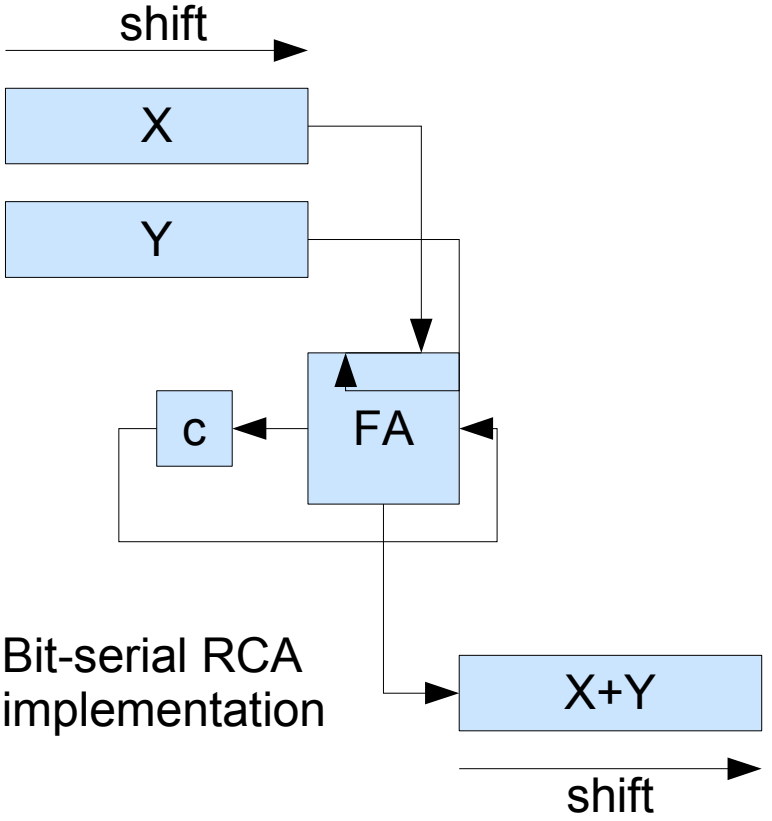
Fast Multipliers

Fast Multiplication

- Additions and Multiplications are building blocks for complex function computation, digital signal processing, etc.
- A number of approaches possible:
  - Bit-serial – compact design, limited resources, relatively slowest, serial process may be an advantage
  - Multioperand addition – cost-effective implementations and high-performance arithmetic units
  - Divide & Conquer – decomposition of long multiplications into smaller tasks
  - Hybrid designs

Bit-Serial Processing

- VLSI implementation advantages:
  - small pin count
  - reduced wire length
  - high clock rate
  - small space
  - low power consumption
- Alternative to pipeline units for parallel processing

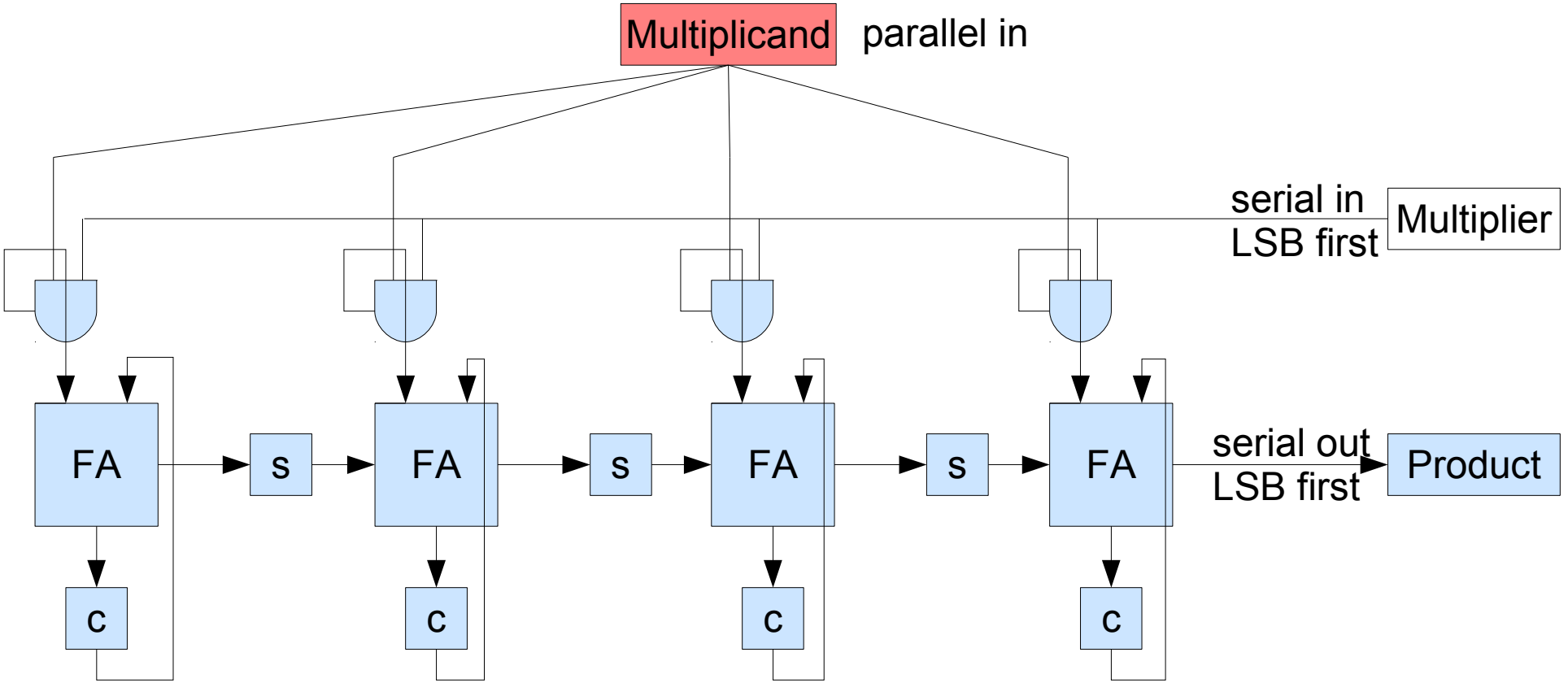


Bit-serial RCA implementation

Bit-Serial Multiplier

- Semi- or fully-serial solutions exists
  - e.g. semi-serial 4×4

    2k cycles needed for k×k-bit multiplication



Multioperand Problem

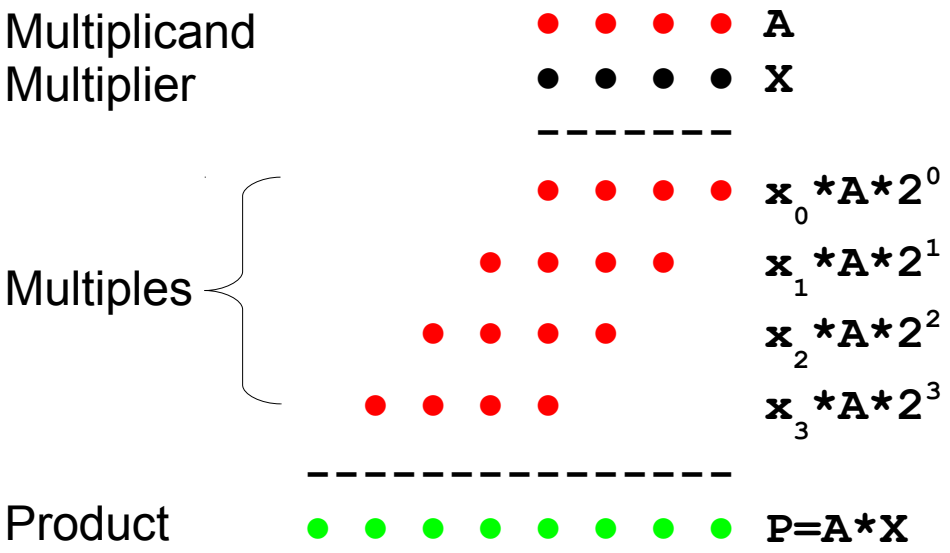
Multiplication as multioperand addition

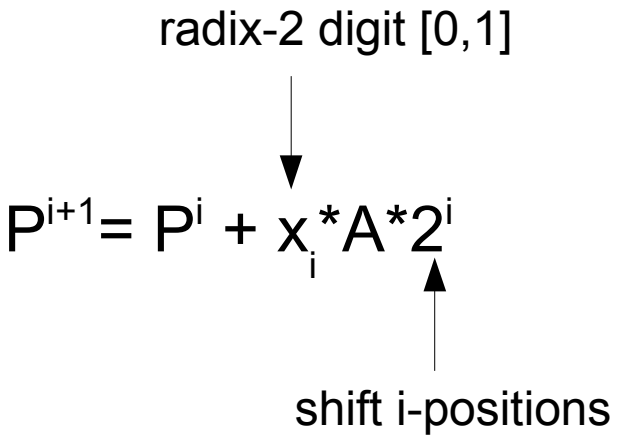
- Possible improvements:
  - add the operands faster → tree & array multipliers (various CSA configurations)
  - reduce the number of operands → high-radix multipliers operation on shorter operands

Basic Multiplication

- Add/Shift algorithm of k-digit numbers, Θ(k log k)

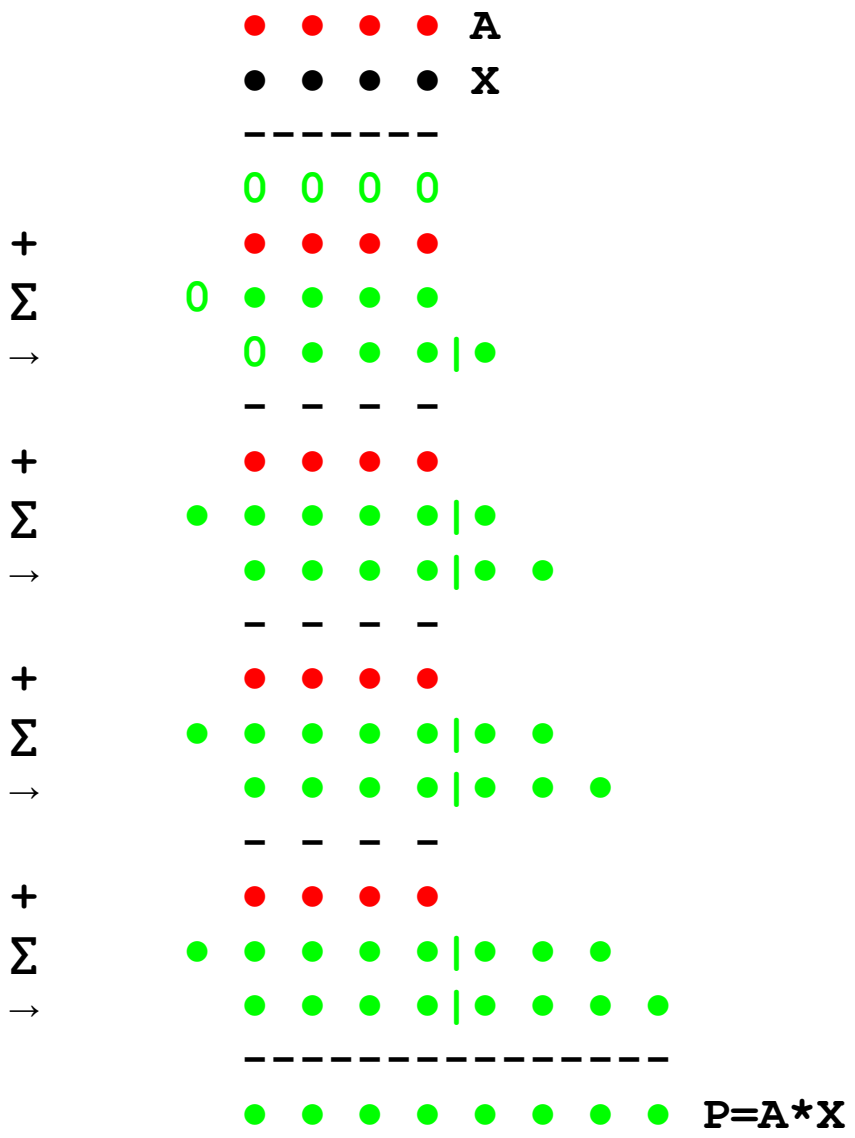
A – Multiplicand, X – Multiplier, P – product



