DIGITAL SYSTEM DESIGN

MULTIPLIERS

WAQAR AHMAD

Faculty of Computer Science and Engineering

Fixed Point vs Floating Point

- Fixed Point
  - A number is represented in integer and fractional parts
  - The position of decimal point is fixed, i.e. range and accuracy is fixed

A 10 bit number $a_1a_2a_3a_4a_5a_6a_7a_8\boxed{.a_9a_{10}}$

Fixing the point

- Floating Point
  - A number is represented in terms of exponent and mantissa
  - The position of decimal point can vary, i.e. we can choose whether to have more range or more accuracy

0.123 may be represented as

$12.3 * 10^{-2}$

Number (Mantissa)

exponent

$1.23 * 10^{-1}$

Fixed Point $Q_{n.m}$ Format

- $Q_{n.m}$ simply means that a K-bit binary number has n bits to represent the integer part and m bits to represent fractional part.

- e.g ($a_1a_2a_3a_4a_5a_6a_7a_8.a_9a_{10}$) is a $Q_{8.2}$ number

- Generally, 2's compliment notation is used to represent the signed numbers

$Q_{n.m}$ Format

Examples

- 01 1101 0000

  in Q2.8 (signed/unsigned) format the number is

  $1 + 1/2 + 1/2^2 + 1/2^4 = 1.8125$

- 11 1101 0000

  in Q2.8 (unsigned) format the number is

  $2+1 + 1/2 + 1/2^2 + 1/2^4 = 3.8125$

- 11 1101 0000

  in Q2.8 (signed) format the number is

  $-2 + 1 + 1/2 + 1/2^2 + 1/2^4 = -0.1875$

Addition in $Q_{n.m}$ Format

- The addition of a $Q_{n1.m1}$ and a $Q_{n2.m2}$ number results in a $Q_{n.m}$ number , such that n is larger of n1 and n2 and m is larger of m1 and m2

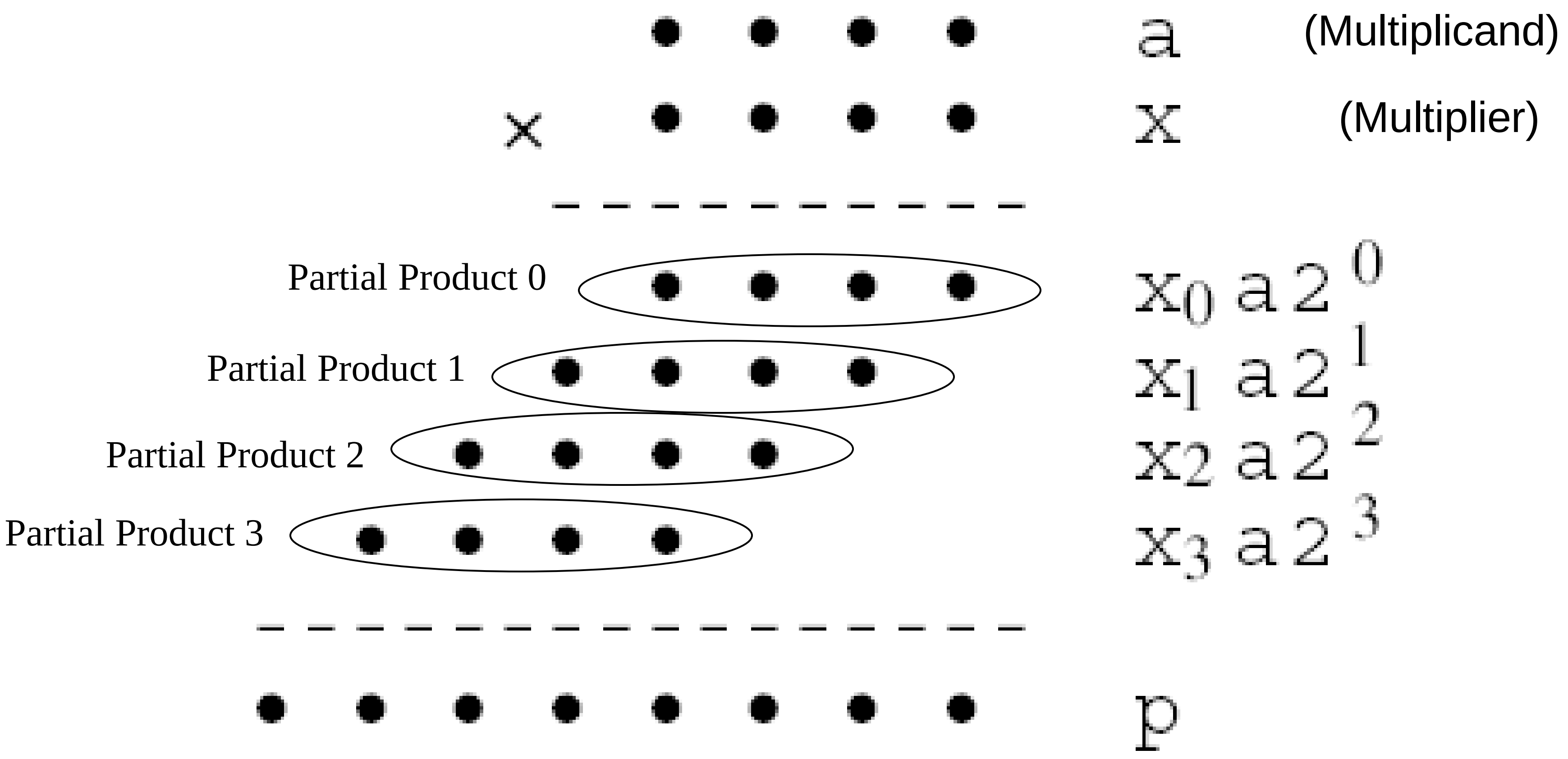
- For addition of two $Q_{n.m}$ numbers,
  - The decimal point should be matched at the same point.
  - If the format of two numbers is not the same then
    - The fractional part should be padded with 0's on the right side.
    - The integer part should be padded with 0's on left side for unsigned numbers
    - The integer part should be sign extended for signed numbers

