Application Specific Integrated Circuits for Digital Signal Processing
Lecture 11

Oscar Gustafsson

Today's topics

- Multiplication
- Distributed arithmetic
- Constant multiplication
- Alternative number representations
  - Floating-point
  - Logarithmic number systems
  - Residue number systems

(If you enjoy this lecture and the previous, take the TSTE18 Digital Arithmetic course)

Multiplication

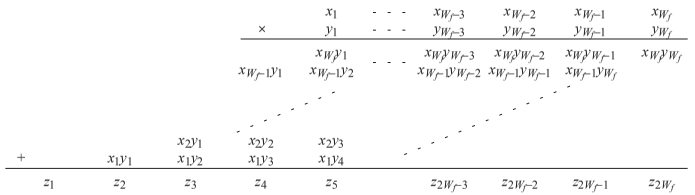
- Multiplication can typically be separated into three sub-problems
  - Generating partial products
  - Adding the partial products using a redundant number representation
  - Converting the redundant number representation to non-redundant form (commonly using a vector merging adder, VMA)

Partial product generation

- Multiplying two unsigned binary numbers X and Y can be written as

$$Z = XY = \sum_{i=1}^{W} x_i 2^{-i} \sum_{j=1}^{W} y_j 2^{-j} = \sum_{i=1}^{W} \sum_{j=1}^{W} x_i y_j 2^{-i-j}$$

- The resulting partial product array is

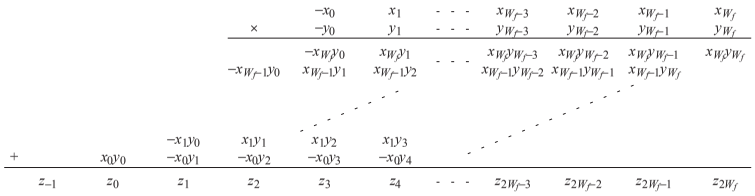


Partial product generation

- For two's complement we can, by using that the sign-bit should be subtracted, write

$$\begin{aligned} Z &= XY \\ &= (-x_0 + \sum_{i=1}^{W_{f_X}} x_i 2^{-i})(-y_0 + \sum_{j=1}^{W_{f_Y}} y_j 2^{-j}) \\ &= x_0 y_0 - x_0 \sum_{j=1}^{W_{f_Y}} y_j 2^{-j} - y_0 \sum_{i=1}^{W_{f_X}} x_i 2^{-i} + \sum_{i=1}^{W_{f_X}} \sum_{j=1}^{W_{f_Y}} x_i y_j 2^{-i-j} \end{aligned}$$

- The resulting partial product array is



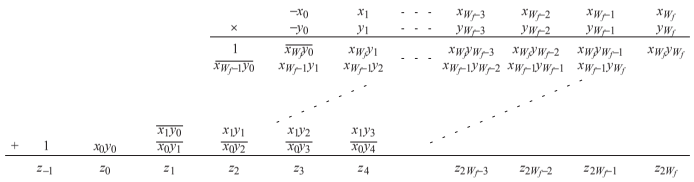
Partial product generation

- To add the partial products we must make all rows have the same length
  - Zero extend at LSB side
  - Sign extend at MSB side
- Adding zeros can be ignored, but additional circuits are required to deal with the sign-bits
- Instead, use the identity $-p = \bar{p} - 1$
- Replace all negative partial products by their inverse

$$\begin{aligned} Z &= XY \\ &= x_0 y_0 + \sum_{j=1}^{W_{f_Y}} (\overline{x_0 y_j} - 1) 2^{-j} + \sum_{i=1}^{W_{f_X}} (\overline{y_0 x_i} - 1) 2^{-i} + \sum_{i=1}^{W_{f_X}} \sum_{j=1}^{W_{f_Y}} x_i y_j 2^{-i-j} \\ &= x_0 y_0 + \sum_{j=1}^{W_{f_Y}} \overline{x_0 y_j} 2^{-j} + \sum_{i=1}^{W_{f_X}} \overline{y_0 x_i} 2^{-i} + \sum_{i=1}^{W_{f_X}} \sum_{j=1}^{W_{f_Y}} x_i y_j 2^{-i-j} - 2\left(1 - 2^{-W}\right) \end{aligned}$$

