# Cost-Effective Multiplication with Enhanced Adders for Multimedia Applications

Zhen Luo and Ruby B. Lee
Princeton University
{zhenluo, rblee}@ee.princeton.edu

## Abstract

*Cost-sensitive consumer multimedia devices based on MPEG and JPEG type algorithms tend to have multiplications by constants, rather than by variables. In this paper, we show how slightly enhanced adders may be used to perform these constant multiplications with higher performance than more expensive hardware multipliers, using low-cost preshift_add instructions. We were able to find the shortest instruction sequences for all 8-bit integer constants and nearly shortest sequences for 12-bit constants and 15-bit constants. We have achieved an average instruction length of 3.055 for 8-bit integer case, 4.2643 and 4.2782 for the two 12-bit constant cases and 5.07673 for the 15-bit constant case. Based on our preshifter design, we evaluate the area and delay of a 16-bit preshift_adder and compare it with a 16×16 multiplier. We show that the simpler preshift_adders achieve a speedup of more than 2X compared to multipliers with similar area cost for typical algorithms like DCT and IDCT.*

## 1. Introduction

Consumer multimedia devices such as DVD players and cameras are very cost-sensitive. The MPEG and JPEG type algorithms they use tend to have multiplications by constants, rather than by variables. In this paper, we focus on cost-effective architectural support for such multiplications. We propose using adders enhanced with pre-shifters to perform efficient constant multiplies. This reduces the cost, as we show in the paper that an integer multiplier takes about three times the latency and three to four times the area over our design of a delay/area efficient preshift_adder to perform the preshift_add instructions. However, it is not easy to find the shortest instruction sequence for each constant multiply. We show our methodology for achieving this using a Directed Acyclic Graph (DAG) approach, to generate the shortest or nearly shortest sequence of instructions for every constant multiplier up to 15 bits. These optimal instruction sequences can be substituted by a programmer or compiler when a multiply by a constant is needed. Our performance results show that we can improve the performance while reducing the cost of constant multiplications.

We use four fixed-point cases to evaluate the instruction sequences that our algorithm generates. We use **CI.F** to denote that the positive constant multiplier has ***I*** bits of integer and ***F*** bits of fraction. The four cases are C8.0, C12.0, C2.10 and C3.12. While we are more interested in cases like C2.10 and C3.12 for our multimedia applications, where constants tend to be fractions with few integer bits, we generate C8.0 and C12.0 sequences, to compare our results with earlier work in [4] on constant multiplication by integers.

Sections 2 and 3 describe our DAG-based search algorithm for finding the shortest instruction sequence for C8.0 case and the nearly shortest instruction sequences for C12.0, C2.10 and C3.12 cases. Section 4 presents our performance results, and comparisons to earlier work [4]. Section 5 presents our design of a preshift_adder. Based on the results from section 4 and 5, we discuss the performance/area gain we achieve on an optimized DCT/IDCT algorithm in section 6.

## 2. Algorithm overview

### 2.1 Candidate instructions

We choose a subset of the HP MAX instruction set [1-3] to implement subword constant multiplication (Table 1). Our goal is to find the shortest instruction sequences composed of these instructions that achieve constant multiplication of N-bit (N≤16) fixed-point numbers.

| | |
|---|---|
| PSHLn, PSHRn | Parallel shift (n = 1 ...15) |
| PADD, PSUB | Parallel add and subtract |
| PSHLxADD[op1,op2]<br>PSHRxADD[op1,op2] | Preshift a signed 16-bit subword op1 by x (x=1,2,3) bits and add it with op2. |
| PAVG[op1, op2] | Parallel Average of op1 and op2 |

**Table 1** Instruction Mix for Subword Constant multiplication

In earlier work [4], the authors used shift_left_and_add instructions to replace constant multiplication by integers. They used a heuristic based on chain rules [5] to generate the instruction sequences for integers from 1 to 10,000. In our work, however, we focus on multiplication by fractional constants, using subword parallel shift_right_and_add instructions in addition to parallel shift_left_and_add instructions, to find the shortest instruction sequences for any fixed-point numbers of 8, 12 and 15 bits.

Subword parallel instructions[2] allow multiple parallel operations on lower-precision data packed into a word, e.g., four 16-bit subwords in a 64-bit word. They enable even higher perfomance using 64-bit adders for four parallel 16-bit operations per cycle, but they are not essential to understanding how the shift_and_add sequences are generated below.