$Q_{n1.m1}$ =	1	1	1	1	1	0	**0**	**0**	$Q_{2.2}$ = - 2 + 1 + 0.5 = -0.5
$Q_{n2.m2}$ =	0	1	1	1	0	1	1	0	$Q_{4.4}$ = 1 + 2+ 4 + 0.25 + 0.125 = 7.375
$Q_{n.m}$ =	0	1	1	0	1	1	1	0	$Q_{4.4}$ = 2 + 4+ 0.5 + 0.25 + 0.125 = 6.875

❝At least one good reason
for studying multiplication and division is that
there is an infinite number of ways
of performing these operations
and hence there is an infinite number of PhDs
(or expenses-paid visits to conferences in the USA)
to be won from inventing new forms of multiplier.❞

Alan Clements
The Principles of Computer Hardware, 1986

Binary Multiplication



Number of partial products = number of bits in multiplier x

Bit-width of each partial product = bit-width of multiplicand a

Multiplication in $Q_{n.m}$ Format

- Four cases
  - Unsigned by Unsigned
  - Signed by Unsigned
  - Unsigned by Signed

$Q_{n1.m1}$	(Multiplicand)
$Q_{n2.m2}$	(Multiplier)
$Q_{(n1+n2).(m1+m2)}$	(Result)

  - Signed by Signed ?

Unsigned by Unsigned Multiplication

- Both the multiplicand and multiplier are unsigned numbers

- Add the partial products in a manner similar to simple multiplication. i.e. Shift each partial product by one bit left and then add

```
        1 1 0 1 = 11.01 in Q2.2 = 3.25
        1 0 1 1 = 10.11 in Q2.2 = 2.75
      ---------
        1 1 0 1
      1 1 0 1 X
    0 0 0 0 X X
  1 1 0 1 X X X
---------------
1 0 0 0 1 1 1 1= 1000.1111 in Q4.4 i.e.8.9375
```

Signed by Unsigned Multiplication

- Multiplicand is signed and Multiplier is unsigned number
- Sign extend each partial product and then add
- Add an extra bit for each partial product

```
        1 1 0 1 = 11.01 in Q2.2 = -0.75
        0 1 0 1 = 01.01 in Q2.2 = 1.25
      ---------
1 1 1 1 1 1 0 1 extended sign bits shown in bold
0 0 0 0 0 0 0 X
1 1 1 1 0 1 X X
0 0 0 0 0 X X X
---------------
1 1 1 1 0 0 0 1 = 1111.0001 in Q4.4 i.e.-0.9375
```

