
DO NOT SHARE
SLIDES AND CLASS MATERIALS
ON ONLINE SITES
Course Hero

Advanced Logic Design
Computer Arithmetic: Distributed Arithmetic

Mingoo Seok
Columbia University

Readings:

Stanley A. White, “Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review,” IEEE ASSP Magazine, July, 1989

Xilinx App Note, ”The Role of Distributed Arithmetic In FPGA Based Signal Processing’

Distributed Arithmetic (DA)

- An efficient technique for calculation of **sum of products** or **vector dot product** or **inner product** or **multiply and accumulate** (MAC)
- MAC operation is very common in all Digital Signal Processing Algorithms
- Initially proposed by Peled and Liu in 1974

So Why Use DA?

- The advantages of DA are best exploited in designing data-path circuits
- Area savings from using DA can be up to 80% and seldom less than 50% in digital signal processing hardware designs
- An old technique that has been revived by the wide spread use of Field Programmable Gate Arrays (FPGAs) for Digital Signal Processing (DSP)
- DA efficiently implements the MAC using basic building blocks (Look Up Tables) in FPGAs

An Illustration of MAC Operation

- The following expression represents a multiply and accumulate operation

$$y = A_1 \times x_1 + A_2 \times x_2 + \square + A_K \times x_K$$

$$i.e. \quad y = \sum_{k=1}^{K} A_k x_k$$

- A numerical example

$$A = [32,42,45,23] \quad x = [42,20,-22,67] \quad (K = 4)$$

$$y = 32 \times 42 + 45 \times 20 + 78 \times (-22) + 23 \times 67$$

$$y = 1344 + 900 - 1716 + 1541 = 2069$$

A Few Points about the MAC

- Consider this

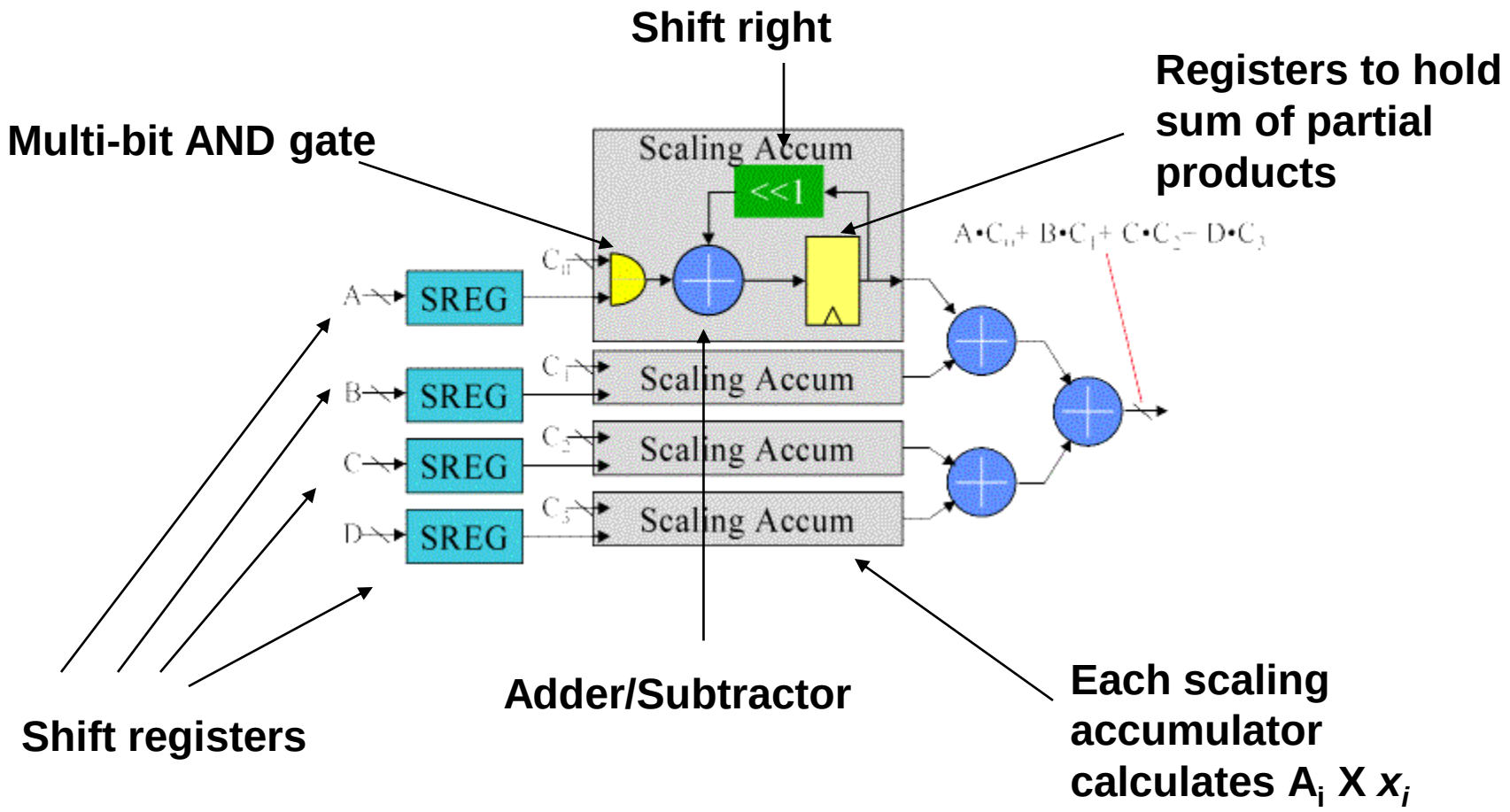
$$y = \sum_{k=1}^{K} A_k x_k$$

Note a few points

- A=[A_1, A_2,…, A_K] is a matrix of "**constant"** values
- x=[x_1, x_2,…, x_K] is a matrix of input **"variables"**
- Each A_k is of M-bits
- Each x_k is of N-bits
- y should be large enough to accommodate the result

A Possible Hardware (not DA yet)

- Let, $A = [C_0, C_1, C_2, C_3] \quad x = [A, B, C, D] \quad (K = 4)$



