論文 / 著書情報
Article / Book Information

Title	DSP Code Optimization Methods Utilizing Addressing Operations at the Codes without Memory Accesses
Authors	NOBUHIKO SUGINO, Hironobu Miyazaki, AKINORI NISHIHARA
出典 / Citation	IEICE Trans. Fundamentals., Vol. E80-A, No. 12, pp. 2562-2571
発行日 / Pub. date	1997,
URL	http://search.ieice.org/
権利情報 / Copyright	本著作物の著作権は電子情報通信学会に帰属します。 Copyright (c) 1997 Institute of Electronics, Information and Communication Engineers.

IEICE TRANS. FUNDAMENTALS, VOL. E80-A, NO. 12 DECEMBER 1997

PAPER

DSP Code Optimization Methods Utilizing Addressing Operations at the Codes without Memory Accesses

Nobuhiko SUGINO†, *Member*, Hironobu MIYAZAKI††, *Nonmember*, *and* Akinori NISHIHARA†††, *Member*

SUMMARY Many digital signal processors (DSPs) employ indirect addressing using address registers (ARs) to indicate their memory addresses, which often leads to overhead. This paper presents methods to efficiently allocate addresses for variables in a given program so that overhead in AR update operations is reduced. Memory addressing model is generalized in such a way that AR can be updated at the codes without memory accesses. An efficient memory address allocation is obtained by a method based on the graph linearization algorithm, which takes account of the number of possible AR update operations for every memory access. In order to utilize multiple ARs, methods to assign variables into ARs are also investigated. The proposed methods are applied to the compiler for μPD77230 (NEC) and generated codes for several examples prove effectiveness of these methods.

key words: *digital signal processor, compiler, code optimization, memory addressing*

1. Introduction

Digital signal processors (DSPs)[1] are now widely used in various applications, such as cellular phones, MODEMs, digital cameras, etc.. To fully utilize the performance of DSPs in real-time applications, efficient (short in execution time) programs are required. Writing such a program, however, is cost and time consuming work even for expert programmers: they are required to have profound knowledge of the processor architecture as well as the processing algorithms. Usually they have to try as many programming techniques as they know, using primitive tools such as assembler or low level (machine level) languages.

To reduce this heavy programming load, software tools such as high-level languages and their compilers are quite helpful. Recently, many C compilers[2]–[6] are supplied by DSP vendors, but these compilers are said to be insufficient for cost-competitive real time applications. Programming tools or compilers are also presented by some researchers[8]–[15]. The authors' group have developed DIMPL (Digital networks IMPlementation Language) and its compiler[14],[15]. In the DIMPL and some other compilers, an efficient program benefits from effective use of registers. Efficient memory access, however, is as important as effective use of registers in order to derive an efficient program.

When a datum is read from or written into a memory (memory access for a datum as a generic term for both cases), memory address corresponding to the datum must be pointed. The ways to point the address are referred to as memory addressing modes. General purpose processors provide various addressing modes such as immediate addressing mode, indirect addressing mode, indexed addressing mode, and so on, so that various data structures are easily implemented. On the contrary, DSPs usually have only simple memory addressing modes due to constraints in DSP hardware design scheme for computational speed, chip size, fabrication cost and etc.. Some DSPs support specific addressing modes for signal processing such as bit-reversed addressing for FFT, though. Indirect addressing mode is popular in many DSPs, which uses address registers (ARs) to point the memory addresses to be accessed. In such memory addressing, AR must be updated before the instruction with memory access. Simple AR update operations, such as AR increment (AR +1) and decrement (AR −1) operations, are quite useful, because they are performed in one instruction cycle together with arithmetic data operations and data transfer operations. But, when AR cannot be updated only by these simple AR operations, additional one instruction cycle is required to substitute memory address for AR, which becomes overhead in a program. Therefore, in order to derive an efficient program, reduction in the number of such overhead codes is very important.

C compilers[2]–[6] usually assign memory addresses of variables according to the declaration order in a source program. In other words, the declaration order determines the number of overhead codes related with memory access. Although programmers can improve the generated assembler program by rearranging memory addresses, it requires additional heavy programming load. The DIMPL compiler[15] determines the memory addresses of variables in the order as they appear in the generated assembler code. Although the codes gen-

Manuscript received Janurary 6, 1997.
Manuscript revised April 28, 1997.

†The author is with Department of Information Processing, Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, Yokohama-shi, 226 Japan.
††The author is with Department of Physical Electronics, Faculty of Engineering, Tokyo Institute of Technology, Tokyo, 152 Japan.
†††The author is with the Center for Research and Development of Educational Technology, Tokyo Institute of Technology, Tokyo, 152 Japan.

erated by this compiler are efficient in terms of execution steps, there are still as many overhead codes introduced by memory accesses as those derived by the C compilers. Although memory sharing by several variables [16] can reduce the total memory area and the AR update amount is expected to be small, the number of overhead codes sometimes increases. In [17], an efficient method to assign memory address to variables is presented. That method models variables in a program and AR operations with a graph. By linearizing the graph, address locations for variables are decided so as to reduce overhead codes. A method to utilize multiple ARs is also proposed for the case that a DSP has multiple ARs [17]. In those methods, only one AR update operation just after each memory access was assumed. Therefore, those methods are effective only when memory is frequently accessed in a program. Such a program is often seen when a DSP has a few arithmetic registers. In the cases where arithmetic registers are effectively used or where there are many arithmetic registers, memory may not be frequently accessed, and those methods do not give efficient results.