Partial product generation

- Possible to add partial products with positive weights and then add the compensation $2\left(1 - 2^{-W}\right) = 10.00\ldots0010$
- The resulting partial product array is



Partial product generation

- A standard multiplier would typically have W rows to generate and add
- In last lecture it was mentioned that for CSD representations at most half of the rows digits are non-zero
- What about encoding one of the terms in CSD?
- CSD requires sequential conversion closely related to carry-propagation
- Modified Booth encoding have a fully parallel conversion
- The modified Booth encoding has at most half of the rows, but on average it has more digits

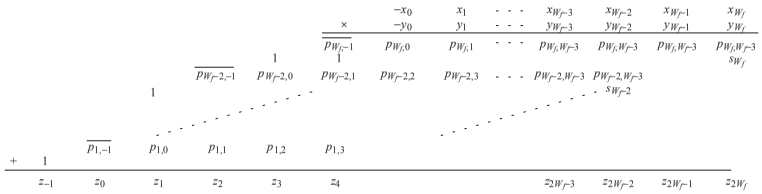
Partial product generation

- The modified Booth encoding can be seen as a radix-4 signed-digit representation with the recoded digits $x_i \in \{-2, -1, 0, 1, 2\}$
- The encoding is based on three bits, but it only replaces two

x_{2k}	x_{2k+1}	x_{2k+2}	r_k	$d_{2k}d_{2k+1}$	Description
0	0	0	0	00	String of zeros
0	0	1	1	01	End of ones
0	1	0	1	01	Single one
0	1	1	2	10	End of ones
1	0	0	-2	$\bar{1}0$	Start of ones
1	0	1	-1	$0\bar{1}$	Start and end of ones
1	1	0	-1	$0\bar{1}$	Start of ones
1	1	1	0	00	String of ones

Partial product generation

- Only half the number of partial product rows are now required
- Must be one bit wider to allow shifts
- Also, to possibly subtract a row, a partial product for adding the LSB must be included
- Resulting partial products array



Partial product generation

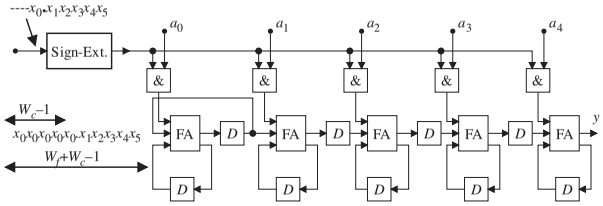
- Possible to define Booth encoding for higher radices
- For radix-8 the digits are $x_i \in \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$
- Multiplication by 3 require one adder in the pre-computation part
- The number of rows are now $W/3$ instead of $W/2$ for radix-4

Partial product accumulation

- Three general ways to adding the partial products
  - Sequential – some of the partial products are added in each cycle
  - Array – regular structure
  - Tree – smallest logic depth but irregular structure

Partial product accumulation – Sequential

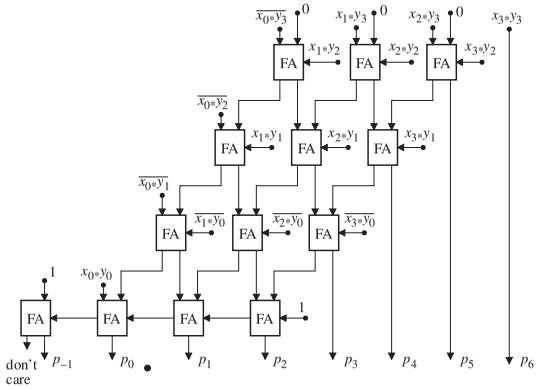
- Add one row or column in each cycle, shift the result and add one more row or column
- Five-bits row-wise sequential accumulator



- Commonly used structure for bit-serial/parallel multipliers

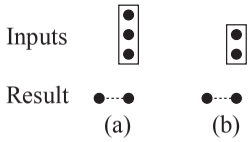
Partial product accumulation – Array

- Use an array of almost identical cells for generation and accumulation of the partial products
- Array multiplier with multiplication time proportional to $2W$



Partial product accumulation – Tree

- Array accumulation gives high regularity, but at the same time the delay increase linearly with the wordlength
- Instead, we can just add all the partial products using simple building blocks such as half and full adders
- A full adder will take three partial products and output two partial products
- A half adder will take two partial products and output two partial products