How does DA work?

- The DA technique is **bit-serial**
- DA is basically a **bit-level rearrangement** of the multiply and accumulate operation
  - Try to replace a multiplier with a LUT
  - The size of address will be the size of the constant vector, (e.g., 4 in the previous example)
- DA hides the explicit multiplications by **LUT**
  - An efficient technique to implement on Field Programmable Gate Arrays (FPGAs)
  - Because FPGAs are a collection of LUTs

Moving Closer to Distributed Arithmetic

- Consider once again $$y = \sum_{k=1}^{K} A_k x_k \qquad \ldots(1)$$
  - Let x_k be a N-bits scaled two's complement number i.e.

$$| x_k | < 1$$

$$x_k : \{b_{k0}, b_{k1}, b_{k2} \ldots\ldots, b_{k(N-1)}\}$$

where b_{k0} is the *sign bit*

  - We can express x_k as $$x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \qquad \ldots(2)$$
  - Substituting (2) in (1),

$$y = \sum_{k=1}^{K} A_k \left[-b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \right]$$

$$y = -\sum_{k=1}^{K} (b_{k0} \bullet A_k) + \sum_{k=1}^{K} \sum_{n=1}^{N-1} (A_k \bullet b_{kn}) 2^{-n} \qquad \ldots(3)$$

Moving More Closer to DA

$$y = -\sum_{k=1}^{K}(b_{k0} \bullet A_k) + \sum_{k=1}^{K}\left[\sum_{n=1}^{N-1}(b_{kn} \bullet A_k)2^{-n}\right] \quad \ldots(3)$$

Expanding this part

$$y = -\sum_{k=1}^{K}(b_{k0} \bullet A_k) + \sum_{k=1}^{K}\left[(A_k \bullet b_{k1})2^{-1} + (A_k \bullet b_{k2})2^{-2} + \square + (A_k \bullet b_{k(N-1)})2^{-(N-1)}\right]$$

$$\begin{aligned} y = &-\left[b_{10} \bullet A_1 + b_{20} \bullet A_2 + \square + b_{K0} \bullet A_K\right] \\ &+ \left[(b_{11} \bullet A_1)2^{-1} + (b_{12} \bullet A_1)2^{-2} + \square + (b_{1(N-1)} \bullet A_1)2^{-(N-1)}\right] \\ &+ \left[(b_{21} \bullet A_2)2^{-1} + (b_{22} \bullet A_2)2^{-2} + \square + (b_{2(N-1)} \bullet A_2)2^{-(N-1)}\right] \\ &\square \\ &+ \left[(b_{K1} \bullet A_K)2^{-1} + (b_{K2} \bullet A_K)2^{-2} + \square + (b_{K(N-1)} \bullet A_K)2^{-(N-1)}\right] \end{aligned}$$

