TSTE18 Digital Arithmetic
Seminar 5

Oscar Gustafsson

Multiplication

- Partial product generation
- Partial product accumulation
- Final adder

- **Squarers**
- **Fast multiplication**
- **Constant multiplication**

Squaring

- Squaring an unsigned binary number X can be written as

$$Z = X^2 = XX = \sum_{i=-L}^{M-1} x_i 2^i \sum_{j=-L}^{M-1} x_j 2^j = \sum_{i=-L}^{M-1} \sum_{j=-L}^{M-1} x_i x_j 2^{i+j}$$

- Consider a six-bit squarer

					x_5	x_4	x_3	x_2	x_1	x_0
				$\times$	x_5	x_4	x_3	x_2	x_1	x_0
					x_0x_5	x_0x_4	x_0x_3	x_0x_2	x_0x_1	x_0x_0
				x_1x_5	x_1x_4	x_1x_3	x_1x_2	x_1x_1	x_1x_0	
			x_2x_5	x_2x_4	x_2x_3	x_2x_2	x_2x_1	x_2x_0		
		x_3x_5	x_3x_4	x_3x_3	x_3x_2	x_3x_1	x_3x_0			
	x_4x_5	x_4x_4	x_4x_3	x_4x_2	x_4x_1	x_4x_0				
x_5x_5	x_5x_4	x_5x_3	x_5x_2	x_5x_1	x_5x_0					

- Every partial product with $i \neq j$ appears twice in the same column

Squaring

- Use this to simplify the partial product array

					x_5	x_4	x_3	x_2	x_1	x_0
				$\times$	x_5	x_4	x_3	x_2	x_1	x_0
x_4x_5	x_3x_5	x_2x_5	x_1x_5	x_0x_5	x_0x_4	x_0x_3	x_0x_2	x_0x_1		x_0
x_5		x_3x_4	x_2x_4	x_1x_4	x_1x_3	x_1x_2		x_1		
		x_4		x_2x_3		x_2				
				x_3						

- This is called a folded squarer
- Approximately half of the partial products compared to a general multiplier
- Middle column contains most partial products

Squaring

- Note that x_3 and x_2x_3 are in the same column
- The value of this expression will be

x_3	x_2	Value	Binary	
0	0	0	0	0
0	1	0	0	0
1	0	1	0	1
1	1	2	1	0

- Replace with x_2x_3 in the next higher column and $\overline{x_2}x_3$ in the same column

					x_5	x_4	x_3	x_2	x_1	x_0
				$\times$	x_5	x_4	x_3	x_2	x_1	x_0
x_4x_5	x_3x_5	x_2x_5	x_1x_5	x_0x_5	x_0x_4	x_0x_3	x_0x_2	$\overline{x_0}x_1$		x_0
x_5	x_3x_4	$\overline{x_3}x_4$	x_2x_4	x_1x_4	x_1x_3	$\overline{x_1}x_2$	x_0x_1			
			x_2x_3	$\overline{x_2}x_3$	x_1x_2					

Squaring

- Can also use signed representations
- Two's complement representation with modified Baugh-Wooley

						$-x_5$	x_4	x_3	x_2	x_1	x_0
					$\times$	$-x_5$	x_4	x_3	x_2	x_1	x_0
1	$\overline{x_4x_5}$	$\overline{x_3x_5}$	$\overline{x_2x_5}$	$\overline{x_1x_5}$	$\overline{x_0x_5}$	x_0x_4	x_0x_3	x_0x_2	x_0x_1		x_0
	x_5		x_3x_4	x_2x_4	x_1x_4	x_1x_3	x_1x_2		x_1		
			x_4		x_2x_3		x_2				
					x_3						
					1						

- Even more partial products in the middle column

Squaring

- Rewrite in a similar way and include the 1

x_3	x_2	Value	Binary	
0	0	1	0	1
0	1	1	0	1
1	0	2	1	0
1	1	3	1	1

- Now, place x_3 in the next higher column and $x_2 + \overline{x_3}$ in the same