In many DSPs with indirect addressing mode, cost-efficient AR update operations (AR ± 1 operations) can be performed at every instruction; AR can be updated not only at the code with memory access but also at the code without memory access [3], [5], [7]. Therefore, further reduction of overhead codes is expected by utilizing AR update at code without memory access. In this paper, the memory allocation algorithm in [17] is extended by considering AR updates at codes without memory accesses. A method to utilize multiple ARs for this issue is also studied.

2. Memory Addressing

A DSP considered here is assumed to have only an indirect addressing mode. Its architecture is depicted in Fig. 1, where the number of ARs is two, just for example. Addressing operations of this DSP have the following features.

Memory addressing mode of a DSP model

- Memory is accessed only by AR indirect addressing.
- In one instruction cycle, the content of AR can be increased or decreased by one (AR ± 1 update operation: hereafter AR± 1) in parallel with an arithmetic operation or a data movement operation.
- All ARs can be simultaneously updated by ± 1 in one instruction cycle, whether one of the ARs is used for memory access or not. except the case of the following AR load operation.
- AR can be updated by substituting an address directly into one of ARs (immediate AR load operation: hereafter, AR load). AR load costs one instruction cycle by itself.

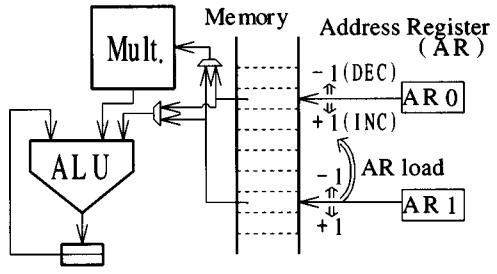


Fig. 1 A model DSP architecture.

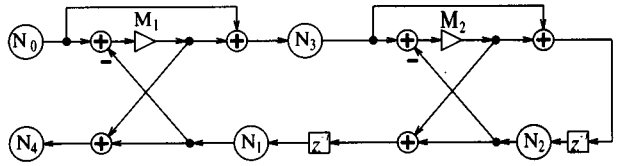


Fig. 2 SFG of 2nd order Lattice Filter.

The third feature shows an apparent difference from the DSP model used in Ref. [17]. When an instruction requires to read a datum from memory or to write a datum into memory, AR must be updated before that data access. In the former model, only one AR is updated by ± 1 just after memory is accessed, so that AR update amount for the following access is within ± 1. In the present DSP model, since ARs can be updated by ± 1 in every instruction except AR load, AR update amount for the following access is up to the number of instruction cycles in this duration. This number is referred to as the address update "length" in this paper. In the case that AR update amount is beyond the length, immediate load operation of the memory address into AR is required. This immediate AR load becomes overhead in a program, because it costs additional one instruction cycle by itself.

Let us think of memory addresses and AR operations with a simple example. The signal flow graph of a second-order allpass lattice filter shown in Fig. 2. The left hand side of Table 1 shows intermediate codes to execute Fig. 2 using two arithmetic registers R_1 and R_2. Variables $N_0 \sim N_4$ are kept in memory space.

The column labeled "Access" in Table 1 shows a variable accessed at each code cycle. The right hand side columns in Table 1 shows the address sequence and AR operations for each address assignment. The mark "*LD*" denotes AR load operation. The first two columns show the comparison of address allocations for the conventional DSP model (column labeled as "Conventional Mode") and the novel DSP model (Column labeled as "Novel Model"), where address allocations of variables for both cases are decided in accordance with the order as they appear in the intermediate code

Table 1 Intermediate code for Fig. 2 and AR operations.

Code Cycle	Intermediate Code	Access	Address AR operation: Address Allocation as Appearance Order — Conventional Model	Address AR operation: Address Allocation as Appearance Order — Novel Model	Address AR operation: Efficient Addr. Alloc. — Novel Model	Multiple ARs
						AR0 AR1
1	$R_1 \leftarrow N_0$	N_0	0) +1	0) +1	0) *LD*	0
2	$R_1 \leftarrow R_1 - N_1$	N_1	1)	1)	3) −1	±0 3
3	$R_1 \leftarrow m_1 \times R_1$	—	−1	−1	) −1	
4	$R_2 \leftarrow R_1$	—			) −1	±0
5	$R_1 \leftarrow R_1 + N_0$	N_0	0) *LD*	0) *LD*	0) +1	+1 0
6	$N_3 \leftarrow R_1$	N_3	2) −1	2) −1	1) *LD*	1
7	$R_2 \leftarrow R_2 + N_1$	N_1	1) *LD*	1) *LD*	3) +1	+1 3 +1
8	$N_4 \leftarrow R_2$	N_4	3) +1	3) +1	4) *LD*	4
9	$R_1 \leftarrow R_1 - N_2$	N_2	4)	4) −1	2) −1	2
10	$R_1 \leftarrow m_2 \times R_1$	—	*LD*	) −1	) ±0	−1 −1
11	$R_2 \leftarrow R_1$	—		) ±0	) ±0	
12	$R_1 \leftarrow R_1 + N_3$	N_3	2) *LD*	2) *LD*	1) +1	+1 1
13	$R_2 \leftarrow R_2 + N_2$	N_2	4) *LD*	4) *LD*	2) +1	2
14	$N_1 \leftarrow R_2$	N_1	1) *LD*	1) *LD*	3) −1	±0 3
15	$N_2 \leftarrow R_1$	N_2	4	4	2	2
Number of AR loads			6	5	3	0

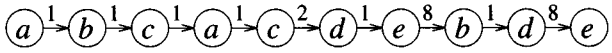
sequence. In the conventional DSP model, 6 AR loads are needed, while 1 AR load between codes 9∼12 is saved in the novel DSP model. Appropriate memory allocation can further reduce the number of AR loads to 3 as shown at the column labeled as "Efficient Address Allocation" for the novel DSP model. Moreover, utilization of multiple ARs improves memory addressing. For this example, by choosing an appropriate AR for every memory access no immediate AR load appears in the resultant program codes, as shown in the rightmost column "Multiple ARs" in Table 1.