Moving Still More Closer to DA

$$y = -\left[b_{10} \bullet A_1 + b_{20} \bullet A_2 + \cdots + b_{K0} \bullet A_K\right]$$
$$+\left[(b_{11} \bullet A_1)2^{-1} + (b_{12} \bullet A_1)2^{-2} + \cdots + (b_{1(N-1)} \bullet A_1)2^{-(N-1)}\right]$$
$$+\left[(b_{21} \bullet A_2)2^{-1} + (b_{22} \bullet A_2)2^{-2} + \cdots + (b_{2(N-1)} \bullet A_2)2^{-(N-1)}\right]$$
$$\vdots$$
$$+\left[(b_{K1} \bullet A_K)2^{-1} + (b_{K2} \bullet A_K)2^{-2} + \cdots + (b_{K(N-1)} \bullet A_K)2^{-(N-1)}\right]$$

$$y = -\left[b_{10} \bullet A_1 + b_{20} \bullet A_2 + \cdots + b_{K0} \bullet A_K\right]$$
$$+\left[(b_{11} \bullet A_1) + (b_{21} \bullet A_2) + \cdots + (b_{K1} \bullet A_K)\right]2^{-1}$$
$$+\left[(b_{12} \bullet A_1) + (b_{22} \bullet A_2) + \cdots + (b_{K2} \bullet A_K)\right]2^{-2}$$
$$\vdots$$
$$+\left[(b_{1(N-1)} \bullet A_1) + (b_{2(N-1)} \bullet A_2) + \cdots + (b_{K(N-1)} \bullet A_K)\right]2^{-(N-1)}$$

Almost there!

$$y = -\left[b_{10} \bullet A_1 + b_{20} \bullet A_2 + \square + b_{K0} \bullet A_K\right]$$
$$+\left[\left(b_{11} \bullet A_1\right) + \left(b_{21} \bullet A_2\right) + \square + \left(b_{K1} \bullet A_K\right)\right]2^{-1}$$
$$+\left[\left(b_{12} \bullet A_1\right) + \left(b_{22} \bullet A_2\right) + \square + \left(b_{K2} \bullet A_K\right)\right]2^{-2}$$
$$\square$$
$$+\left[\left(b_{1(N-1)} \bullet A_1\right) + \left(b_{2(N-1)} \bullet A_2\right) + \square + \left(b_{K(N-1)} \bullet A_K\right)\right]2^{-(N-1)}$$

$$y = -\sum_{k=1}^{K}(b_{k0}) \bullet A_k + \sum_{n=1}^{N-1}\left[b_{1n} \bullet A_k + b_{2n} \bullet A_2 + \square + b_{Kn} \bullet A_K\right]2^{-n}$$

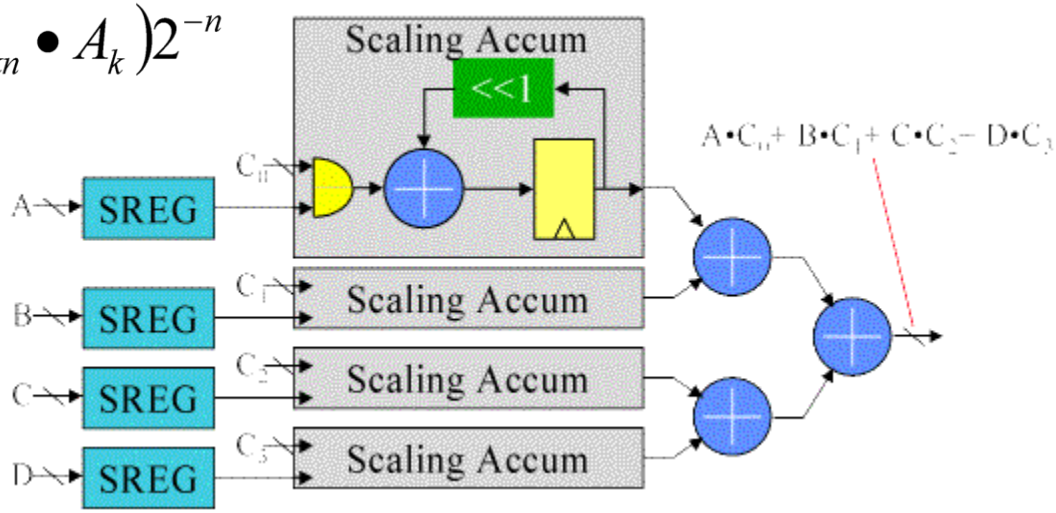
$$y = -\sum_{k=1}^{K} A_k \bullet (b_{k0}) + \sum_{n=1}^{N-1}\left[\sum_{k=1}^{K} A_k \bullet b_{kn}\right]2^{-n} \qquad \ldots(4)$$

