PAPER Special Section on Selected Papers from the 17th Workshop on Circuits and Systems in Karuizawa

Memory Allocation and Code Optimization Methods for DSPs with Indexed Auto-Modification

Yuhei KANEKO[†a)], ***Nonmember***, **Nobuhiko SUGINO**[†b)], ***Member***, ***and*** **Akinori NISHIHARA**[†c)], ***Fellow***

SUMMARY A memory address allocation method for digital signal processors of indirect addressing with indexed auto-modification is proposed. At first, address auto-modification amounts for a given program are analyzed. And then, address allocation of program variables are moved and shifted so that both indexed and simple auto-modifications are effectively exploited. For further reduction in overhead codes, a memory address allocation method coupled with computational reordering is proposed. The proposed methods are applied to the existing compiler, and generated codes prove their effectiveness.

key words: *indirect addressing, memory addressing, memory allocation, scheduling method*

1. Introduction

Digital Signal Processors (DSPs) are often used to realize real-time applications for their high performance. By using DSPs, various real-time algorithms can be realized flexibly. When a real-time signal processing is implemented on DSPs, efficient program codes or program codes short in execution time are required. In order to generate such efficient program codes, it is preferable to exploit hardware resources in DSPs as much as possible. For the reason, compilers, which can be generated efficient codes from high-level languages, are required. For example, in Refs. [6], [7] DIMPL (Digital network IMPlementation Language) and its compiler have been proposed. In these compilers, efficient program codes are derived by effective use of registers and arithmetic resources in a DSP.

A typical DSP architecture is shown in Fig. 1. In many DSPs, program variables in memory are usually accessed indirectly through an address register (AR) in address generation unit (AGU). Moreover, simple AR auto-modifications or update operations are often provided for array access. Since these auto-modifications are executed at the AGU in parallel with other arithmetic operations, memory allocation is very important issue to reduce overhead codes by memory access.

Some of the DSPs provide AR auto-modification by an index (IX) register. By use the IX register, further reduction in overhead codes is expected. References [1], [3] etc were proposed. However, heuristic memory allocation methods for such memory addressing are hardly known. In this paper, indirect addressing DSP with the index register modification is assumed, and a new memory allocation method is presented.

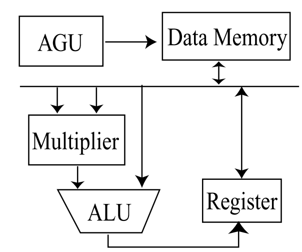


Fig. 1 DSP architecture.

Although the proposed memory allocation method is applied to memory accesses after computational ordering, more efficient memory access can be achieved for other computational orders. For further reduction in overhead codes over memory accesses, a memory access sequence is rearranged by computational ordering, while memory addresses are allocated for program variables.

2. Memory Addressing

In many DSPs, program variables stored in memory space are accessed by indirect addresing mode. In this paper, the processor of the following indirect addressing mode with indexed and simple auto-modifications is assumed.

Memory Addressing Mode

1. Memory addresses are pointed by AR.
2. AR can be post-modified for the next access by the following modifications.

Auto modification by 1 ($AR \leftarrow AR \pm 1$)

AR can be increased or decreased by one (AR) at every memory access.

Indexed auto-modification ($AR \leftarrow AR \pm IX$)

The assumed DSPs provide AR modification by an index register. AR can be increased or decreased by IX ($AR \leftarrow AR \pm IX$) at every memory access.
3. IX can be increased or decreased by one ($IX \leftarrow IX \pm 1$) at every memory access, where IX value is a positive integer.

Manuscript received July 2, 2004.
Manuscript revised October 8, 2004.
Final manuscript received December 9, 2004.
[†]The authors are with Tokyo Institute of Technology, Yokohama-shi, 226-8502 Japan.
a) E-mail: kaneko@ae.titech.ac.jp
b) E-mail: sugino@ae.titech.ac.jp
c) E-mail: aki@ec.ss.titech.ac.jp
DOI: 10.1093/ietfec/e88–a.4.846

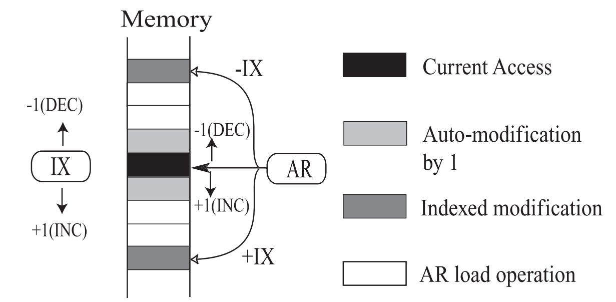


Fig. 2 Assumed indirect addressing mode.

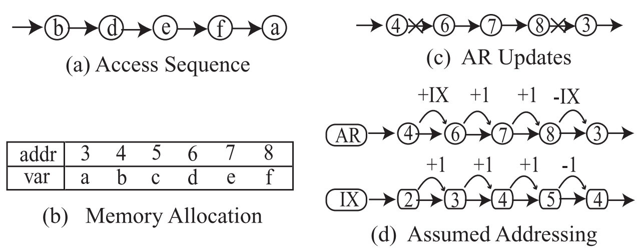


Fig. 3 Memory addressing.