### 2.2 Representing instruction sequence with DAG

We use a **DAG** (Directed Acyclic Graph) to describe all possible instruction sequences for constant multiplication. The root of this DAG is register R1 that contains the variable multiplicand and each node in the DAG corresponds to one of the instructions in table 1. An edge is added from node i to node j if the instruction on node j uses the result of node i as its operand and we call node i a **parent** of node j. Since all our instruction candidates takes two operands (for shifts, one operand is a constant) and generates one result, each node in the DAG has two input edges and one output edge. We define the **sub-DAG** of the root to be itself and we recursively define the sub-DAG of any node N to be the union of node N and the sub-DAGs of all its parents. Any instruction sequence can be modeled as a sub-DAG. For example, for the following instruction sequence,

```
1. PSHR3ADD R1 R1 R2   ;R2=[1.001]2×R1
2. PSHL1ADD R1 R1 R3   ;R3=[11]2×R1
3. PSHR3ADD R2 R3 R4   ;R4=[11.001001]2×R1
4. PSHR1ADD R1 R3 R5   ;R5=[11.1]2×R1
5. PSHR3ADD R4 R5 R6   ;R6=[11.111001001]2×R1
```

its corresponding sub-DAG is shown Figure 1. A sub-DAG shows the data flow of an instruction sequence. All the instruction sequences that have the same data flow correspond to the same sub-

0-7803-5482-6/99/$10.00 ©2000 IEEE

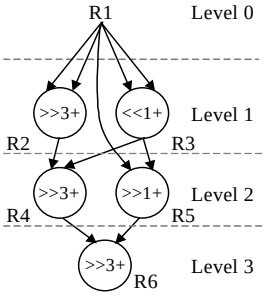



**Figure 1** A Sample DAG

DAG. For example, if we reorder the above instruction sequence by exchanging instruction 3 and 4, the new instruction sequence has the same data flow as the original one; thus they correspond to the same sub-DAG in Figure 1.

We define the **size** of a given node to be the number of nodes in its sub-DAG except the root. Thus the size of a given node corresponds to the length of the instruction sequence resulting in this node. Similarly, we define the size of a set of nodes to be the number of nodes in the union of their sub-DAGs except for the root. Since DAG is acyclic, we use **level** to show the longest distance between the root and a certain node. The level of the root is defined to be 0 and the level of a certain node S is recursively defined as 1 + the largest level of its parents. In our case, R1 is the only root.

The structure of a DAG shows the amount of parallelism within the computation. For a sub-DAG of level L, the minimum cycles needed to compute the final result is L cycles.

### 2.3 General constraints

For a given constant multiplication **MUL R1, C, R2** (R1 contains the variable multiplicand, C is the constant, R2 contains result), we impose two constraints when generating instruction sequences.

**a. No excess overflow**

If the original constant multiplication does not produce any overflow, neither should the replaced instruction sequence. Thus for all the intermediate results of the instruction sequence, their absolute value should be less than the final result. For example, replacing R1×31 by R1<<5 – R1 is not allowed since |R1<<5| > |R1×31|. This is different from the approach used in [4], in which excess overflow is allowed. In MAX-2, overflow is handled through saturation, thus the approach in [4] will suffer from loss of precision when excess overflow occurs.

**b. No direct loss of precision**

Consider instruction sequences for multiplication by a constant of k-bit fraction, the loss of precision in the final result is inevitable because of the rounding in intermediate instructions. However, some instruction sequences are preferred to others. For example, for the instruction sequences below for multiplication by $[0.011]_2$,

| | |
|---|---|
| PSHR R1 2 R2 | PSHR R1 3 R2 |
| PSHR1ADD R2 R2 R3 | PSHL1ADD R2 R2 R3 |

The left sequence is preferred because the 3rd least significant bit of R1 is kept after the first instruction and used in getting the final result in the second instruction. The problem with the right sequence is that it shifts too much to the right: the 3rd least significant bit is lost and it has to shift left again to get the result. To solve this problem, we do not allow any fractional intermediate results to shift left, i.e., we do not allow instructions like PSHLxADD[Ra, Rb] or PSHL[Ra, n], if Ra is not a multiple of R1.

