Instruction Set and Functional Unit Synthesis for SIMD Processor Cores

Nozomu Togawa[†,‡] Koichi Tachikake[††] Yuichiro Miyaoka[††] Masao Yanagisawa[††] Tatsuo Ohtsuki[††]

[†]Dept. of Information and Media Sciences, The University of Kitakyushu
[††]Dept. of Electronics, Information and Communication Engineering, Waseda University
[‡]Advanced Research Institute for Science and Engineering, Waseda University
1-1 Hibikino, Wakamatsu, Kitakyushu 808-0135, Japan
Tel: +81-93-695-3264 Fax: +81-93-695-3368
E-mail: togawa@env.kitakyu-u.ac.jp

Abstract—This paper focuses on SIMD processor synthesis and proposes a SIMD instruction set/functional unit synthesis algorithm. Given an initial assembly code and a timing constraint, the proposed algorithm synthesizes an area-optimized processor core with optimal SIMD functional units. It also synthesizes a SIMD instruction set. The input initial assembly code is assumed to run on a full-resource SIMD processor (virtual processor) which has all the possible SIMD functional units. In our algorithm, we introduce the SIMD operation decomposition and apply it to the initial assembly code and the full-resource SIMD processor. By gradually reducing SIMD operations or decomposing SIMD operations, we can finally find a processor core with small area under the given timing constraint. The promising experimental results are also shown.

I. Introduction

Let us consider a b-bit functional unit. It can execute a single b-bit operation. By modifying it slightly, it can also execute n-parallel b/n-bit sub-word operations. These operations are called *packed SIMD type operations* or *SIMD operations* [4], [7], [12]. A functional unit modified to execute SIMD operations is called a *SIMD functional unit*. For example, a 32-bit SIMD adder can execute a single 32-bit addition or 4-parallel 8-bit additions. A micro processor with SIMD functional units is called a *SIMD processor*. It can be effectively applied to image processing.

Generally, a SIMD operation has very many parameters. We can configure so many different SIMD operations and we can have so many SIMD functional unit configurations. However, a particular image application program often uses very limited SIMD operations which leads to a limited number of SIMD functional unit configurations. We consider that appropriate configuration for a image processor core is required depending on application programs as well as hardware costs.

Processor synthesis or ASIP synthesis has been studied for many years such as in [1], [3], [6], [9], [11], [17]. Tensilica develops the Xtensa system for application-specific processor synthesis [14]. All the systems proposed so far, however, focus on conventional micro processor cores and/or DSP cores and then they do not deal with automatic SIMD processor synthesis.

Now let us pick up a single SIMD operation *op*. It is usually composed of several SIMD sub-operations, such as an arithmetic sub-operation, a shift sub-operation, and a bit-saturation sub-operation. We can consider the following two cases for executing the SIMD operation by SIMD instructions:

Case 1: We can consider a single SIMD instruction i which directly executes *op* in one clock cycle. A SIMD functional unit executing i must be complex and may have a large area and delay.

Case 2: We can also consider a decomposed SIMD instruction set of $\{i_1, i_2, \cdots, i_n\}$ (n is the number of sub-operations in *op*). *op* can be executed by a combination of $i_1, i_2, \cdots, i_n$. A SIMD functional unit executing an instruction $i \in \{i_1, i_2, \cdots, i_n\}$ must be simple and may have a small area and delay. A decomposed SIMD instruction can be commonly used in several SIMD operations.

By introducing SIMD operation decomposition, we can have a compact set of SIMD instructions. Our previous study on SIMD processor synthesis is appeared in [13] but it does not deal with SIMD instruction decomposition.

In this paper, we focus on SIMD processor synthesis and propose a SIMD instruction set/functional unit synthesis algorithm. The algorithm is based on [15]. Given an initial assembly code and a timing constraint of execution time, the proposed algorithm synthesizes an area-optimized processor core with SIMD functional units. It also outputs a new assembly code under a synthesized SIMD instruction set. The input initial assembly code is assumed to run on a full-resource SIMD processor (virtual processor) which has all the possible SIMD functional units. The initial assembly code includes complex SIMD instructions. In our algorithm, we introduce SIMD operation decomposition and apply it to the initial assembly code and full-resource SIMD processor. By gradually reducing SIMD operations or decomposing SIMD operations, we can finally find a processor core with small area under the given timing constraint. We expect that we can have a processor core which has appropriate SIMD functional units for running an input application program.

