S-38.220
Postgraduate Course on Signal Processing in Communications,
FALL - 99

Numeric strength reduction

Giedrius ZAVADSKIS

Email: giedrzav@cc.hut.fi

Date: 15.11.1999

ABSTRACT

The following paper gives an introduction to numerical strength reduction. The latter is the last step in algorithmic strength reduction – the optimization of binary arithmetic's in DSP computations. The most time and power consuming operation in DSP is multiplication. Without regard of being implemented in software or hardware, multiplication is usually divided into simpler shift and add operation. Number strength reduction technique addresses the simplification of this operation. The multiplication optimization method known as subexpression elimination is presented in the paper. Also a concept of canonic signed digit is introduced.

1. INTRODUCTION

Most DSP algorithms are combination of delays, adders and multipliers. The latter are composed of shift-registers and adders that are distributed in time or in space. The numerical strength reduction method uses tricks to share the results of addition between several multiplications.

2. CANONIC SIGNED DIGIT

Consider a binary number 1111. Multiplication by this number requires 3 shifts and 3 additions. However this number may be expressed as 1000 – 0001. In this case 3 shifts and one subtraction are required. The number may be expressed in a different way:

$$1000\bar{1}$$

Here the minus above the last digit signifies subtraction.
The above method of displaying numbers is known as Canonic signed digit (CSD).
In general CSD is a set of $\{-1, 0, 1\}$. The property of CSD is that represents binary numbers with minimum count of non-zero digits. Since only non-zero elements create additions in multiplication operation, CSD is an efficient way of number representation for multiplication operations.
However CSD arithmetical units are more complicate that those for ordinary binary applications.

3. SUBEXPRESSION ELIMINATION

When one variable is multiplied by a set of constants, usually these constants may be expressed one by another by simple arithmetical operations. Therefore expressing constant through other ones lead to reduced number of operations. The main method for simplification of constant multiplication is Subexpression elimination. This is a numerical transformation of the constant multiplications that can lead to efficient hardware implementation in terms of area, speed and power.
The main point of subexpression elimination is to find common structures in set of constant and to reuse them. Let ’s look at one example.
Consider that a variable must be multiplied by a= 13 and b=27. Then by converting the constants as binary numbers, as shown in table 3-1, we can see that 1st and 4th bits are the same for both constants. Therefore multiplication with common term 1001 has to be done once.

*Table 3-1: **E**xample 3-1. [1]*

Constant	Decimal	Binary
a	13	001101
b	27	011011

Furthermore be the common term happens in constant *b* twice: once with the shift 1 and another with the no shift, i.e.:

$b \text{ x } x = (x \text{ x } 1001) << 1 + (x \text{ x } 1001)$

here <<1 means one shift in the register.

In the similar manner multiplication *a* becomes: $a \times x = x \times 1001 + x \times 0100$
If the above given algorithm is used, the required number of shifts is 3 and required number of additions is also 3.
If conventional multiplication is used then the total number of addition is 5 shift and 5 adds.

As and alternative, just by inspecting constants *a* and *b* in either decimal or binary form, that b=2a +1. This scheme also gives total number of 3 additions and 3 shifts.

The basic algorithm consists of 5 steps.
Step 1. All the constants are expressed in binary form. Signed, unsigned and CSD representation are possible.
Step 2. The non-zero bit matches are determined between all constant pairs.
Step 3. The best match, with the maximum number of identical bits, is chosen.
Step 4. The identical bits or so-called 'redundancy' are removed from all constants.
Step 5. Continue with the step 2, for the new set of constants, until no improvement is achieved.

An example of the algorithm follows:
Consider variable multiplication by a set of constants {237, 182, 93}.
The constants are expressed in binary as it is shown in table 3-2. The number of bit matches is also shown in the table. We can see that the greatest match is between constants *a* and *c*. Therefore this match '01001101' is being removed from the constants *a* and *c*.

Table 3-2: Example 3-2. [1]

Constant	Decimal	Binary	Match		
a	237	11101101	-	3	4
b	182	10110110	3	-	2
c	93	01011101	4	2	-

Then a'=11101101-01001101 = 10100000 and
c' = 01011101-01001101 = 00010000

These results are displayed in table 3-3. By inspecting the table one can see, that there are still 2 matches left among a' and *b*. This match '10100000' is remove in the second iteration.