By checking the instruction sequences generated, we consider these two constraints sufficient. Figure 1 shows an example for multiplication of $[11.111001001]_2$. We apply these two constraints to find the instruction sequences in all our algorithms.

## 3. Searching for Optimal Instruction Sequences

Our objective is thus to generate all possible nodes of the same result and choose one among them with the smallest size. We cannot just throw away the other nodes, however, since they could be used in the sub-DAG of some other nodes with the smallest size for their results. Nor can we throw away a node whose corresponding instruction sequence violates **constraint a** in Section 2.3, since this node could also be used in the shortest sequence for some other results. For example, although (R1<<5 – R1) is not a valid instruction sequence for constant multiplication by $[11111]_2$, we still have to keep this node since it is used in the shortest instruction sequence for $[111110]_2$, (R1<<5 – R1)<<1. Because of this, we have to keep a huge number of intermediate nodes. We estimate [7] that for level i, both the temporal and spatial complexity of the straightforward full search algorithm is greater than $8^{(2^i-1)}$, making full search impractical for i > 3.

The above full search algorithm can be refined by removing the redundant nodes. A node N is redundant if there are one or more nodes $N_i$ (i = 1, 2, 3... k) such that

a) $\text{Level}(N_i) \le \text{Level}(N)$; i = 1, 2, 3 … k
b) $\text{Size}(\{N_i\}) \le \text{Size}(N)$; i = 1, 2, 3 … k
c) $\cup\text{Values}(N_i) \supseteq \text{Values}(N)$; i = 1, 2, 3 … k

Here ***Values***(N) is defined as the set of the results of all the nodes in node N's sub-DAG.

Based on this criterion, we found the following redundant nodes:

a) Its result equals to any of the nodes' result in its sub-DAG.
b) If B = PSHL[A, n1], C = PSHL[B, n2], C is redundant, since C = PSHL[A, (n1 + n2)]
c) If C = PSHL[A, x] (x = 1,2,3), D = C + B, D is redundant, since D = PSHLxADD[A, B]
d) If C = PSHL[A, $x_1$], D = $\text{PSHL}_{x2}$ADD[C, B] ($x_1+x_2 \le 3$), D is redundant, since D = $\text{PSHL}_{(x1+x2)}$ADD[A, B]

Still, the complexity for the full search algorithm is too high even for 12-bit constants. We had to use the following heuristics to cut down the search space:

a) Remove nodes with negative results. For an instruction sequence with negative intermediate results, we can almost always reconstruct the instruction sequence so that it is composed of all positive intermediate results.
b) Keep only the intermediate nodes that has the smallest size for its result at the current level (there could exist nodes of smaller size for this result at higher levels). In this way, we throw away huge quantities of intermediate nodes that are very unlikely to be used in the shortest instruction sequences.
c) For C3.12 case, the complexity is still too high even after we apply heuristics a) and b). By keeping only one smallest-size node for every result at each level instead of all the smallest-size nodes as in b), we found that none of the instruction sequences in case C3.12 requires more than 7 instructions and very few constants require 7 instructions. Thus we only had to find all the nodes of size 6 or less since we already obtained instruction sequences of length 7 for other nodes. We observe that:
   1. Any node X of size 6 will not be used in the sub-DAG of any other node Y, since SIZE(Y) ≥ 7.
   2. Any node X of size 5 can not be combined with any node Y that is not in X's sub-DAG to generate new nodes. Otherwise we need one instruction to generate the new node from X and Y, 5 instructions to generate X and at least another

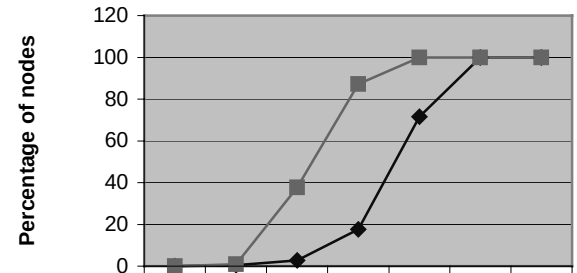



**Figure 2** Distribution of optimal instruction sequences in C3.12

one instruction to generate Y since Y is not in X's sub-DAG. Thus the size of the new nodes generated this way is at least 7.