The Final Reformulation

Lets See the change of hardware

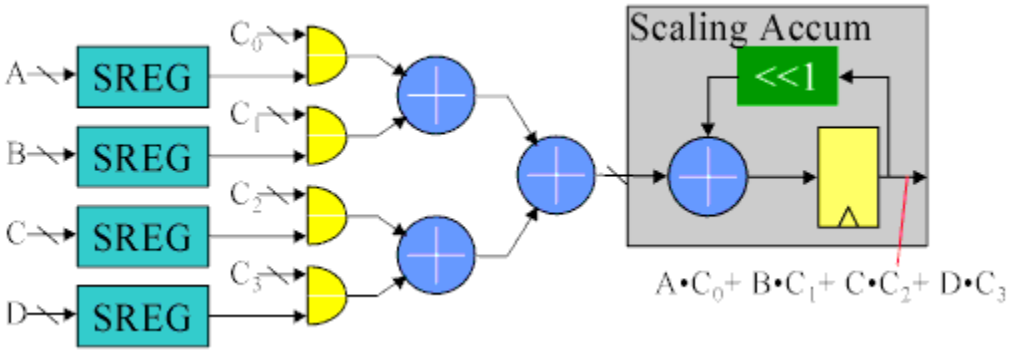
$$y = -\sum_{k=1}^{K}(b_{k0} \bullet A_k) + \sum_{k=1}^{K}\sum_{n=1}^{N-1}(b_{kn} \bullet A_k)2^{-n}$$

Our Original Equation

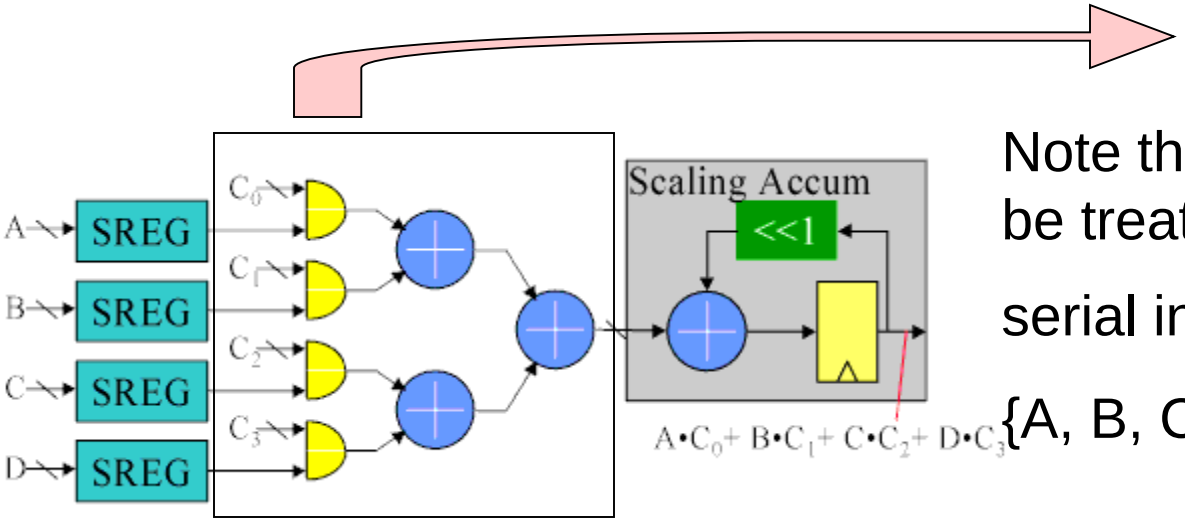


$$y = -\sum_{k=1}^{K} A_k \bullet (b_{k0}) + \sum_{n=1}^{N-1}\left[\sum_{k=1}^{K} A_k \bullet b_{kn}\right]2^{-n}$$