Unsigned by Signed Multiplication

- Multiplicand is unsigned and Multiplier is signed number
- No need to sign extend the partial products.
- For the last partial product, take 2’s compliment of the multiplicand and then simply add all the partial products

```
      1 0 0 1 = 10.01 in Q2.2 = 2.25 (unsigned)
      1 1 0 1 = 11.01 in Q2.2 = -0.75 (signed)
    ---------
      1 0 0 1
    0 0 0 0 X
  1 0 0 1 X X
1 0 1 1 1 X X X 2's compliment of the positive multiplicand 01001
---------------
1 1 1 0 0 1 0 1  = 1110.0101 in Q4.4 i.e.-1.6875
```

Signed by Signed Multiplication

- Multiplicand and Multiplier both are signed numbers
- A redundant sign bit is produced always
  - Can be removed by shifting the result one bit left

$Q_{n1.m1}$	(Multiplicand)
$Q_{n2.m2}$	(Multiplier)
$Q_{(n1+n2).(m1+m2)}$	(Left shift by one bit)
$Q_{(n1+n2-1).(m1+m2+1)}$	(Result)

Signed by Signed Multiplication

- If Multiplier is signed positive number
  - Add an additional bit to each partial product
  - Sign extend each partial product
  - Add all the partial products
  - Left shift the result, discarding the redundant bit, to get the correct result

```
          1 1 0 = Q1.2 = -0.5 (signed)
          0 1 0 = Q1.2 = 0.5 (signed)
        -------
    0 0 0 0 0 0
    1 1 1 1 0 X
    0 0 0 0 X X
  -------------
    1 1 1 1 0 0 = Q1.5 format 1_11000=-0.25
```

Signed by Signed Multiplication

- If Multiplier is signed negative number
  - Add an additional bit to each partial product
  - Sign extend each partial product
  - For the last partial product, take 2’s compliment of the multiplicand and then simply add all the partial products
  - Add all the partial products
  - Left shift the result, discarding the redundant bit, to get the correct result

```
        1 1. 0 1 = -0.75 in Q2.2 format
        1. 1 0 1 = -0.375 in Q1.3 format
      -----------
  1 1 1 1 1 1 0 1
  0 0 0 0 0 0 0 X
  1 1 1 1 0 1 X X
  0 0 0 1 1 X X X
 -----------------
  0 0 0 0 1 0 0 1 = shifting left by one 00.010010 in Q2.6 format is 0.28125
```

Truncation in $Q_{n.m}$ Format

- As we have seen that the product size of two Q format numbers will be double the size of the original number
- We may need to sacrifice on precision by truncating some low precision bits of the product

- Simple truncation

0111 0111	in Q4.4 is	7.4375
	Truncated to Q4.2 gives	
0111_01		7.25

Truncation in $Q_{n.m}$ Format

- Rounding off followed by Truncation
  - Direct truncation may provide bias in results
  - Not suitable for sensitive applications

  - Thus, first round and then truncate technique may be used

  - Add a ‘1’ to the bit on the right side of truncation point
  - Then truncate the result

Truncate Here

	0111 0111	in $Q_{4.4}$ is	7.4375
Rounding	+1		
	0111 1001		
Now truncate to $Q_{4.2}$	0111_10		7.5

HARDWARE MULTIPLIERS

Shift and Add Algorithms

Right Shift

- For each bit in multiplier, (Starting from LSB)
  - If bit = 1
    - Add multiplicand to result
    - Shift result right by 1
  - If bit = 0
    - Shift result right by 1

Left Shift

- For each bit in multiplier, (Starting from MSB)
  - If bit = 1
    - Shift result left by 1
    - Add multiplicand to result
  - If bit = 0
    - Shift result left by 1

Shift and Add Algorithms

```
a          1 0 1 0
x          1 0 1 1
==============
           1 0 1 0
         1 0 1 0 -
       0 0 0 0 - -
     1 0 1 0 - - -
==============
   0 1 1 0 1 1 1 0
```