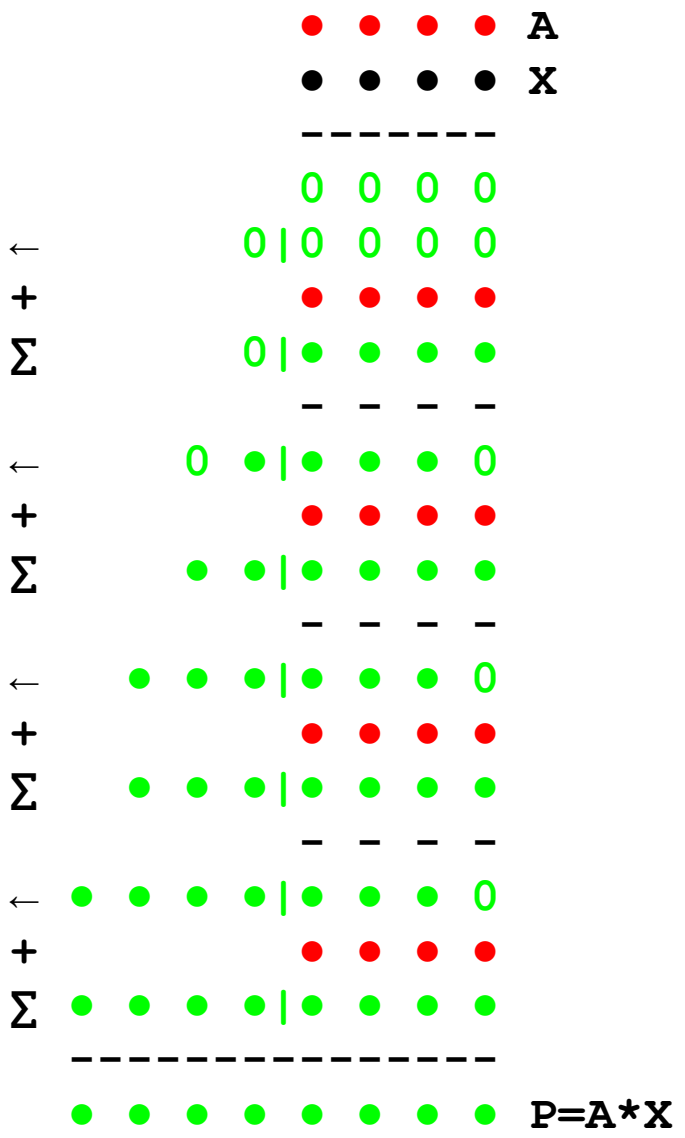


- In radix-2, multiple (the term x_i*A) is either 0 or A, thus simple
- Unnecessary demand of resources (long adder)

Shift Right(Left) Approach



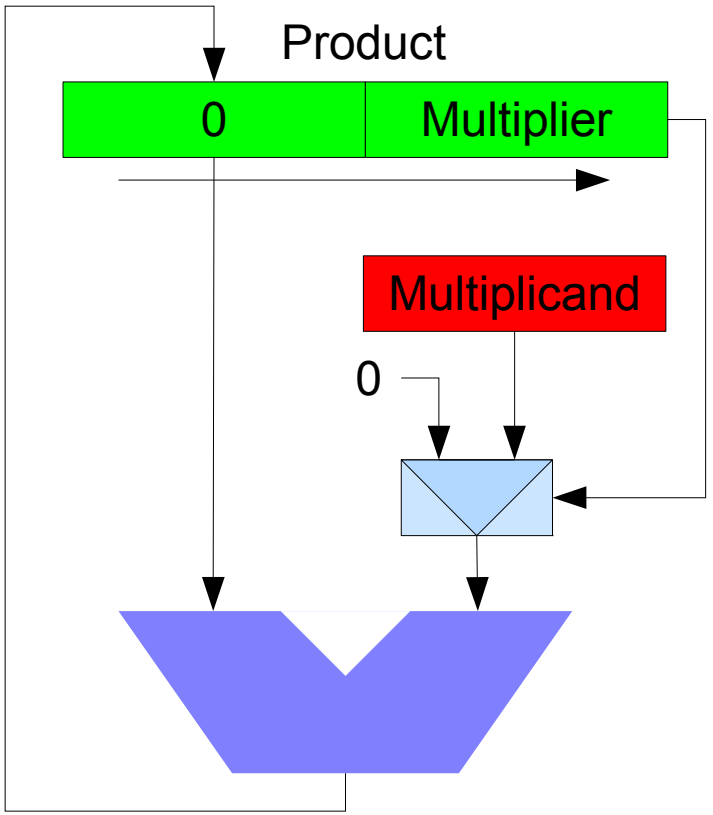
$$P^{i+1} = (P^{i}_{(upper)} + x_i * A) * 2^{-1}$$



$$P^{i+1} = 2^{1} * P^{i} + x_{k-i-1} * A$$

Realizations

Hardware Shift Right



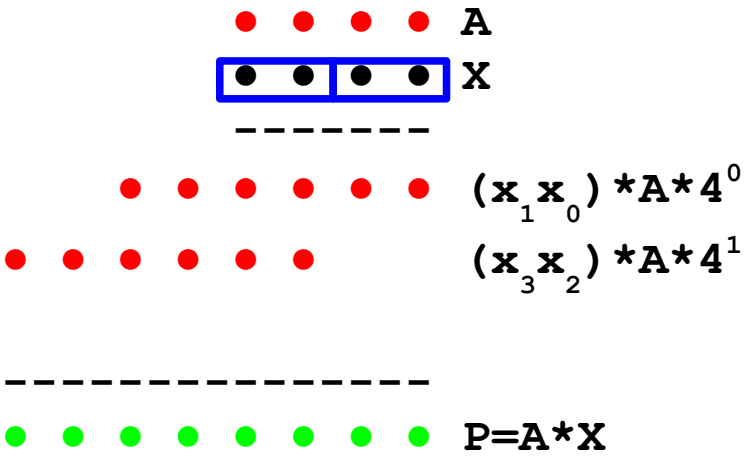
Software Shift Left

```
* initialize
        move.w   #A,RA
        move.w   #X,RX
        move.l   #0,RP
        move.b   #k,RC
* start
next    lsl.l    #1,RP
        lsl.w    #1,RX
        bcc      skip
        add.l    RA,RP
skip    sub.b    #1,RC
        bne      next
* done
```

Motorola 68000 Assembler

Radix-4 Multiplication

- k-digit radix-2 number = ½k-digit radix-4 number
- Require multiples: 0, A, 2A, 3A, thus not so simple



radix-4 digit [0,3]

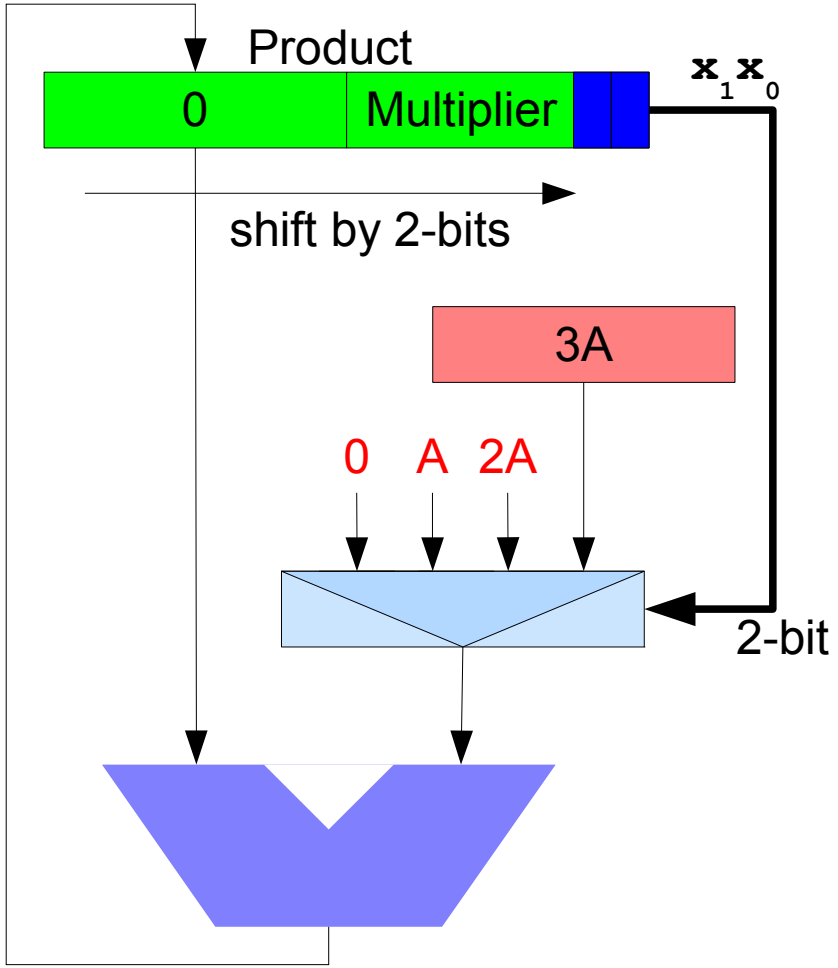
$$P^{i+1} = P^i + x_i * A * 4^i$$

shift i-positions in radix-4

- 0, A, 2A can be provided easily (2A by shifting)
- 3A (yet another addition ?)

Precomputation of Multiples

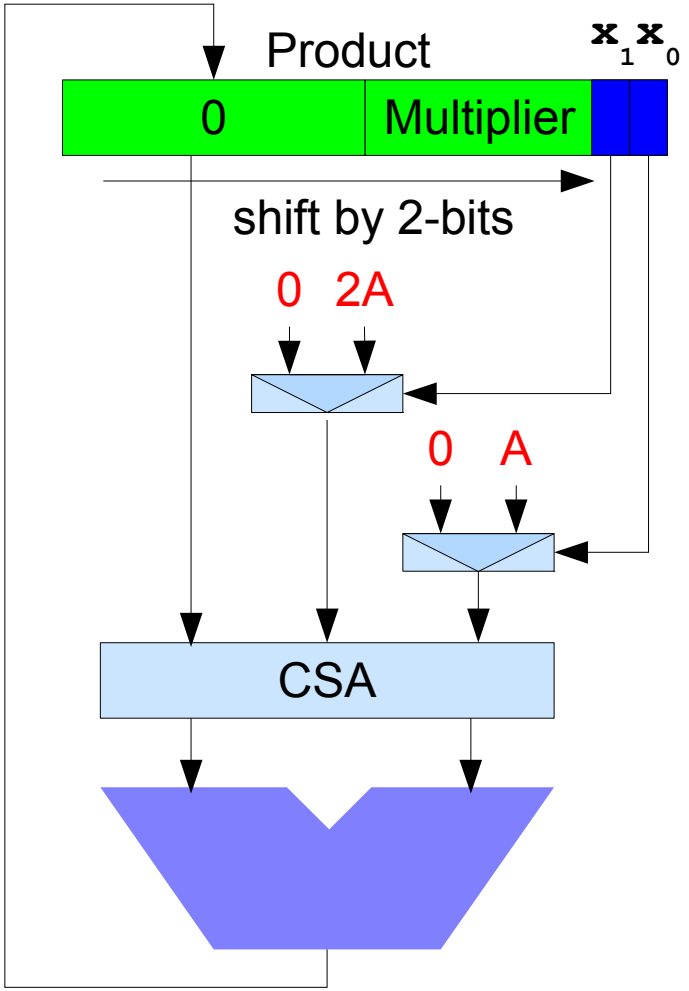
- Compute 3A once and use many times later
- Solution does not scale well to radix-8, radix-16,...



Radix-4 Multiplier

Computing Multiples with CSA

- CSA's offer a solution to fast computation of 3A (and other in higher radices) multiples



Partial tree architecture

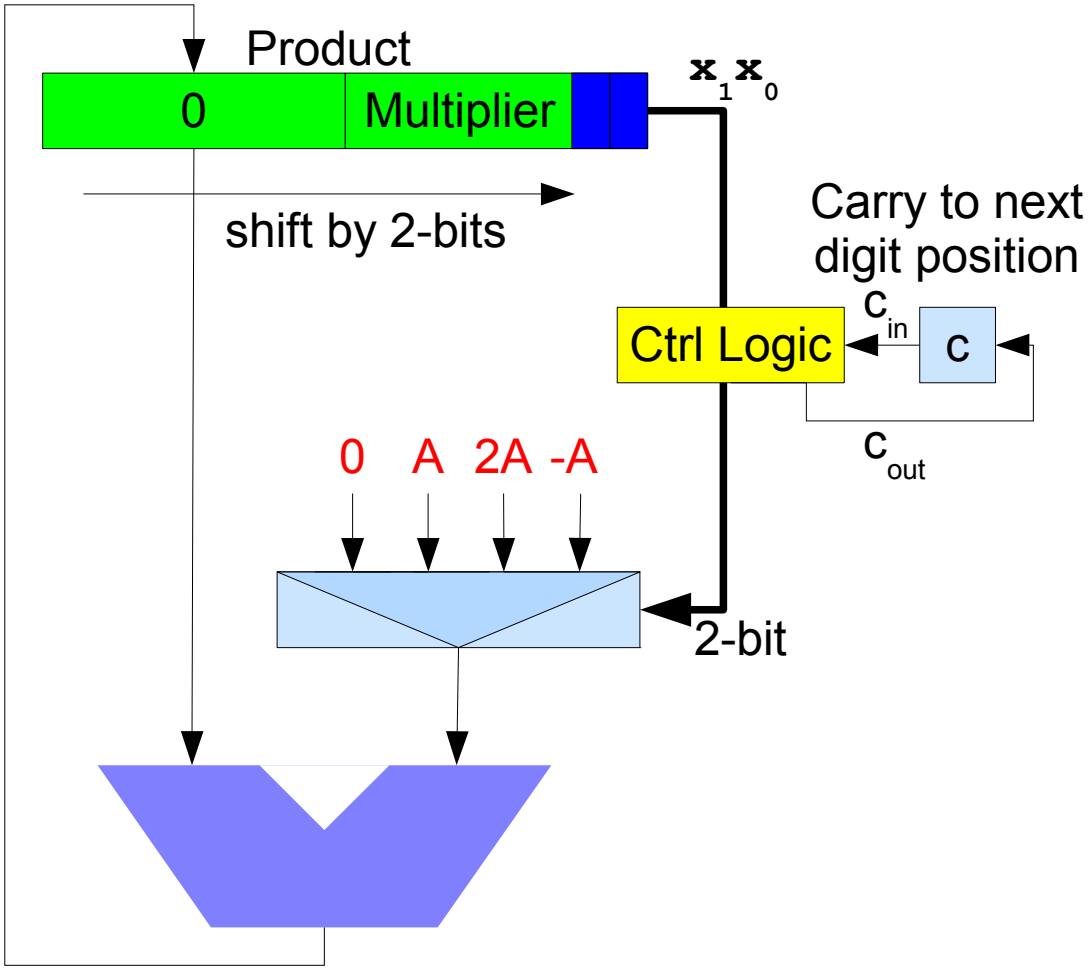
CSA'a introduce a delay in each step of multiplication. In higher radices this delay can be significant.