As shown in the above example, memory access for indirect addressing contains "memory allocation" issue for efficient address mapping of variables and "AR assignment" issue for effective use of multiple ARs. Programmers usually take account of these two issues at the same time and devise an efficient memory access pattern. It is, however, not easy to formulate them as an algorithm. In order to simplfy the overall problem, memory allocation and AR assignment are considered as distinct problems in this paper. Methods for memory allocation problem and AR assignment problem are proposed in Sects. 3 and 4, respectively.

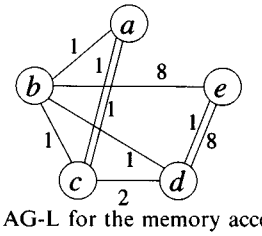
3. Memory Allocation Methods

3.1 Novel Access Graph

In Ref. [17], a given memory access sequence is modeled by an access graph (AG) where each vertex and edge denote the variable to be accessed and the necessity of AR update, respectively. In this paper an AG has an additional label referred to as "length" on the corresponding edge [18]. The length means the number of instruction cycles between the two accesses. This



(a) Memory Access in 5th order Wave Digital Lattice Filter



(b) The AG-L for the memory access in (a)

Fig. 3 An AG-L example.

new AG is called as an AG-L (AG with Length). As a memory allocation, memory addresses are decided for vertices, and then AR update operations such as AR ±1 operations are assigned to each edge. For example, the AG-L of the memory access sequence in Fig. 3 (a) is shown in Fig. 3 (b). The numbers labeled on each arrow in Fig. 3 (a) and beside each edge in Fig. 3 (b) denote the length, which is the number of instructions which can include AR ±1 operations. In this example it is assumed that ARs are initialized at the beginning of the access sequence which corresponds to the processing of one signal sample. When a circulating memory access is required, the algorithms proposed in this paper are applicable only with slight modification in input access sequences. When a given AG-L is a "line graph" as shown in Fig. 4 (a), no immediate AR load is required by putting memory addresses for variables just as the order of the nodes in that line graph. Such a line graph is defined as a simple chain sequence or a simple edge sequence in the graph theory [21]. A general AG-L, however, includes some fork vertices as shown in Fig. 4 (b)

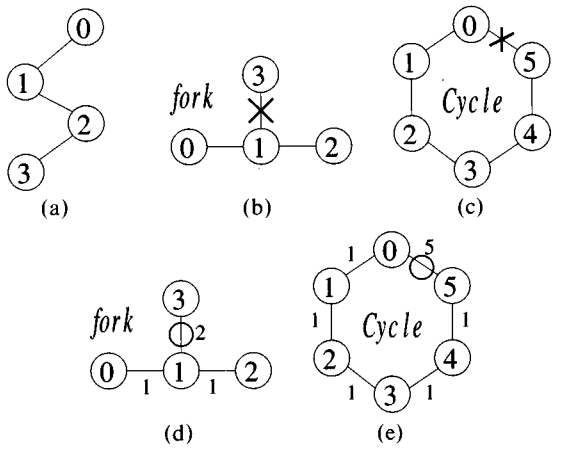


Fig. 4 Classes of AG-Ls.

and some cycles as shown in Fig. 4(c). Our approach is to linearize a given AG-L by removing appropriate edges from the AG-L, so that an address location for every variable is easily decided. In return, immediate AR loads are required at the memory access corresponding to the removed edges (in Figs. 4(b) and (c) marked by $\times$) in the conventional DSP model. Since the lengths of edges are considered in this paper, for some forks like an AG-L shown in Fig. 4(d) and for some cycles like an AG-L shown in Fig. 4(e), no AR load is required after an appropriate address allocation (marked by $\circ$). Although memory allocations for these illustrated cases are easily decided, memory allocation for a general AG-L is not a trivial problem. The optimum memory allocation for a given AG-L is given by examining all the possible combination of memory address and variables. However that method costs a plenty of time and is not appropriate for practical use [20]. Therefore heuristic methods are employed as written in Refs. [17]–[20].