II. Processor Model and Instruction Set

Our processor architecture model is shown in Fig. 1 [8], [10], [13], [16]. The model is composed of a *processor kernel* and extra *hardware units*. A *processor core* is constructed by adding several hardware units to a processor kernel.

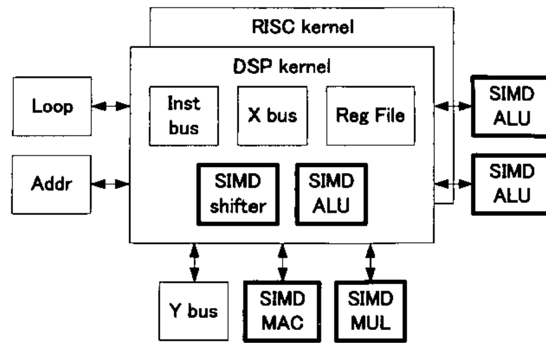


Fig. 1. Processor kernels and hardware units.

TABLE I
SIMD OPERATION TYPES.

SIMD operation class	SIMD operation type
(a) Arithmetic operation	Addition (ADD) Subtraction (SUB) Multiplication (MUL) Multiply and Addition (MAC)
(b) Shift operation	Arithmetic Right Shift (SRA) Arithmetic Left Shift (SLA) Logical Left Shift (SLL)
(c) Bit extend/extract operation	Bit Extend (EXTD) Bit Extract (EXTR)
(d) Data move operation	Data Move (EXCH)

TABLE II
SIMD FUNCTIONAL UNIT AND ITS EXECUTING SIMD OPERATION TYPES.

SIMD functional unit type	SIMD operation types
SIMD Shifter (sft)	SRA, SLA, SLL
SIMD ALU (alu)	ADD, SUB
SIMD Multiplier (mul)	MUL
SIMD MAC unit (mac)	MAC
SIMD Bit extractor/extender (ext)	EXTD, EXTR
SIMD Data move unit ($exch$)	EXCH

A. 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 [2]. A DSP-type kernel has the three pipeline stages (IF, ID, and EXE) as in the DSP processors of [5]. Each processor kernel has Harvard architecture and consists of (c-i) a bus for an instruction memory, (c-ii) a bus for an X data memory (X-bus), (c-iii) a register file for general-purpose registers, and (c-iv) an ALU and a barrel shifter. Data bus width of the instruction memory and the X data memory can be changed but their address bus width is fixed to 16 bits. The number of registers and their bit width in the register file can be changed. In our processor model, data bus width of the X data memory is the same as the bit width of the register file. Data bus width of the instruction memory is determined based on a synthesized instruction set.

B. Hardware Units

Our processor core can have extra hardware units: (1) SIMD functional units, (2) a Y-bus for Y data memory, (3) addressing units, and (4) hardware loop units. In this section, we focus on SIMD operations and SIMD functional units.

SIMD operation: Our SIMD processor executes four classes of SIMD operations: (a) SIMD arithmetic operations, (b) SIMD shift operations, (c) SIMD bit extend/extract operations, and (d) data move operations.

As an example of (a) *SIMD arithmetic operations*, we show two types of SIMD multiplications in Figs. 2 (a) and (b). In Fig. 2(a), two four-packed data are multiplied and the four results are packed into a single register. In Fig. 2(b), the lower two sub-words of two four-packed data are multiplied and the two results are packed into a single register. Such a SIMD arithmetic operation has the *SIMD parameters* of (0) operation type (see Table I), (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. A *SIMD shift operation* has the same parameters of SIMD arithmetic operations.

A *SIMD bit extend operation* constructs $n/2$-packed data from n-packed data (Fig. 3(a)). A *SIMD bit extract operation* constructs $2 \times n$-packed data from n-packed data (Fig. 3(b)). A *SIMD data move operation* gives new n-packed data by rearranging old n-packed data. They have similar SIMD parameters as the SIMD arithmetic operations.

If we give a particular value to each SIMD parameter, we can determine a particular *SIMD operation*. For example, we can consider a SIMD operation `MUL_4_sr2s` which 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 (<u>mul</u>tiplication, <u>4</u> packing data, <u>s</u>igned, <u>r</u>ight shift by <u>2</u> bits, and <u>s</u>aturated).

SIMD functional unit: Our SIMD processor has six types of SIMD functional units listed in Table II. Table II also shows SIMD operation types which each functional unit can execute.