4. When a required AR update cannot be achieved by use of operations in 2, an "AR load" operation, which directly substitutes a memory address into the AR, is required. This AR load operation costs one instruction cycle only by itself, so that it becomes an overhead over memory accesses.

The assumed memory addressing mode is depicted in Fig. 2.

For such an indirect addressing processor, the features of $AR \pm IX$ operation can be illustrated with an example memory access subsequence: b-d-e-f-a, where "b," "d," "e," "f," and "a" are program variables and they are accessed in this order (Fig. 3(a)). Assume a memory allocation for a program variable set V={a, b, c, d, e, f} as is given in Fig. 3 (b). If we do not use any $AR \leftarrow AR \pm IX$ operations, we have the memory access sequence shown in Fig. 3(c). Since "address distance" at b-d and f-a are $ad(b, d) = 2 > 1$ and $ad(f, a) = 5 > 1$, respectively, we need AR load operations at b-d and f-a.

In this paper, $AR \pm IX$ operation is used to update AR by ±2 or more. Furthermore, IX is updated by ±1 in parallel with the AR update operation. For the given example, when $AR \pm IX$ operations at b-d and f-a are assumed, no AR load is required as shown in Fig. 3(d). Note that IX is updated by +3 during this access sequence. When the update amount exceeds the IX value, however, we need an AR load operation, which becomes an overhead code in the program code.

3. Conditions Requiring AR Load Operation

3.1 Address Distance and Distance Sequence

Consider a memory access subsequence s_k, s_{k+1} in a memory

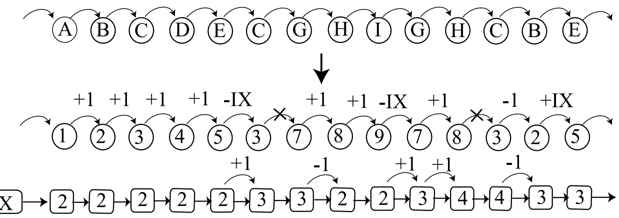


Fig. 4 AR updates example.

access sequence, i.e.

$$acc\_seq = \ldots\ldots\ldots \text{-}\, s_k \text{-} s_{k+1} \text{-} \ldots\ldots\ldots \tag{1}$$

where s_k, s_{k+1} denote program variables. When these program variables are allocated to memory addresses a_k, a_{k+1}, respectively, "address distance" or AR modification amount ad (s_k, s_{k+1}) is written as

$$ad(s_k, s_{k+1}) = |a_{k+1}(=\text{address of } s_k{+}1) - a_k \\ (= \text{address of } s_k)| \tag{2}$$

The "distance sequence" for overall access sequence is written as

$$dis\_seq\ (acc\_seq) = ad_1 \text{-}\ ad_2\ \text{-}\ ad_3 \ldots \tag{3}$$

where ad_n denotes the address distance at n-th update.

For example, assume a set of program variables V={A, B, C, D, E, F, G, H, I} and a memory access sequence

$$acc\_seq_1 = \ldots\text{-A-B-C-D-}\underline{\text{E-C-G}}\text{-H-}\underline{\text{I-G-H-C}}\text{-B-E-}\ldots \tag{4}$$

as shown in Fig. 4. An initial memory allocation for these variables is given by some methods. If program variables are allocated to a memory,

$$alloc1(\text{A,B,C,D,E,F,G,H,I}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{5}$$

then, two AR load operations (mark x in Fig. 4) are required.

The sequence can be rewritten

$$acc\_seq_1 = \ldots\text{-1-2-3-4-5-3-7-8-9-7-8-3-2-5-}\ldots \tag{6}$$

as shown in Fig. 4. The distance sequence becomes.

$$dis\_seq(acc\_seq_1) = \ldots\text{-1-1-1-1-}\underline{\text{2-4}}\text{-1-1-}\underline{\text{2-1-5}}\text{-1-3-}\ldots \tag{7}$$

3.2 AR Update by $AR \leftarrow AR \pm IX$ Operations

When the $AR \leftarrow AR \pm IX$ operation is used for three consecutive memory accesses, the following condition holds.

Condition 1

Consider a memory access subsequence s_1-s_2-s_3, or a_1-a_2-a_3, after memory allocation, where a_1, a_2, a_3 are memory addresses for s_1, s_2, s_3, respectively. Its distance subsequence or subsequence of AR modification amounts is d_1-d_2, where d_k denotes address distance or AR modification amount and is written as d_k = | a_{k+1} (=address of s_k+1) − a_k (=address of s_k)|. If

$$|d_1 - d_2| > 1, \tag{8}$$

at least one AR load is required at this subsequence.

As an example, consider memory access underlined subsequence in Eq. (4).

$$acc\_subseq_1 = \text{E-C-G} \tag{9}$$

When $AR \leftarrow AR \pm IX$ is used for an AR update at E-C, IX at C-G takes the value of 1,2,3, respectively, since IX can be updated by ±1 at every instruction cycle with memory access. For ad(C, G) = 4, an AR load is required. Similar can be said if we use $AR \leftarrow AR \pm IX$ at C-G. Therefore, at least one AR load is required at the subsequence E-C-G.

