Towards the Least Complex Time-Multiplexed Constant Multiplication

Levent Aksoy
INESC-ID
Lisboa, Portugal

Paulo Flores
INESC-ID/IST TU Lisbon
Lisboa, Portugal

José Monteiro
INESC-ID/IST TU Lisbon
Lisboa, Portugal

***Abstract*—The multiplication of a variable by a single constant selected from a set of fixed constants at a time, called the time-multiplexed constant multiplication (TMCM), is frequently used in digital signal processing (DSP) systems. Existing algorithms implement the TMCM operation using multiplexers (MUXes), adders/subtractors, and shifts, and reduce its complexity by merging single/multiple constant multiplication graphs and by sharing the basic structures. This paper introduces ARION, that exploits the most common partial terms in the TMCM design on top of the previously proposed DAGfusion algorithm, which merges the single constant multiplication graphs. Experimental results show that ARION obtains significantly better solutions than prominent TMCM methods.**

I. Introduction

The multiple constant multiplications (MCM) operation, that realizes the multiplication of fixed constants by a variable, dominates the complexity of many DSP systems, *e.g*, filters and linear transforms. Since the implementation of a multiplier in hardware is expensive in terms of area and the constants are determined beforehand, the constant multiplications are generally realized using shifts and adders/subtractors. Note that shifts can be implemented using wires without representing any hardware cost. The prominent MCM algorithms [1]–[4] find the fewest number of adders/subtractors by sharing the partial products among the constant multiplications.

However, realizing constant multiplications simultaneously in a parallel architecture increases the design complexity significantly as the number and size of constants increase. Hence, constant multiplications are generally implemented in a folded architecture, where at each time, a single constant selected from a set of constants is multiplied by an input variable, which is called the TMCM operation. In addition to its straightforward implementations given in [5], the TMCM operation can be realized using a set of basic structures that consists of an adder, a subtractor, or an adder/subtractor (determined by a select input), all of which may include MUXes at its inputs. Thus, the TMCM problem is defined as finding a set of basic structures that realizes the TMCM operation and leads to a TMCM design with minimum complexity. In existing algorithms [5]–[10], the complexity of the TMCM operation is determined based on the design platform, *i.e.*, field programmable gate arrays (FPGA) or application specific integrated circuits (ASIC). However, the exact algorithm [6] can only be applied to a small number of constants and there exists no approximate method that ensures the best solution on every TMCM operation. The reader is referred to [5], [10] for a detailed overview on previously proposed TMCM algorithms.

This paper introduces our algorithm, called ARION, that targets single-output TMCM operations for the ASIC design and combines both the exploitation of the most common partial terms and merging of single constant multiplication (SCM) graphs which is realized by DAGfusion [5].

II. ARION: A TMCM Algorithm

ARION takes a set of multiple positive[1] constants C of the TMCM operation as an input and returns a set of basic structures realizing TMCM. Its main steps are as follows:

1) Find a solution to C using DAGfusion and compute its implementation cost $cost_{DAG}$ as described in [5].
2) In a preprocessing phase, determine the non-redundant target set T, that ARION will be applied to, from C.
3) Add T to an empty set, called Y, that will include the sets of constants which are required to realize T. Determine the initial node(s) of the decision tree that will be used to decide which realization of a set of constants will be chosen in order to have the minimum design complexity.
4) Take an element from Y, Y_i. Find alternative possible realizations of Y_i using a basic structure with the smallest cost. Add the partial sets of constants (PSC) required to realize Y_i to the set Y. Update the decision tree with these realizations and their implementation costs.
5) Find the implementation of Y_i using DAGfusion and compute its cost.
6) If all elements of Y are not handled yet, go to Step 4.
7) Find a set of basic structures that realize C with minimum complexity, $cost_{ARION}$, using the decision tree and the solution of DAGfusion on each Y_i found in Step 5.
8) If $cost_{ARION} < cost_{DAG}$, return the solution found by ARION. Otherwise, return the solution of DAGfusion.

These steps are explained briefly through a simple TMCM example with $C = \{8, 46, 58, 32\}$ in the next subsections.

A. Step 1: Finding a Solution with DAGfusion

The solution of DAGfusion on C is given in Fig. 1(a), where a number next to an edge denotes the amount of left shift and select inputs of MUXes are not given for the sake of clarity.