Check:
10 × 11
= 110
= 64 + 32 +
8 + 4 + 2

Right-shift algorithm

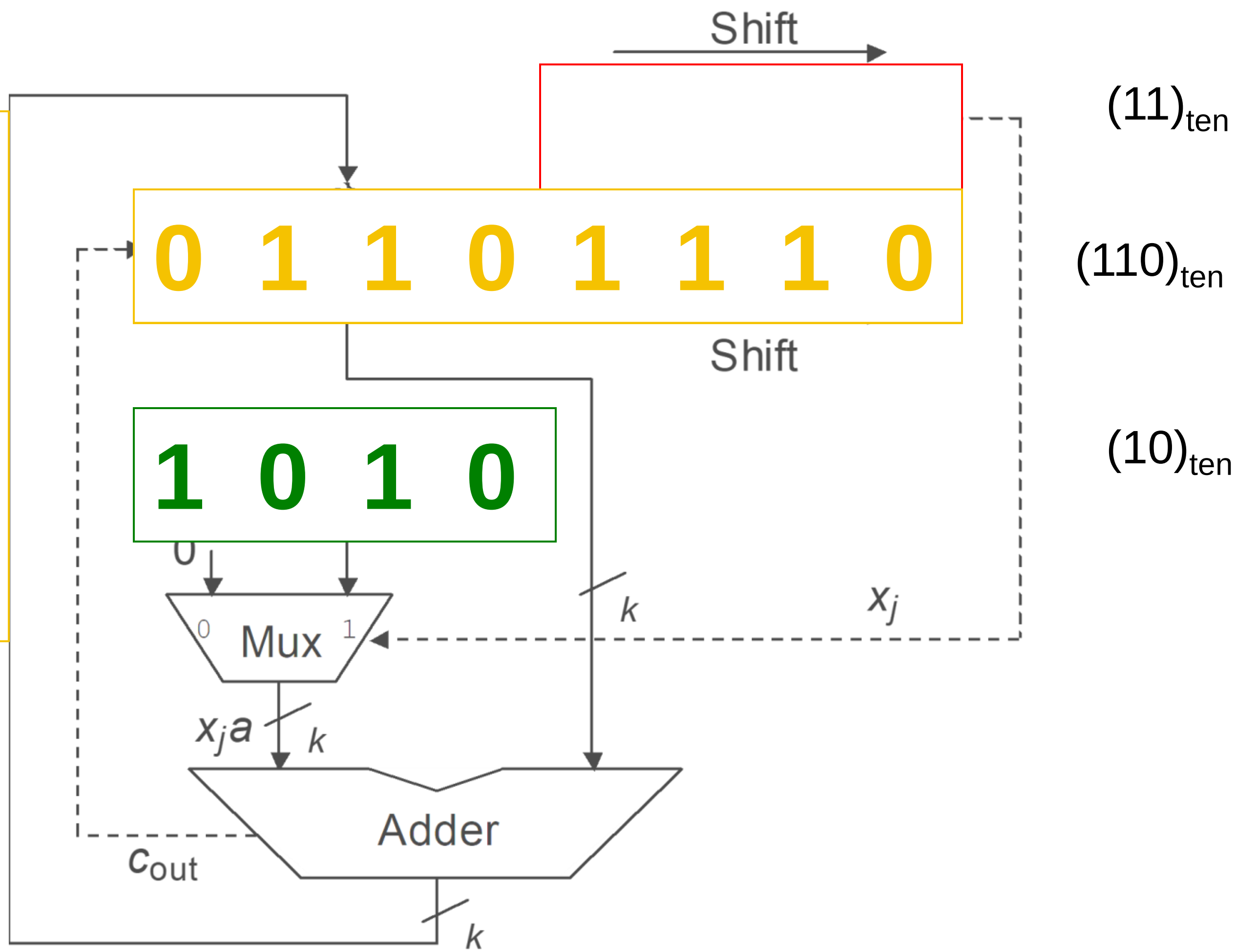
a	1 0 1 0	
x	1 0 1 1	
$p^{(0)}$	0 0 0 0	
$+x_0a$	1 0 1 0	
$2p^{(1)}$	0 1 0 1 0	
$p^{(1)}$	0 1 0 1	0
$+x_1a$	1 0 1 0	
$2p^{(2)}$	0 1 1 1 1	0
$p^{(2)}$	0 1 1 1	1 0
$+x_2a$	0 0 0 0	
$2p^{(3)}$	0 0 1 1 1	1 0
$p^{(3)}$	0 0 1 1	1 1 0
$+x_3a$	1 0 1 0	
$2p^{(4)}$	0 1 1 0 1	1 1 0
$p^{(4)}$	0 1 1 0	1 1 1 0