## 4. Results

By applying the improved full-search algorithm, we were able to find the shortest instruction sequences for C8.0 case. By using the heuristics, we found the nearly shortest instruction sequences for C12.0, C2.10 and C3.12 cases. We list the number of constants that requires an instruction sequence of length L (L = 1, ...7) in table 2. In Figure 2, we show the percentage of constants that require an instruction sequence length ≤ L and the percentage of the corresponding nodes of level ≤ L for case C3.12.

| Length = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Avg |
|---|---|---|---|---|---|---|---|---|
| Case 8.0 | 10 | 38 | 118 | 88 | 0 | 0 | 0 | 3.059 |
| Case 12.0 | 14 | 66 | 433 | 1894 | 1685 | 2 | 0 | 4.2643 |
| Case 2.10 | 15 | 79 | 440 | 1795 | 1748 | 17 | 0 | 4.2767 |
| Case 3.12 | 19 | 117 | 763 | 4856 | 17701 | 9307 | 3 | 5.0767 |

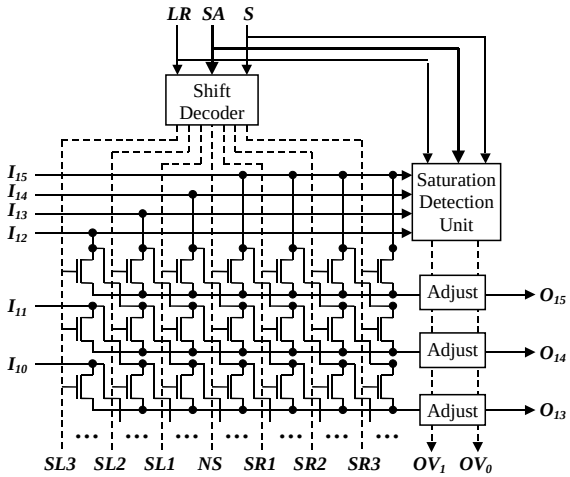
**Table 2** Length of optimal instruction sequences

To compare our results with the results listed in Figure 1 of [4], we made a similar table for the 12.0 in Table 3. In this table, row r lists the first few constants that requires an instruction sequence of length r. We use the bold letters to show the difference between our table and the table in [4].

| r | Least Values of n such that length(n) = r |
|---|---|
| 1 | 2,3,4,5,8,9,16,32,64,128,256,512,1024,2048 |
| 2 | 6,7,10,11,12,13,15,17,18,19,20,21,24,25 |
| 3 | 14,22,23,26,28,29,30,**31**,35,38,39,42,43 |
| 4 | 58,78,86,92,106,110,114,115,116,118,119 |
| 5 | 466,474,618,622,678,683,686,687,691,698 |
| 6 | 3802,**3806** |

**Table 3** Results for C12.0 and comparison to [4]

From the above table, we can see that apart from 31 and 3806, our result is as good as that is given in [4]. Note since we do not allow excess overflow as stated in section 2.3, the instruction length is expected to be longer in our case. For example, as the authors in [4] pointed out, it is impossible to use 2 instructions to implement constant multiplication by 31 if no excess overflow is allowed. We are not exactly sure if that is also the case for 3806, since we do not know the instruction sequence that generates 3806 in [4].




**Shift Decoder**

$SL3 = S \cdot LR \cdot SA_1 \cdot SA_0$ $\quad SR3 = S \cdot \overline{LR} \cdot SA_1 \cdot SA_0$

$SL2 = S \cdot LR \cdot SA_1 \cdot \overline{SA_0}$ $\quad SR2 = S \cdot \overline{LR} \cdot SA_1 \cdot \overline{SA_0}$

$SL1 = S \cdot LR \cdot \overline{SA_1} \cdot SA_0$ $\quad SR1 = S \cdot \overline{LR} \cdot \overline{SA_1} \cdot SA_0$

$NS = \overline{S}$

**Saturation Detection**

$OV_1 = S \cdot LR \cdot \overline{I_{15}} \cdot (I_{14} + SA_1 \cdot I_{13} + SA_1 \cdot SA_0 \cdot I_{12})$

$OV_0 = S \cdot LR \cdot I_{15} \cdot (\overline{I_{14}} + SA_1 \cdot \overline{I_{13}} + SA_1 \cdot SA_0 \cdot \overline{I_{12}})$

**Adjustment Unit**

$ADO_{15} = \overline{OV_1} \cdot (OV_0 + ADI_{15})$

$ADO_i = OV_1 + \overline{OV_0} \cdot ADI_i$ For $i \neq 15$

**Figure 3** Preshift_adder Design

## 5. Preshift_adder

We now look at the hardware cost of a preshift_adder and compare it to a multiplier.