						$-x_5$	x_4	x_3	x_2	x_1	x_0
					$\times$	$-x_5$	x_4	x_3	x_2	x_1	x_0
1	$\overline{x_4x_5}$	$\overline{x_3x_5}$	$\overline{x_2x_5}$	$\overline{x_1x_5}$	$\overline{x_0x_5}$	x_0x_4	x_0x_3	x_0x_2	x_0x_1		x_0
	x_5	x_3x_4	$\overline{x_3}x_4$	x_2x_4	x_1x_4	x_1x_3	x_1x_2		x_1		
				x_3	$x_2 + \overline{x_3}$		x_2				

- Several other approaches to merge a number of partial products before accumulating them has been proposed

Multiplication through squaring

- It is possible to multiply through squaring

$$(a+b)^2 = a^2 + 2ab + b^2 \Rightarrow ab = \frac{(a+b)^2 - a^2 - b^2}{2} \quad (1)$$

- More suitable for table-based implementation than logic-based
- For a table-base realization assuming input wordlength N

Multiplier-based		Squarer-based	
Table	Table size	Table	Table size
ab	2^{2N}	$(a+b)^2$	2^{N+1}
		a^2	2^N
		b^2	2^N
Total	2^{2N}	Total	2^{N+2}

- Additional cost is one addition and two subtractions

Multiplication through squaring

- Alternatively

$$(a+b)^2-(a-b)^2 = 4ab \Rightarrow ab = \frac{(a+b)^2-(a-b)^2}{4} \quad (2)$$

- For a table-base realization assuming input wordlength N

Multiplier-based		Squarer-based	
Table	Table size	Table	Table size
ab	2^{2N}	$(a+b)^2$	2^{N+1}
		$(a-b)^2$	2^{N+1}
Total	2^{2N}	Total	2^{N+2}

- Additional cost is one addition and two subtractions

Fast multiplication

- Consider a polynomial multiplication $(A_0 + A_1X)(B_0 + B_1X)$
- (Motivation: $X = j \Rightarrow$ complex multiplication, $X = 2^{\frac{W}{2}} \Rightarrow$ long multiplication)
- Normally, four multiplications are required: A_0B_0, A_0B_1, A_1B_0, and A_1B_1
- However, three are enough

Fast multiplication

- Let the result of the multiplication be $(A_0 + A_1X)(B_0 + B_1X) = C_0 + C_1X + C_2X^2$
- The unisolvence theorem states that an N-term polynomial is uniquely defined by its values in N points
- Evaluate the polynomial in three points, e.g., $X = \{0, 1, \infty\}$

$$\begin{aligned} A_0B_0 &= C_0 \\ (A_0+A_1)(B_0+B_1) &= C_0 + C_1 + C_2 \\ A_1B_1 &= C_2 \end{aligned}$$

- Or on matrix form

$$\begin{bmatrix} B_0 & 0 & 0 \\ 0 & B_0+B_1 & 0 \\ 0 & 0 & B_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix}$$

Fast multiplication

- The result of the polynomial multiplication is C_0, C_1, and C_2, so solve for those:

$$\begin{aligned} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} B_0 & 0 & 0 \\ 0 & B_0+B_1 & 0 \\ 0 & 0 & B_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_0 & 0 & 0 \\ 0 & B_0+B_1 & 0 \\ 0 & 0 & B_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \end{aligned}$$

- So

$$\begin{aligned} C_0 &= A_0B_0 \\ C_2 &= A_1B_1 \\ C_1 &= (A_0+A_1)(B_0+B_1) - C_0 - C_2 \end{aligned}$$

Fast multiplication

- Higher-order polynomials can be used
- Karatsuba, Cook-Toom, Gauss, ...
- Evaluating in different points gives different equations
- Evaluating on the unit circle $\Rightarrow$ DFT/FFT
  - Efficient for high-order polynomials
  - Main complexity in matrix operations rather than multiplications
- Applications in FIR filters