Bit Level Rearrangement



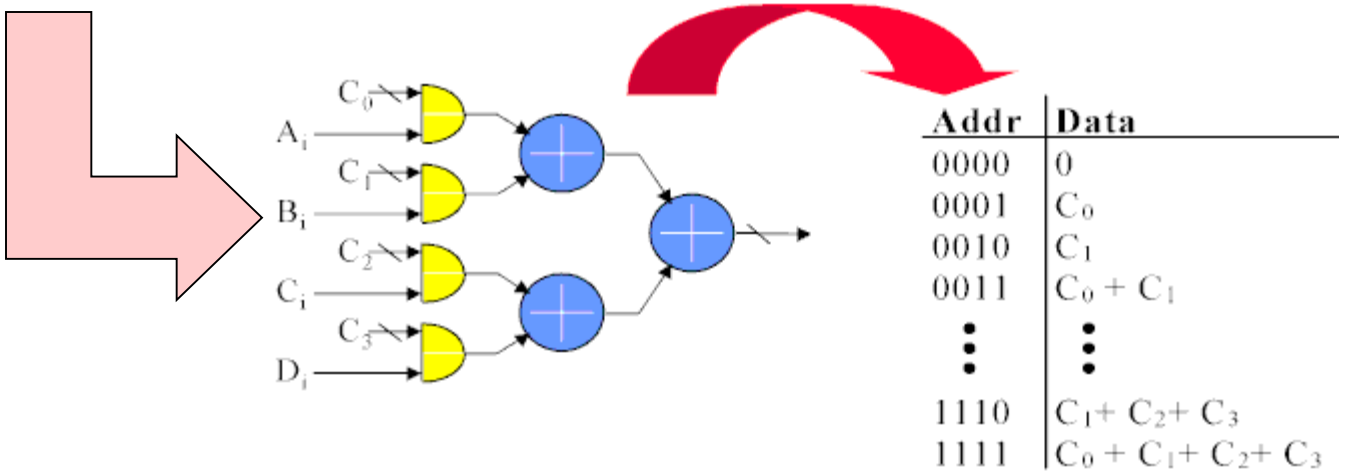
So where does the ROM come in?



Note this portion. It can be treated as function of

serial inputs bits of

{A, B, C,D}



The ROM Construction

$$y = -\sum_{k=1}^{K} A_k \bullet (b_{k0}) + \sum_{n=1}^{N-1}\left[\sum_{k=1}^{K} A_k \bullet b_{kn}\right]2^{-n} \quad \ldots(4)$$

- $\left[\sum_{k=1}^{K} A_k b_{kn}\right]$ has only(?) 2^K possible values i.e.

$$\left[\sum_{k=1}^{K} A_k b_{kn}\right] = f_n(b_{1n} b_{2n} \square\ b_{Kn}) \quad \ldots(5)$$

- (5) can be **pre-calculated** for all possible values of $b_{1n}\, b_{2n} \ldots b_{Kn}$
- We can store these in a look-up table of ***2^K* words** addressed by ***K-bits*** *i.e.* $b_{1n}\, b_{2n} \ldots b_{Kn}$

Lets See An Example

- Let number of taps K=4
- The fixed coefficients are A_1 =0.72, A_2= -0.3, A_3 = 0.95, A_4 = 0.11

$$y = \sum_{n=1}^{N-1}\left[\sum_{k=1}^{K} A_k b_{kn}\right]2^{-n} + \sum_{k=1}^{K} A_k(-b_{k0}) \quad \ldots(4)$$

