sr7w R7,R8,R9 || MUL_4_sr7w R10,R11,R12
```

The first instruction is executed by using mul_1 and mul_2 and the second instruction is also executed by using mul_1 and mul_2.

Consider to eliminate the sub-function corresponding to `MUL_2_ur4s` in mul_2. The configuration of mul_2 is changed so that it can execute only `MUL_4_sr7w`. We have the new SIMD functional unit mul_2' as shown in Fig. 6. mul_2' is composed of a multiplier for two data and a 7-bit right shifter. Comparing the configuration of mul_2 and that of mul_2', the hardware cost of mul_2' must be smaller than that of mul_2. However, mul_2' cannot execute `MUL_2_ur4s`. Then if we eliminate the sub-function corresponding to `MUL_2_ur4s` in mul_2, we must update the above assembly code as follows:

```
MUL_2_ur4s R1,R2,R3
MUL_2_ur4s R4,R5,R6
MUL_4_sr7w R7,R8,R9 || MUL_4_sr7w R10,R11,R12
```

The first instruction is executed by mul_1 and the second instruction is also executed by mul_1. The third instruction is executed by using mul_1 and mul_2'.

In this way, we try to eliminate each of the hardware units, sub-functions of hardware units, and registers in Step 1. Then in Step 3, we update an assembly code according to a new processor core configuration.

Based on this algorithm, we can reduce redundant sub-functions in SIMD functional units and then we can find an optimal processor core configuration.

4 Experimental Results and Conclusion

The proposed hardware/software partitioning algorithm has been implemented in the C language on Sun Ultra Workstation. The algorithm was applied to the Alpha Blend (image size of 640 × 480 pixels) and the Copying Machine Application (image size of 640 × 480 pixels). The basic bit width of a processor core is set to be 32 bits and the number of instructions executed concurrently is set to be four.

Tables 4 and 5 show the experimental results. In the tables, Const shows timing constrains, Area shows synthesized processor core area, Time shows execution time for running an application program, and Hardware configuration shows hardware configuration for synthesized processor cores. In the tables, SIMD functional unit configuration is shown as follows: Assume that a synthesized SIMD processor core has one ALU and two SIMD ALUs, $salu1$ and $salu2$, where $salu1$ and $salu2$ have two SIMD ALU instructions and one ALU insturction, respectively. This ALU confiugration is shown as $(1, 2[2, 1])$.

The tables indicate that, our hardware/software partitioning algorithm configures appropriate SIMD functional units depending on the given application programs and timing constraints. If a similar timing constraint is given to a non-SIMD processor core and a SIMD processor core, an area of a SIMD processor core

Table 4. Experimental results (Alpha Blend).

	Consts	Area	Time	Hardware configuration								
	[ms]	[μm^2]	[ms]	Kernel	#ALUs	#SFTs	#MULs	#MACs	#Regs	Y-mem	Addr unit	HW loop
Non SIMD ([17])	18.0	11,591,021	17.740	DSP	2	1	2	3	(7, 3, 1)	Yes	X[1,2], Y[1,2]	Yes
	20.0	5,672,754	18.923	DSP	2	1	1	1	(6, 3, 1)	Yes	X[1,2], Y[1,2]	Yes
	22.0	3,839,427	20.106	DSP	2	1	1	0	(8, 3, 1)	Yes	X[1,2], Y[1,2]	Yes
	24.0	3,672,783	22.471	DSP	2	1	1	0	(8, 3, 0)	Yes	X[1,2], Y[1,2]	No
	32.0	3,549,223	30.750	DSP	2	1	1	0	(7, 3, 0)	Yes	X[1,2], Y[1,2]	No
Packed SIMD	4.5	6,299,770	4.421	DSP	0, 2[2,2]	1, 0	0, 3[1,1,1]	0, 2[1,1]	(8, 3, 1)	Yes	X[1,2], Y[1,2]	Yes
	5.0	4,065,268	4.886	DSP	0, 1[3]	1, 0	0, 2[1,1]	0, 1[1]	(8, 3, 1)	Yes	X[1,2], Y[1,2]	Yes
	6.0	2,873,058	5.584	DSP	0, 1[3]	1, 0	0, 1[1]	0, 0	(10, 3, 0)	Yes	X[1,2], Y[1,2]	No
	7.0	2,857,929	6.981	DSP	0, 1[3]	1, 0	0, 1[1]	0, 0	(12, 0, 0)	Yes	No	No
	18.0	2,656,377	13.962	DSP	0, 1[3]	1, 0	0, 1[1]	0, 0	(9, 0, 0)	Yes	No	No
	20.0	2,656,377	13.962	DSP	0, 1[3]	1, 0	0, 1[1]	0, 0	(9, 0, 0)	Yes	No	No

Table 5. Experimental results (Copying Machine Application).