Left-shift algorithm

a		1 0 1 0
x		1 0 1 1
$p^{(0)}$		0 0 0 0
$2p^{(0)}$	0	0 0 0 0
$+x_3a$		1 0 1 0
$p^{(1)}$	0	1 0 1 0
$2p^{(1)}$	0 1	0 1 0 0
$+x_2a$		0 0 0 0
$p^{(2)}$	0 1	0 1 0 0
$2p^{(2)}$	0 1 0	1 0 0 0
$+x_1a$		1 0 1 0
$p^{(3)}$	0 1 1	0 0 1 0
$2p^{(3)}$	0 1 1 0	0 1 0 0
$+x_0a$		1 0 1 0
$p^{(4)}$	0 1 1 0	1 1 1 0

Right Shift and Add (Example)

```
a           1 0 1 0
x           1 0 1 1
==============
            1 0 1 0
          1 0 1 0 -
        0 0 0 0 - -
      1 0 1 0 - - -
==============
   0 1 1 0 1 1 1 0
```



Hardware realization of the sequential multiplication algorithm with additions and right shifts.

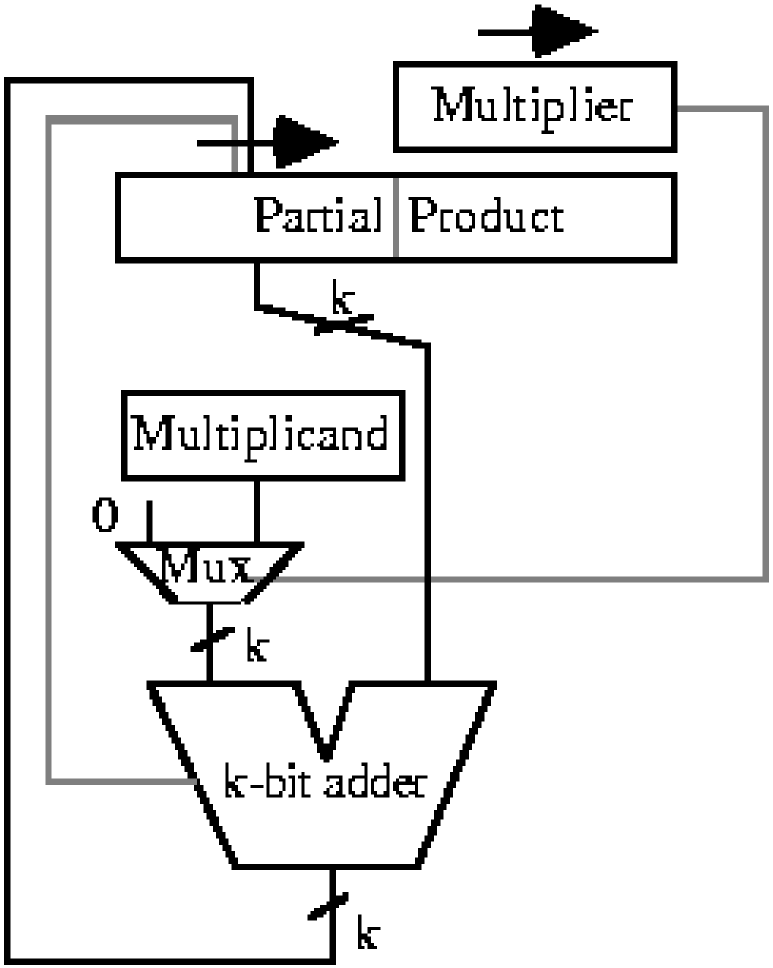
Hardware to Multiply

For 32 x 32 multiplication

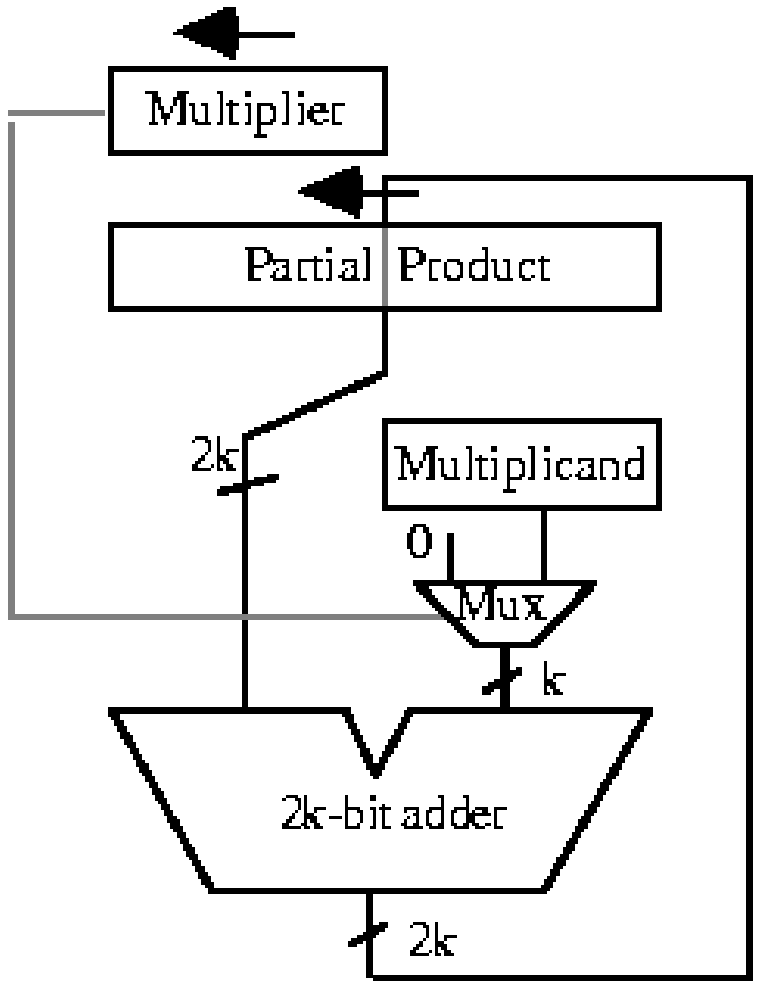
- 64-bit register for product