3.2 Length and Address Location

In the conventional method [17], the number of removed edges is important, because it determines the number of AR loads. For the novel DSP model, edge removal does not necessarily lead to the introduction of AR load. In this case the length has an important role. For some partial AG-Ls in a given AG-L, such as AG-Ls shown in Figs. 4(d) and (e), efficient memory allocations can be decided. Although the optimal address allocation is found by examining all the possible combinations of such partial AG-Ls, that method is not easy to be formulated as an algorithm. Therefore, AR update length is taken into account in the graph linearization procedure [17]. When an AR update amount at the edge removed by the linearization process is more than the length of that edge, AR load is required. No AR load is required, when an AR update amount is less than or equal to the length of the corresponding edge. In other

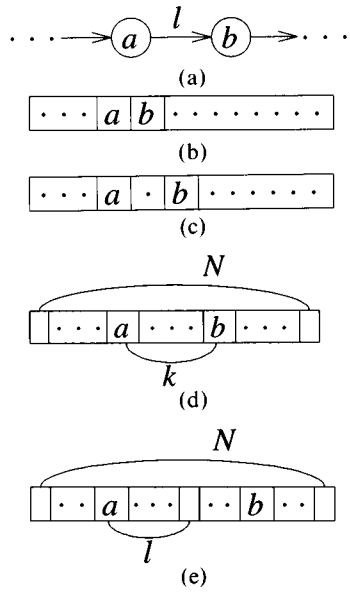


Fig. 5 Memory Locations and Edge Lengths.

words, in order not to introduce an AR load at an edge, two variables at both ends of the edge are preferably located at close addresses within the edge length. Intuitively, an edge with large length tends to need no AR load compared to that with small length. So edges of large length are given high priority to be removed in the graph linearization. In the algorithm, however, a measure to evaluate the effect of edge length is needed.

Suppose variable b is accessed l cycles after an access of variable a as shown in Fig. 5(a), where the length of this AR update l is labeled on the edge. When we have N variables in a given memory access sequence, the number of the possible address allocations is

$$N!.$$

In some of $N!$ address allocations, variable b is located in the next address to variable a as shown in Fig. 5(b). The number of such address allocations is given by

$$2 \cdot (N-1) \cdot (N-2)!,$$

where 2 means that variables a and b are positioned in 2 ways. Similarly, the number of possible allocations in which b is located in two addresses away from a as shown in Fig. 5(c) is

$$2 \cdot (N-2) \cdot (N-2)!.$$

In general, the number of possible allocations in which b is located in k addresses away from a as shown in Fig. 5(d) is

$$2 \cdot (N-k) \cdot (N-2)!.$$

The number of possible allocations in which address distance between the two variables is more than l as

shown in Fig. 5 (e) is given by

$$\sum_{k=l+1}^{k=N} 2 \cdot (N-k) \cdot (N-2)!. \\ = (N-l-1)(N-l) \cdot (N-2)!.$$

Such allocations require AR load at this AR update. Note that $l < N$ is assumed, because AR update of $l \geqq N$ is easily realized by use of AR$\pm$1 operations.

On the other hand, when we remove the edge between variable a and b in the graph linearization, variables a and b are not located in the adjacent addresses. The number of possible allocations in which b is not located in the next address to a is given by

$$N! - 2 \cdot (N-1) \cdot (N-2)!.$$

Consequently, the ratio of the number of possible allocations in which address distance of the two variables is more than l to the number of overall possible allocations in which the two variables are not located in the adjacent addresses is given by

$$P(N,l) = \frac{(N-l-1)(N-l) \cdot (N-2)!}{N! - 2 \cdot (N-1) \cdot (N-2)!} \quad (1)$$

$$= \frac{(N-l)(N-l-1)}{(N-1)(N-2)}. \quad (2)$$

$P(N,l)$ estimates the number of AR load at an edge with length l, when this edge is removed in the graph linearization procedure. Thus $P(N,l)$ is used in a cost function to select edges to be removed in the graph linearization algorithm.

3.3 Memory Allocation Algorithm

Reference [17] presents heuristic algorithm based on the graph linearization, named ALOMA (Address Load Operation Minimization Algorithm). That memory allocation algorithm eliminates forks and cycles in a given AG by removing appropriate edges. In return, AR loads are required at the removed edges. Therefore, the number of removed edges is evaluated as a cost function in the graph linearization algorithm.

For the novel DSP model, an efficient memory allocation is achieved by effective use of AR update operations at the codes without memory access. Hence, the new parameter "length" is very important in a memory allocation method for an AG-L. Since memory space is addressed in one dimensional manner, it is reasonable to assign addresses to variables along with line graphs. Therefore, the graph linearization procedures in ALOMA are applied to AG-L with some modifications; to select edges to be removed, the number of edges in each edge group [17] as well as the length of each edge [18] is evaluated by a new cost function. This method is named ALOMA-L (ALOMA with Length). ALOMA-L eliminates forks and cycles in a given AG-L by removing appropriate edge groups in the AG-L. Although AR loads are necessary at the removed edges in ALOMA, necessity of AR loads is ambiguous in the graph linearization procedure of ALOMA-L. The exact number of AR loads for a given AG-L is not apparent until memory addresses are decided.

The ALOMA-L is given as the followings.

ALOMA-L

1. Remove edges with length $l \geqq |V| - 1$, where $|V|$ denotes the number of variables.
2. For all the vertices $v \in V$ in G, evaluate $\mathrm{fork}(v)$, the fork count of a node v defined by

$$\mathrm{fork}(v) = \max\{\deg(v) - 2, \quad 0\}, \quad (3)$$

where $\deg(v)$ denotes the order of node v.