Radix-4 Multiplier

Multiples Generation with Carry

- Multiples can be handled as (in radix-4):
  - 3A = Carry – A
  - 4A = Carry

x1	x0	cin	Multiple	cout
0	0	0	0	0
0	0	1	A	0
0	1	0	A	0
0	1	1	2A	0
1	0	0	2A	0
1	0	1	-A	1
1	1	0	-A	1
1	1	1	0	1



Radix-4 Multiplier

Multiples Generation – Example

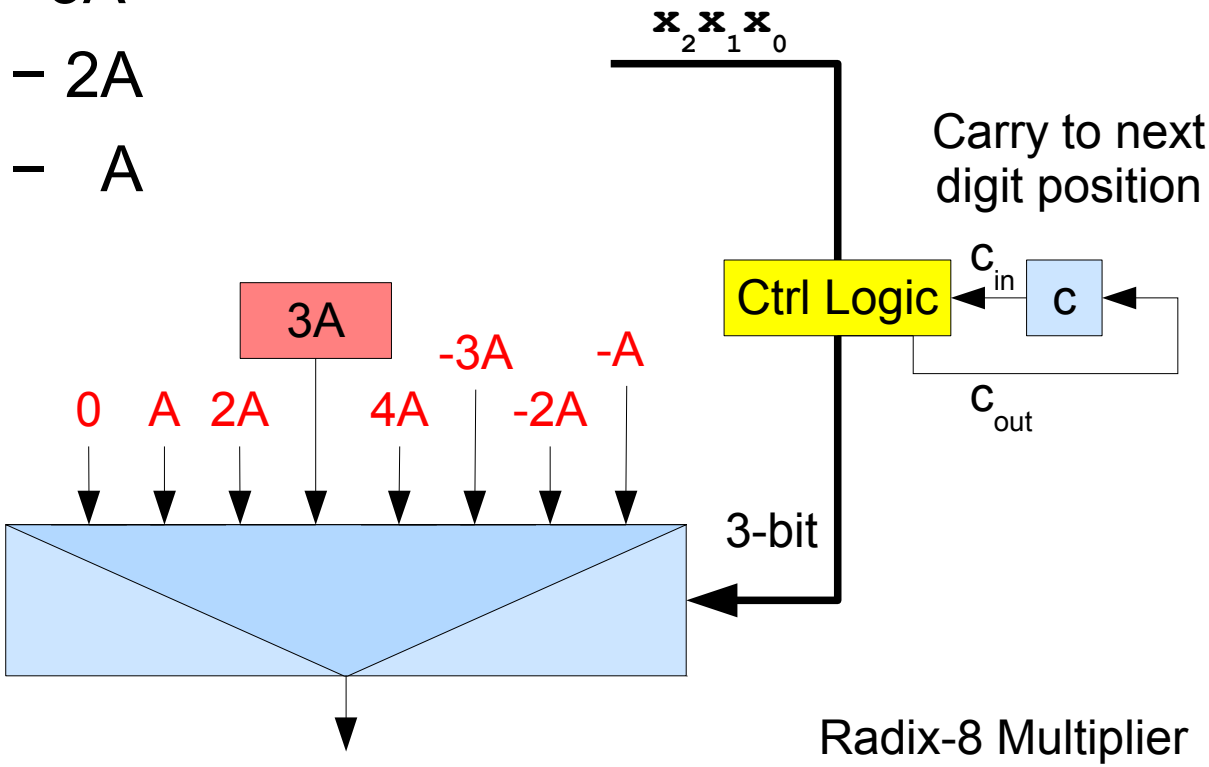
```
      0101          → 5 (A)
      0111          → 7
      ----
      0000
+     1011      (3A → -A & Carry generated
Σ     1011
→     001011
+     0101       +1   (1A + Carry)
Σ     100011
→     00100011
-------------
      00100011      → 35
```

Only 2-steps of algorithm in radix-4

Radix-4 Multiplication

Multiples Generation with Carry

- Method scales to radix-8 (precomputation needed), but is not practical for higher radices
  - radix-8: 3A must be precomputed
    - 5A → Carry − 3A
    - 6A → Carry − 2A
    - 7A → Carry − A
    - 8A → Carry



Radix-8 Multiplier

Booth's Recoding – Signed Arithmetics

- Any chain of ones (1) in radix-2 numbers can be handled by just two operations: subtraction at the beginning and addition at the end of chain

$$\ldots 0011 \ldots 1100 \ldots = 2^{j} + 2^{j-1} + \ldots + 2^{i+1} + 2^{i} = 2^{j+1} - 2^{i}$$

$j \quad i$

- Radix-2 operands can be converted (recoded) to BSD (redundant, radix-2, signed-digit system with set [-1,1])

$$\ldots 0011 \ldots 1100 \ldots = \ldots 0100 \ldots 0 \; -1 \; 00 \ldots$$

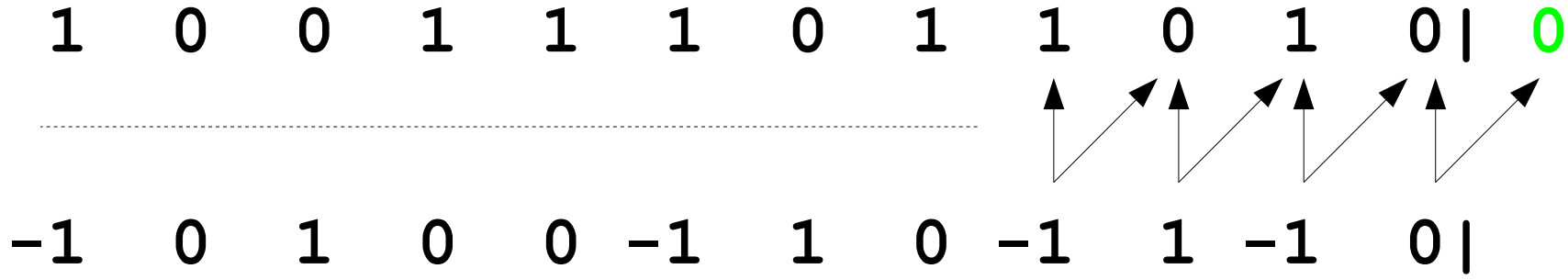
$j \quad i \qquad\qquad j \quad i$

- Recoding properly handles negative (2's complement) numbers – signed multiplication possible

Radix-2 Recoding

NBC (radix-2, digit set [0,1]) ⟶ BSD (radix-2, [-1,1])

- 00 or 11 – inside chain of 0's or 1's – recode as 0
- 10 – beginning of 1's chain – recode as -1
- 01 – end of 1's chain – recode as 1

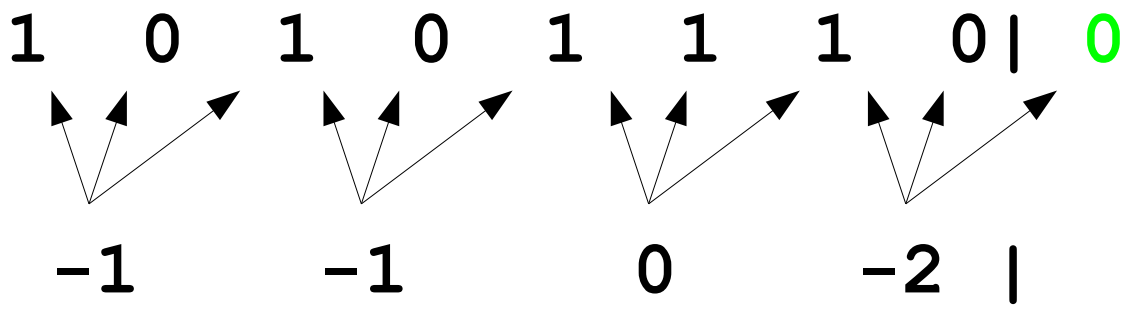


- Recoding is fast – no carry propagation
- Recoded i-digit depends directly on i-th and i-1 digits

Radix-4 Recoding

NBC (radix-2, digit set [0,1]) → radix-4 [-2,2]

- recoding is fast and direct
- each recoded digit depends on 3 digits
- recoded number has k/2 digits



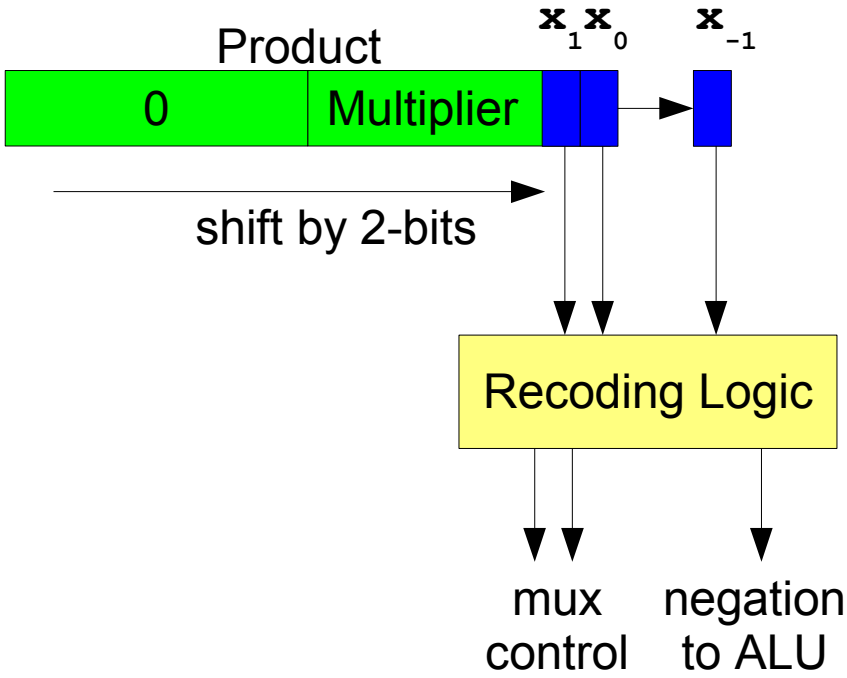
Radix-4 Recoding Logic

radix-2 [0,1]	radix-2 [-1,1]	radix-4 [-2, 2]
000	0 0	0
001	0 1	1
010	1 -1	1
011	1 0	2
100	-1 0	-2
101	-1 1	-1
110	0 -1	-1
111	0 0	0

- Recoded i-digit depends directly on i-th and i-1 digits
- Recoding in higher radices is equally simple

Multiplication with Recoding

- If multiplier is recoded (radix-4 [-2,2]), than possible multiples are: 0, ±1A, ±2A, thus simple
- 3-bit of multiplier are needed for coding each radix-4 digit of multiplier