Partial product accumulation – Tree

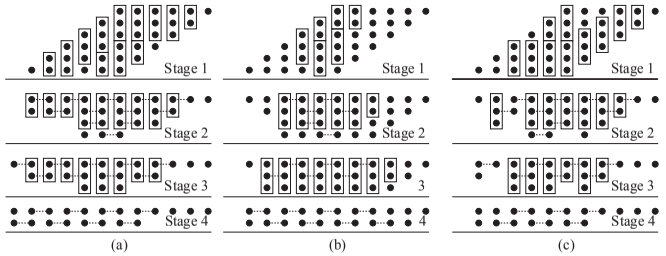
- The approach to add all partial products as fast as possible by first adding as many full adders as possible and then as many half adders as possible is known as the Wallace multiplier
- The drawback of the Wallace multiplier is the excessive use of half adders
  - Half adders does not decrease the number of partial products, it only moves them to different columns

Partial product accumulation – Tree

- An alternative approach is to try to balance the height of the columns first to later reduce as much as possible
- This approach is called the Dadda multiplier
- The drawback is that it hardly use any half adders and, hence, the carry-propagation may become longer
- The reduced area tree tries to find the best of both algorithms, fastest possible reduction, but considering half adders

Partial product accumulation – Tree

- Example reduction steps of the discussed algorithms: (a) Wallace, (b) Dadda, and (c) Reduced area



Tree structure	Full adders	Half adders	VMA length
Wallace	16	13	8
Dadda	15	5	10
Reduced area	18	5	7

Vector merging adder

- Many partial products accumulation schemes results in a redundant representation
- This must be converted to non-redundant form, typically by addition
- It should be noted that the delay for the different output bits from the accumulation typically differs quite much
- Can design parallel prefix adders where some inputs are received later on

Distributed arithmetic

- Distributed arithmetic is an efficient scheme for computing inner products of a fixed and a variable data vector

$$Y = a^T X = \sum_{i=1}^{N} a_i X_i$$

- For two's complement this can be rewritten as

$$\begin{aligned} Y &= -\sum_{i=1}^{N} a_i x_{i0} + \sum_{k=1}^{Wf} \left[\sum_{i=1}^{N} a_i x_{ik} \right] 2^{-k} \\ &= -F_0(x_{10}, x_{20}, ..., x_{N0}) + \sum_{k=1}^{Wf} F_k(x_{1k}, x_{2k}, ..., x_{Nk}) 2^{-k} \end{aligned}$$

where

$$F_k(x_{1k}, x_{2k}, ..., x_{Nk}) = \sum_{i=1}^{N} a_i x_{ik}$$

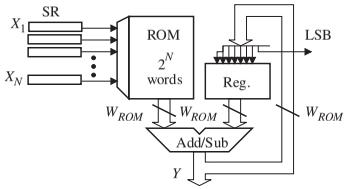
Distributed arithmetic

- F_k is a function of N binary variables, ith variable being the kth bit in x_i
- F_k can take on only 2^N values so it can be precomputed and stored in a look-up table
- Example: $Y = a_1X_1 + a_2X_2 + a_3X_3$ where $a_1 = (0.0100001)_{2C}$, $a_2 = (0.1010101)_{2C}$, and $a_3 = (1.1110101)_{2C}$
  - The look-up table looks like

x_1	x_2	x_3	F_k	F_k
0	0	0	0	$(0.0000000)_{2C}$
0	0	1	a_3	$(1.1110101)_{2C}$
0	1	0	a_2	$(0.1010101)_{2C}$
0	1	1	$a_2 + a_3$	$(0.1001010)_{2C}$
1	0	0	a_1	$(0.0100001)_{2C}$
1	0	1	$a_1 + a_3$	$(0.0010110)_{2C}$
1	1	0	$a_1 + a_2$	$(0.1110110)_{2C}$
1	1	1	$a_1 + a_2 + a_3$	$(0.1101011)_{2C}$

Distributed arithmetic

- Can be realized as

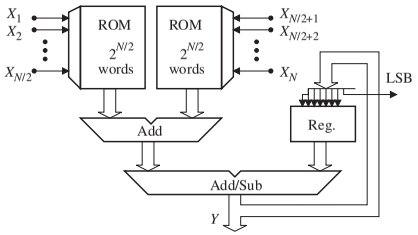


- The computation is based on Horners rule

$$y = ((...((0 + F_{Wf})2^{-1} + ... + F_2)2^{-1} + F_1)2^{-1} - F_0) \quad (1)$$