Fast multiplication

- Fewer multiplications but more additions/subtractions
- Effectiveness determined based on relative cost
- Figures from GNU Multiple Precision (GMP) Library

Number of words required to use algorithm for long multiplication

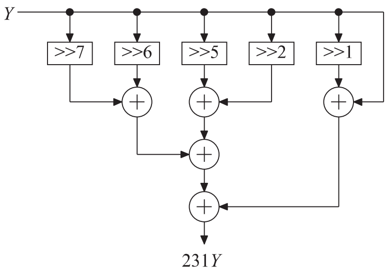
Algorithm	ARM A15	Core 2	Core i7
2×2	23	23	26
3×3	90	65	89
4×4	262	179	214
7×6	351	268	327
9×8	557	357	466
FFT	5760	4736	6784

Multiplication by a constant

- Each partial product row is either the input data or zero

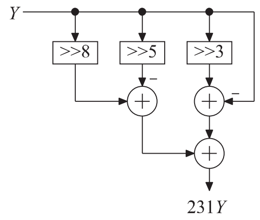
$$Z = XY = Y \sum_{i=1}^{W} x_i 2^{-i} = \sum_{i=1}^{W} Y x_i 2^{-i} \tag{3}$$

- To add W words, $W - 1$ adders are required
- If the coefficient X is known beforehand it is not required to use $W - 1$ adders
- Example: $X = 231$



Multiplication by a constant

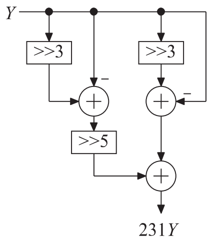
- Is there a way to find a representation with fewer non-zero positions?
- MSD/CSD is a good choice here



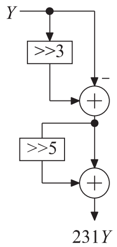
- No major difference between adders and subtracters
- Coefficients are not explicitly represented in CSD, the CSD representation rather determines the structure
- Minimum number of non-zero digits equal to minimum number of adders?

Multiplication by a constant

- No!
- Rewrite as

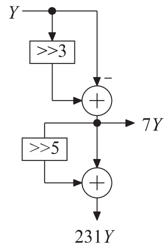


- The two first subtracters compute the same result
- Better only do it once

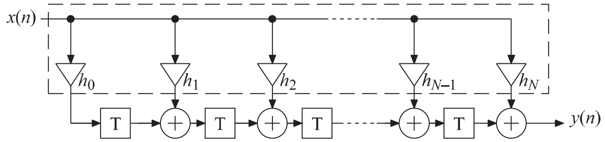


Multiplication by a constant

- A careful inspection gives that we have a free multiplication by 7

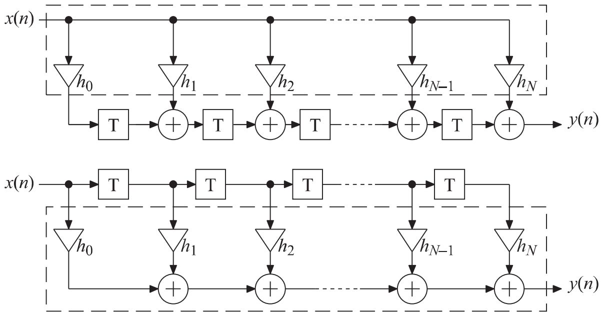


- Can this be useful?
- Transposed direct form FIR filters



Multiplication by a constant

- In fact transposing gives the direct form FIR filter



- The problem of multiplying a single input data with several constant coefficients is known as multiple constant multiplication (MCM)
- Efficient technique to realize constant multiplications using as few adders and subtracters as possible
- Can easily be generalized to linear transforms as well (matrix-vector multiplications)

Multiplication by a constant

- The multiple constant multiplication problem:
  - Given a set of coefficients, S, find a realization using as few additions and subtractions as possible such that the input is multiplied with all coefficients in S
- Two main techniques:
  - Sub-expression sharing (easy to solve hard problems, representation dependent)
  - Adder graphs (hard to solve hard problems, representation independent)