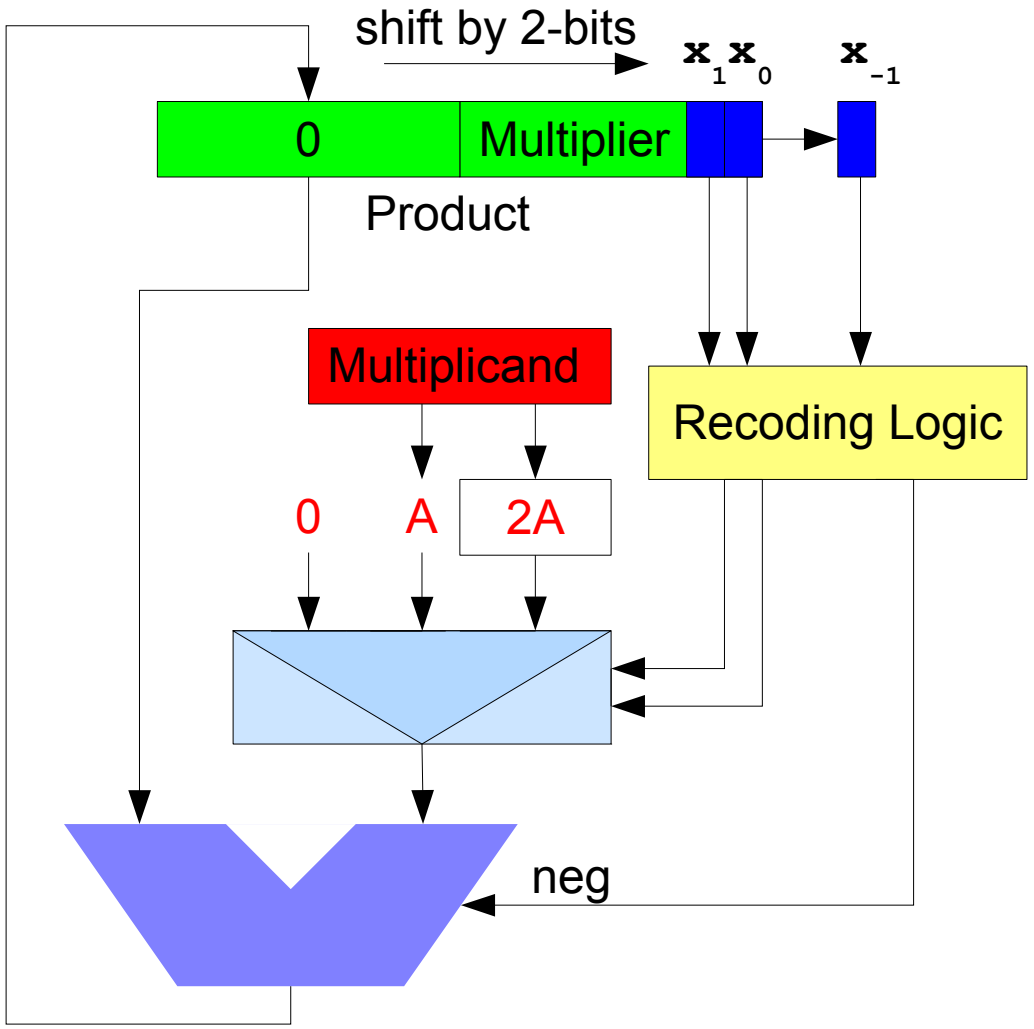


Booth's Algorithm

Radix-4 Multiplier

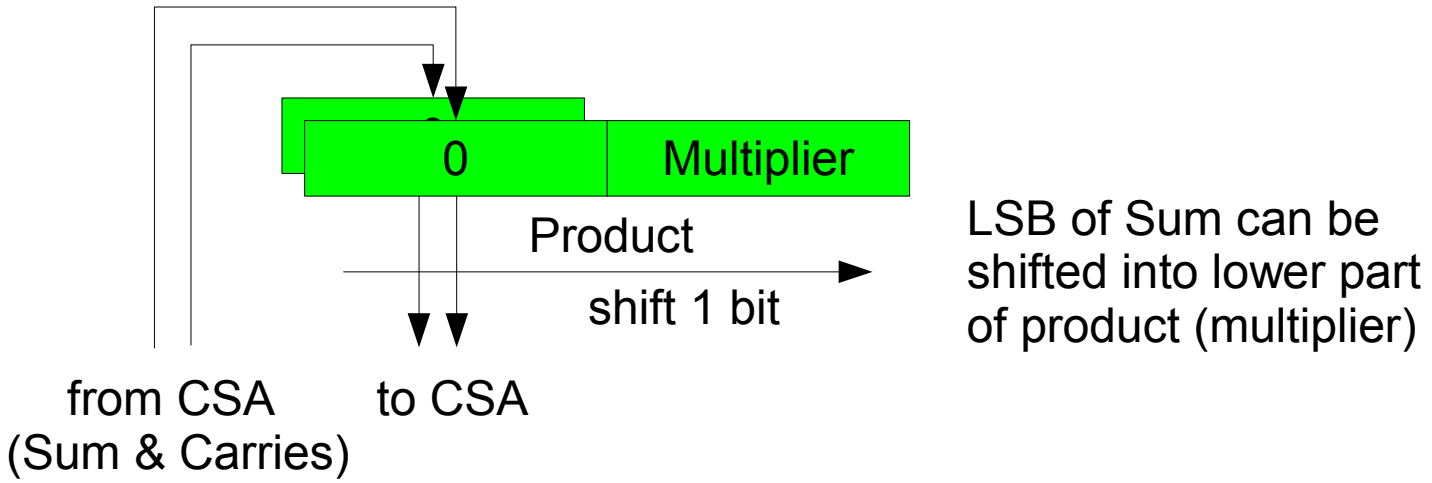
Radix-4 Multiplier – Signed Arithmetics

with Booth's Recoding



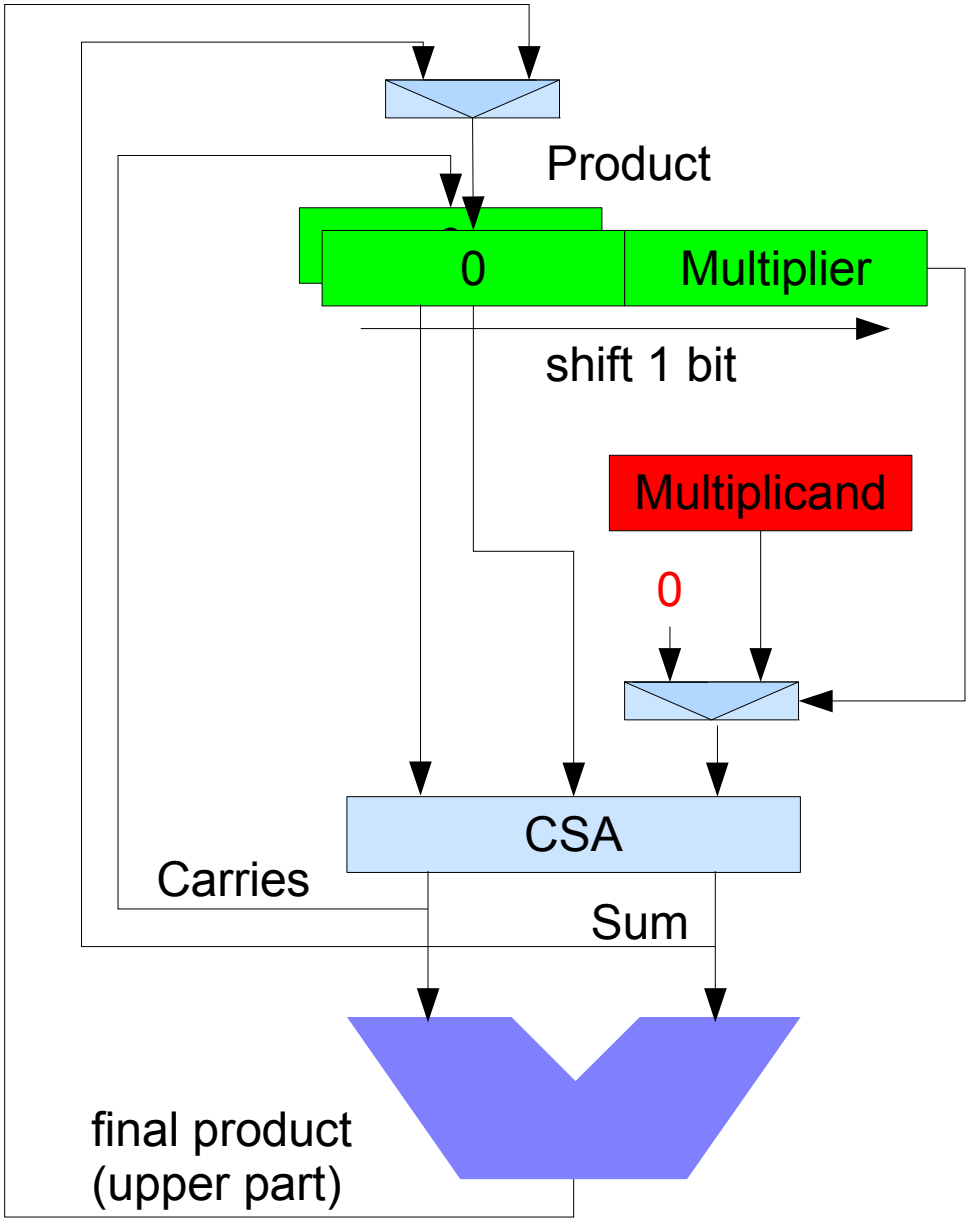
Improvement with CSA's

- Partial additions can be done with CSA,
  thus significantly faster, Θ(k) instead of Θ(k log k)
- Upper part of product must be kept as redundant number, proper for CSA operation
- Final conversion will require CPA once,
  for the upper part of product only



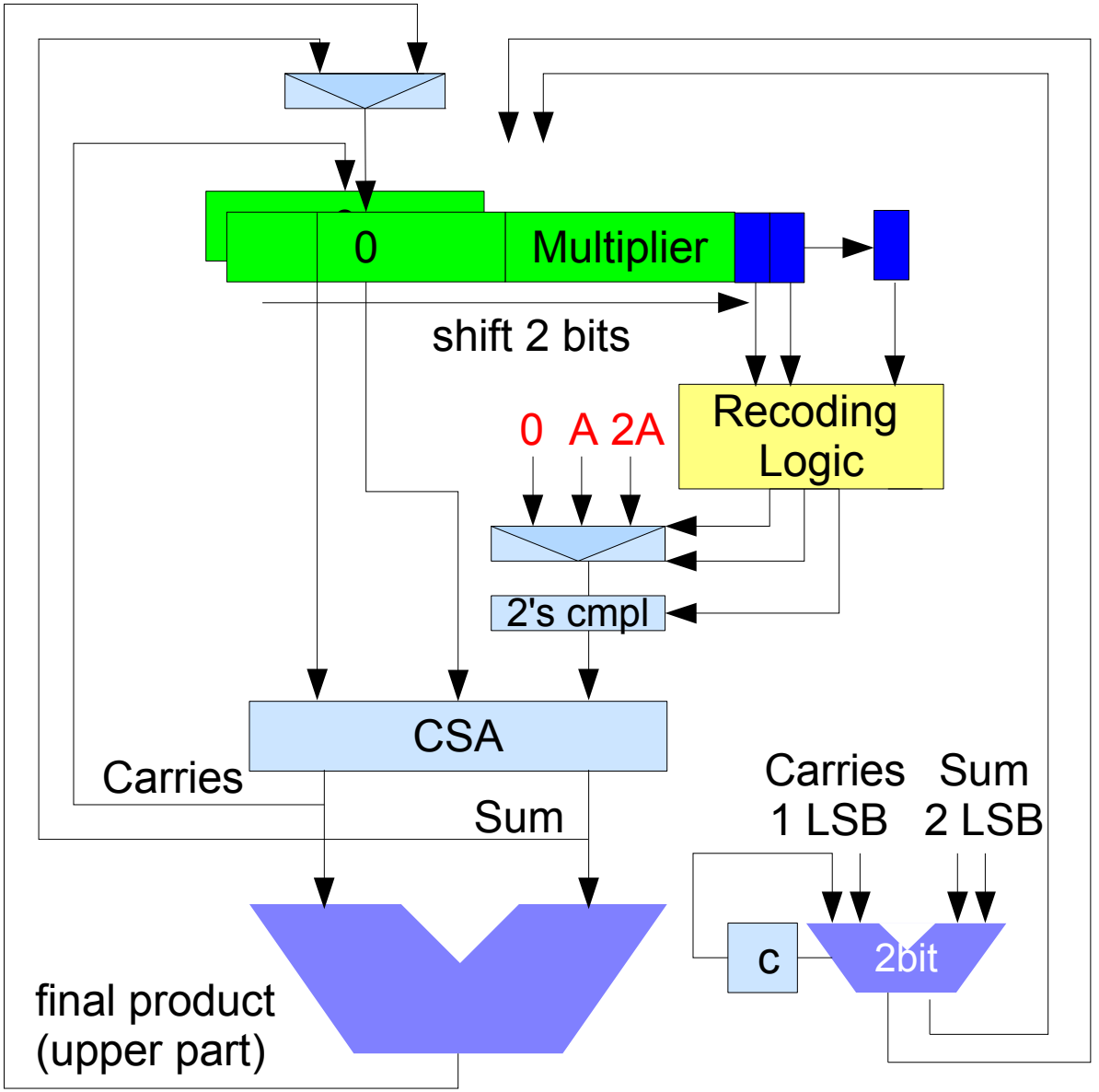
Fast Radix-2 Multiplier

- Few more resources compared to simple sequential architecture
- Latency Θ(k+log k) instead of Θ(k log k)