B. Step 2: Preprocessing Phase

We eliminate the repeated constants of C because they are redundant. Note that this elimination requires a combinational logic to map the primary select input of the TMCM operation

[1] It is always assumed that the sign of a constant is handled where the constant multiplication is required using an adder/subtractor.

978-1-4799-0524-9/13/$31.00 ©2013 IEEE

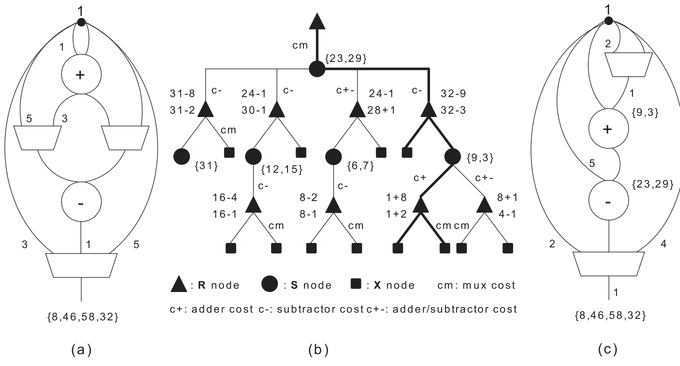

Fig. 1. Some results obtained by ARION on $\{8,46,58,32\}$: (a) solution of DAGfusion; (b) decision tree constructed in ARION; (c) solution of ARION.

into the select inputs of MUXes and adders/subtractors. The smallest amount of left shift ls_c on the constants of C is determined, and the constants are divided by 2^{ls_c}. The constants of C, that are 1, denoting the input variable, or its shifted versions, are removed from C to a set called I and the elements of the target set T are determined as the elements of C. Note that when $|I| \geq 1$ and T is not empty, we need a MUX, whose inputs are the elements of I and the output of the TMCM operation realizing T which will be found. When $|I| \geq 2$ and T is empty, a MUX is also required. Otherwise, no MUX is needed. For our example, ls_c is 1, I is found as $\{4, 16\}$, T is determined as $\{23, 29\}$, and the MUX, whose output is shifted by ls_c times, is depicted at the bottom of Fig. 1(c).

C. Step 3: Determining the Initial Nodes of the Decision Tree

We add the target set T to Y and determine the initial node(s) of a decision tree. As shown in the decision tree of Fig. 1(b) constructed for our example, it includes three different types of nodes: i) an **S** node denotes a set of constants in Y, Y_i; ii) an **R** node stands for a realization of Y_i; and iii) a **X** node represents a set of constants including only 1, *i.e.*, the input variable, or its shifted versions. An edge presents a connection between an **R** node and an **S** or **X** node, where its value denotes the cost of an operator realizing the **S** or **X** node. If no operator is required for the realization of the **S** or **X** node, the value of the associated edge is zero, which is not shown in Fig. 1(b).

Thus, if a MUX is used in Step 2, the decision tree starts with an **R** node that is connected to an **S** node denoting Y_1, *i.e.*, T, with an edge whose value is the cost of this MUX. Otherwise, its initial node is an **S** node denoting Y_1. For our example, the initial nodes can be found in Fig. 1(b).

D. Step 4: Finding Promising Realizations

This step is applied to Y_i including more than one constant. Otherwise, the solution of DAGfusion (Step 5) is used.

As done in [4], we find all possible realizations of each constant of Y_i by decomposing its nonzero digits in two partial terms when it is defined under minimal signed digit (MSD) representation. Note that a signed digit number system uses the digit set $\{\overline{1}, 0\ 1\}$, where $\overline{1}$ denotes -1, and in MSD, a constant may have alternative representations, all with minimum number of nonzero digits. Fig. 2(a) presents the possible realizations of the elements of $Y_1 = \{23, 29\}$. Removing the same realizations of 23 and 29, *i.e.*, $-1 + 24$ and $32 - 3$, respectively, there exist 5 implementations for both constants.

Our aim is to find a realization (an adder or a subtractor) for each constant in Y_i such that these operations include the minimum number of distinct partial terms at their inputs. The reason behind this is common partial terms reduce the sizes of MUXes in the basic architecture and the number of elements in PSC. This problem is formalized as a 0-1 integer linear programming (ILP) problem.