- 32-bit register for multiplier
  - Shift right after each step (low bits fall off)
- 32-bit register for multiplier
- 32-bit ALU
  - Add multiplier/zeroes to upper-half of product
- Control hardware

32 shift and add cycles

Hardware Shift and Add (Right)



Hardware Shift and Add (Left)



Sequential Multiplication of 2's Compliment Numbers

Positive Multiplier

a	1 0 1 1 0	
x	0 1 0 1 1	
$p^{(0)}$	0 0 0 0 0	
$+x_0a$	1 0 1 1 0	
$2p^{(1)}$	1 1 0 1 1 0	
$p^{(1)}$	1 1 0 1 1	0
$+x_1a$	1 0 1 1 0	
$2p^{(2)}$	1 1 0 0 0 1	0
$p^{(2)}$	1 1 0 0 0	1 0
$+x_2a$	0 0 0 0 0	
$2p^{(3)}$	1 1 1 0 0 0	1 0
$p^{(3)}$	1 1 1 0 0	0 1 0
$+x_3a$	1 0 1 1 0	
$2p^{(4)}$	1 1 0 0 1 0	0 1 0
$p^{(4)}$	1 1 0 0 1	0 0 1 0
$+x_4a$	0 0 0 0 0	
$2p^{(5)}$	1 1 1 0 0 1	0 0 1 0
$p^{(5)}$	1 1 1 0 0	1 0 0 1 0