3. For all the edge groups (u,v) in G, evaluate $\mathrm{cycle}(u,v)$, the cycle flag of edge group (u,v), by

$$\mathrm{cycle}(u,v) = \begin{cases} 1, & \text{edge group } (u,v) \text{ is} \\ & \text{included in a cycle.} \\ 0, & \text{edge group } (u,v) \text{ is} \\ & \text{not included in any cycles.} \end{cases} \quad (4)$$

4. For all the edge groups (u,v) in G, evaluate $\mathrm{benefit}(u,v)$, the cost function defined by

$$\mathrm{benefit}(u,v) = \frac{\mathrm{fork}(u) + \mathrm{fork}(v) + \mathrm{cycle}(u,v)}{w(u,v)} \quad (5)$$

In this cost function, weighting function $w(u,v)$ is given by

$$w(u,v) = \sum_{l=1}^{N-1} a_l(u,v) \cdot P(N,l), \quad (6)$$

where N denotes the number of variables, and $a_l(u,v)$ denotes the number of edges with length l connected between vertices u and v. $P(N,l)$ is given by Eq. (2). Consequently, the cost function $\mathrm{benefit}(u,v)$ estimates how many forks and cycles are eliminated after removal of edges between vertices u and v.

5. Find an edge group with the maximum benefit and remove it from G.
6. Repeat 2$\sim$5, until G becomes a line graph or line graphs.
7. For the vertices in the derived line graph, memory addresses are decided.

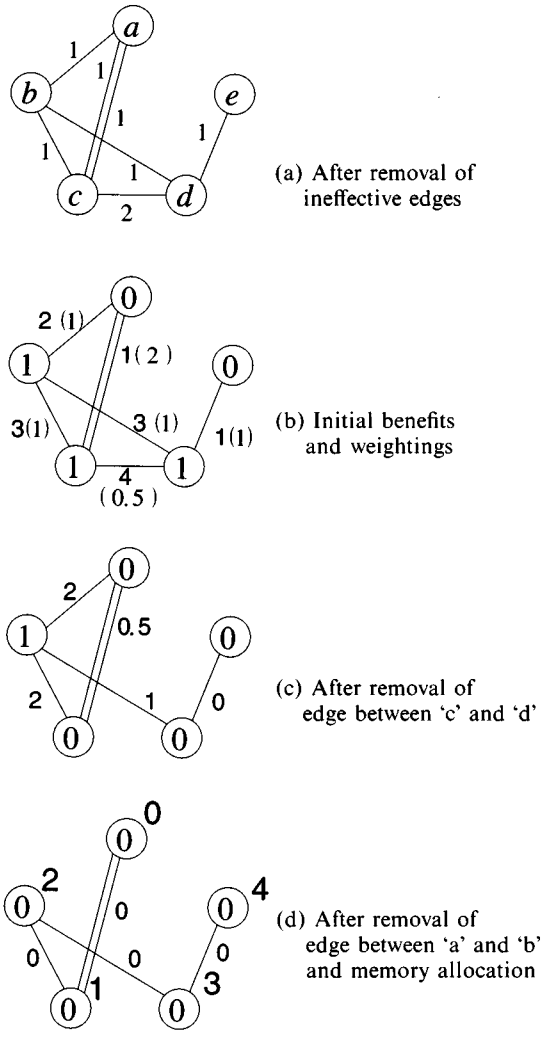


Fig. 6 Procedure of memory allocation.

Procedures of the ALOMA-L with the novel cost function are now explained using an example AG-L shown in Fig. 3 (a), where length is labeled near each edge. At the first procedure of ALOMA-L, redundant edges such as edges '(b, e)' and '(d, e)' with the length 8 (which is greater than the number of nodes) are removed (Fig. 6 (a)), because memory accesses of these edges are realized in any memory allocation.

Figure 6 (b) shows the AG-L to be linearized, where the number labeled at every vertex denotes its fork count, the number in parentheses labeled on every edge denotes its weigthing function, and the bold number labeled on every edge group denotes its benefit. The same notations are used in the following Figs. 6 (c),(d). Note that weighting functions remain unchanged, during the following graph linearization procedure, and so they are omitted in the following figures.

Second, edge group (c, d) of the maximum benefit "4" is removed from the AG-L. After this removal, fork counts of vertices, cycle flags and benefits of edge groups

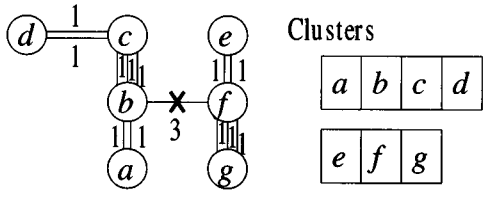


Fig. 7 An example AG-L for cluster allocation.

are renewed as shown in Fig. 6 (c).

In Fig. 6 (c), edge groups (a, b) and (b, c) have the same benefits "2." Here, we assume the method selects and removes the former edge group (a, b) (Fig. 6 (d)). Then, benefits of all the remaining edges become "0," and a line graph results. Finally, memory addresses for vertices a through e are assigned along the line graph, which are labeled as boldfaced superscripts on vertices (Fig. 6 (d)).

3.4 Cluster Allocation

ALOMA-L sometimes partitions a given AG-L into several line graphs which are called "clusters" in this paper. For example, an AG-L in Fig. 7 is linearized by removing edge (b, f) and 2 line graphs, a, b, c, d and e, f, g, are derived. Locations of these clusters onto memory space are very important to reduce the number of required AR loads. For the example in Fig. 7, one may allocate these 2 clusters as

$$a\text{–}b\text{–}c\text{–}d\text{–}e\text{–}f\text{–}g.$$

In this case, one AR load is required at edge (b, f) of length 3, because address distance between b and f is 4. In the case where the clusters are replaced as

$$e\text{–}f\text{–}g\text{–}a\text{–}b\text{–}c\text{–}d,$$

or in the case where elements in the first cluster are reversed as

$$d\text{–}c\text{–}b\text{–}a\text{–}e\text{–}f\text{–}g,$$

no AR load is required, because AR update amount at edge (b, f) is within its length. In this paper, the most efficient cluster allocation is selected among all possible allocations. This method is called cluster allocation method. The examination of all the possible allocations finishes in relatively short period, because the number of clusters does not increase so much as the size of a given AG-L.