Table 3-3: Example 3-2. First iteration

Constant	Binary	Match		
a'	10100000	-	2	0
b	10110110	2	-	1
c'	00010000	0	1	-
redun(a,c)	01001101	-	-	-

After second iteration the constant a'' and b' look like this:

a''=10100000-10100000 = 00000000 and

b ’ = 10110110-10100000 = 00010110

The results are summarized in Table 3-4. There is one match between b and c but eventually it will not decrease number of operations, therefore the end of subexpression elimination has been reached. All three constants may be expressed in the following way:

a= redun(a,c) + redun(a,b)
b= redun(a,b) + 00010110
c= redun(a,c) + 00010000

Table 3-4: Example 3-2. Second iteration

Constant	Binary
a ’ ’	00000000
b ’	00010110
c'	00010000
redun(a,c)	01001101
redun(a,b)	10100000

The complexity analysis of the problem is shown in table 3-5.

Table 3-4: Example 3-2. Complexity analysis

Constant	without SE	with SE
a	5 shifts + 5 adds	1 add
b	5 shifts + 4 adds	3 shifts + 3 add
c	4 shifts + 4 adds	1 shift + 1 add
redun(a,c)	-	3 shifts + 3 adds
redun(a,b)	-	2 shifts + 1 add
Total	14 shifts + 13 adds	9 shifts + 9 adds

As a result of the application of subexpression elimination 5 shifts and 4 additions have been saved, what is over ¼.

4. LINEAR TRANSFORMATIONS

Consider a general form of linear transformation that is done by a matrix multiplication:

$$\mathbf{y} = \mathbf{T} * \mathbf{x}$$

Here **y** is a output vector of length *m*, **x** is an input vector of length *n*, while **T** is a matrix of the seize *m* x *n*. If **T** is a matrix of constant then each member of output vector may be expressed by the following formula

$$y_i = \sum_{j=1}^{n} t_{ij} x_j, \qquad i = 1,\dots,m$$

Since it is constant multiplication, subexpression elimination can be applied. In this occasion it consists of 3 steps.
Step 1 is the algorithm described in previous chapter applied to constants in the columns of the matrix **T**.

In the Step 2 unique products with respect to x_1, x_2,…,x_n are determined. The output values of y_1, y_2,…,y_m are constructed as the sums of these products.
Step 3 is devoted for minimization of addition number for obtaining output values. The algorithm is similar to the one discussed in chapter 3.
An example of the substructure elimination in linear transformations is discussed further.
Example 4-1. Consider the following transform matrix:

$$\mathbf{T} = \begin{bmatrix} 7 & 8 & 2 & 13 \\ 12 & 11 & 7 & 13 \\ 5 & 8 & 2 & 15 \\ 7 & 11 & 7 & 11 \end{bmatrix}$$

Step 1. Each element in the column vector **x** of variable is multiplied by the constants in the respective column of the transform matrix **T**. Therefore the subexpression elimination has to be done only on the transform matrix columns. The result is shown in table 5-1.

Table 4-1: Subexpressions required for each column.

Column 1	Column 2	Column 3	Column 4
0101 *0010* *1100*	1000 1011	0010 0111	1001 0100 0010
* x_1	* x_2	* x_3	* x_4

Step 2. Constants of each column are multiplied by the variable x from respective row. The unique multiplications are the following:

p_1=0101 x x_1;	p_2=0010 x x_1;	p_3=1100 x x_1;
p_4=1000 x x_2;	p_5=1010 x x_2;	
p_6=0010 x x_3;	p_7=0111 x x_3;	
p_8=1001 x x_4;	p_9=0100 x x_4;	p_{10}=0010 x x_4;

Then the output value become:

$y_1 = p_1 + p_2 + p_4 + p_6 + p_8 + p_9$;
$y_2 = p_3 + p_5 + p_7 + p_8 + p_9$;
$y_3 = p_1 + p_4 + p_6 + p_8 + p_9 + p_{10}$;
$y_4 = p_1 + p_2 + p_5 + p_7 + p_8 + p_{10}$;

Step 3. The sums given above are simplified in this step. In order to simplify the operation each output term is express in the binary format, where positions in the word mean the presence or absence of the multiplicative term.

y_1 = 1101010110; y_2 = 0010101110;
y_3 = 1001010111; y_4 = 1100101101;

From the above expression it may be seen, that there are 5 matches between y_1 and y_3, while y_2 and y_4 have 3 bits in common. After evaluating these matches, output terms may be expressed:

$y_1 = p_2 + (p_1 + p_4 + p_6 + p_8 + p_9)$;
$y_2 = p_3 + p_9 + (p_5 + p_7 + p_8)$;
$y_3 = (p_1 + p_4 + p_6 + p_8 + p_9) + p_{10}$;

$y_4 = p_1 + p_2 + p_{10} + (p_5 + p_7 + p_8)$;

5. POLYNOMIAL EVALUATION

Another area for application of subexpression elimination is polynomial multiplication. Consider the following polynomial:

$$x^{11} + x^9 + x^4 + x^2 + x$$

The straightforward evaluation of the polynomial will require:

10 + 8 +3 + 1 = 22 multiplications

However term x^4 may be expressed as $x^4 = x^2 * x^2$. Usage of this equation leads to saving of 2 multiplications.
A more general technique is to express exponent value in the binary format, as the following:

11 = 1011; 9=1001; 4=0100; 2=0010; 1=0001.