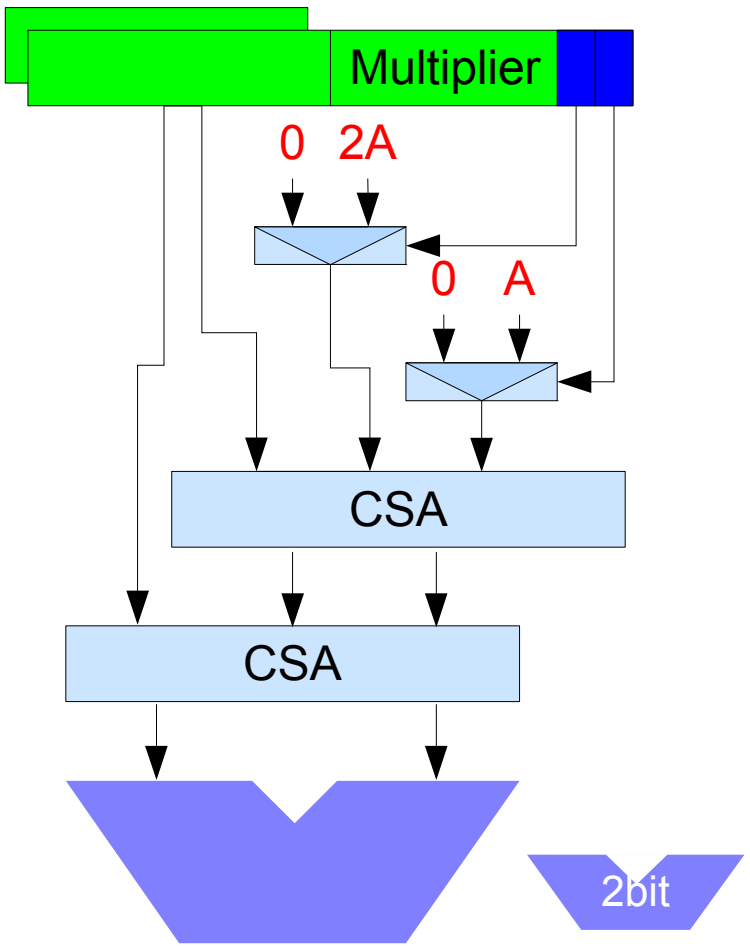
Fast Radix-4 Multiplier

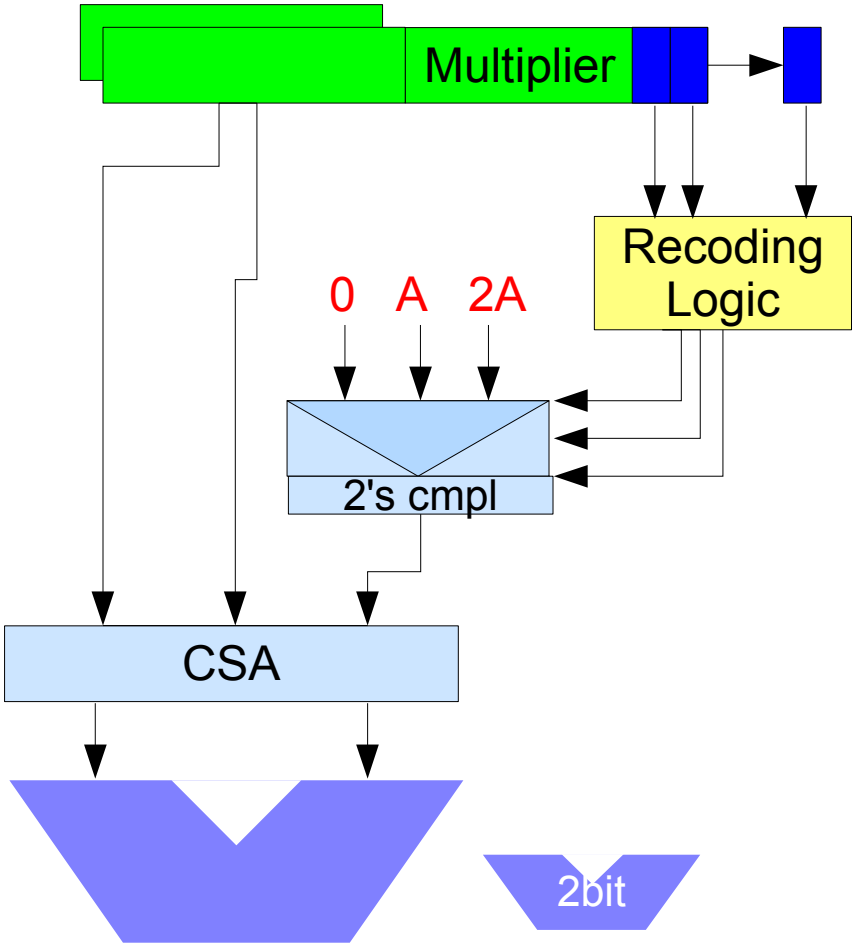
- Latency Θ(k+log k) but with radix-4 digits - fast
- Separate 2's compl. unit is needed
- Computation of shift-in 2-bits is required



Unsigned vs Signed Multiplication

- Partial Tree and Recoding Architectures
- Both architectures scale well for higher radices

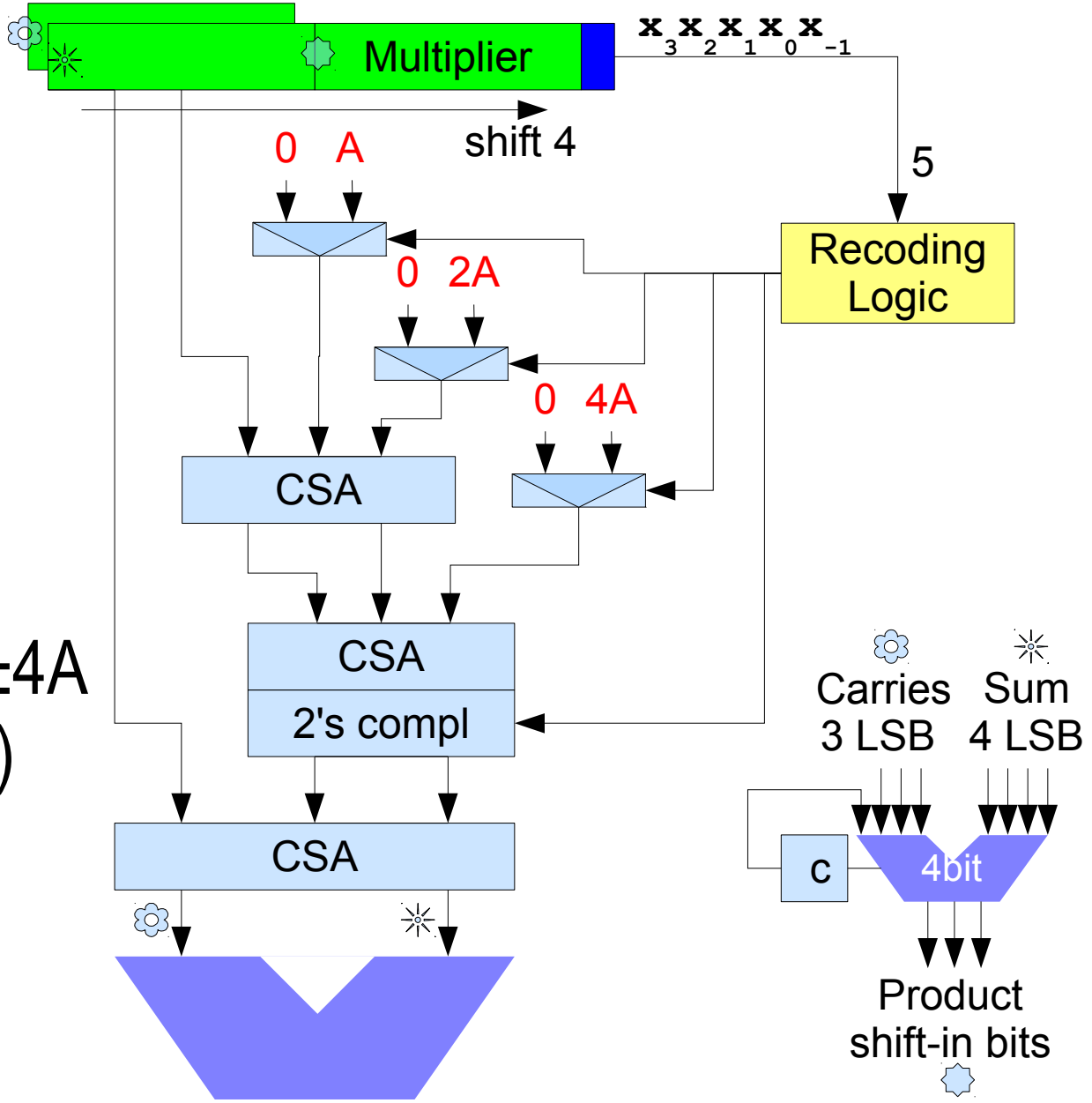




Radix-4 Multipliers

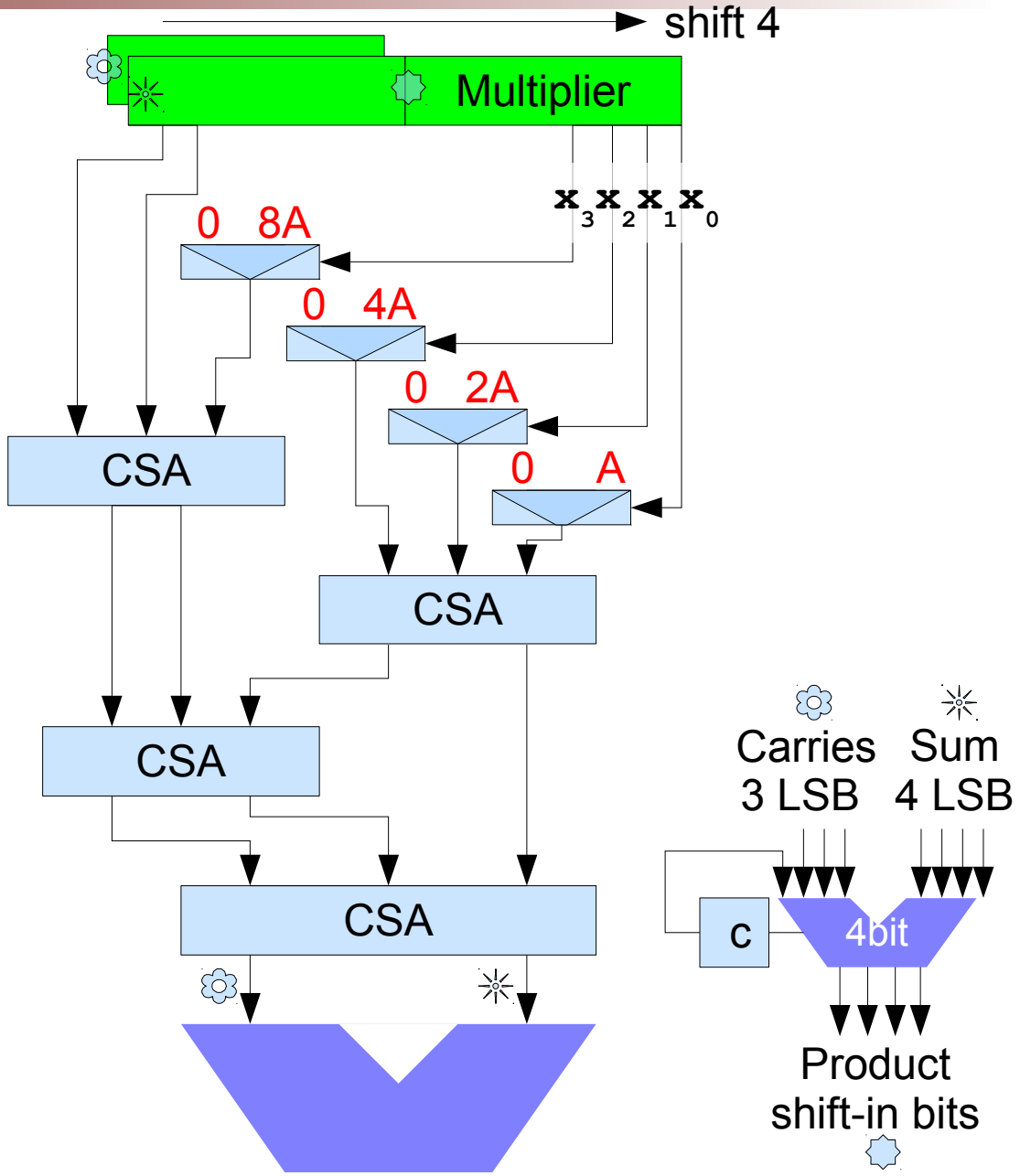
Radix-16 with Recoding – Example

- Shift by radix-16 digit (4 bits)
- Recoding use [-4,4] digit set (5 bit analysis)
- Multiples needed: 0,±1A,±2A,±3A,±4A (all with CSA tree)
- Signed arithmetic