We represent the possible implementations of constants in a Boolean network that includes only AND and OR gates. An operation (an adder or a subtractor) realizing the constant is represented by an AND gate. For an adder, two AND gates are generated and are denoted as $AND_{p_1+p_2}$ and $AND_{p_2+p_1}$ due to the commutative law of addition. These two AND gates are also combined with an OR gate, denoted as $OR_{p_1\&p_2}$, indicating that both of them generate the same constant with the same partial terms, but on different inputs. For a subtractor, we assume that the first input is the partial term with the positive sign and the second input is the one with the negative sign, and we generate an AND gate denoted as $AND_{p_1-p_2}$. For each constant c_j of Y_i, an OR gate, OR_{c_j}, is generated to combine all possible realizations of c_j. Each partial term p_k at the first or second input of an operation is denoted as an optimization variable, $O_1|p_k|$ or $O_2|p_k|$, respectively. The network generated for $Y_1 = \{23, 29\}$ is shown in Fig. 2(b).

The objective function of the 0-1 ILP problem is obtained as a linear combination of optimization variables, where the cost value of each optimization variable is 1. Its constraints are obtained by finding the conjunctive normal form (CNF) formulas of each gate in the network and expressing each clause of the CNF formulas as a linear inequality, as described in [11]. For example, a 2-input AND gate, $c = a \wedge b$, is translated to CNF as $(a+\overline{c})(b+\overline{c})(\overline{a}+\overline{b}+c)$ and converted to linear constraints as $a-c \geq 0$, $b-c \geq 0$, $-a-b+c \geq -1$. Also, the outputs of OR gates associated with constants, OR_{c_j}, are set to 1, since they need to be implemented.

This problem is given to a generic 0-1 ILP solver SCIP (http://scip.zib.de/) and a minimum solution is found. If the solution does not include any common partial terms, then this process is terminated and the solution of DAGfusion (Step 5) is determined to be the only realization of Y_i. Otherwise, the operation in the basic structure is determined as: an adder, if all selected operations (the outputs of AND gates set to 1 by SCIP) are adders; a subtractor, if all selected operations are subtractors; and an adder/subtractor, otherwise. Also, the PSC and the set of shifted versions of the input variable (SIV) are determined based on the first and second inputs of all selected operations. To do so, the partial terms on the first (second) inputs of all selected operations are grouped in a set G_1 (G_2), and the procedure given in Step 2 is applied, where G_1 (G_2), PSC, and SIV correspond to C, T, and I in Step 2, respectively. For our example, according to a solution of SCIP, 23 and 29 are realized as $31-8$ and $31-2$, respectively, which require a basic structure including a subtractor, leading to $G_1 = \{31, 31\}$ and $G_2 = \{8, 2\}$. For G_1, PSC is found as $\{31\}$ and SIV is

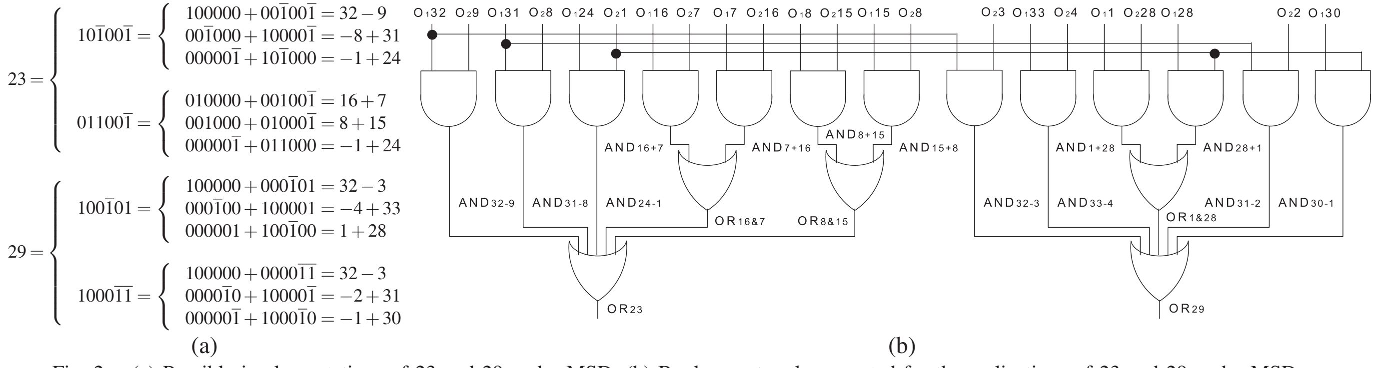

Fig. 2. (a) Possible implementations of 23 and 29 under MSD; (b) Boolean network generated for the realizations of 23 and 29 under MSD.