3.5 The Optimal/Suboptimal Address Allocation

One can find the optimal memory allocation for a given AG-L by examining all the possible combination of memory address and variables, but this exhaustive search method costs a plenty of time. As an experimental example, memory allocation for a 5th-order wave

Table 2 Comparison of Memory Allocation Methods.

Filter (Order)	# of Steps	# of Variables	# of Accesses	Appearance Order # of AR loads	ALOMA [17] # of AR loads	ALOMA-L # of AR loads	ALOMA-L with Cluster Allocation # of AR loads	# of Clusters	Suboptimal (Iterative ALOMA-L) # of AR loads	Optimal (Exhaustive) # of AR loads
WDLF (5)	46	5	12	3	3	1	1	1	1	1
WDF (5)	71	14	33	7	11	8	8	2	7	5
WDF (7)	113	23	53	17	15	15	14	2	11	14*
WDF (9)	145	32	75	29	31	22	20	3	15*	—
WDF (11)	183	40	93	42	37	29	27	4	23*	—
WDF (17)	298	64	159	75	72	66	64	6	58*	—

*: Not yet finished. The minimum derived until now.
WDF: Wave Digital Filter WDLF: Wave Digital Lattice Filter

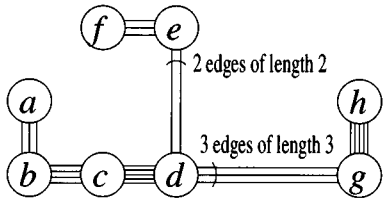


Fig. 8 An example AG-L of multiple selections.

digital filter (hereafter WDF) of 14 variables, which will also be used in the comparison of memory allocation methods, took about 3days for the exhaustive search on a workstation SUN Super Sparc20. Meanwhile, exhaustive search for 7th order WDF of 22 variables is estimated to take more than 500 million years. Therefore, some other methods to find a near optimal location is needed, as a reference to evaluate the proposed method.

At the selection of an edge group in the AG-L linearization procedure, sometimes there are several edge groups having the maximum benefit. In the ALOMA-L, one edge group among them is selected, but the number of AR loads in the resultant memory allocations is dependent on this selection. For an example AG-L in Fig. 8, two edge groups, (d, e) and (d, g) have the same cost $\frac{10}{7}$. Two AR loads are required after the removal of edge group (d, e), while no AR load is required after removal of edge group (d, g). If all the possible selections in edge groups of the same benefit are examined, a near optimal or sub-optimal memory allocation is found. This method is called 'Iterative ALOMA-L.' The followings are the procedures of Iterative ALOMA-L.

Iterative ALOMA-L

1. Build an AG-L for a given memory access sequence.
2. Remove edges whose lengths are greater than the number of variables.
3. For all the edge groups (u, v) in G, evaluate benefit(u, v).
4. Find an edge group with the maximum benefit and remove it from G. If multiple edge groups are of the maximum benefit, select one of them and mark the rest edge groups as 'not-yet-selected' edge groups.
5. Repeat 3–4, until G becomes a line graph.
6. For the vertices in the derived line graph, decide memory addresses and count the number of AR loads.
7. Back to the last selection of edge groups, select one among the not-yet-selected edge groups, and repeat 3–6.
8. Repeat 3–7, until all the 'not-yet-selected' edge groups are tried.
9. Output the memory address allocation of the minimum AR loads.

The computational time of the iterative ALOMA-L is less than that of the exhaustive search method, but it still exponentially increases; this method took 34 seconds for a 7th order WDF, but it did not complete for a 9th order WDF in 2 weeks.

3.6 Allocation Results

The above proposed methods are applied to the DIMPL compiler [14], [15] for μPD77230 (NEC) [7]. This compiler reads the filter network description, decides the computational ordering (scheduling), and generates an efficient code in terms of program length by utilizing arithmetic units and registers effectively. The proposed method is applied just after the code generation part. A program for μPD77230 often includes many codes without any memory accesses, because this DSP has 8 arithmetic registers. Codes are generated for several filter examples. Here, wave digital filters (WDF) of different orders are mainly employed, because they give complicated memory access sequences. The numbers of program steps, variables, and memory accesses are tabulated in the left side of Table 2. The table shows the comparison of the proposed methods to the existing

methods [17] in terms of the number of AR loads in the derived codes. The column "Appearance Order" shows result derived by a simple memory allocation method which assigns addresses to variables as the order of their appearance in generated assembler codes. That method is similar to memory allocation methods in C compilers [3], [4], [6], which assign addresses of variables as their declaration order. The numbers of AR loads in the optimal/suboptimal memory allocations are also shown in the same table.

From the table, the method with cluster allocation gives the better results in the heuristic methods. The number of AR loads derived by that method is slightly more than that of optimal/suboptimal results derived by the exhaustive method and iterative ALOMA-L. The exhaustive and the iterative ALOMA-L, however, require tremendous time and computational cost. From the table, the exhaustive method is not completed after long computational time even for rather small examples, and gives inefficient results. The iterative ALOMA-L gives the better results in rather short time. On the contrary, computational time of the method with cluster allocation is not so much. The column labeled "# of clusters" proves this fact, because the number of clusters remains small even for large sized access sequences. It is thus concluded that the proposed method with cluster allocation is the best among three.