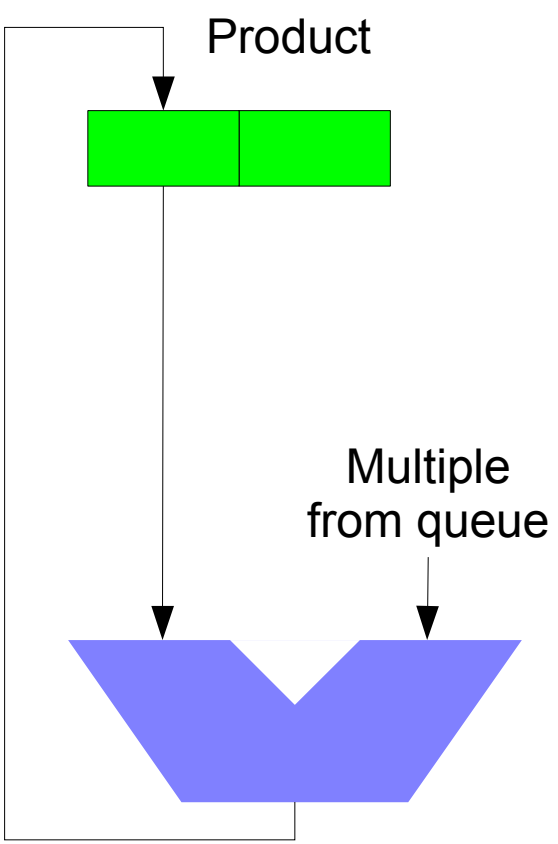
Radix-16 with Partial Tree – Example

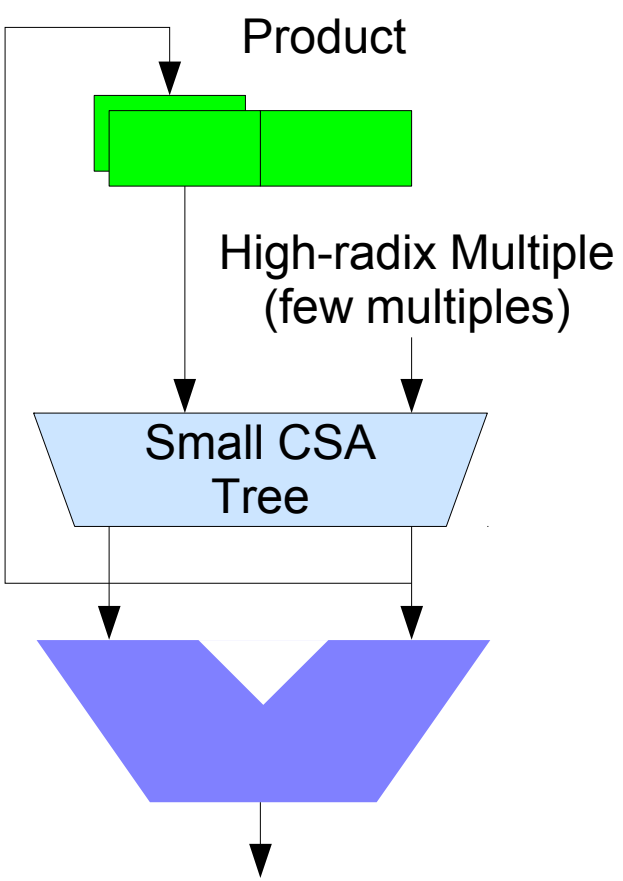
- Shift by radix-16 digit (4 bits)
- Multiples 0 – 15A generated with 3-level CSA tree
- Unsigned arithmetic

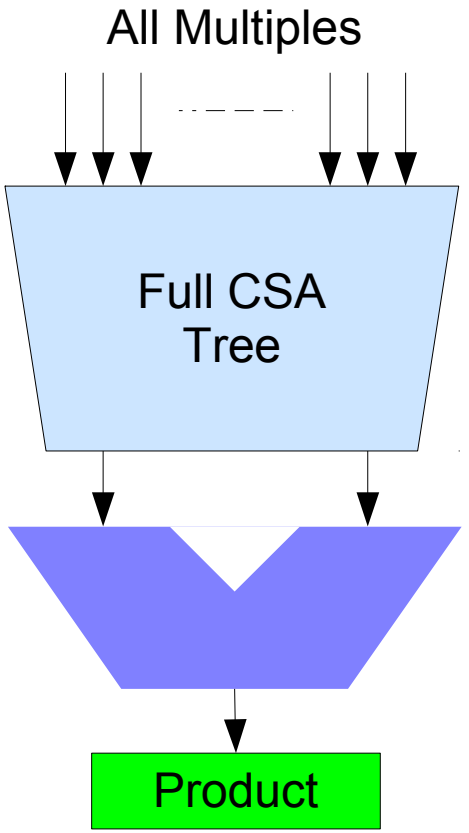


Multipliers – Big Picture

Basic radix-2 sequential → Speed up → High-radix or Partial Tree ← Economize ← Full Tree Multiplier



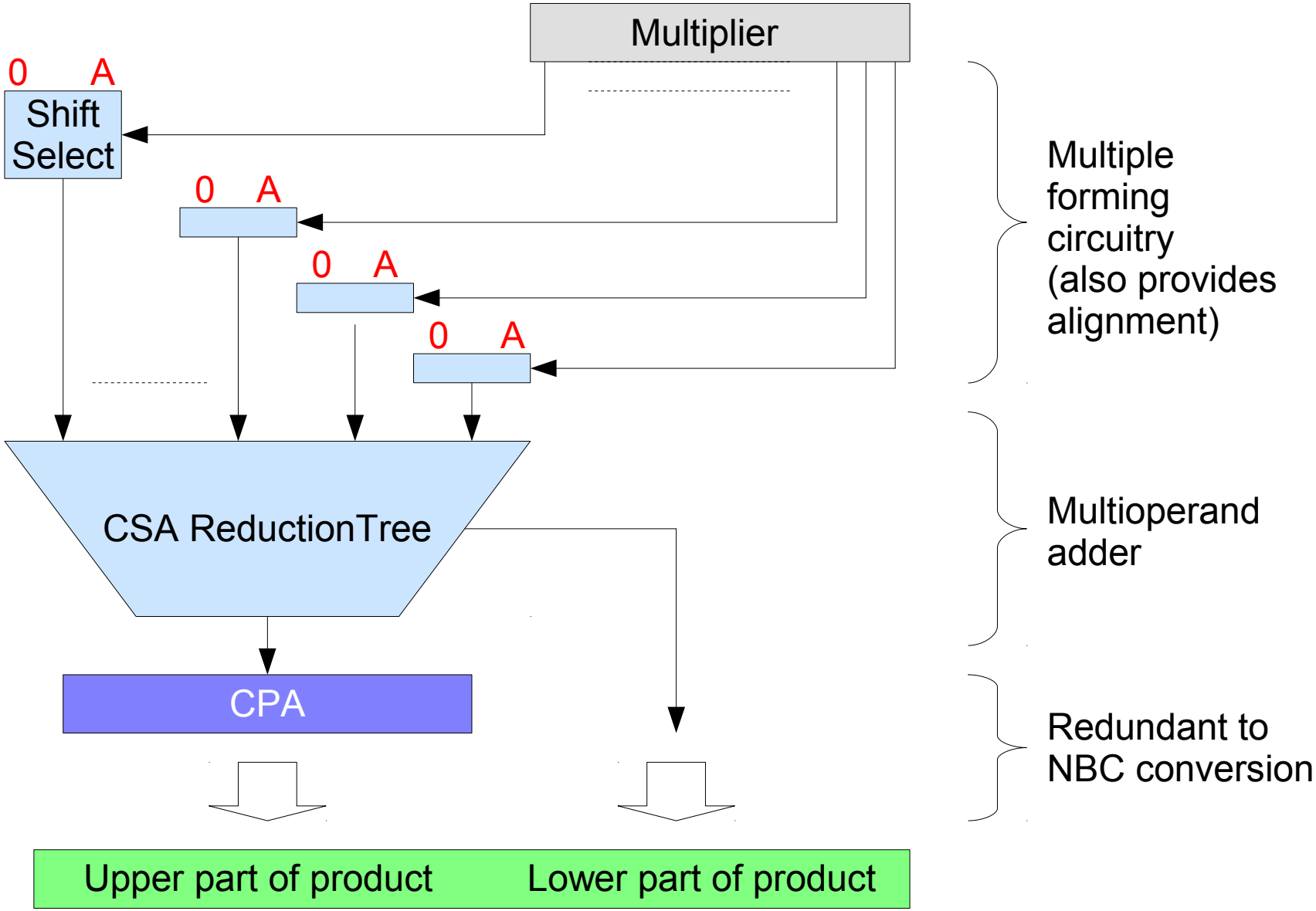




Full Tree Multipliers

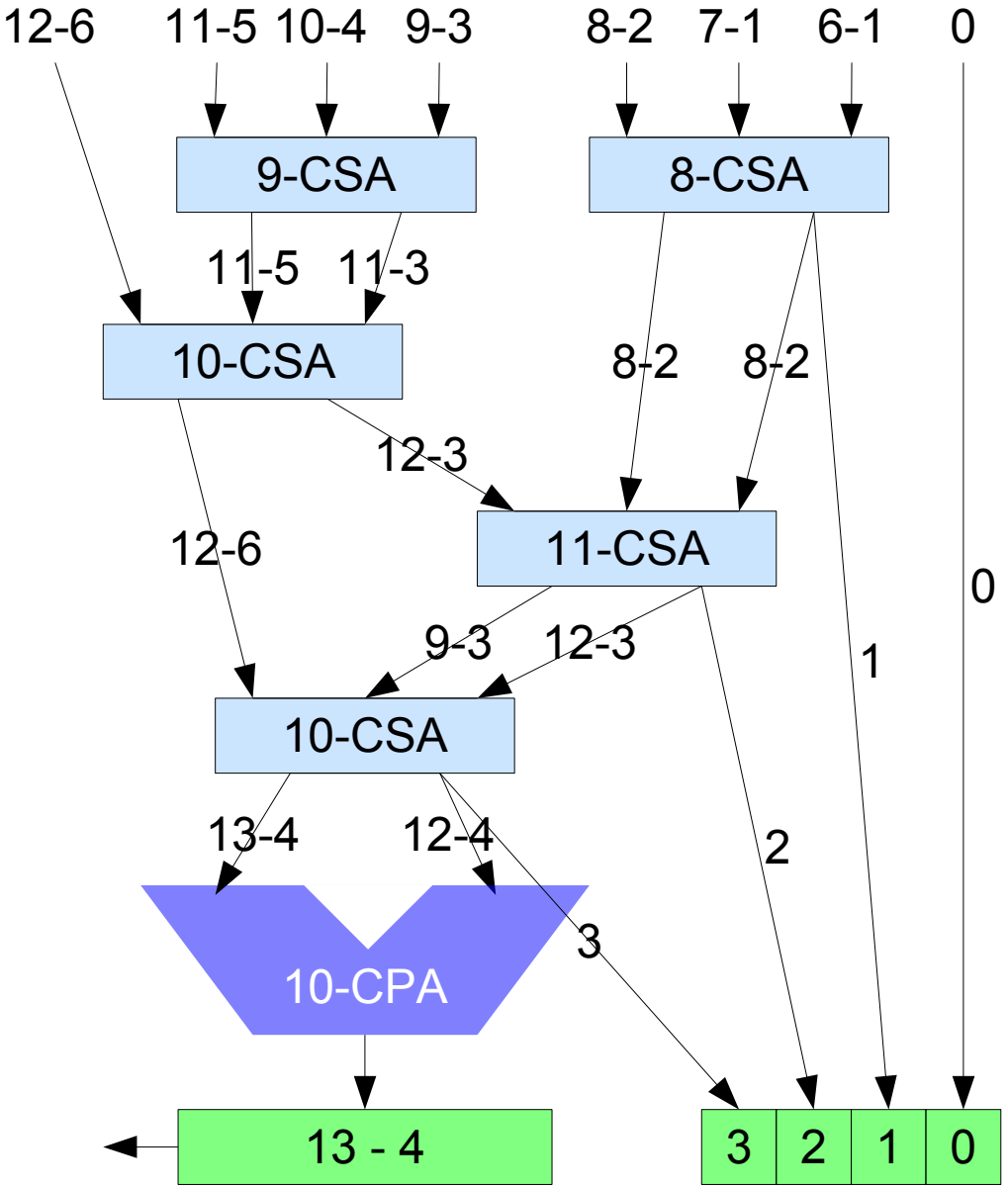
- All multiples provided at the same time
- Generation logic for high-radix multiples
- Reduction tree with CSA (or RCA)
- Final CPA (for part of product only)
- Irregular (Wallace & Dadda trees) or regular configurations
- Best performance: Θ(log k)
- Highest cost (size)
- Suitable for pipeline processing