empty. For G_2, PSC is empty and SIV is found as $\{4,1\}$ which requires a MUX whose output is shifted by 1 times.

There may exist more than one minimum solution to this problem, all of which lead to different realizations of Y_i with different costs. We iteratively obtain another minimum solution by adding a single constraint to the 0-1 ILP problem, that prohibits the previous solution to be a solution again, and by solving this problem. In general, this constraint has the form of $AND_1 + AND_2 + \ldots + AND_m \leq m-1$, where m denotes the number of constants in Y_i and AND_j with $1 \leq j \leq m$ denotes the AND gate set to 1 for each constant of Y_i in the previous solution. If all operations in the previous solution are adders, rather than the AND gates, the OR gates $OR_{p_1 \& p_2}$, combining two possible realizations, are stated in this constraint to avoid a similar solution. If this solution includes a number of distinct partial terms greater than the minimum (found without any of these constraints in the first 0-1 ILP problem), the process of finding alternative solutions is stopped. It is also terminated when up to 100 minimum solutions[2] are considered. All other realizations of 23 and 29 with minimum number of distinct partial terms at their inputs are found as i) $24-1$ and $30-1$; ii) $24-1$ and $28+1$; iii) $32-9$ and $32-3$.

After possible realizations of Y_i are found, the actual cost of each basic structure is computed as described in [5]. The cost values of all PSC in each realization of Y_i are also estimated. It is assumed that $max(Oscm(c_j))$ adders/subtractors, where $Oscm(c_j)$ denotes the minimum number of operations required to realize SCM of c_j in the PSC [1], are needed and both inputs of an adder/subtractor include a MUX. Thus, all realizations of Y_i are sorted according to the actual cost value of the basic structure plus the estimated cost values of its all PSC. Then, at most 5 realizations[3] with the smallest cost value are selected to be possible realizations of Y_i. All PSC required in these realizations are added to Y.

The realizations of Y_i (denoted as **R** nodes) and all PSC (denoted as **S** nodes) and SIV (denoted as **X** nodes) in its realizations are added to the decision tree with the implementation cost of an operator. While the edge value between an **S** node and an **R** node represents the cost of an operation, the edge value between an **R** node and a **X** or **S** node stands for the cost of a MUX if it is required (Fig. 1b).

[2]Although more solutions can yield less complex TMCM designs, the runtime of ARION is increased due to more 0-1 ILP problems to be solved.

[3]Although more possible realizations can lead to better solutions, the runtime of ARION is increased due to more PSC to be considered.

E. Step 5: Finding an Alternative Realization with DAGfusion

We apply DAGfusion to Y_i, find a solution, compute its implementation cost, and record this value. The reason behind that is it may obtain a realization of Y_i which is never considered in Step 4, since the possible implementations of each constant in Y_i are limited to its MSD representation. DAGfusion is also used to realize Y_i including one constant with minimum number of operations, *e.g.*, 31 in Fig. 1(b).

F. Step 7: Finding a Solution with Minimum Complexity

After all elements of Y are considered, starting from the **X** nodes in the decision tree, the minimum implementation cost of each Y_i is computed considering its possible realizations. Then, this value is compared to that found using DAGfusion on the same Y_i and the implementation (the one obtained in Step 4 or in Step 5), that has the minimum value, is determined. This process iterates level by level on the decision tree until the initial node is reached. Then, starting from the initial node, we choose the implementations with the minimum cost, forming a set of operations and MUXes, and return $cost_{ARION}$. For our example, these implementations are shown in bold lines in Fig. 1(b) that lead to Fig. 1(c).