4. AR Assignment

4.1 AR Assignment Algorithms

In the case of DSPs with multiple ARs, AR assignment for each memory access is as important as memory allocation. For simplicity, every variable in a given access sequence is assigned to one of ARs in this paper, rather than AR is selected for every access. Although this method restricts memory space accessed by each AR and reduces the choices in memory access pattern, practically efficient memory access patterns are derived under this limitation for many examples.

The diagram of AR assignment method is illustrated in Fig. 9. At first, V, a set of variables in a given memory access sequence is partitioned into k subsets (V_1 through V_k), where k is equal to the number of ARs. Each subset is accessed by an individual AR, and its memory location is decided using the ALOMA-L. Consequently, a given memory access sequence is divided into k subsequences. Then, other partitionings and memory allocations are iteratively tried, until the best partitioning which minimize the cost function is found. The cost function $f(P)$, which is the number of AR loads in this paper, controls the partitioning at the next iteration. For the partitioning method, the min-cut algorithm [22], [23], which is well known in graph theory, is applied.

From the third feature of the DSP model, when

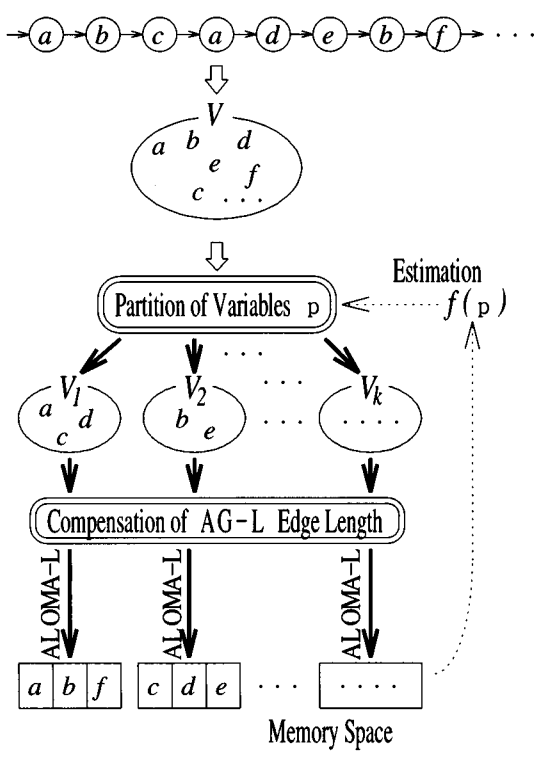


Fig. 9 AR assignment diagram.

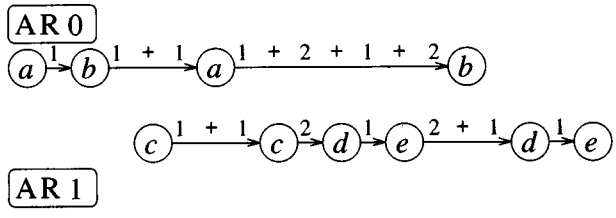


Fig. 10 Multiple ARs and their effects on the lengths of the edges.

one of ARs is currently used for memory access in an instruction cycle, other ARs can be updated at the same cycle. For example, we assume memory access sequence in Fig. 3 (a) is divided into two subsequences in Fig. 10. In the upper subsequence, AR0 is updated during the memory c, d, and e are accessed using AR1. Note that a code with memory access pointed by one of ARs is regarded as a code without memory access from the viewpoint of other ARs. Therefore, before memory allocation by ALOMA-L, the method compensates lengths of all the edges in each partitioned subset in the manner shown in Fig. 10.

4.2 AR Assignment Results

The above method is applied to DIMPL compiler, which is mentioned in 3.6. The target DSP μPD77230 has 2 memory pages and each memory page is accessed by 2 ARs. In this paper, however, 2 fictitious ARs are

Table 3 Comparison of AR assignment for multiple ARs.

Filter (Order)	Conventional Method				Novel Method			
# of ARs	1	2	3	4	1	2	3	4
WDF (5)	11	3	0	0	8	0	0	0
WDF (7)	20	4	2	0	14	0	0	0
WDF (9)	31	7	3	2	20	0	0	0
WDF (17)	72	29	11	3	64	7	0	0

WDF: Wave Digital Filter

added and total 4 ARs are assumed. The numbers of immediate AR loads in generated codes are tabulated in Table 3. From this comparison, the novel AR assignment method saves more overhead codes than the conventional method.

5. Conclusions

In this paper, a method to derive an efficient memory allocation is improved by utilizing AR operations at the codes without memory accesses. The method uses a novel graph model for representing address allocation of variables and AR operations. The memory allocation method employs a graph linearization algorithm with a novel cost function for the selection of edges to be removed. For the comparison, iterative method to derive a suboptimal memory allocation is presented. These methods are applied to DIMPL compiler for μPD77230, and the derived results are compared in terms of the numbers of overhead codes in generated codes together with the results derived by the existing simple memory allocation method. Codes generated by the proposed method can save up to 40% of immediate AR loads. The number of overhead codes in the proposed method is comparable to the optimal/suboptimal results. The proposed methods are easily applicable to C or other compilers, because only an access sequence of variables is used as an input. A method to derive an efficient memory access pattern for multiple ARs is also investigated. The results derived by this method include less overhead code than those derived by the existing method. Methods to derive further efficient memory access pattern which take memory addressing hardware into account are now studied.