General Structure



Example – 7x7 CSA Tree

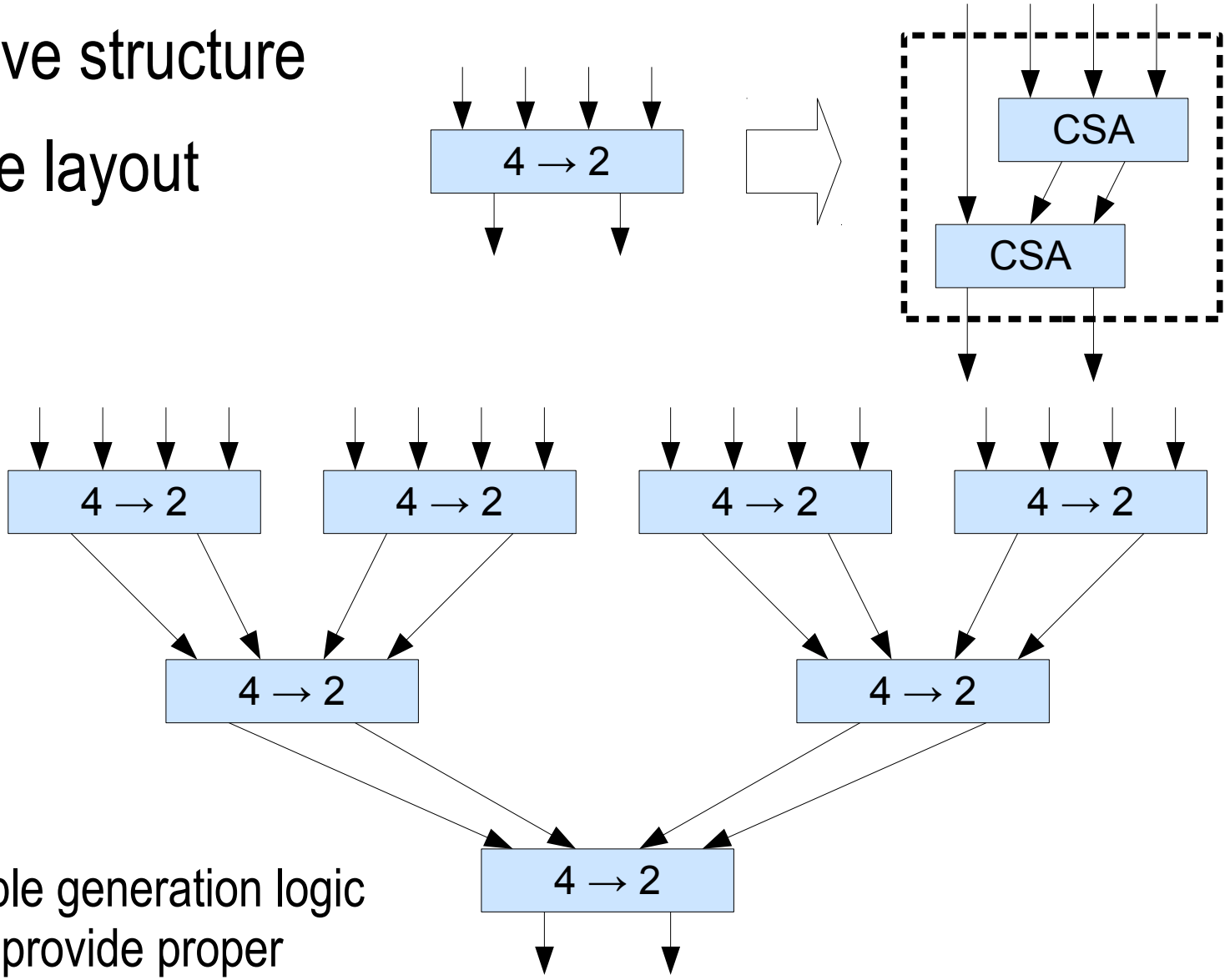
- Irregular reduction tree
- Variable-size CSA's
- Multiples by alignment of operands
- Lower bits of product available directly
- CPA only for upper part of product



Numbers indicate bit-positions of multiples, partial results and product

Regular Reduction Trees

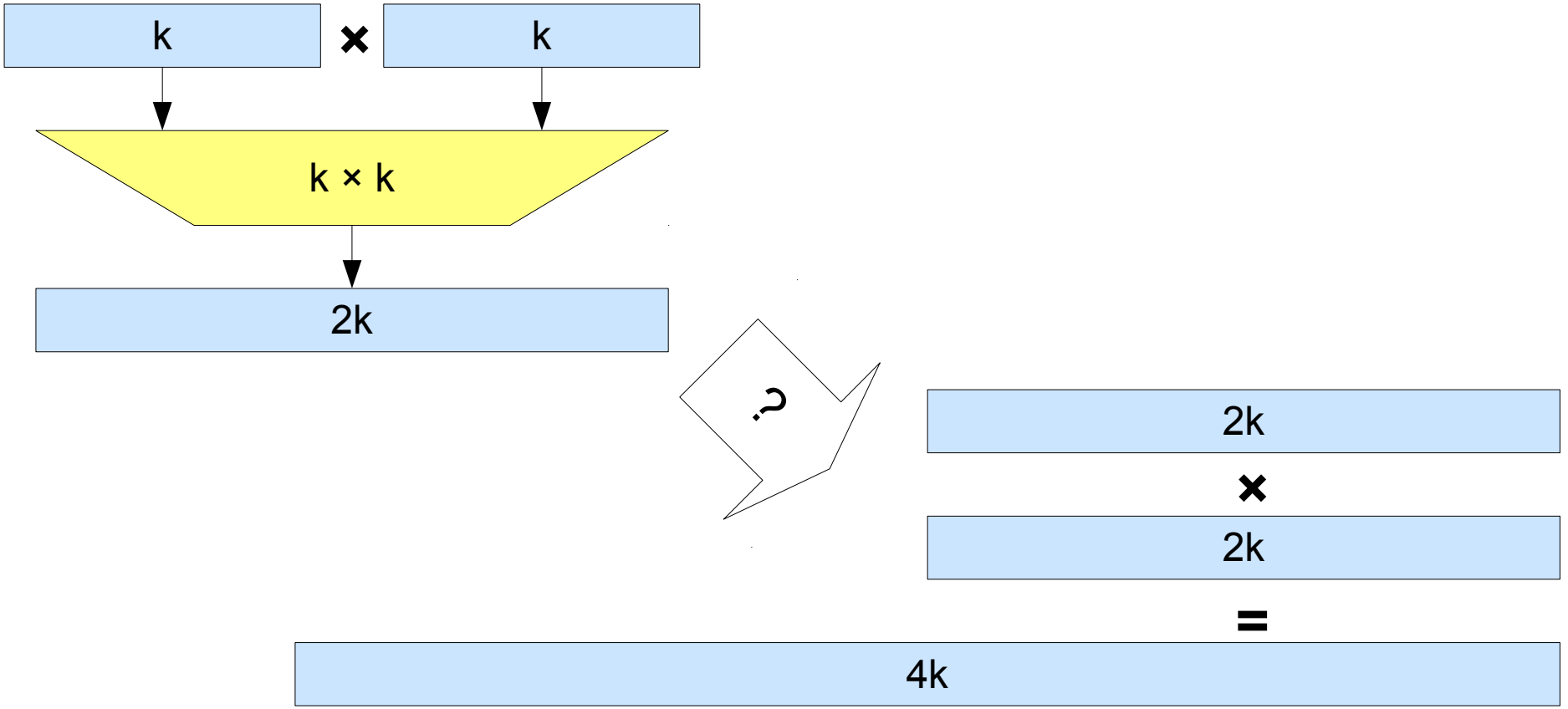
- Recursive structure
- Scalable layout



Multiple generation logic must provide proper operands for each block

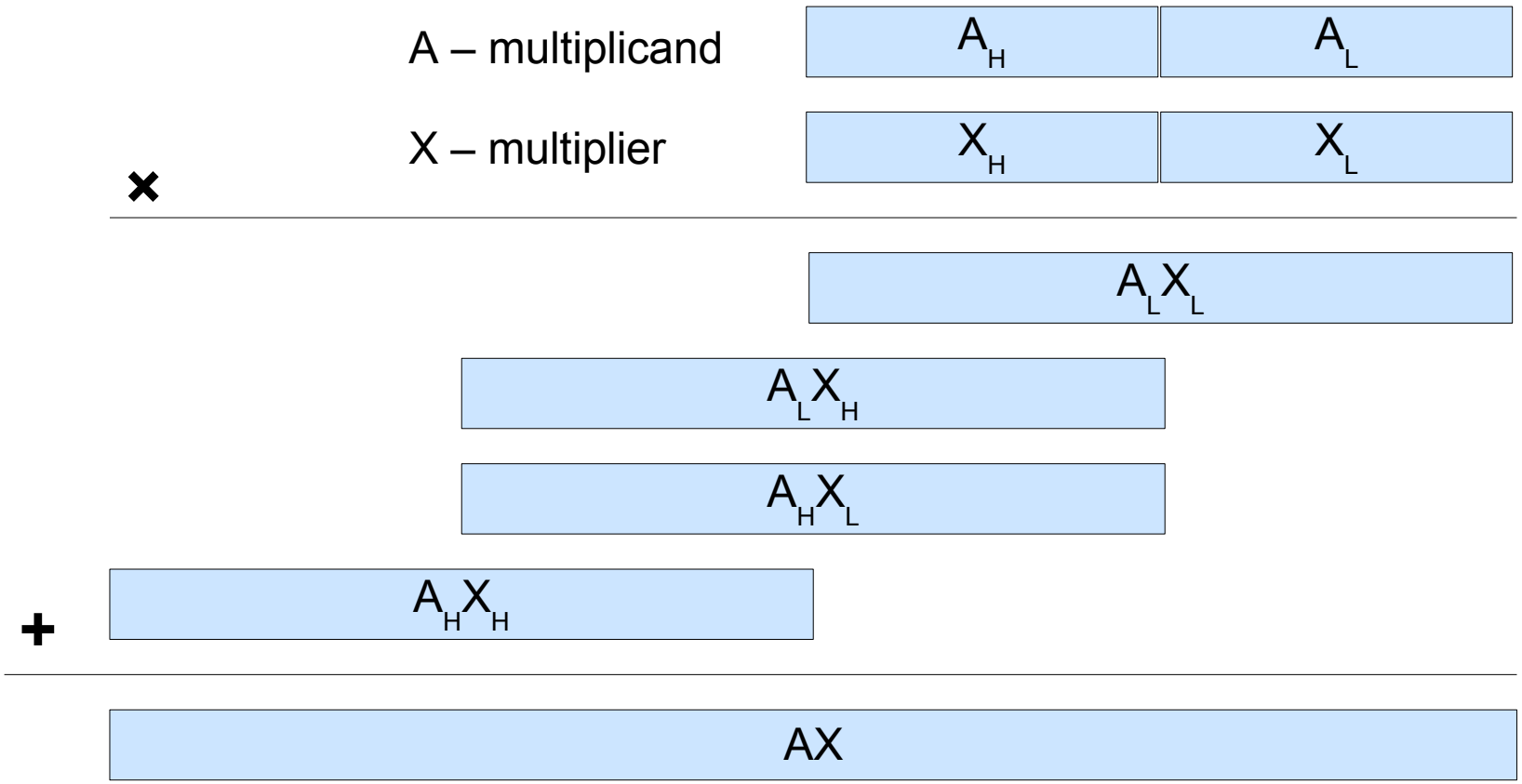
Multiplication of Long Numbers

- How to multiply long numbers with shorter multiplication units?
- e.g. 2k x 2k numbers with k-bit multiplier?



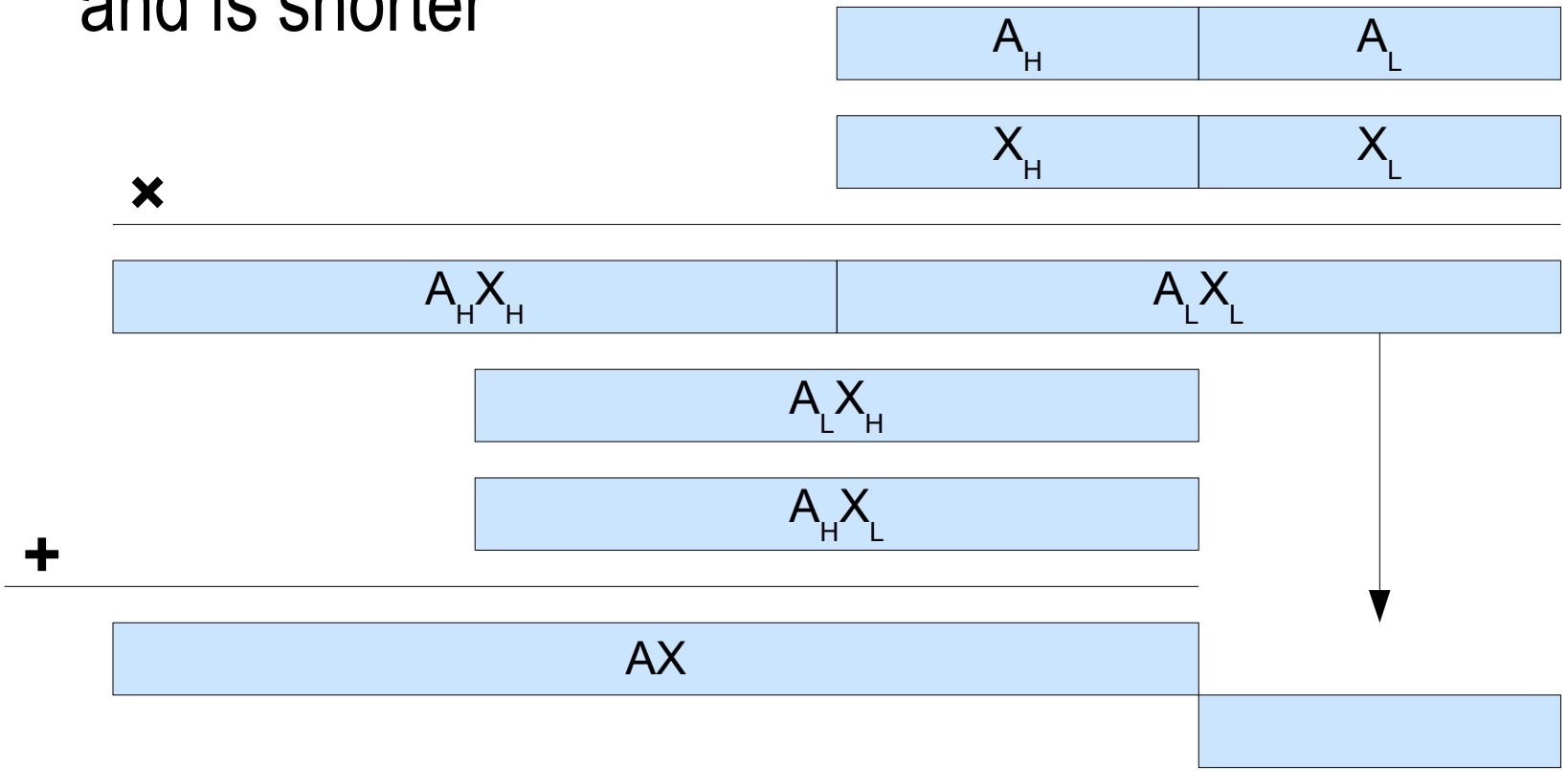
Divide & Conquer

- Split number into smaller parts (2k ⟶ 2 x k)
- Compute partial products (4 x 2k)
- Add the partial products (∑4 x 4k)