A SIMD functional unit can have one or more SIMD operations. For example, let us consider a SIMD multiplier mul_0 which has the SIMD operations of `MUL_4_sr2s` and `MUL_2_sr2s`. mul_0 can execute one of the two SIMD operations, `MUL_4_sr2s` and `MUL_2_sr2s`, in a single clock cycle.

C. Instruction Set

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). A parallel instruction executes more than one

basic instructions. All the combination of basic instructions cannot be a parallel instruction. Our processor synthesizer determines which basic instructions should be included in a processor core and which combination of basic instructions should be a parallel instruction.

SIMD Instructions Our SIMD processor has SIMD instructions. The description of a SIMD instruction is the same as a SIMD operation. For example, our SIMD processor can have a SIMD instruction of `MUL_4_sr2s`. Since there are too many SIMD instruction instances, the proposed algorithm synthesizes which SIMD instructions are included in an instruction set.

Note that a single SIMD operation is executed by a single SIMD instruction or a sequence of SIMD instructions. For example, the SIMD operation `MUL_4_sr2s` is executed by a single SIMD instruction of "`MUL_4_sr2s R1,R2,R3`." It is also executed by the sequence of SIMD instructions of:

```
MUL_4h_s    R1,R2,R3
MUL_4l_s    R1,R2,R4
EXTR_4_sr2s R3,R4,R3
```

where Rx ($x = 1, \cdots$) shows a general-purpose register. In the latter case, the SIMD instructions sequentially execute the SIMD operation of `MUL_4_sr2s`.[1] In this case, the SIMD operation `MUL_4_sr2s` is *decomposed* into three SIMD sub-operations.

III. An Instruction Set and Functional Unit Synthesis Algorithm for SIMD Processor Cores

We have been developing a hardware/software cosynthesis system for SIMD processor cores [8], [10], [13], [16].

The system is composed of *Process 1: full-resource compiling*, *Process 2: hardware/software partitioning*, and *Process 3: hardware/software generation*. Given an application program in C and a set of its application data, our system synthesizes a processor core description and generates an object code and a software environment (compiler, assembler and simulator) under *the timing constraint*. The objective is to minimize the hardware area of a processor core.

In this section, we focus on Process 2 in our SIMD processor synthesis and propose a new algorithm with a SIMD instruction set/functional unit synthesis.

A. Problem Definition

First we define our SIMD instruction set/functional unit synthesis problem. Assume that a SIMD processor core configuration P and its corresponding instruction set $I(P)$ is given. We can have a clock period $T(P)$ for P [16]. When an application program ap is compiled into an assembly code $C_{ap}(P)$ under $I(P)$, we can also have a total clock cycle $N_{ap}(P)$ to run the application program on P. Then the execution time $T_{ap}(P)$ to run the application program on P is expressed by:

$$T_{ap}(P) = N_{ap}(P) \times T(P). \quad (1)$$

The SIMD processor core configuration P has an area cost of $A(P)$, which is expressed by:

$$A(P) = A_{kernel}(P) + \sum_{u \in U(P)} A(u) \quad (2)$$

where $A_{kernel}(P)$ is an area cost of the processor kernel of P, $U(P)$ is a set of hardware units in P, and $A(u)$ is an area cost for each hardware unit $u \in U(P)$.

A full-resource SIMD processor FP is a virtual processor core which has all the hardware units including SIMD functional units with all possible SIMD operations. Each of the SIMD instructions in FP executes a complex SIMD operation in a single clock cycle, i.e., all the SIMD operations are not decomposed. Then, for an input application program ap, we can construct an initial assembly code $C_{ap}(FP)$ which is run on FP.

Then our SIMD instruction set/functional unit synthesis problem is defined as follows:

Definition 1 *Given an initial assembly code $C_{ap}(FP)$ and a timing constraint T_{max}, find a new processor core configuration P, a new instruction set $I(P)$, and a new assembly code $C_{ap}(P)$, under the constraint of $T_{ap}(P) \leq T_{max}$ so as to minimize $A(P)$.*

B. The Algorithm

The proposed algorithm is an extended version of the algorithm in [15] so that it can deal with SIMD instructions and SIMD functional units. Our approach is heuristic but we expect that it can find a globally good solution in a practical time since it simultaneously optimizes the numbers, types, and functions of hardware units including SIMD functional units.

The algorithm is composed of Phase 1 and Phase 2.

B-I Phase 1. Configure an Initial Processor Core P_i

Phase 1 determines an initial processor core P_i. First, 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 the *bit width of a register file* is set to $b_{knl,fu}$. The number of registers in a register file is given as a maximum number of registers appeared in an input assembly code. The data bus width of an instruction memory is determined based on the instructions used in an assembly code.