3.3 AR Update by $AR \leftarrow AR \pm IX$ and $AR \leftarrow AR \pm 1$ Operations

From the assumed AR update model, IX can be updated by ±1 at every memory access, whether AR is updated or not. Therefore, when a subsequence between two $AR \leftarrow AR \pm IX$ operations is realized only by $AR \leftarrow AR \pm 1$, IX can be updated by number of memory access included in the subsequence. In such a case, the following condition holds.

Condition 2

Consider the subsequence s_1-…-s_{k+3}, or a_1-…-a_{k+3} after memory allocation, where a_1-…-a_{k+3} are memory addresses for s_1-…-s_{k+3}, respectively.Its distance subsequence is d_1-d_2-…-d_{k+1}-d_{k+2}, where $d_k = |a_{k+1}(s_k{+}1) - a_k(s_k)|$. Assume that $d_1, d_{k+2} \geq 2$, and $d_2, \ldots, d_{k+1} \leq 1$. If

$$||d_1 - d_{k+2}| - k| > 1 \tag{10}$$

at least one AR load is required at this subsequence.

As an example, consider the following memory access subsequence from Eq. (4).

$$acc\_subseq_2 = \text{I-G-H-C} \tag{11}$$

For this subsequence, we have distance subsequence dis_seq(I-G-H-C)=2-1-5.

Since $||d_1 - d_2| - k| = ||2 - 5| - 1| > 1$ (k=1), at least one AR load is required at this subsequence.

When the $AR \leftarrow AR \pm IX$ operation is used for AR update at I-G, IX takes the value of 2,3,4 at H-C, since IX can be updated by ±1 at every memory access. For ad (H, C)=5, an AR load is required. Similar can be said if we use $AR \leftarrow AR \pm IX$ at H-C. Therefore, at least one AR load is required at the subsequence I-G-H-C.

3.4 Cost-Intensive Variables

According to the result in Condition 1, underlined subsequences in Eq. (4) require AR load operations. In this paper, such a subsequence is called as a "cost-intensive subsequence," and variables in a cost-intensive subsequence are called as "cost-intensive variables," where cost means the number of AR loads.

4. Proposed Memory Allocation Method

4.1 Lower Variance in the AR Modification

In this section, a memory allocation method, which utilizes $AR \leftarrow AR \pm IX$ operations, is proposed.

4.1.1 Access Graph

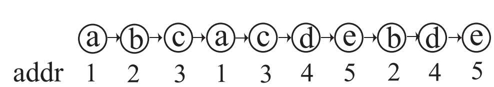
A given memory access sequence is modeled with an access graph (AG), where each vertex and edge denote the variable to be accessed and the required update, respectively [4]–[6]. Program variables are allocated to the memory as their appearance. For example, the AG of the memory access sequence in Fig. 5(a) is shown in Fig. 5(b).

4.1.2 Variance of the AR Modification

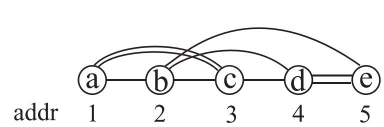
Since IX can be increased or decreased by one at every memory access, large modification in IX, and hence higher variance in AR update amount, as illustrated in Fig. 6(b), is not preferable. In this subsection, the method, which variance in AR update amount becomes lower, is introduced. The purpose of this variable allocation method is to decide IX value of each update approximately.

The detail procedures are:

1. Decide an initial memory allocation for program variables V. Program variables are allocated to the memory in the accessed turn.
2. Find the AR update(s) u-v of the longest address distance (ex. a-g in Fig. 6(b)). If multiple updates of the same distance exist, choose one whose variable, which is allocated at the lowest address.
3. Choose the variable w of the lower memory address (ex. "a" in Fig. 6(b)) in the AR update u-v. Exchange w with variable next to w, so that the AR modification amount at the AR update u-v decreases. If no additional AR modification longer than u-v newly appears, take the new memory allocation and restart from step 2. Otherwise, try the same exchange for the other variable of u-v (ex. "g" in Fig. 6(b)).
4. If no exchange occurs at step 3, try the same exchange for the next longest address distance (or rest updates in step 2).
5. If no exchange occurs at step 4, find u-v, Try the same



(a) Memory Access Sequence



(b) An AG for the Memory Access in (a)

Fig. 5 An AG example.

exchange with the next nearest variable from w.
6. Repeat 5, until no further exchanges occur.

For an example, consider a part of memory access sequence

$$acc\_seq = \ldots\text{-f-h-e-f-b-d-e-d-b-g-a-b-c-a-g-f-}\ldots \quad (12)$$

An initial memory allocation for program variables V={a, b, c, d, e, f, g, h} is determined as their appearance in the sequence, i.e.,

$$addr\_alloc\ (\mathrm{V}) = \{1, 2, 3, 4, 5, 6, 7, 8\} \quad (13)$$

The access sequence can be rewritten as

$$acc\_seq = \text{-6-8-5-6-2-4-5-4-2-7-1-2-3-1-7-6-}\ldots \quad (14)$$

A part of the distance sequence becomes

$$dis\_seq(acc\_seq) = \ldots\text{-2-3-1-4-2-1-1-2-5-6-1-1-2-6-1-}\ldots \quad (15)$$

Figure 6(a) shows the *dis_seq* along with the order of AR updates except the case of address distance $ad = 1$.