Sub-expression sharing

- Given a representation, the result is computed by shifting and adding/subtracting the input
- Assuming there are N terms, $N-1$ adders are required
- Both the terms and the adders can be ordered arbitrarily
- **If we order them in a clever way, it is possible to reduce the number of adders**

Sub-expression sharing

- The concept of generalized sub-expression sharing is like
  1. Represent each required result as a sum of signed-digits in a given representation
     - CSD appears to be a good choice, but there will in general be better choices (which are hard to find)
  2. For each required result find and count possible sub-expressions
     - Sub-expression characterized by the origin of the two terms, the difference in the non-zeros position and if the non-zeros have the same or opposite signs, i.e., the sub-expressions $100\bar{1}$ and $\bar{1}001$ are the same
  3. If there are common sub-expressions, select one to replace and replace instances of it by introducing a new symbol in place of the sub-expression
     - Common approach is to select the most frequent sub-expression and replace all instances
     - Greedy optimization, so not always the globally best choice
  4. If there were sub-expressions replaced, go to Step 2 otherwise the algorithm is done.

Sub-expression sharing example

- Multiply X_1 with 13 and 21, i.e., compute $\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} 13 \\ 21 \end{bmatrix} [X_1]$
- Select representation: Binary $\Rightarrow 13 = (1101)_2, 21 = (10101)_2$ **(four adders required)**
  Hence $Y_1 = \sum_i a_{i,1} 2^i X_1$ and $Y_2 = \sum_i a_{i,2} 2^i X_1$
- Count sub-expressions:
  - For Y_1: $\mathbf{11}01$, $\mathbf{1}1\mathbf{0}1$, $1\mathbf{101}$
  - For Y_2: $\mathbf{101}01$, $\mathbf{1}010\mathbf{1}$, $10\mathbf{101}$
  - Frequency

Sub-expression	Frequency
$11 \Leftrightarrow 2X_1 + X_1 \Leftrightarrow 3X_1$	1
$101 \Leftrightarrow 4X_1 + X_1 \Leftrightarrow 5X_1$	3
$1001 \Leftrightarrow 8X_1 + X_1 \Leftrightarrow 9X_1$	1
$10001 \Leftrightarrow 16X_1 + X_1 \Leftrightarrow 17X_1$	1

Sub-expression sharing example

- Select sub-expression and replace:
  - Most frequent one is $101 \Leftrightarrow 4X_1 + X_1 \Leftrightarrow 5X_1$
  - Define $X_2 = 4X_1 + X_1$ **(one adder)**
  - New formulation of the expression

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} 13 \\ 21 \end{bmatrix} [X_1] = \begin{bmatrix} 8 & 1 \\ 16 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

  so

$$Y_1 = \sum_i \sum_j a_{i,j,1} 2^i X_j$$

  and

$$Y_2 = \sum_i \sum_j a_{i,j,2} 2^i X_j$$

  or in general

$$Y_k = \sum_i \sum_j a_{i,j,k} 2^i X_j$$

  - Note that we only could replace two of the expected three sub-expressions

Sub-expression sharing example

- Count sub-expressions:
  - Slightly more complicated to illustrate, but as each result now consists of two terms, there is only one sub-expression for each

Sub-expression	Frequency
$8X_1 + X_2 \Leftrightarrow 13X_1$	1
$X_1 + 4X_2 \Leftrightarrow 21X_1$	1

- No more savings are obtainable, so we can just compute the remaining sub-expressions to obtain the final result **(two adders)**
- Three adders are required in total, so one is saved