Next, 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. For example, if an input assembly code uses three loop registers, we add the hardware loop unit with three loop registers to a processor kernel.

Finally, we must synthesize a set of SIMD functional units in P_i as follows.

[1] In MUL_4h_s (or MUL_4l_s), the higher (or lower) two sub-words of the two four-packed data given by R1 and R2 are multiplied and the two results are packed into a single register R3 (or R4) as in Fig. 2(b). Then EXTR_4_sr2s extracts appropriate bits of R3 and R4, shifts the resultant data, and packs them into R3.

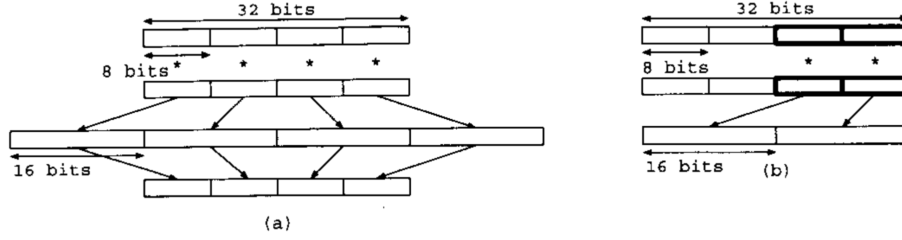


Fig. 2. SIMD multiplications. (a) Four 8-bit multiplications. (b) Two 16-bit bit-extend multiplications.

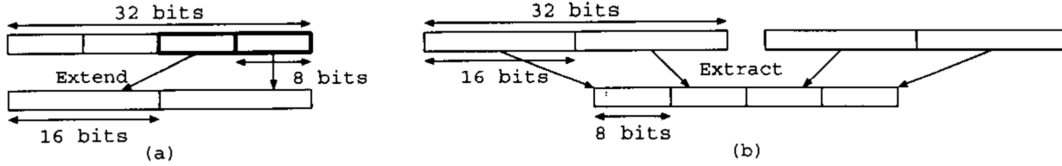


Fig. 3. (a) Bit extend operation and (b) bit extract operation.

Initial SIMD functional unit synthesis: The configuration of each SIMD functional unit is determined in the following way. For each SIMD functional unit type $t \in \{sft, alu, mul, mac, ext, exch\}$, let I_t be a set of the SIMD instructions in an input assembly code which can be executed by a SIMD functional unit with type t. We construct a SIMD functional unit with type t so that it has all the SIMD operations corresponding to I_t.

Example 1 Let us assume that an input assembly code includes the SIMD instructions of `MUL` (normal 1-pack multiplication), `MUL_4_ur4s` (multiplication for four-packed data), and `MUL_2_sr7w` (multiplication for two-packed data). In this case, we construct a SIMD multiplier which has SIMD operations of `MUL`, `MUL_4_ur4s`, and `MUL_2_sr7w`. The SIMD multiplier can execute each of the SIMD instructions `MUL`, `MUL_4_ur4s`, and `MUL_2_sr7w` in one clock cycle. □

The number of each functional unit is determined in the following way. If a maximum of n_t instructions are executed concurrently for I_t in an input assembly code, we add n_t functional units with the type of t to a processor kernel.

Example 2 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 SIMD multipliers whose configuration is shown in Example 1 to a processor kernel. □

The constructed *initial processor core* P_i may include redundant SIMD operations in a SIMD functional unit but they will be reduced in Phase 2. Since P_i includes all the hardware units required for an input assembly code, the initial assembly code can be executed on P_i and furthermore we expect that it can satisfy the given timing constraint.

B-II Phase 2: Determine a SIMD instruction set and SIMD functional unit

Based on the parameters determined by Phase 1, Phase 2 determines a processor core configuration P, i.e., it determines (1) a processor kernel type (RISC or DSP), (2) the number of general-purpose registers, (3) whether the Y data memory is actually 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) SIMD functional unit configuration, depending on an input assembly code and timing constraint. Phase 2 also determines an instruction set $I(P)$ for P.