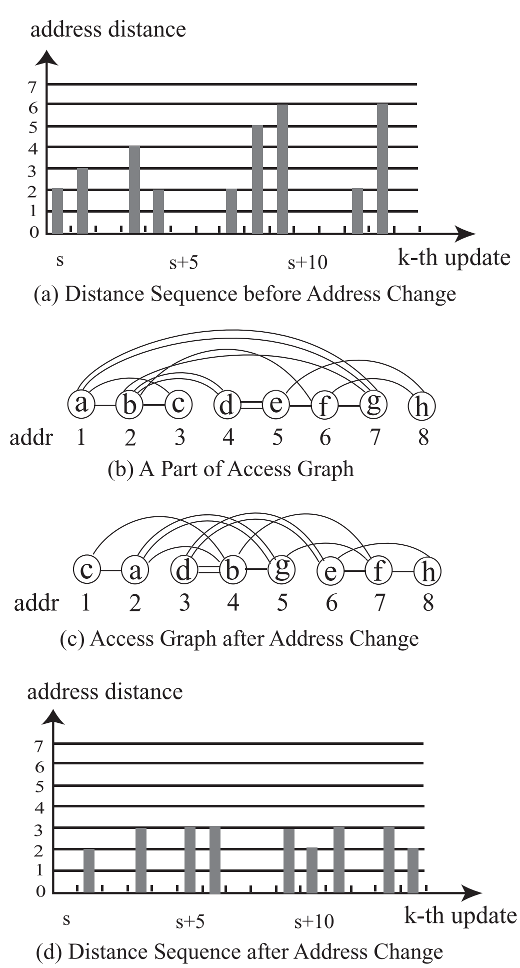
For AG in Fig. 6(b), a-g is one of the updates of the largest address distance in Fig. 6(b), and program variables a and g are exchanged with their neighbor variables b, f, and h. The similar procedures are repeated, and, as the result, address allocation

$$addr\_alloc(\mathrm{V}) = \{2, 4, 1, 3, 6, 7, 5, 8\} \quad (16)$$

is derived. The distance sequence becomes

$$dis\_seq(acc\_seq) = \text{1-2-1-3-1-3-3-1-1-3-2-3-1-3-2}. \quad (17)$$

Figures 6(c) and (d) show the AG and *dis_seq* after the address exchange. Since variance in AR modification in Fig. 6(d) is lower than that in Fig. 6(a), the address allocation in Fig. 6(c) gives a memory allocation to decide IX value of each update approximately.



(a) Distance Sequence before Address Change
(b) A Part of Access Graph
(c) Access Graph after Address Change
(d) Distance Sequence after Address Change

Fig. 6 Variance in AR modification amount.

4.2 Move-and-Shift Operations

In order to reduce AR loads, it is also very important to prevent the conditions in Condition 1 and Condition 2. In this paper, cost-intensive variables are moved to obtain memory allocations with less AR loads.

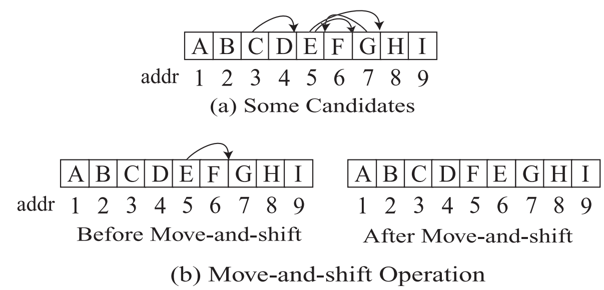
As an example, consider the underlined subsequence in Eq. (4). According to Condition 1 in Sect. 3.2, acc_subseq_1= E-C-G (Eq. (9)) is a cost-intensive subsequence, and variables E, C and G are cost-intensive variables. These cost-intensive variables are changed their addresses by using a variable movement.

This variable movement scheme is called "move-and-shift operations." Cost-intensive variables are moved by applying move-and-shift operations so that cost-intensive subsequences are reduced. There are some move-and-shift operations for reducetion in a certain cost-intensive subsequence, as shown in Fig. 7(a). For example, the memory allocation shown in Fig. 7(b) is obtained as a result of one of a move-and-shift.

Figure 8 shows the AR updates after move-and-shift operation (memory allocation is shown in Fig. 7(b)). The AR load operation required in Fig. 4 is reduced as a result of a move-and-shift operation.

4.3 Move-and-Shift Operation within Depth n

Although AR loads can be reduced by the move-and-shift



(a) Some Candidates
(b) Move-and-shift Operation

Fig. 7 Memory allocation.

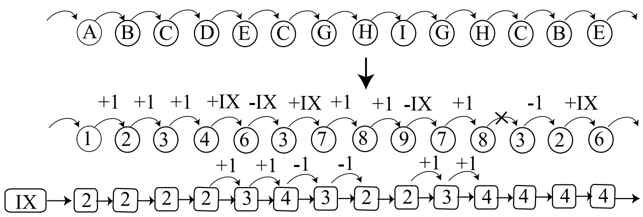


Fig. 8 AR updates after move-and-shift.