Distributed arithmetic

- The memory requirement can be high if the number of inputs is high
- Note that the memory is containing the sum of the coefficients
- One way to reduce the memory requirements is to divide the memory into two memories and add the outputs before the shift-accumulator
- The memory requirements are then reduced from 2^N to $2 \times 2^{\frac{N}{2}} = 2^{\frac{N}{2}+1}$



- Naturally, it is possible to divided the memories into more parts

Distributed arithmetic

- An alternative way to reduce the memory requirements is as follows
- Use the identity $X = \frac{1}{2}[X - (-X)]$ which gives

$$X = -(x_0 - \bar{x_0})2^{-1} + \sum_{k=1}^{Wf}(x_k - \bar{x_k})2^{-k-1} - 2^{-(W_f+1)} \quad (2)$$

- $(x_k - \bar{x}_k)$ can only evaluate to 1 or -1 which leads to

$$Y = \sum_{k=1}^{Wf} F_k(x_{1k}, \ldots, x_{Nk})2^{-k-1} - F_0(x_{10}, \ldots, x_N)2^{-1} + F(0, \ldots, 0)2^{-(W_f+1)}$$

where

$$F_k(x_{1k}, x_{2k}....., x_{Nk}) = \sum_{i=1}^{N} a_i(x_{ik} - \bar{x_{ik}}).$$

Distributed arithmetic

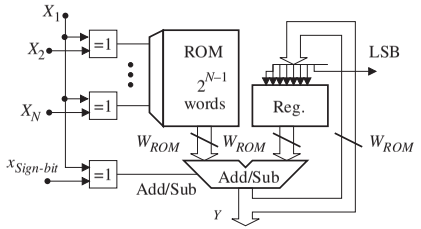
- The look-up table contents for $N = 3$ are

x_1	x_2	x_3	F_k	u_1	u_2	A/S
0	0	0	$-a_1 - a_2 - a_3$	0	0	A
0	0	1	$-a_1 - a_2 + a_3$	0	1	A
0	1	0	$-a_1 + a_2 - a_3$	1	0	A
0	1	1	$-a_1 + a_2 + a_3$	1	1	A
1	0	0	$+a_1 - a_2 - a_3$	1	1	S
1	0	1	$+a_1 - a_2 + a_3$	1	0	S
1	1	0	$+a_1 + a_2 - a_3$	0	1	S
1	1	1	$+a_1 + a_2 + a_3$	0	0	S

- The table is anti-symmetric and the addresses can be modified to utilize this (selecting X_1 as control variable) as $u_1 = x_1 \oplus x_2$, $u_2 = x_1 \oplus x_3$, and $A/S = x_1 \oplus s_{sign-bit}$

Distributed arithmetic

- The memory size is halved using this approach

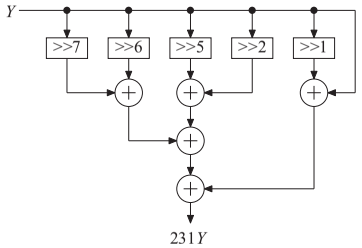


Multiplication by a constant

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

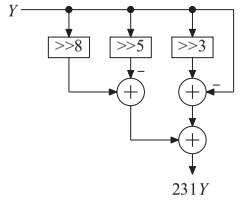
$$Z = XY = Y \sum_{i=1}^{W} x_i 2^{-i} = \sum_{i=1}^{W} Y x_i 2^{-i} \qquad (3)$$

- 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

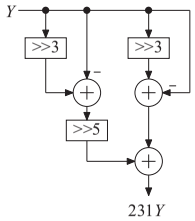
- Is there a way to find a representation with fewer non-zero positions?
- MSD/CSD is a good choice here



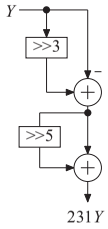
- No major difference between adders and subtracters
- Coefficients are not explicitly represented in CSD, the CSD representation rather determines the structure
- Minimum number of non-zero digits equal to minimum number of adders?