Then these terms can be expressed as a multiplication of x^{n^2}, where n denotes position of bit in exponent expressed in binary format. For x^{n^2} the following equality holds:

$$x^{n^2} = x^{(n-1)^2} * x^{(n-1)^2},$$

i.e. $x^2 = x * x$; $x^4 = x^2 * x^2$; $x^8 = x^4 * x^4$.

Then the original polynomial can be rewritten in this way:

$$(x^8 * x) * x^2 + (x^8 * x) + x^4 + x^2 + x$$

If this scheme is used, the number of multiplications have been reduced to 5!

6. SUBEXPRESSION SHARING IN DIGITAL FILTERS

Consider FIR filter implementation shown in figure 5-1. This type of filter structure is known as transposed direct-form or data-broadcast. The property of this filter is that input value is fed in multipliers at the same instant. Therefore it is possible to apply subexpression elimination on this instance.

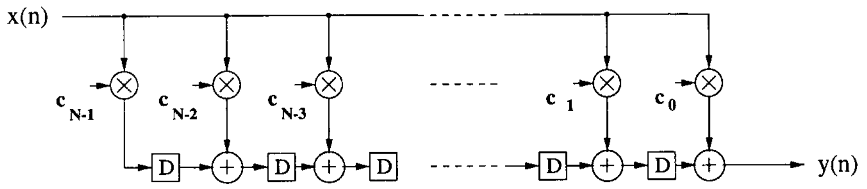

Figure 6-1: Data-broadcast structure of FIR filter [1]

The general algorithm is composed of 6 steps.
In Step 1 a table is constructed. The rows in the table denote delays, while the filter coefficients digits are distributed over the columns. In the column 0 the most significant bits are entered. Only non-zero digits are entered into the table.
In the same step 'best' subexpressions of size 2 are found. The best subexpression is chosen by cost function, that is mostly influenced by number of occurances.
In Step 2 each occurrence of subexpression is removed and replaced by value 2 or – 2 in the upper-left cell of the subexpression. Positive value means addition, while the negative value means subtraction in the higher level of subexpression elimination.
Step 3 is devoted for making of records subexpressions eliminated.
At Step 4 it is checked if any subexpression can be eliminated. If so, it is returned to Step 2, otherwise it continues with Step 5.
In Step 5 the complete definition of the filter is written down.
There might occur subexpression definitions with negative shifts. These shifts are eliminated in Step 6.
To explain the algorithm in detail an example is considered.
Example 6-1. There is a 4-tap FIR filter given with the following equation:

$$y(n) = \bar{1}.01010000010 * x(n) + 0.\bar{1}000\bar{1}0\bar{1}0\bar{1}0\bar{1} * x(n-1)$$

$$+ 0.\bar{1}0010000010 * x(n-2) + 1.0000010\bar{1}000 * x(n-4)$$

Step 1. These values are written down in the table 6-1. The matches are also shown.

Table 6-1: Iteration 1. [1]

0	1	2	3	4	5	6	7	8	9	10	11
-1		1		1						1	
	-1				-1		-1		-1		-1
	-1			1						1	
1						1		-1			

Step 2. The subexpression are removed, as shown in table 6-2. There is also shown the subexpressions that will be removed in the next iteration.

Table 6-2: Iteration 1 with subexpression eliminated. [1]

0	1	2	3	4	5	6	7	8	9	10	11
-1		2		1						2	
					-2		-1				-2
	-2										
						1		-1			

Step 3. The subexpression removed is recorded:

$x2 = x1 - x1[-1] >> (-1)$

Negative shift may happen in this step. They are removed in the further steps

Step 4. Another iteration is done, and it is returned to step 2. These are the results of the second iteration are shown below.

Table 6-3: Results of iteration 2. [1]

0	1	2	3	4	5	6	7	8	9	10	11
-1		3								2	
					-3						-2
	-2										
						1		-1			

$x3 = x2 + x1 >> 2$

Step 5. The filter definition is written down:

$x2 = x1 - x1[-1] >> (-1)$
$x3 = x2 + x1 >> 2$
$y = -x1 + x3 >> 2 + x2 >> 10 \quad - x3[-1] >> 5 - x2[-1] >> 11$
$-x2[-2] >> 1 \quad + x1[-3] >> 6 - x1[-3] >> 8$

Step 6. Negative shifts have to be removed. The variables that have the negative shift are flipped to create positive shift:

$x2 = x1 - x1[-1] >> (-1) \quad \Rightarrow \quad x2 = x1 >> 1 - x1[-1]$

Then all the other equations are modified due to this change:

$x3 = x2 + x1 >> 3$
$y = -x1 + x3 >> 1 + x2 >> 9 \quad - x3[-1] >> 4 - x2[-1] >> 10$
$-x2[-2] \quad + x1[-3] >> 6 - x1[-3] >> 8$

The filter structure is shown in fig.6-2.