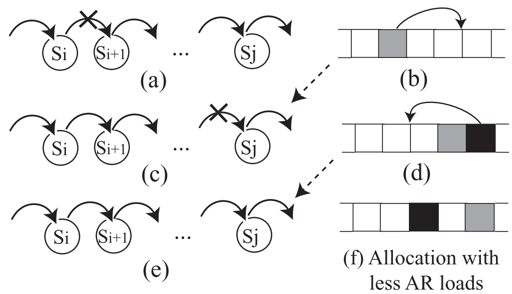


Fig. 9 Updates for each memory allocation.

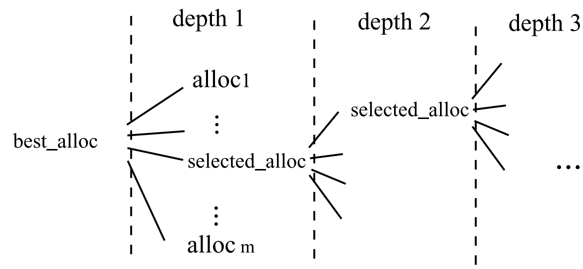


Fig. 10 Move-and-shift operation within depth *n*.

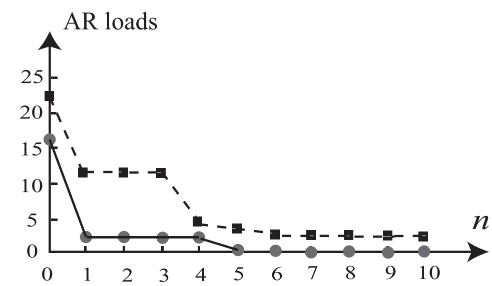


Fig. 11 The maximum depth *n* and the number of AR loads (Wave digital filter (11th order)).

operations in Sect. 4.2, total number of AR loads for a given access sequence may not be reduced. The memory allocation method for reduction in AR loads is shown in subsection.

The procedures are:

1.An initial memory allocation for program variables V is decided by the method in Sect. 4.1. Best allocation (*best_alloc*) is an initial allocation at this step, where best allocation means the allocation with lowest AR loads.

2.Choose a cost-intensive subsequence in an access sequence, and move-and-shift operations are applied. Some variable allocations are generated in this step.

3.Choose the memory allocation with the lowest AR loads as the *selected_alloc*.

4.

Case 1.

If *ARload* (*selected_alloc*) < *ARload* (*best_alloc*), *selected_alloc* newly becomes best_alloc, and go back to step 2, where *ARload* ( ) means the number of AR loads required for each memory allocation.

Case 2.

If *ARload*(*selected_alloc*) = *ARload*(*best_alloc*)), a cost-intensive subsequence is reduced. However, an AR load is newly required at another AR update as shown in Fig. 9(c). In this case, depth n is incremented (shown in Fig. 10), and go back to step 2 to reduce a newly required AR load.

Case 3.

If *ARload* (*selected_alloc*) > *ARload* (*best_alloc*), the numbers of AR load operations are increased by a move-and-shift operation. Try the next cost-intensive subsequence from the step 2.

For an example access sequence in Fig. 9(a), suppose that we apply a move-and-shift in order to reduce a cost-instensive subsequence, and an AR load is newly required at another AR update (Fig. 9(c)). In such a case, we may reduce an AR load, after the move-and-shift is iteratively applied (Fig. 9(e)). In this paper, when an AR load is reduced after *n* repetitions of moving program variables, it is refered as a "move of depth *n*" (depth 2 in this example).

An allocation with less AR loads is obtained by repeating move-and-shift operations. However, since a long compiling time will be required when depth *n* becomes large, compiler users can specify "the maximum value of *n*" in this method. Move-and-shifts are applied within Depth *n*.

4.4 Variance in AR Update and Maximum Depth

In this section, the effectiveness of the method to suppress variance in AR modification amount is shown. Also, the appropriate value of maximum depth *n* is evaluated.

As an example, 11th order wave digital filter is used. For its memory access sequence, the different memory allocation methods in the maximum depth are applied. As a preprocess of the memory allocation, variance in AR modification amount is suppressed by using the method mentioned in Sect. 4.1.

The numbers of resultant AR loads are plotted as the solid line in Fig. 11. For the comparison, the memory allocation results without the above preprocess are also plotted as the dashed line in the same figure. In this paper, a method to suppress variance in AR modification amount is applied as a preprocess before the memory allocation.

Also, the maximum depth is chosen as 5 in this paper. This value may not be enough. However, the maximum depth is taken fewer so that the memory allocation method can be applied to a large program.

Table 1 The number of the AR loads.

	proposed method	existing method[1]	existing method[3]
WDF(5)	0	0	2
WDF(7)	0	1	2
WDF(9)	0	1	6
WDF(11)	0	2	8
WDF(13)	3	4	
WDF(15)	2	8	
WDF(17)	4	12	
FFT 8	0	2	
FFT 16	0	2	

WDF(n): Wave Digital Filter (n-th order)

4.5 Memory Allocation Results

The proposed methods are applied to the DIMPL compiler for the assuming DSP model, and codes for several examples are generated. Table 1 shows a comparison of the proposed methods to the existing methods [1], [3] in terms of the number of AR loads in the generated codes. The existing method [3] decides such a memory allocation that $AR \leftarrow AR \pm 1$ ops. are used as much as possible, and then $AR \leftarrow AR \pm IX$ ops. are taken into account. In [1], the "move-and-shift operation" is not repeated to obtain a memory allocation. Thus, compiler time is expected to be reduced, but extra AR load operations are sometimes required.