G. Step 8: Determining the Final Solution

If $cost_{ARION}$ is smaller than $cost_{DAG}$, then the solution found in Step 7 is returned. Otherwise, the solution of DAGfusion found in Step 1 is returned. For our example, final solution is the one obtained in Step 7.

III. Experimental Results

In this section, we present the results of ARION on randomly generated instances and on two benchmark sets, and compare with those of prominent TMCM algorithms. Note that ARION was written in MATLAB and was run on a PC with Intel Xeon at 2.4GHz and 10GB memory.

For the comparison of ARION with DAGfusion, we used randomly generated instances where the bitwidth of constants (bwc) varies in between 6 and 14 in increment of 2 and the number of constants (n) ranges between 4 and 20 in increment of 4. For each group, there were 30 instances, a total of 750 instances. Table I presents the results of algorithms, where #*BS*

TABLE I
SUMMARY OF ALGORITHMS ON RANDOMLY GENERATED INSTANCES.

bwc	*n*	4	8	12	16	20
6	*#BS*	19	24	13	17	19
	Avg Gain	10.1	9.4	5.0	7.5	6.5
	Max Gain	26.1	22.2	15.2	25.1	16.7
	CPU DAGfusion	0.2	0.7	1.1	1.6	2.2
	CPU ARION	5.8	17.8	33.5	36.7	25.2
8	*#BS*	23	25	13	18	15
	Avg Gain	10.8	7.4	5.0	4.5	4.8
	Max Gain	29.7	17.2	21.0	21.8	19.3
	CPU DAGfusion	0.2	1.2	1.8	2.7	3.1
	CPU ARION	16.2	50.8	83.0	137.1	206.9
10	*#BS*	14	20	15	15	11
	Avg Gain	5.9	8.7	5.1	5.4	5.5
	Max Gain	28.8	20.6	22.5	15.6	16.7
	CPU DAGfusion	0.2	2.3	7.2	16.8	30.3
	CPU ARION	40.9	102.6	228.4	439.2	718.9
12	*#BS*	22	20	19	15	12
	Avg Gain	9.0	9.1	8.2	5.0	6.9
	Max Gain	28.8	25.0	33.4	19.6	26.8
	CPU DAGfusion	0.2	6.8	32.5	63.5	82.7
	CPU ARION	80.5	218.4	417.9	607.3	1159.1
14	*#BS*	26	19	20	18	16
	Avg Gain	11.6	12.5	7.5	7.5	8.2
	Max Gain	34.5	32.0	25.2	19.3	30.9
	CPU DAGfusion	0.3	15.1	55.1	94.9	160.5
	CPU ARION	174.2	838.0	955.6	1087.9	1184.1

TABLE II
AREA COST ESTIMATION FOR DATA SET A.

Method	Add	Sub	Add/Sub	MUX	Cost
DAGfusion	0	0	1 (12-bit) 1 (14-bit)	1 (14-bit) 3×1 1 (16-bit) 7×1	4704
[9]	1 (10-bit)	1 (12-bit)	1 (10-bit) 2 (11-bit) 1 (12-bit) 1 (16-bit)	1 (8-bit) 2×1 1 (9-bit) 2×1 2 (10-bit) 2×1 1 (11-bit) 2×1 1 (12-bit) 2×1 1 (16-bit) 2×1	9578
[10]	0	0	1 (12-bit) 1 (14-bit)	1 (14-bit) 3×1 1 (11-bit) 4×1 1 (16-bit) 4×1	4648
ARION	1 (12-bit)	0	1 (14-bit)	1 (10-bit) 2×1 1 (11-bit) 2×1 1 (14-bit) 2×1 1 (16-bit) 6×1	4500

denotes the number of better solutions that ARION found with respect to DAGfusion in terms of design complexity, computed using the 0.18-μm standard cell library as described in [5] when the bitwidth of the input variable (*bwi*) was 16. Also, *Avg Gain* and *Max Gain* denote the average and maximum gain in percentage obtained by ARION over DAGfusion, respectively. The average runtime of both methods is presented in seconds.