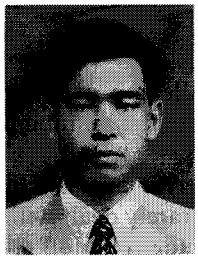
Acknowledgement

The authors are grateful to Prof. N. Fujii and Ass. Prof. S. Takagi of Tokyo Institute of Technology, for the valuable discussions. This work is supported by the Research Body of CAD21 (Computer Aided Design for 21th Century) in Tokyo Institute of Technology.

References

[1] M. Levy and A. Coyle, "EDN's 1996 DSP-chip directory," EDN, vol.41, no.5 pp.40–103, March 1996.
[2] R. Simar, Jr. and A. Davis, "The application of hign-level langueges to single chip digital signal processors," 1988 ICASSP, pp.1678–1681, 1988.
[3] Texas Instruments, "TMS320C30 C Compiler—Reference Guide," 1990.
[4] J. Hartung, S.L. Gay, and S.G. Haigh, "A practical C language compiler/optimizer for real-time implementations on a family of floating point DSPs," 1988 ICASSP, pp.1674–1677, 1988.
[5] MOTOROLA, "DSP56000/DSP56001 Digital Signal Processor User's Manual," 1990.
[6] Motorola, "56KCCx Full Kernighan and Richie C Compiler User's Manual."
[7] NEC, "μPD77230 Users Manual," 1987.
[8] E.A. Lee, W.-H. Ho, E.E. Goei, J.C. Bier, and S. Bhattacharyya, "Gabriel: A design environment for DSP," IEEE Trans. ASSP, vol.37, no.11, pp.1751–1761, Nov. 1989.
[9] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Multirate signal processing in ptolemy," Proc. ICASSP 1991, pp.1245–1248, April 1991.
[10] N. Kumamoto, K. Aoki, and H. Kunieda, "VIRGO: Hierarchical DSP code generator based on vectorized signal flow graph description," IEICE Trans., vol.E75-A, no.8, pp.1004–1013, Aug. 1992.
[11] B. Wess, "Automatic code generation for integrated digital signal processors," Proc. ISCAS 1991, pp.33–36, June 1996.
[12] W. Kreuzer, M. Gotschlich, and B. Wess, "A retargetable optimizing code generator for digital signal processors," Proc. ISCAS 1996, II, pp.257–260, May 1996.
[13] M. Karjalainen, T. Altosaar, and P. Alku, "QuickSig—An object oriented signal processing environment," Proc. ICASSP 1988, pp.1682–1685, May 1988.
[14] 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 Trans., vol.J71-A, no.2, pp.327–335, 1988.
[15] N. Sugino, S. Ohbi, and A. Nishihara, "Computational ordering of digital network under the pipeline constraints and its application to compiler for DSPs," Proc. ECCTD'89, pp.395–399, Sept. 1989.
[16] S. Ohbi, N. Sugino, and A. Nishihara, "Automatic DSP code generation with effective utilization of storage resources—Improvement in DSP compiler DIMPL," Proc. of 4th Digital Signal Processing Symposium, pp.37–41, Dec. 1989.
[17] N. Sugino, S. Iimuro, A. Nishihara, and N. Fujii, "DSP code optimization utilizing memory addressing operation," IEICE Trans. Fundamentals, vol.E79-A, no.8, pp.1217–1224, Aug. 1996.
[18] N. Sugino, H. Miyazaki, S. Iimuro, and A. Nishihara, "Improved code optimization method utilizing memory addressing and its application to DSP compiler," Proc. ISCAS 1996, II, pp.249–252, May 1996.
[19] N. Sugino, H. Miyazaki, and A. Nishihara, "An improved memory address allocation for DSP code optimization," A-105, Proc. 1996 Society Conference of IEICE, Sept. 1996.
[20] N. Sugino and A. Nishihara, "A memory addressing optimization method for DSP code improvement and its application to compilers," Proc. of 11th Digital Signal Processing Symposium, pp.693–698, Dec. 1996.
[21] V. Chachra, P.M. Ghare, and J.M. Moore, "Applications of graph theory algorithms," Elsevier North Holland, 1979.
[22] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," Bell System Technical Journal, vol.49, pp.291–307, Feb. 1970.
[23] C.M. Fiducca and R.M. Mattheyses, "A linear-time heuris-

tic for improving network partitions," Proc. 19th DAC, pp.175–181, 1982.



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 a lecturer of department of information processing, interdisciplinary graduate school of science and engineering. His main research interests are in hardware and software for digital signal processing, especially in software development tools for digital signal processors. Dr. Sugino is a member of IEEE.

Hironobu Miyazaki was born in Fujisawa, Kanagawa, Japan on March 11, 1973. He received B.E. degree of physical electronics, and M.E. degree of information processing from Tokyo Institute of Technology in 1995 and 1997, respectively. Since 1997, he has been with Sony corporation, where he works on software for commercial audio products.

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 Trans. Fundamentals. from 1990 to 1994, and an Associate Editor of the IEEE Transactions on Circuits and Systems II from 1996 to 1997. He was 1995–1996 Student Activities Committee Chair of IEEE Region 10 (Asia Pacific Region). Dr.Nishihara is a member of IEEE, EURASIP, ECS and JET.