### 5.1 Preshifter design

A preshift_adder is different from a normal adder in that there is a 3-bit preshifter before the adder. Figure 3 shows a preshifter design. In this figure, ***$I_{15}$-$I_0$*** is the 16-bit data input and signals ***S***, ***LR*** and ***$SA_0$-$SA_1$*** are inputs to the ***Shift Decoder*** to determine how many bits $I_{15}$-$I_0$ should be preshifted. When S = '1', do preshift; S= '0', no preshift. When LR = '1', preshift left; LR = '0', preshift right. SA[1-0] record how many bits to shift.

The ***Saturation Detection*** Unit is used to detect if there is any overflow during preshift_left by x bit (x = 1, 2, 3). When the input number is positive ($I_{15}$ = '0'), a positive overflow will occur (***$OV_1$*** = '1') if any of $I_{14}$ … $I_{15\text{-}k}$ is '1'. Similarly, when the input number is negative ($I_{15}$ = '1'), a negative overflow will occur (***$OV_0$*** = '1') if any of $I_{14}$ … $I_{15\text{-}k}$ is '0'. The preshifted inputs will be

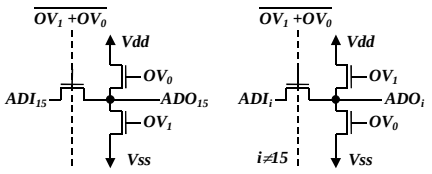



**Figure 4** Adjustment Unit design

| | Preshift_&_ Adder | Multiplier (w. Norm) |
|---|---|---|
| Area ($mm^2$) | 1.77 | 5.9 |
| Normed Area | 1 | 3.3 |
| Delay (ns) | 3.3 | 9.6 |
| Normed Delay | 1 | 2.9 |

**Table 4** Area and Delay Comparison

| IDCT (instrs) | 1.41421 | 2.61313 | 1.08239 | 0.76537 |
|---|---|---|---|---|
| C2.10 | 3 | 5 | 4 | 4 |
| C3.12 | 5 | 6 | 4 | 5 |
| DCT | 0.70711 | 0.54120 | 1.30658 | 0.38268 |
| C2.10 | 4 | 4 | 4 | 3 |
| C3.12 | 4 | 5 | 4 | 5 |

| IDCT(cycles) | Config 1 | Config 2 | Config 3 |
|---|---|---|---|
| C2.10 | 29 | 24 | 12 |
| C3.12 | 29 | 27 | 14 |
| DCT | Config 1 | Config 2 | Config 3 |
| C2.10 | 29 | 24 | 12 |
| C3.12 | 29 | 26 | 13 |

**Table 5** DCT/IDCT Performance Comparison of different configurations

adjusted to the maximum value (0x7fff) in case of a positive overflow and to the minimum value (0x8000) in case of a negative overflow. Figure 3 shows the logic of the Shift Decoder, Saturation Detection Unit and Adjustment Unit.

As we can see from Figure 3, the efficient implementation of adjustment unit is vital to minimizing the total area and delay overhead we add to the shifter. Instead of using the 6-transistor CMOS logic, we exploit the fact that $OV_1$ and $OV_0$ will never be '1' at the same time and design a 3-transistor unit. The design is shown in Figure 4. From Figure 3 and 4, our preshifter design has a size that is roughly 10/7 of a 7-bit shifter and an addition delay of one simple gate.

### 5.2 Area and delay estimation and comparison

We use the data from a 300-MHz 16-bit video processor built in 1993 at NEC with 0.5 $\mu$m BiCMOS technology [8-10] for an estimation [7] of the area and delay of a multiplier verses a preshift_adder. The estimated data are listed in table 4.

## 6. Multiplier versus preshift_adder

Table 4 shows that a subword multiplier with normalization unit takes more than 3 times the area and close to 3 times the delay than the preshift_adder. Furthermore, a subword multiplier can only be used for multiplication while a preshift_adder can be used for both preshift_and_addition and normal addition and subtraction. Thus it is possible to achieve better performance by replacing the multiplier by two to three preshift_adders.