Check:
$^{-}10 \times 11$
$= {}^{-}110$
$= {}^{-}512 + 256 + 128 + 16 + 2$

Sequential Multiplication of 2's Compliment Numbers

Negative Multiplier

a	1 0 1 1 0	
x	1 0 1 0 1	
$p^{(0)}$	0 0 0 0 0	
$+x_0a$	1 0 1 1 0	
$2p^{(1)}$	1 1 0 1 1 0	
$p^{(1)}$	1 1 0 1 1	0
$+x_1a$	0 0 0 0 0	
$2p^{(2)}$	1 1 1 0 1 1	0
$p^{(2)}$	1 1 1 0 1	1 0
$+x_2a$	1 0 1 1 0	
$2p^{(3)}$	1 1 0 0 1 1	1 0
$p^{(3)}$	1 1 0 0 1	1 1 0
$+x_3a$	0 0 0 0 0	
$2p^{(4)}$	1 1 1 0 0 1	1 1 0
$p^{(4)}$	1 1 1 0 0	1 1 1 0
$+(-x_4a)$	0 1 0 1 0	
$2p^{(5)}$	0 0 0 1 1 0	1 1 1 0
$p^{(5)}$	0 0 0 1 1	0 1 1 1 0

Check:
$^{-}10 \times {}^{-}11$
$= 110$
$= 64 + 32 + 8 + 4 + 2$

Parallel Multiplication

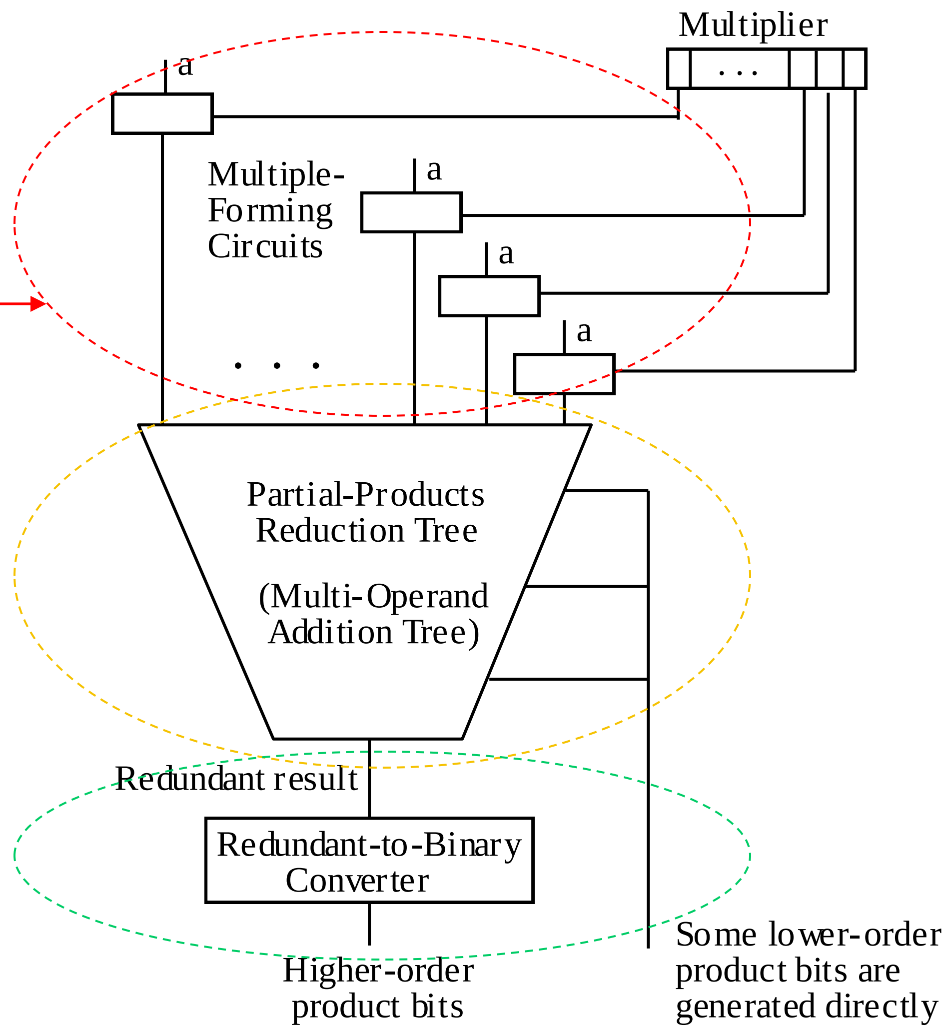
- Instead of 32 cycles, use 32 adders
- Each adds 0 or multiplicand
- Arrange in tree so results cascade properly
- Result pulls each bit from the appropriate adder