	Consts	Area	Time	Hardware configuration							
	[ms]	[μm^2]	[ms]	Kernel	#ALUs	#SFTs	#MULs	#Regs	Y-mem	Addr unit	HW loop
Non SIMD ([17])	50.5	8,753,937	50.295	DSP	4	1	4	(69, 6, 1)	Yes	X[1,2], Y[1,2]	Yes
	100.0	5,086,785	99.421	DSP	2	1	1	(48, 6, 0)	Yes	X[1,2], Y[1,2]	No
	250.0	3,944,657	249.138	DSP	2	1	1	(31, 6, 0)	Yes	X[1,2], Y[1,2]	No
	500.0	2,668,161	499.446	DSP	1	1	1	(12, 6, 0)	Yes	X[1,2], Y[1,2]	No
Packed SIMD	5.7	10,698,879	5.688	DSP	0, 4[2,2,2,2]	1, 0	0, 4[1,1,1,1]	(69, 6, 1)	Yes	X[1,2], Y[1,2]	Yes
	10.0	5,500,743	9.783	DSP	2, 2[2,2]	1, 0	0, 1[1]	(44, 6, 0)	Yes	X[1,2], Y[1,2]	No
	50.0	3,155,438	49.837	RISC	1, 1[2]	1, 0	0, 1[1]	(10, 0, 0)	Yes	No	No
	100.0	2,696,463	99.881	DSP	1, 1[2]	1, 0	0, 1[1]	(6, 6, 0)	Yes	X[1,2], Y[1,2]	No

#ALUs for SIMD cores: (#ALUs, #SIMD ALUs[#SIMD instructions in SIMD ALU1,. . .])
#SFTs for SIMD cores: (#Shifters, #SIMD Shifters[#SIMD instructions in SIMD Shifter1, . . .])
#MULs for SIMD cores: (#MULs, #SIMD MULs[#SIMD instructions in SIMD MUL1, . . .])
#MACs for SIMD cores: (#MACs, #SIMD MACs[#SIMD instructions in SIMD MAC1, . . .])
#Regs: (#General registers, #Address registers, #Loop registers)
Addr unit: Address unit configuration. X[1,2] (or Y[1,2]) means that the X (or Y) data memory has the addressing unit with post increment operation.

can be smaller than that of a non-SIMD processor core for both application programs. Tables 4 and 5 show that the area of SIMD processor core is 22–53 % smaller than that of non-SIMD processor cores configured under the similar timing constraints. Because the numbers of functional units and reginsters added to SIMD processor cores are smaller than that of non-SIMD processor cores.

By using our new hardware/software partitioning algorithm, we can find a processor core architecture with small area satisfying a given timing constraint. Now our algorithm is a greedy heuristic approach, but for larger applications we may need more efficient heuristics. In the future, we will improve our algorithm so that it can optimize the configuration of each SIMD functional unit by reducing several sub-functions at once. Thus we will have globally optimized hardware/software partitioning.

Acknowledgement

This research is supported in part by STARC (Semiconductor Technology Academic Research Center).

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] 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.

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

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

[5] Intel, *MMX Technology Architecture Overview*, http://www.intel.com/technology/itj/q31997/articles/art 2.htm, 1997.

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

[7] H. Liu and D. F. Won, "Integrated partitioning and scheduling for hardware/software codesign," in *Proc. International Conference on Computer Design*, 1998.

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

[9] MIPS Technologies, *MIPS Extension for digital media with 3D*, 1997.

[10] M. Mittal, A. Peleg, and U. Weiser, "MMX technology architecture overview," *Intel Technology Journal*, 3rd Quarter, 1997.

[11] Y. Miyaoka, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A hardware unit generation algorithm for a hardware/software cosynthesis system of digital signal processor cores with packed SIMD type instructions," *Transactions on Information Processing Society of Japan*, vol.43, no.5, pp.1191–1201, 2002, (in japanese).

[12] E. F. Nurprasetyo, A. Inoue, H. Tomiyama, and H. Yasuura, "Soft-core processor architecture for embedded system design," IEICE Trans. on Electron, vol.E81-C, no.9, pp.1416–1423, 1998.

[13] N. Nonogaki, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A parallelizing compiler in a hardware/software cosynthesis system for image/video processor with packed SIMD type instruction sets," IEICE Technical Report, VLD2000-139, ICD2000-215, 2001, (in japanese).

[14] 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.

[15] Sun Microsystems, *VIS Instruction Set User's Manual*, 1997.

[16] Tensilica, *Xtensa Microprocessor: Overview Handbook*, http://www.tensilica.com/.

[17] N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A hardware/software cosynthesis system for digital signal processor cores," *IEICE Trans. on Fundamentals*, vol. E82-A, no. 11, pp. 2325–2337, 1999.

[18] N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A hardware/software cosynthesis system for digital signal processor cores with two types of register files," *IEICE Trans. on Fundamentals*, vol. E83-A, no. 3, 2000.