Since there is almost always enough parallelism in the multimedia applications, we can calculate the latency of an application if we know the instruction mix. For example, AAN DCT/IDCT kernel [6] use 5 constant multiplication and 29 additions. The constants for IDCT are 1.41421, 2.61313, 1.41421, 1.08239 and 0.76537. The constants for DCT are 0.70711, 0.54120, 0.70711, 1.30658 and 0.38268. We compare the performance of 3 different configurations. For each 16-bit subword, Config 1 has one multiplier and 1 normal adder, Config 2 has two preshift_adders and Config 3 has four preshift_adders. We found the instruction sequence length for each constant and compared the cycle count of different configurations in Table 5.

Config 3, which has a similar area cost as Config 1, has around 2.4X performance speedup over Config 1 for C2.10 case and 2.1X performance speedup for C3.12 case. Config 2, which has half the area cost as Config 1, has around 1.2X performance speedup over Config 1 for C2.10 case and 1.1X speedup for C3.12 case. Much of the performance gain comes from the fact that the subword multiplier can only be used for multiplication, thus its area is not efficiently used. Thus for subword arithmetic unit design, we could substitute 2 to 4 preshift_addersfor each subword multiplier, except now we have to use the word multiplier to do the subword variable multiplication. However, since variable multiplication is hardly used in multimedia applications, the system with only preshift_adders will win in most cases.

## 7. Conclusion

We have devised a DAG-based algorithm to find the shortest instruction sequence with preshift_and_add and other MAX instructions to perform constant multiplication. We used the full search algorithm to find the shortest instruction sequence for 8-bit integers and achieved an average length of 3.059. By applying some heuristics, we found the close-to-optimum instruction sequences for constant multiplication by 12-bit integers, 2-bit integer with 10-bit fractions and 3-bit integer with 12-bit fractions. The average lengths are 4.2643, 4.2767, 5.07673 instructions respectively.

We also did a preshifter design and compared the area and performance of our preshift_adder to a 16×16 bit multiplier. We showed that the multiplier takes more than 3 times the area and close to 3 times the latency. By replacing the multiplier by our preshift_adders, we have demonstrated 2× performance speedup with similar area for representative kernels like AAN DCT/IDCT.

**References:**


[1] Ruby Lee, "Accelerating Multimedia with Enhanced Microprocessors", *IEEE Micro*, Vol. 15, No. 2, Apr. 1995, pp.22-32

[2] Ruby Lee, "Subword Parallelism with MAX-2", *IEEE Micro*, Vol. 16, No. 4, Aug. 1996, pp.51-59

[3] Gerry Kane, *PA-RISC 2.0 Architecture*, ISBN 0-13-182734-0, Prentice Hall, 1996

[4] Daniel Magenheimer etc, "Integer Multiplication and Division on the HP Precision Architecture", *IEEE Transcations on Computers*, Vol. 37, No. 8, Aug. 1988

[5] D. Knuth, *The Art of Computer Programming*, Vol. 2, *seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1981, pp.444-446

[6] Arai Yukihiro, Agui Takeshi and Nakajima Masayuki, "A Fast DCT-SQ Scheme for Images", Transactions for the IEICE, Vol. E 71, No.11, Nov. 1998

[7] Zhen Luo and Ruby Lee, "Subword Constant Multiply by Preshift_and_add", Princeton University Department of Electrical Engineering Technical Report #CE-L99-003.

[8] Toshiaki Inoue etc., "A 300-MHz 16-b BiCMOS Video Signal Processor", *IEEE Journal of Solid-State Circuits*, Vol. 28, No. 12, Dec. 1993, pp.1321-1328

[9] Masahiro Nomura etc., "A 300-MHz 16-b BiCMOS Digital Signal Processor Core LSI", *IEEE Journal of Solid-State Circuits*, Vol. 29, No. 3, Mar. 1994, pp.290-296

[10] Kazumasa Suzuki etc., "A 2.4-ns, 16-BIT, 0.5 $\mu$m CMOS Arithmetic Logic Unit for Micorprogrammable Video Signal Processors LSIs", *Proc. Of IEEE Custom Integrated Circuits Conference*, 1993, pp.12.4.1-12.4.5