- We need $2^K = 2^4$ = 16-words ROM

ROM: Address and Contents

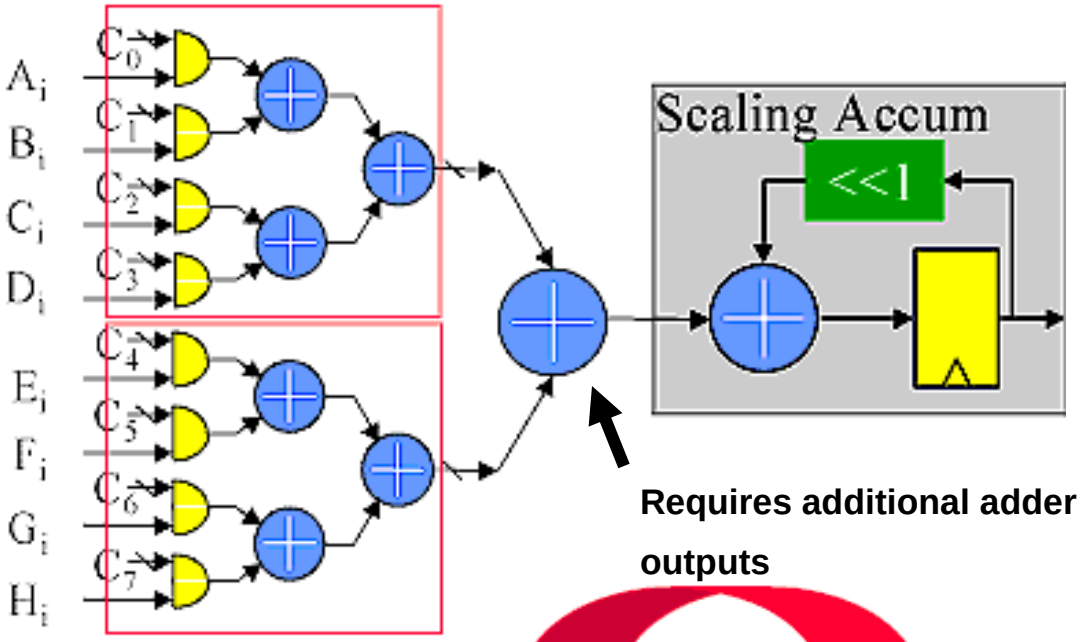
$$\left[\sum_{k=1}^{4} A_k b_{kn}\right] = A_1 b_{1n} + A_2 b_{2n} + A_3 b_{3n} + A_4 b_{4n}$$

b_{1n}	b_{2n}	b_{3n}	b_{4n}	Contents
0	0	0	0	0
0	0	0	1	A_4=0.11
0	0	1	0	A_3=0.95
0	0	1	1	A_3+ A_4=1.06
0	1	0	0	A_2=-0.30
0	1	0	1	A_2+ A_4= -0.19
0	1	1	0	A_2+ A_3=0.65
0	1	1	1	A_2+ A_3 + A_4=0.75
1	0	0	0	A_1=0.72
1	0	0	1	A_1+ A_4=0.83
1	0	1	0	A_1+ A_3=1.67
1	0	1	1	A_1+ A_3 + A_4=1.78
1	1	0	0	A_1+ A_2=0.42
1	1	0	1	A_1+ A_2 + A_4=0.53
1	1	1	0	A_1+ A_2 + A_3=1.37
1	1	1	1	A_1+ A_2 + A_3 + A_4=1.48

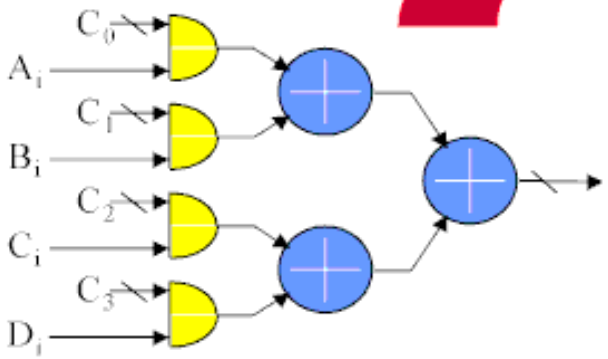
Key Issue: ROM Size

- The size of ROM is very important for high speed implementation as well as area efficiency
- ROM size grows exponentially with each added input address line
- The number of address lines are equal to the number of elements in the vector of constants i.e. K
- Elements up to 16 and more are common => 2^{16}=64K-words of ROM (!)
- We have to reduce the size of ROM

Decomposing the ROM



Requires additional adder to the sum the partial outputs



Addr	Data
0000	0
0001	C_0
0010	C_1
0011	$C_0 + C_1$
⋮	⋮
1110	$C_1 + C_2 + C_3$
1111	$C_0 + C_1 + C_2 + C_3$