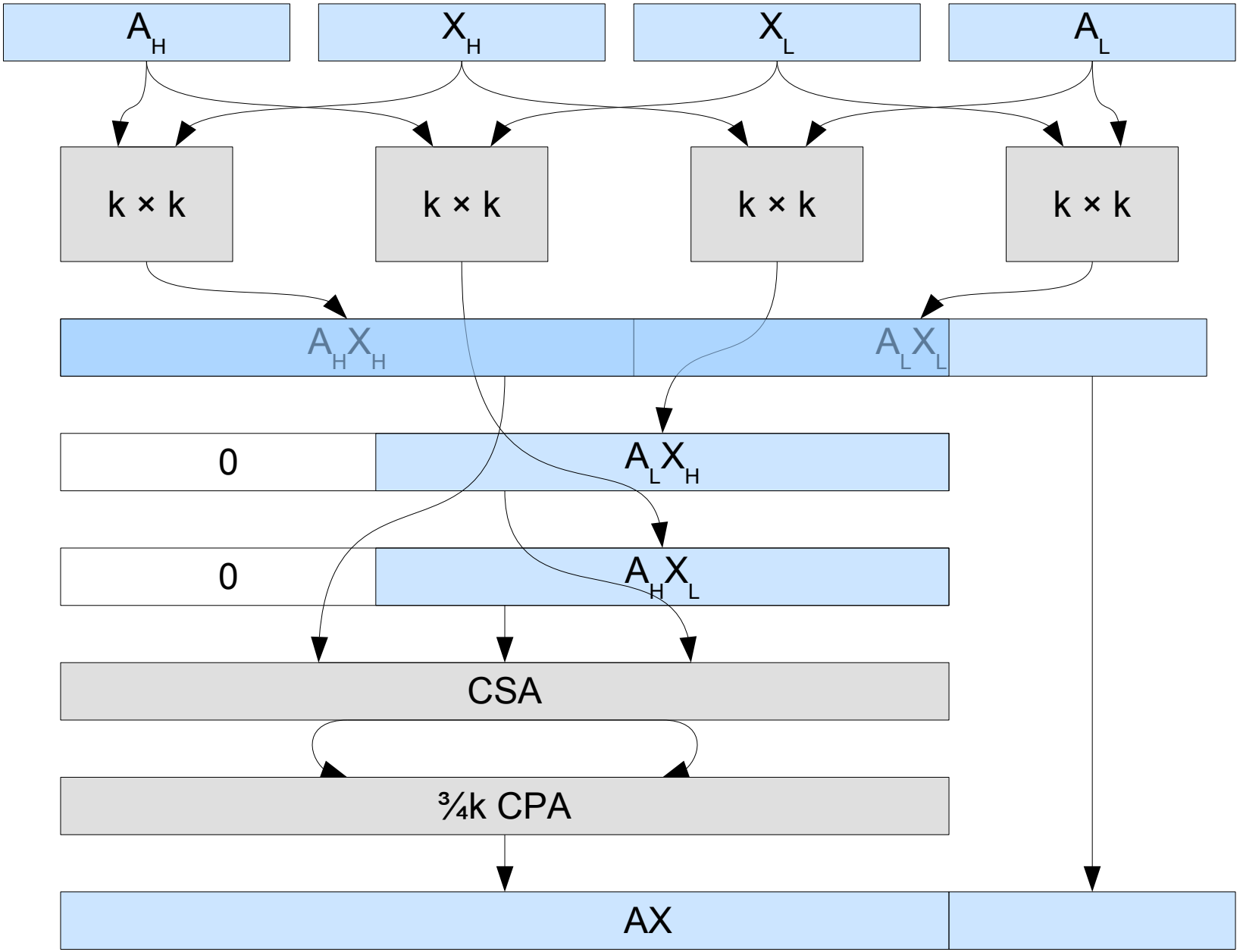


Divide & Conquer

- Least significant part of product is obtained directly
- There are non-overlapping partial products
- Final addition requires less partial products and is shorter

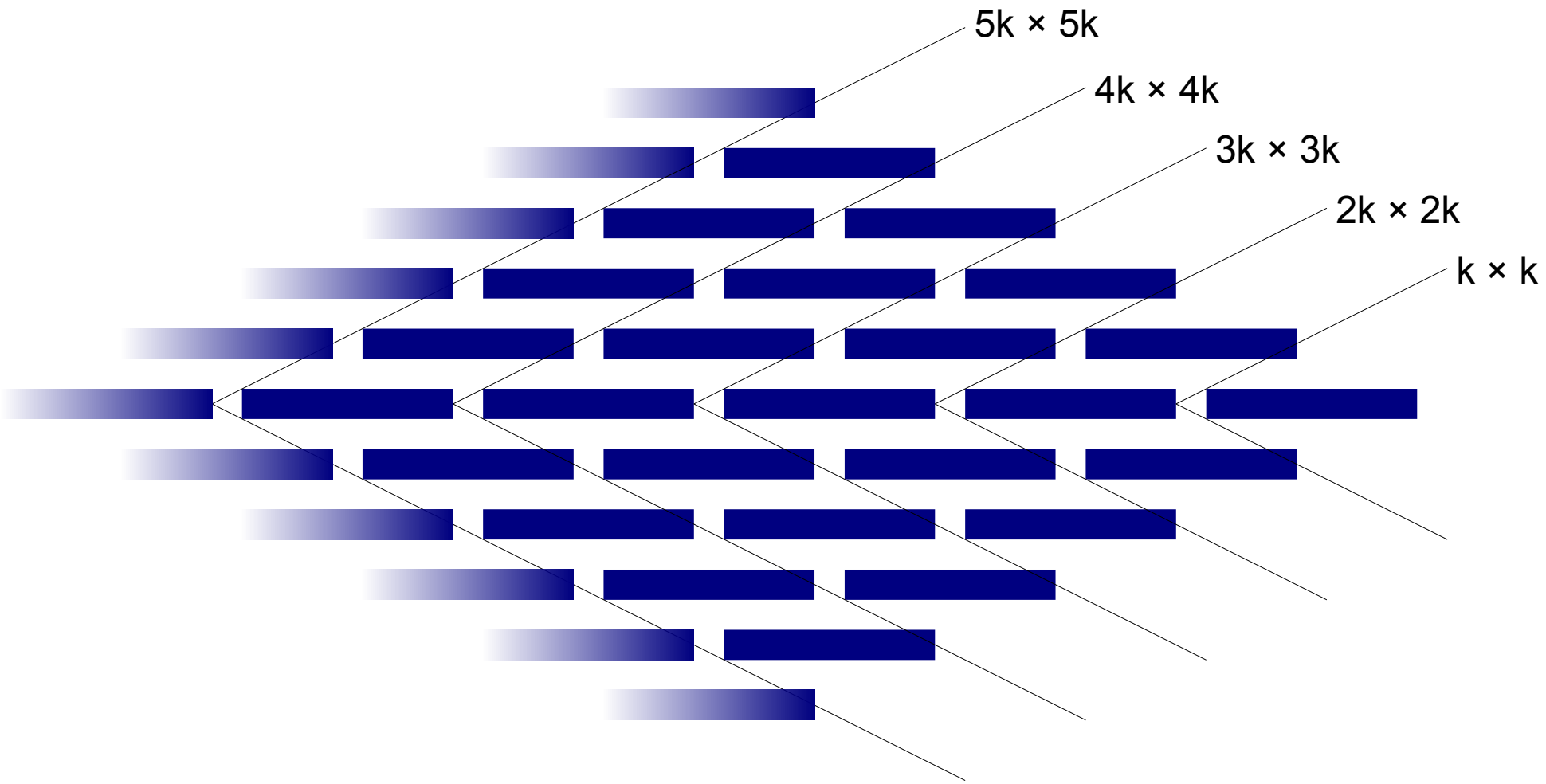


2k x 2k Divide & Conquer

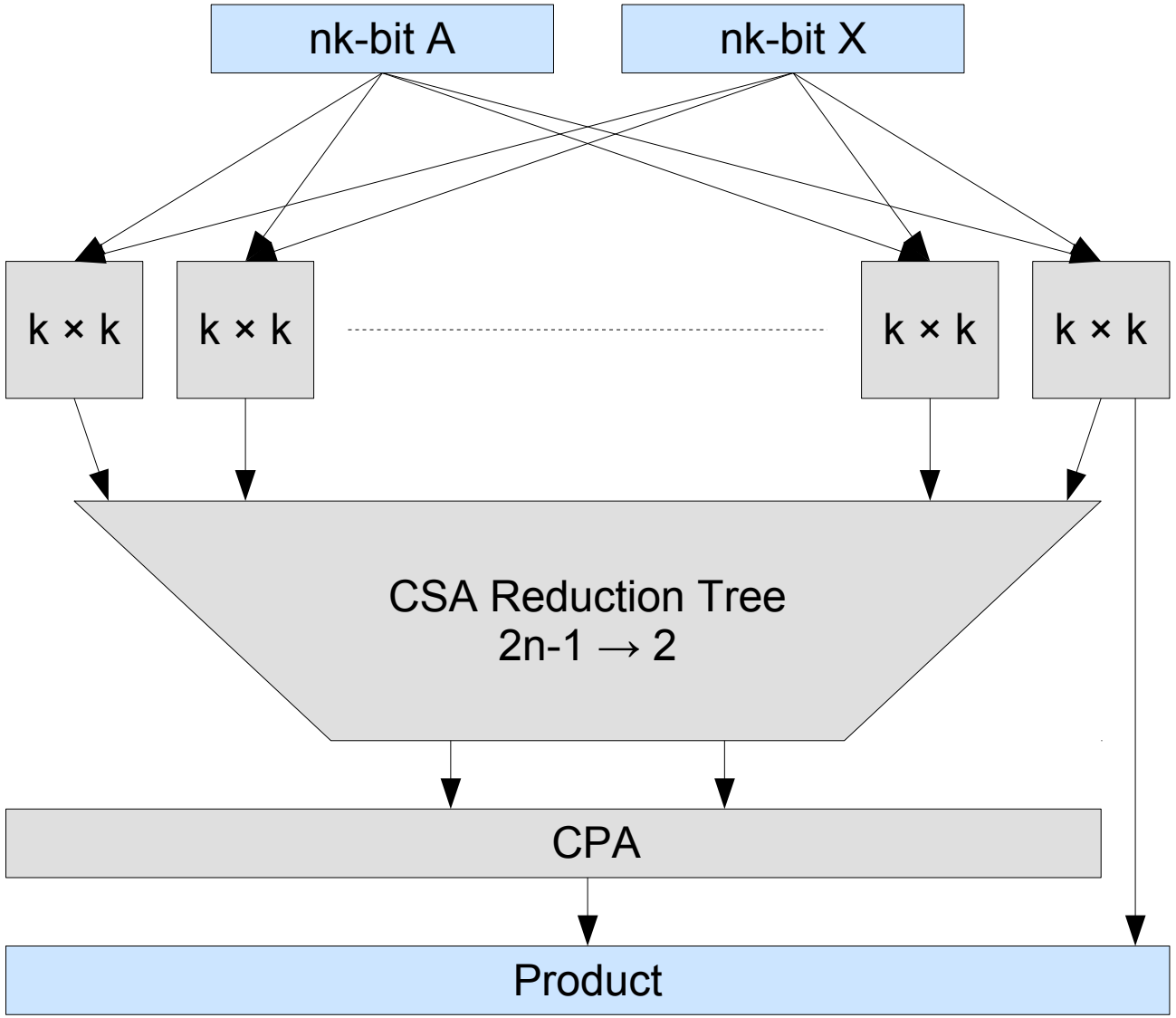


General Divide & Conquer

- Multiplication of two n×k-bit numbers with k-bit multipliers produces n×n partial products



General Divide & Conquer



n × n multipliers