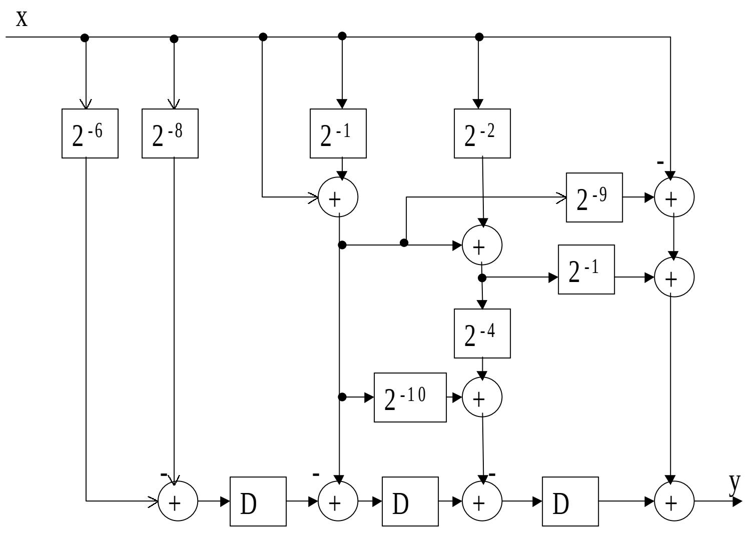

Figure 6-2: Structure of FIR filter from example 6-1[1]

7. TWO MOST COMMON SUBEXPRESSIONS IN CSD REPRESENTATION

ince CSD have the minimum number of non-zero bits they have an advantage in multiplications done by digital filters. It can be shown that statistically the most common subexpressions are x-x>>2 and x+x>>2. These subexpressions correspond to bit patterns $10\bar{1}$ and 101.

Asymptotically CSD of the length W may be divided into W/18+1 pairs of $10\bar{1}$, W/18+1 pairs of 101, and W/9+1 isolated positive – 1 and negative – $\bar{1}$ bits.

Therefore any FIR filter can be built only out of subexpressions $10\bar{1}$, 101 and 1. This allow implementation of the digital filter with three buses as shown in fig.7.1.

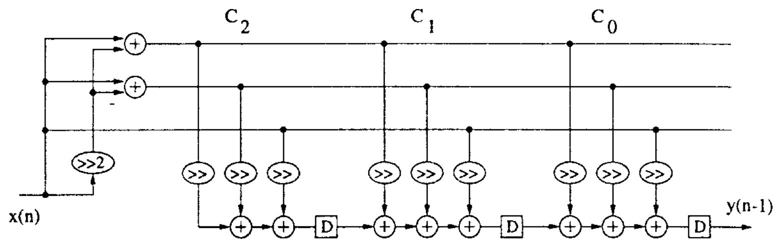

Figure 7-1: FIR filter implementation for terms $10\bar{1}$ and 101[1]

In this structure $10\bar{1}$ and 101 precomputed beforehand. These values are put to 2 buses of total 3. The third is devoted for *x* value. To get value to be stored in the delay element 2 addition are required. The input to adders are shifted appropriately.

8. CONCLUSIONS

The paper considers numerical strength reduction algorithms – the last step in algorithmic strength reduction processes. The main method for this is subexpression elimination. The essence of this method is to find the common structure, compute them once and reuse at each next occurrence. Application in linear transforms, polynomial evaluation and FIR data-broadcast structures have been considered.

REFERENCES

[1] K.K. Parhi, VLSI Digital Signal Processing: NUMERICAL STRENGTH REDUCTION, Chap 15, J. Wiley &Sons, 1999

[2] M.Potkonjak, M.B Srivastava and A.P.Chandrakasan, "Multiple constant multiplication: efficient and versatile framework and algorithms for exploring common subexpression elimination," IEEE trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 2., pp. 151-165, Feb. 1996

[3] A.Chatterjee, R.K. Roy, and M.A.d'Abreu, "Greedy hardware optimization for linear digital circuits using number splitting and refactoriztion," IEEE Trans. on VLSI systems, vol.1., no.4, pp. 423-431, Dec. 1993