Note that the maximum depth in the proposed method is set to 5. Memory allocation results of the proposed method need less AR loads than those of the existing allocation methods [1], [3].

5. Memory Allocation Methods Coupled with Computational Ordering

5.1 Computational Ordering and Memory Access Sequence

The proposed method in the last chapter allocates memory addresses of the program variables for the memory access sequence given for the code, which is generated for a source program. The order of load instructions and store instructions depend on computational order. Therefore, if other computational order are obtained, different memory allocation results for the alternative memory access sequence is given, and hence further reduction in overhead codes is expected. In this chapter, computational rearrangement in memory access sequence is taken into account, an improved memory allocation method is shown.

5.1.1 Precedence Relation

A computational ordering must satisfy precedence relations. For the program in Fig. 12(a), its "precedence relations" are shown in Fig. 12(b). These relations, for example, indicates that F must be calculated before J is calculated.

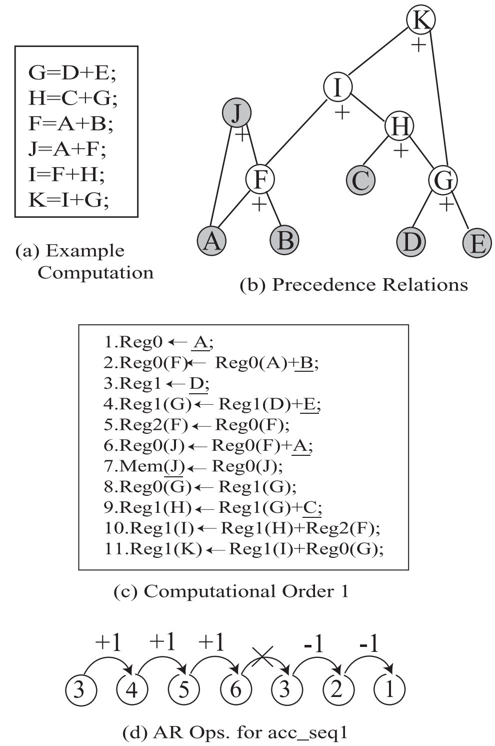


Fig. 12 Computational order and memory access order.

5.1.2 Memory Access Order

For a given program, we may take alternative computational orders to achieve the same calculation. According to the computational order, memory access sequence is derived. For the above example program, Fig. 12(c) shows a computational order, where the underlined variables are program variables kept in memory space. According to this computational order, the memory access sequence becomes *acc_seq1*=A-B-D-E-A-J-C. For the simplicity of explanation, we now assume the indirect addressing only of $AR \leftarrow AR \pm 1$ operation. For the memory access sequence *acc_seq*1, memory allocation *alloc*1 (A, B, C, D, E, J)={3, 4, 1, 5, 6, 2} minimizes the required AR load, which is shown by mark x in Fig. 12(d).

When we take an alternative computational order in Fig. 13(a), memory allocation *alloc*2 (A, B, C, D, E, J)={2, 1, 6, 4, 5, 3} requires no AR load as shown in Fig. 13(b).

As shown by above example, computational order is very important in reduction of overhead codes over memory accesses. Therefore, memory allocation and computational order are simultaneously considered in the following sections.

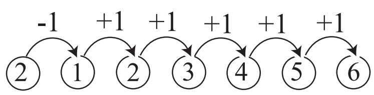
5.2 Program Variables in Memory Space

5.2.1 Load Instructions

In Fig. 12(a), F=A+B and F=B+A give the same calcu-

```
1.Reg0 ← A;
2.Reg0(F) ← Reg0(A)+B;
3.Reg1(F) ← Reg0(F);
4.Reg0(J) ← Reg0(F)+A;
5.Mem(J) ← Reg0(J);
6.Reg0 ← D;
7.Reg0(G) ← Reg0(D)+E;
8.Reg2(G) ← Reg0(G);
9.Reg0(H) ← Reg0(G)+C;
10.Reg0(I) ← Reg1(F)+Reg0(H);
11.Reg0(K) ← Reg0(I)+Reg2(G);
```

(a) Computation Order2

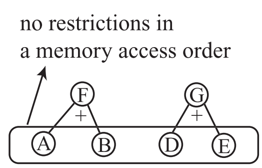


(b) AR Ops. for acc_seq2

Fig. 13 AR Ops. for *acc_seq2*.

```
1.Reg0 ← D;
2.Reg0(G) ← Reg0(D)+E;
3.Reg1 ← B;
4.Reg1(F) ← Reg1(B)+A;
5.Reg2(F) ← Reg1(F);
6.Reg1(J) ← Reg1(F)+A;
7.Mem(J) ← Reg1(J);
```

(a) Schedule Example



(b) Load Instructions

Fig. 14 Load instructions.

lation. Fuhermore, there exist no precedence relations in F=A+B and G=D+E. Thus, lines 1-7 in Fig. 12 (a) can be rewritten as shown in Fig. 14(a). Therefore, there are no restrictions in access orders of program variables "A," "B," "C," "D" (Fig. 14(b)).