Firstly, we assume that a processor core has a RISC-type kernel or a DSP-type kernel. Then, for each of kernels, we reduce the parameters in (1)–(6) one by one while the processor core satisfies the timing constraint. Finally, we pick up the processor core with the smaller area. We can find a processor core architecture which has a small area with 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/register u is defined as:

$$T_{rate}(u) = \frac{T_1(u) - T_0}{A_0 - A_1(u)}, \tag{3}$$

where A_0 and T_0 refer to an area cost and execution time of the processor core before eliminating u, and $A_1(u)$ and $T_1(u)$ refer to an area cost and execution time of the processor core after eliminating u. All these values are computed by the area/delay estimator in [16]. Step 2 finds u_{min} which gives minimum $T_{rate}(u_{min})$ and actually eliminates u_{min} from a current processor core. By using the $T_{rate}(u)$ value, we can effectively reduce an area cost of a processor core with

Inputs: Assembly code $C_{ap}(P_i)$, initial processor core P_i, and timing constraint T_{max}.

Outputs: New processor core P, new assembly code $C_{ap}(P)$, and new instruction set $I(P)$

Phase 2. For P_i, we assume DSP-type kernel or RISC-type kernel. For each of the kernels, let $P \leftarrow P_i$ and execute Steps 1–4. Between them, output the processor core with the smaller area, its corresponding assembly code, and instruction set.

Step 1. For each u in the hardware units/registers in P, try to eliminate u.

Step 2. Evaluate the $T_{rate}(u)$ value. For u_{min} which gives the minimum $T_{rate}(u_{min})$ value satisfying $T_{ap}(P') \leq T_{max}$, eliminate u_{min} from P and update P to P'.

Step 3. Update the assembly code and instruction set according to P'.

Step 4. Let $P \leftarrow P'$. While there exists a hardware unit/register which satisfies Step 2, repeat Steps 1–3. Otherwise finish.

Fig. 4. The algorithm of Phase 2.

satisfying a timing constraint. See [13] for discussion on T_{rate} design.

In the following, we discuss Step 1 and Step 3 for SIMD functional units and SIMD operations.

SIMD operation reduction and assembly code/instruction set update (Step 1 and Step 3): In Step 1 and Step 3, we can try to reduce hardware units/registers other than SIMD functional units in the same way as in [15]. For example, in case a hardware loop unit is eliminated from a processor core, we replace the instruction using the hardware loop unit with a normal conditional jump instruction. Then we discuss here how to try to eliminate a SIMD operation in a SIMD functional unit.

Let $FU(P)$ be a set of SIMD functional units in P. Let $OP(fu)$ be a set of SIMD operations in $fu \in FU(P)$. Now we try to eliminate a SIMD operation $op \in OP(fu)$ from fu. We can consider the two cases:

Case A: There is another functional unit $fu' \in FU(P)$ such that $op \in OP(fu')$.

Case B: There is no functional unit $fu' \in FU(P)$ such that $op \in OP(fu')$.

Case A: In this case, the SIMD instruction corresponding to *op* can be executed by fu' instead of fu. Thus we simply eliminate *op* from fu and construct a new processor core P'.

Case B: In this case, we eliminate *op* from fu by *SIMD operation decomposition.* An arithmetic SIMD operation which gives n-packed data from two n-packed data is called a *full SIMD operation.* Fig. 2(a) shows an example of a full SIMD operation. When the type of a full SIMD operation *op* is addition (ADD), subtraction (SUB), multiplication (MUL), or multiply and addition (MAC), *op* is called a *decomposable SIMD operation.* We will decompose only a decomposable SIMD operation. Then it will be executed by a sequence of decomposed SIMD instructions.

Let $op \in OP(fu)$ for a SIMD functional unit fu be a decomposable SIMD operation. Generally *op* is composed of (1) n-parallel SIMD arithmetic sub-operation followed by (2) n-parallel SIMD shift sub-operation and bit-saturation sub-operation. n-parallel SIMD arithmetic sub-operation is further composed of two $n/2$-parallel SIMD arithmetic sub-operations ((1-1) and (1-2)). Then we can decompose the operation *op* into:

(1-1) $n/2$-parallel arithmetic sub-operation for $n/2$ upper sub-words,

(1-2) $n/2$-parallel arithmetic sub-operation for $n/2$ lower sub-words, and

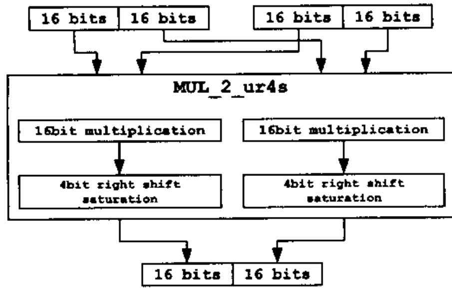
(2) n-parallel SIMD shift sub-operation and bit-saturation sub-operation.