Observe that ARION can find significantly better solutions than DAGfusion, where the maximum gain is 34.5% obtained on a TMCM instance with 14-bit 4 constants. On average, it obtains better solutions than DAGfusion, where the maximum *#BS* value is 26 out of 30 instances found on 14-bit 4 constants. However, its runtime increases as *n* and *bwc* increase, since the number of sets of constants considered in ARION is increased. At most, it is 555 times slower than DAGfusion, obtained on TMCM instances with 14-bit 4 constants.

For the comparison of ARION with prominent TMCM algorithms, it was applied to two data sets, A and B, used in [10]. Tables II and III present the results of algorithms, where *Cost* is computed using the 0.18-μm standard cell library when *bwi* is 8 as in [10]. Note that the results of other algorithms were taken from [10] as reported.

TABLE III
AREA COST ESTIMATION FOR DATA SET B.

Method	Add	Sub	Add/Sub	MUX	Cost
DAGfusion	1 (19-bit)	1 (16-bit)	1 (12-bit)	2 (12-bit) 2×1 1 (19-bit) 2×1 1 (16-bit) 3×1 1 (20-bit) 3×1	6365
[9]	1 (11-bit) 1 (12-bit) 1 (17-bit)	0	1 (11-bit)	3 (11-bit) 2×1 1 (14-bit) 2×1	5074
[10]	1 (17-bit)	1 (11-bit) 1 (12-bit) 1 (14-bit)	0	1 (14-bit) 2×1 1 (16-bit) 2×1 1 (17-bit) 2×1 1 (18-bit) 3×1	5986
ARION	1 (10-bit)	1 (15-bit) 1 (18-bit)	0	1 (11-bit) 2×1 1 (12-bit) 3×1 1 (18-bit) 3×1	4713

Observe that the solutions of ARION lead to the least complex TMCM designs on sets A and B, where the gain over the second best solution is 3.18% and 7.11%, respectively.

IV. CONCLUSIONS

This paper introduced a TMCM algorithm ARION that finds alternative basic structures realizing a given set of constants with the smallest complexity and incorporates an efficient TMCM method DAGfusion to consider another possible implementation of the set of constants. In ARION, finding such basic structures was formalized as a 0-1 ILP problem and the alternative realizations were handled with the use of a decision tree. Experimental results showed that ARION can find significantly better solutions than prominent TMCM algorithms.

V. ACKNOWLEDGMENT

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

REFERENCES

[1] O. Gustafsson, A. Dempster, and L. Wanhammar, "Extended Results for Minimum-Adder Constant Integer Multipliers," in *ISCAS*, 2002, pp. 73–76.

[2] A. Dempster and M. Macleod, "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters," *IEEE TCAS II*, vol. 42, no. 9, pp. 569–577, 1995.

[3] Y. Voronenko and M. Püschel, "Multiplierless Multiple Constant Multiplication," *ACM Tran. on Algorithms*, vol. 3, no. 2, 2007.

[4] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, "Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications," *IEEE TCAD*, vol. 27, no. 6, pp. 1013–1026, 2008.

[5] P. Tummeltshammer, J. Hoe, and M. Püschel, "Time-Multiplexed Multiple-Constant Multiplication," *IEEE TCAD*, vol. 26, no. 9, pp. 1551–1563, 2007.

[6] N. Sidahao, G. Constantinides, and P. Cheung, "Multiple Restricted Multiplication," in *FPL*, 2004, pp. 374–383.

[7] R. Turner and R. Woods, "Highly Efficient, Limited Range Multipliers for LUT-based FPGA Architectures," *IEEE TVLSI*, vol. 12, no. 10, pp. 1113–1117, 2004.

[8] N. Sidahao, G. Constantinides, and P. Cheung, "A Heuristic Approach for Multiple Restricted Multiplication," in *ISCAS*, 2005, pp. 692–695.

[9] S. Demirsoy, I. Kale, and A. Dempster, "Reconfigurable Multiplier Blocks: Structures, Algorithm and Applications," *Circuits, Systems and Signal Processing*, vol. 26, no. 6, pp. 793–827, 2007.

[10] J. Chen and C.-H. Chang, "High-Level Synthesis Algorithm for the Design of Reconfigurable Constant Multiplier," *IEEE TCAD*, vol. 28, no. 12, pp. 1844–1856, 2009.

[11] P. Barth, "A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," Max-Planck-Institut Fur Informatik, Tech. Rep., 1995.