5.2.2 Store Instructions

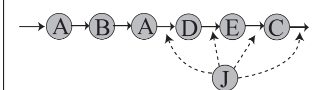
In general, a program variable is written into memory space immediately after its value is computed. For example, program variable J in Fig. 12(b) needs to be stored into memory space. According to the computational order in Fig. 13(a), the variable J is stored into memory space just after the computation F+A (line 5 in Fig. 13(a)). However, we can keep such a program variable on an arithmetic register and postpone the execution of store instruction. For the above example, we can compute G=D+E before the store of J, so that lines 1-7 in Fig. 13(a) is rewritten as shown in Fig. 15(a). The choices in store timing of program variable J are illustrated in Fig. 15(b).

5.3 Computational Re-ordering Method

For a given program, there exist alternative computational orders, that satisfy precedence relations, and hence the alternative memory access sequences. In this section, the computational ordering method coupled with computational re-ordering is proposed. An overall diagram of the proposed method is shown in Fig. 16.

```
1.Reg0 ← A;
2.Reg0(F) ← Reg0(A)+B;
3.Reg1(F) ← Reg0(F);
4.Reg0(J) ← Reg0(F)+A;
5.Reg2 ← D;
6.Reg2(G) ← Reg2(D)+E;
7.Mem(J) ← Reg0(J);
```

(a) Schedule Example



(b) Choices in Orders of Store Instruction

Fig. 15 Store instructions.

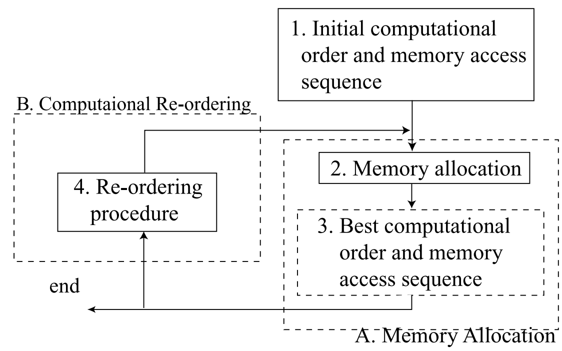
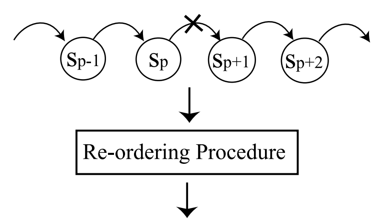


Fig. 16 Memory allocation and code optimization.



computational order	access sequence	instruction cycles
order 1	acc_seq 1	cycle (order 1)
order 2	acc_seq 2	cycle (order 2)
⋮	⋮	⋮
order m	acc_seq m	cycle (order m)

Fig. 17 Memory access ordering.

5.3.1 Initial Computational Order

At the procedure 1 in Fig. 16, an initial computational order is determined by use of the conventional computational ordering method. Although this method exploits computational resources or arithmetic units (e.g., arithmetic registers and ALU) as much as possible, overhead codes over memory accesses are not taken into account.

5.3.2 Computational Re-ordering

Memory access sequence is changed by the computational re-ordering at procedure 4 in Fig. 16. The detailed procedure of the re-ordering is shown in Fig. 17.

Suppose a memory access sequence

$$acc\_seq = s_1 \text{-} s_{2-} \ldots\ldots \text{-} s_p \text{-} s_{p+1} \text{-} \ldots\ldots \text{-} s_{q-1} \text{-} s_q \quad (18)$$

is given, where there required an AR load between p-th and (p+1)-th memory accesses (Fig. 16 left-handside). In the re-ordering procedure, computational orders commutative with s_p and/or s_{p+1} is considered, so that the memory access order around s_p-s_{p+1} changes. By use of flexibility in computational orders mentioned in Sect. 5.2, m computational order candidates $order_k$ $(1 \leq k \leq m)$ are listed as shown in Fig. 17. For each candidate $order_k$, the number of instruction cycles is counted as $cycle\ (order_k)$.

5.3.3 Memory Allocation

According to computational orders $order_k$ $(1 \leq k \leq m)$, memory access sequences acc_seq_k $(1 \leq k \leq m)$ are derived, and memory allocations are determined. For each memory allocation alloc (acc_seq_k), the number of required AR loads, AR*load* (*alloc* (acc_seq_k)) is counted.

Computational orders are evaluated by the cost function

$$cost\ (order_k) = cycle\ (order_k) + \mathrm{AR}load\ (\mathrm{alloc}\ (acc\_seq_k)) \quad (1 \leq k \leq m) \quad (19)$$

The $order_k$ with the lowest $cost\ (order_k)$ is chosen for the re-ordering in the next iteration.

5.3.4 Outline of the Re-ordering Method

An outline of the re-ordering method applied in this paper is shown below.
1. An initial computational order is given by the conventional method. Although it gives efficient program codes in terms of arithmetic instruction cycles, overhead in memory accesses are not considered.
2. Memory allocations are determined for m different memory access sequences (given at procedure 1 or procedure 4). The number of required AR loads is counted for each memory access sequence or computational order.
3. Chose the computational order with the lowest *cost* k.
4. Re-ordering procedure determines m kinds of computational order. Repeat 2-4 no further re-ordering occurs.