A SIMD functional unit executing $n/2$-parallel arithmetic sub-operation has much smaller area cost than a SIMD functional unit executing n-parallel arithmetic operation. Furthermore, since the sub-operations of (1-1) and (1-2) do not have any particular shift operations or bit-saturation operations, they can be shared by many SIMD instructions. Overall we can reduce a processor core area by decomposing a decomposable SIMD operation in the above way.

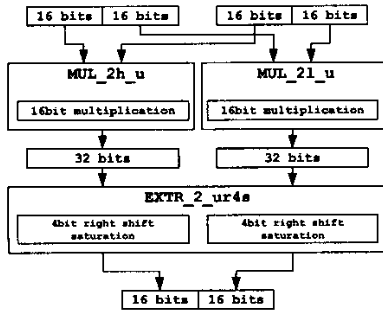
Example 3 Let us consider a decomposable SIMD operation `MUL_2_ur4s`. It can be executed by a single SIMD instruction `MUL_2_ur4s` as shown in Fig. 5(a). Fig. 5(a) can be decomposed into Fig. 5(b). First, `MUL_2_ur4s` is composed of 2-parallel 16-bit multiplication followed by 2-parallel 4-bit right shift and bit-saturation. The 2-parallel 16-bit multiplication can be further decomposed into two 1-parallel 16-bit multiplications, (1-1) `MUL_2h_u` and (1-2) `MUL_2l_u` for the upper sub-word and the lower sub-word of the input two data, respectively. The 2-parallel 4-bit right shift and bit-saturation operation is executed by (2) `EXTR_2_ur4s` (Fig. 5(b)). After all, `MUL_2_ur4s` is decomposed into `MUL_2h_u`, `MUL_2l_u`, and `EXTR_2_ur4s`. Note that the intermediate results in Fig. 5(b) must have 32-bit width to keep the correct results. □

According to SIMD operation decomposition, we update a set $FU(P)$ of SIMD functional units, assembly code $C_{ap}(P)$, and instruction set $I(P)$. Assume that a decomposable SIMD operation $op \in OP(fu)$ is decomposed into a set $DOP = \{op_s | op_s$ is a decomposed sub-operation for $op\}$.

$FU(P)$ is updated as follows: We first eliminate *op* from fu. For all of each SIMD sub-operation $op_s \in DOP$, if there exists a SIMD functional unit $fu' \in FU(P)$ such that $op_s \in OP(fu')$, $FU(P)$ is unchanged since fu' can execute op_s. If there exists no such SIMD functional unit and there exist a functional unit $fu'' \in FU(P)$ which has the same operation type as op_s, then we add op_s to fu'', i.e, $OP(fu'') \leftarrow OP(fu'') \cup \{op_s\}$. Otherwise, we construct a new SIMD functional unit fu_{new} which includes only op_s and add it to $FU(P)$, i.e., $FU(P) \leftarrow FU(P) \cup \{fu_{new}\}$. According to a new set of SIMD functional units, we update a processor core P to P'.

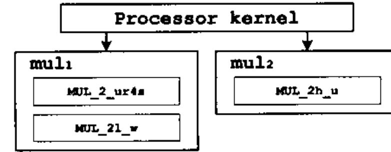


(a)



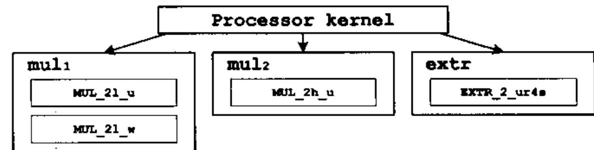
(b)

Fig. 5. Non-decomposed SIMD operation (a) and its decomposed SIMD sub-operations (b).




```
MUL_2_ur4s  R0,R1,R2 || MUL_2h_u R3,R4,R5
MUL_21_w    R6,R7,R8 || MUL_2h_u R9,R10,R11
```

(a)




```
MUL_2h_u    R0,R1,R12  || MUL_21_u R0,R1,R2
EXTR_2_ur4s R12,R2,R2  || MUL_2h_u R3,R4,R5
MUL_21_w    R6,R7,R8   || MUL_2h_u R9,R10,R11
```

(b)

Fig. 6. Processor core and assembly code before SIMD operation decomposition (a) and after SIMD operation decomposition (b).