Multiplication by a constant

- No!
- Rewrite as

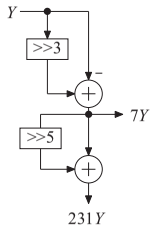


- The two first subtracters compute the same result
- Better only do it once

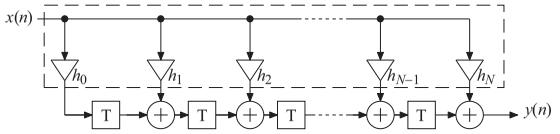


Multiplication by a constant

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

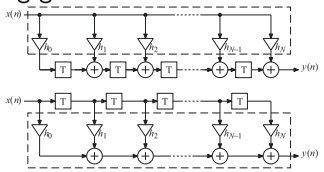


- Can this be useful?
- Transposed direct form FIR filters



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)
- 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

- 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 subexpressions
     - 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 53 and 317
- Select representation $53 =$ $, 317 =$
- Count sub-expressions

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5

- Select sub-expression and replace

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5

- Realization

Floating-point representation

- Floating-point numbers consists of two parts, the mantissa (or significand), M, and the exponent (or characteristic), E, with a number, X, represented as

$$X = M2^E \tag{4}$$

  assuming base 2
- Larger dynamic range, but lower precision compared to using the same number of bits and fixed-point
- A general floating-point representation is redundant since

$$M2^E = \frac{M}{2}2^{E+1}$$

- Therefore, most processors use normalized numbers, i.e., the mantissa does not have any leading zeros
- This also means that there is no need to store the first one of the mantissa since it is always there

Floating-point representation

- Earlier, there was a rather large difference between different processor manufacturers what the floating-point format looked like
- Today, most processors have conformed to the IEEE 754 standard
- The single precision IEEE 754-number is defined as binary32
- The binary32 format has a sign bit, eight exponent bits using excess-127 representation, and 23 bits for the mantissa plus a hidden leading one
- The represenation can be visualized as

$$\underbrace{s}_{\text{Sign}}\ \underbrace{e_{-7}e_{-6}e_{-5}e_{-4}e_{-3}e_{-2}e_{-1}e_0}_{E\ \text{8 bits biased exponent}}\ \underbrace{f_1f_2f_3f_4f_5f_6\ldots f_{22}f_{23}}_{F\ \text{23 bits unsigned fraction}}$$

- The value of the floating-point number is given by

$$X = (-1)^s\ 1.F\ 2^{E-127}$$

Floating-point representation

- There are some special cases defined as

	$F = 0$	$F \neq 0$
$E = 0$	0	Denormalized
$E = 255$	$\pm\infty$	NaN

- Denormalized numbers are used to extend the dynamic range
- The hidden one limits the smallest positive number to $2^{1-127} = 2^{-126}$
- A denormalized number has a value of

$$X = (-1)^s\ 0.F\ 2^{-126}$$

- With denormalized numbers it is possible to represent $2^{-23}2^{-126} = 2^{-149}$
- The implementation cost of denormalized numbers is high, and, hence, they are not always included

Floating-point representation

- Extended formats are also defined for intermediate results of certain operations
- For the various precisions the following wordlengths are defined

Property	binary32	binary64	binary128
Total bits	32	64	128
Mantissa bits	$23+1$	$52+1$	$112+1$
Exponent bits	8	10	14
Bias	127	1023	16383
Ext. mantissa bits	≥ 32	≥ 64	≥ 128
Ext. exponent bits	11	15	20

Floating-point operations

- Addition and subtraction
  - Require that the mantissas are algined based on the exponents
  - Then, the operation can be performed
  - Final result should in general be normalized
- Multiplication and division
  - Operates on exponent and mantissa separately
  - Exponents have to be added or subtracted
  - Mantissas are multiplied or divided
  - Final result should in general be normalized

Residue number systems

- Decompose the data into smaller parts and perform operations on each part
- The decomposition is based on modulo computations
- The set of moduli should be relative prime
- Advantage: Shorter carry chains
- Disadvantage: Overflows etc very hard to detect, hard to scale
- Example: Add 7 and 6 using moduli 3 and 5

Logarithmic number systems

- (Floating-point number system without mantissa)
- Large dynamic range
- Advantages: multiplication, division, and exponentiation are simple, corresponding to addition, subtraction, and multiplication of the exponents
- Disadvantage: Addition and subtraction are hard and require a look-up table