Speed Concerns

- We considered One Bit At A Time (1 BAAT)
- No. of Clock Cycles Required = N
- If K=N, then essentially we are taking 1 cycle per dot product. Not bad!
- Opportunity for parallelism exists but at a cost of more hardware
- We could have 2 BAAT or up to N BAAT in the extreme case
- N BAAT. One complete result/cycle

Illustration of 2 BAAT

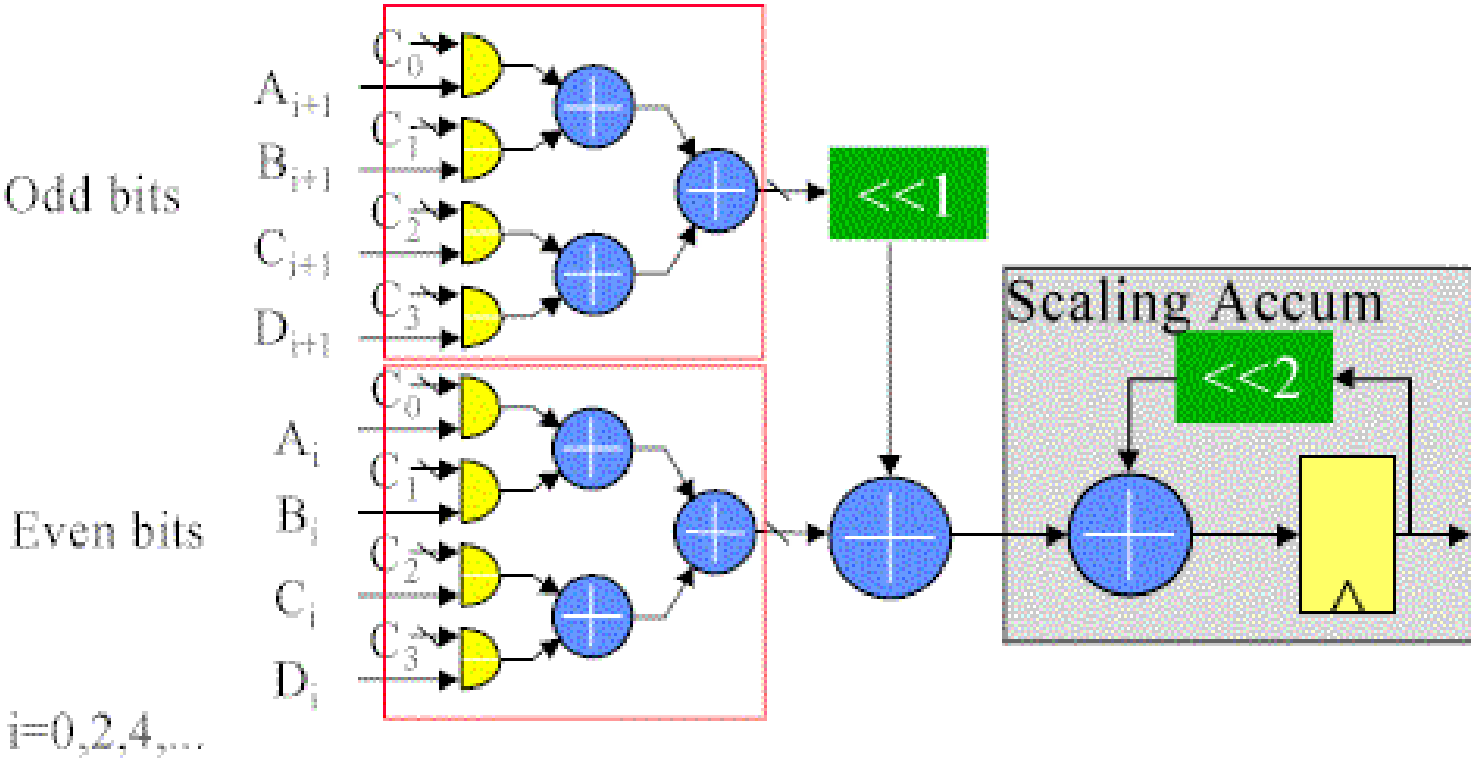


Illustration of N BAAT (N=4)