Sub-expression sharing

- Problems faced:
  - How to select a suitable representation?: $21 = 10101$ and $7 = 100\bar{1}$ has no common expressions, but $3 \cdot 7 = 10\bar{0}\bar{1}0 + 100\bar{1} = 110\bar{1}\bar{1} = 10101 = 21$
  - How to detect collisions, e.g., how many usable sub-expressions in 101010101?
  - Some sub-expressions are "hidden", i.e., there is no suitable representation that will reveal it
  - Which sub-expression to select? Frequency is good, but the number of cascaded adders will increase the delay (and power consumption because of increased switching)
  - Which sub-expressions to replace?
- Typically, we will have to make heuristic decisions for most of these issues
- Still: a well defined way to obtain a good solution

Adder graphs

- Look at the problem from a different perspective
  - In an (well designed) FIR filter, the tail coefficients are often small
  - Two-term sub-expressions which are also coefficients, will eventually be computed although they may not be the most frequent for the initial iterations
  - Makes sense to compute them initially and benefit from them in later iterations
- Only consider odd positive integers as even and fractional numbers can be obtained by shifting
- If a negative coefficient is required, it can often be solved by replacing a subsequent addition with a subtraction or vice versa

Adder graph algorithm

- Form a set R of the coefficients in S by taking the absolute value and shifting the coefficients to be odd integers
- Form a set of already computed coefficients, A, initially consisting of the coefficient 1
- As long as there are coefficients in R
  - Compute all possible partial results that can be obtained by shifting and adding the coefficients in A
    $$C = \left|2^i a \pm 2^j b\right|$$
    where a and b are coefficients in A
  - If any of the coefficients in R is present in C, it can be computed using a single adder, which clearly is the optimal
  - Move those coefficients from R to A and iterate
- If none of the coefficients in R is present in C, we still need to pick a coefficient from C such that the algorithm can converge later on
- This is the hard part and several heuristics have been proposed
- Note that this approach is totally independent of a bit-level representation of the coefficients

Adder graph algorithm example

- Coefficients $S = \{6, -21, 37\}$
- First, create $R = \{3, 21, 37\}$
- In the first iteration $C = \{\mathbf{3}, 5, 7, 9, 15, 17, 31, 33, 63, 65, \dots\}$
- 3 is in C, so move it to A: $R = \{21, 37\},\ A = \{1, 3\}$
- Next iteration gives $C = \{5, 7, 9, 11, 13, 15, 17, 19, \mathbf{21}, 23, 25, 27, 29, 31, 33, 35, 45, 47, \dots\}$
- This gives $R = \{37\},\ A = \{1, 3, 21\}$
- Now $C = \{5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, \mathbf{37}, 39, \dots\}$
- And $R = \emptyset,\ A = \{1, 3, 21, 37\}$ so the algorithm has converged

Higher dimension problems

- In the sub-expression sharing problem we had

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

- This means that there is no difference between a sub-expression and an input so we can start with multiple inputs
- Useful for constant matrix-vector multiplications such as linear transforms, e.g., DCTs
- Each row can be expressed as

$$Y_k = \sum_i \sum_j a_{i,j,k} 2^i X_j$$

- Also possible to introduce more shift dimensions, e.g., time, although shifts in time can hardly be argueed to be as cheap as arithmetic shifts

$$Y_k = \sum_i \sum_j \sum_l a_{i,j,k,l} 2^i z^{-l} X_j$$

Higher dimension problems

- Using this with sub-expression sharing is actually rather straightforward
- Each result (in this general case a multiple input FIR filter) can be expressed as

$$Y_k = \sum_i \sum_j \sum_l a_{i,j,k,l} 2^i z^{-l} X_j$$

- Each two non-zero $a_{i,j,k,l}$ terms forms a possible sub-expression
- The same concepts are possible to use for the adder graph approach using the following two modifications
  - For multiple inputs, the coefficients in R and A are now vectors, with A initialized as the rows from an identity matrix
  - For shift in time, the possible results are computed as

$$C = \left| 2^i z^{-k} a \pm 2^j z^{-l} b \right|$$

- However, there are typically quite a number of partial results to be determined before a matrix row or an FIR filter transfer function is obtained in C making it very challenging