Example 4 Assume that the processor core P has SIMD functional units as in Fig. 6(a). If the SIMD operation MUL_2_ur4s in mul_1 is decomposed, we have sub-operations of MUL_2h_u, MUL_21_u, and EXTR_2_ur4s. Since mul_2 has the SIMD operation MUL_2h_u, it is executed by mul_2. Since mul_1 is a multiplier and it has the same operation type as MUL_21_u, we add it into mul_1 as a new SIMD operation. Since we do not have a SIMD bit extractor in P, we add a SIMD bit extractor having EXTR_2_ur4s into P. Thus we finally obtain a new processor core P' as shown in Fig. 6(b). Since area for MUL_2_ur4s is much larger than that for the total area of MUL_21_u and EXTR_2_ur4s, area cost of the new processor core P' can be reduced. □

Since a SIMD operation op is decomposed into DOP, a SIMD instruction corresponding to op is eliminated from the assembly code $C_{ap}(P)$ and instruction set $I(P)$. Instead, each $op_s \in DOP$ is added to an instruction set. In the assembly code, op is replaced with DOP. Figs. 6(a) and (b) show the example of updating an assembly code. We may need extra clock cycles for a new assembly code but can reduce an area cost for a new processor core.

We try to eliminate each decomposable SIMD operation by decomposing it, and then we actually decompose the decomposable SIMD operation if it gives minimum T_{rate} value among other hardware unit/register reduction trials. By repeating this process, we expect that we can have a compact SIMD instruction set and its corresponding SIMD functional units.

IV. Experimental Results

The proposed SIMD instruction set/functional unit synthesis algorithm has been incorporated into our SIMD processor synthesis system. 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 maximum number of basic instructions and SIMD instructions executed concurrently is set to be four. In this experiment, we used an Intel Pentium III (850MHz)-based PC with 256MB memory. Also, we assumed the Hitachi VDEC libraries (0.35μm-CMOS) to obtain processor area and speed. For comparison, the system proposed in [15] is also applied to both applications. The system proposed in [15] deals with a normal DSP processor core.

Tables III and IV show the experimental results. In the tables, Consts 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 two SIMD ALUs, $alu1$ and $alu2$, where $alu1$ and $alu2$ have three SIMD ALU operations and four ALU operations, respectively. This ALU configuration is shown as $(2[3,4])$.

The tables indicate that, our algorithm configures appropriate

TABLE III

EXPERIMENTAL RESULTS (ALPHA BLEND).

	Consts [ms]	Area [μm^2]	Time [ms]	Hardware configuration							CPU time [sec]
				Kernel	#ALUs	#MULs	#MACs	#Regs	Addr unit	HW loop	
[15]	20.0	17,305,942	28.108	DSP	2	2	3	(8, 3, 1)	X[1,2], Y[1,2]	Yes	0.84
	40.0	7,524,999	33.730	DSP	2	1	1	(8, 3, 0)	X[1,2], Y[1,2]	No	3.21
	60.0	6,882,647	59.330	DSP	1	1	1	(7, 3, 0)	X[1,2], Y[1,2]	No	4.54
	80.0	6,790,695	73.713	DSP	1	1	1	(5, 3, 0)	X[1,2], Y[1,2]	No	5.19
	100.0	6,744,719	98.883	DSP	1	1	1	(4, 3, 0)	X[1,2], Y[1,2]	No	5.53
	120.0	6,698,743	104.277	DSP	1	1	1	(3, 3, 0)	X[1,2], Y[1,2]	No	6.03
Ours	4.0	3,345,320	4.610	DSP	1[2]	3[1,1,1]	3[1,1,1]	(8, 3, 1)	X[1,2], Y[1,2]	Yes	7.20
	5.0	2,961,206	4.866	DSP	1[2]	3[1,1,1]	2[1,1]	(8, 3, 1)	X[1,2], Y[1,2]	Yes	32.59
	10.0	1,396,746	9.791	DSP	1[3]	1[2]	0	(11, 3, 0)	X[1,2], Y[1,2]	No	59.44
	20.0	1,275,480	15.131	DSP	1[3]	1[2]	0	(8, 3, 0)	X[1,2], Y[1,2]	No	56.67
	30.0	1,213,906	21.140	DSP	1[3]	1[2]	0	(7, 3, 0)	X[1,2], Y[1,2]	No	68.44
	40.0	1,132,862	36.939	DSP	1[3]	1[2]	0	(5, 3, 0)	X[1,2], Y[1,2]	No	92.85

TABLE IV

EXPERIMENTAL RESULTS (COPYING MACHINE).