5.4 Results

The proposed memory address allocation method coupled with computational reordering is applied to the DIMPL compiler for the above-mentioned DSP model, where the maximum depth is set to 5. Codes for several examples are generated and the numbers of their AR loads are shown in Table 2. The numbers at the "initial computation" column is the same results shown in Table 1. The number of AR loads is reduced by the proposed re-ordering method.

In the table, "(+ number)" denotes the number of additional arithmetic instructions associated by the computational re-ordering method. From the table, no additional instruction cycle is required for these examples. The numbers in the "iteration" column are the loop count in Fig. 16. The "memory allocation" column denotes the number of times the proposed memory allocation methods was applied.

Table 2 The number of the AR loads.

	initial computaion	re-ordering method	iteration	memory allocation
WDF(5)	0	0	0	1
WDF(7)	0	0	0	1
WDF(9)	0	0	0	1
WDF(11)	0	0	0	1
WDF(13)	3	1(+0)	2	17
WDF(15)	2	0(+0)	2	19
WDF(17)	4	0(+0)	4	48
FFT 8	0	0	0	1
FFT 16	0	0	0	1

WDF(n): Wave Digital Filter (n-th order)

6. Conclusions

In this paper, a new memory allocation method for indirect addressing processor with an indexed auto-modification is proposed. A memory allocation method in cooperated with computational re-scheduling method is also proposed. These methods are applied to the existing compiler and their effectiveness is shown by the generated codes for several examples. A further reduction in AR loads, efficient memory addressing for multiple ARs, and memory allocation methods with less computational complexity must be studied.

References

[1] Y. Kaneko, N. Sugino, and A. Nishihara, "Memory allocation method for indirect addressing with an index register," 2002 IEEE Asia Pacific Conference on Circuit and Systems, pp.199–202, Singapore, Dec. 2002.
[2] K. Miyahara, Y. Kaneko, and N. Sugino, "DSP code optimization technique with consideration in both computational ordering and memory access," IEICE, Digital Signal Processing Symposium in Japan, 21, May 2002.
[3] R. Seno, N. Sugino, and A. Nishihara, "Memory allocation method for indirect addressing with index register," 14th Digital Signal Processing Symposium in Japan, pp.617–622, Nov. 1999.
[4] B. Wess and S. Frohlich, "DSP data memory layouts optimized for intermidiate address pointer updates," Proc. IEEE APCCAS 1998, pp.451–454, Nov. 1998.
[5] N. Sugino, S. Yoshida, and A. Nishihara, "Code optimization method for DSPs with multiple memory addressing registers and its application to compiler," IEEE TENCON 1996, pp.619–624, 1996.
[6] N. Sugino, A. Toshikiyo, E. Watanabe, and A. Nishihara, "Computational ordering of digital signal processing networks and its application to compilers for signal processors," IEICE Tarns. Fundamentals (Japanese Edition), vol.J71-A, no.2, pp.327–335, Feb. 1988.
[7] N. Sugino, S. Ohbi, and A. Nishihara, "Computational ordering of digital network under the pipeline constraints audits application to compiler for DSPs," Proc. ECCTD'89, pp.395–399, Sept. 1989.

Yuhei Kaneko was born in Isesaki, Gunma, Japan January 23, 1978. He received B.E. degree in electronics from Chiba University in 2000 respectively. Since 2003, he has been with Tokyo Institute of Technology, where he is now master cource student of Department of Advanced Applied Electronics, Interdisciplinary Graduate School of Science and Technology. His main research interests are compiler techniques for VLIW processors, and single chip multi-core processors.

Nobuhiko Sugino was born in Yokkaichi, Mie, Japan on November 19, 1964. He received B.E., M.E. and Dr. Eng. degrees in physical electronics from Tokyo Institute of Technology in 1986, 1989 and 1992, respectively. Since 1992, he has been with Tokyo Institute of Technology, where he is now Associate Professor of Department of Advanced Applied Electronics, Interdisciplinary Graduate School of Science and Technology. His main research interests are compiler techniques for VLIW processors, and single chip multi-core processors. He is also interested in hardware and software for digital signal processing. Dr. Sugino is a member of IEEE.

Akinori Nishihara received the B.E., M.E. and Dr. Eng. degrees in electronics from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. Since 1978 he has been with Tokyo Institute of Technology, where he is now Professor of the Center for Research and Development of Educational Technology. His main research interests are in one- and multi-dimensional signal processing, and its application to educational technology. He served as an Associate Editor of the IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences from 1990 to 1994, and then an Associate Editor of the Transactions of IEICE Part A (in Japanese) from 1994 to 1998. He was an Associate Editor of the IEEE Transactions on Circuits and Systems II from 1996 to 1997 and Editor-in-Chief of Transactions of IEICE Part A (in Japanese) from 1998 to 2000. He received Best Paper Awards of the IEEE Asia Pacific Conference on Circuits and Systems in 1994 and 2000, a Best Paper Award of the IEICE in 1999, and IEEE Third Millennium Medal in 2000. He also received a 4th LSI IP Design Award in 2002. Prof. Nishihara is a Fellow of IEEE, and a member of EURASIP, European Circuits Society, and Japan Society for Educational Technology.