	Consts [ms]	Area [μm^2]	Time [ms]	Hardware configuration						CPU time [sec]
				Kernel	#ALUs	#MULs	#Regs	Addr unit	HW loop	
[15]	50.0	8,976,039	50.295	DSP	4	4	(69, 6, 1)	X[1,2], Y[1,2]	Yes	8.11
	100.0	4,271,449	99.520	DSP	2	1	(64, 6, 0)	X[1,2], Y[1,2]	No	118.98
	200.0	2,694,991	197.947	DSP	2	1	(25, 6, 0)	X[1,2], Y[1,2]	No	360.32
	300.0	2,331,193	286.531	DSP	2	1	(16, 6, 0)	X[1,2], Y[1,2]	No	413.40
	400.0	2,088,661	398.082	DSP	2	1	(10, 6, 0)	X[1,2], Y[1,2]	No	442.25
	450.0	2,048,239	444.014	DSP	2	1	(9, 6, 0)	X[1,2], Y[1,2]	No	443.49
Ours	5.0	6,985,419	5.543	DSP	4[4,4,4,4]	4[1,1,1,1]	(69, 6, 1)	X[1,2], Y[1,2]	Yes	68.05
	20.0	2,843,812	19.430	DSP	2[3,3]	1[1]	(41, 6, 0)	X[1,2], Y[1,2]	No	1853.19
	40.0	1,994,750	38.277	DSP	2[3,3]	1[1]	(20, 6, 0)	X[1,2], Y[1,2]	No	2255.04
	60.0	1,749,592	55.765	DSP	2[3,3]	1[2]	(15, 6, 0)	X[1,2], Y[1,2]	No	1993.81
	80.0	1,440,960	79.271	DSP	1[4]	1[2]	(11, 6, 0)	X[1,2], Y[1,2]	No	2081.89
	100.0	1,440,960	79.271	DSP	1[4]	1[2]	(11, 6, 0)	X[1,2], Y[1,2]	No	2078.97

#ALUs for SIMD cores: #SIMD ALUs[#SIMD operations in SIMD ALU1,. . .] (one of the ALUs is include in Kernel.)
#MULs for SIMD cores: #SIMD MULs[#SIMD operations in SIMD MUL1,. . .]
#MACs for SIMD cores: #SIMD MACs[#SIMD operations 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.
Kernel includes a SIMD shifter, which just has minimum shift-related instructions.

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 [15] and a SIMD processor core (proposed algorithm), an area cost of a SIMD processor core can be 1/10 compared with a non-SIMD processor core.

V. CONCLUSIONS

This paper proposed an instruction set/functional unit synthesis algorithm for SIMD processor cores. Experimental results show the effectiveness of the proposed algorithm.

In the current system, our system considers only timing constraints but will incorporate constraints for power dissipation as well as specific configuration of hardware units in the future.

ACKNOWLEDGMENT

This research was supported in part by fund from the MEXT (Ministry of Education, Culture, Sport, Science and Technology) via Kitakyushu innovative cluster project and Grant-in-Aid for Scientific Research No. 15700071.

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

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

[4] Intel, *MMX Technology Architecture Overview*, http://www.intel.com/technology/itj/q31997/articles/art_2.htm, 1997.

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

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

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

[8] 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).

[9] 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.

[10] 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).

[11] 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.

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

[13] K. Tachikake, N. Togawa, Y. Miyaoka, J. Choi, M. Yanagisawa, and T. Ohtsuki, "A hardware/software partitioning algorithm for SIMD processor cores," *Proc. IEEE Asia and South Pacific Design Automation Conference 2003 (ASP-DAC 2003)*, pp. 135–140, 2003.

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

[15] 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.

[16] N. Togawa, Y. Kataoka, Y. Miyaoka, M. Yanagisawa, and T. Ohtsuki, "Area and delay estimation in hardware/software cosynthesis for digital signal processor cores," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E83-A, no. 11, pp. 2639–2647, 2001.

[17] J.-H. Yang, B.-W. Kim, S.-J. Nam, J.-H. Cho, S.-W. Seo, C.-H. Ryu, Y.-S. Kwon, D.-H. Lee, J.-Y. Lee, J.-S. Kim, H.-D. Yoon, J.-Y. Kim, K.-M. Lee, C.-S. Hwang, I.-H. Kim, J.-S. Kim, K.-I. Park, K.-H. Park, Y.-H. Lee, S.-H. Hwang, I.-C. Park, and C.-M. Kyung, "MetaCore: An application specific DSP development system," in *Proc. 35th DAC*, pp. 800–803, 1998.