Full Tree Architecture

1. Multiple-forming circuits (Partial Product Generation)

2. Partial products reduction tree

3. Redundant-to-binary converter



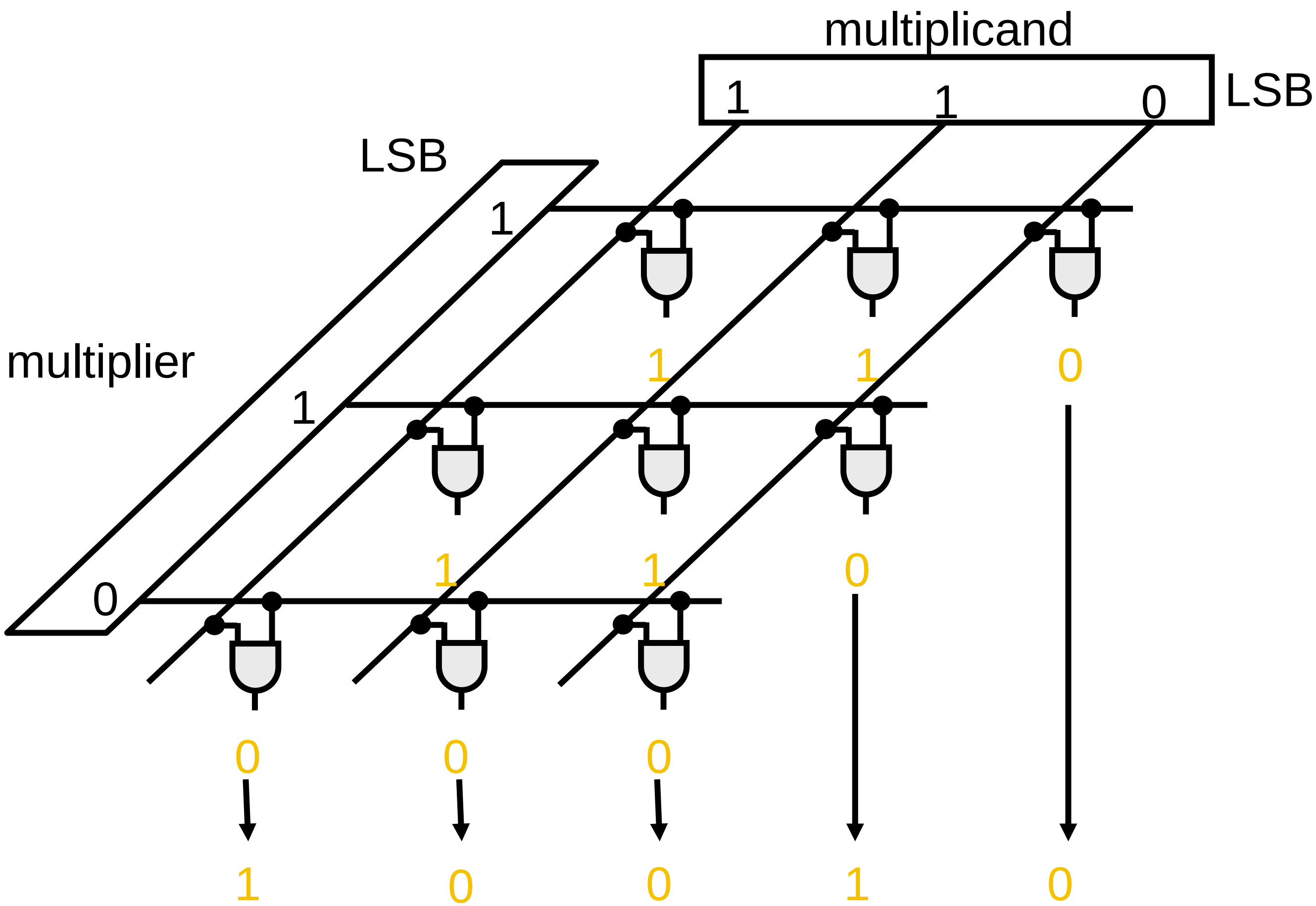
Array/Combinational Multipliers

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		partial product 1
			C_{12}	C_{11}	C_{10}			1st row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		1st row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			partial product 2
		C_{22}	C_{21}	C_{20}				2nd row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			2nd row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				partial product 3
	C_{32}	C_{31}	C_{30}					3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	final product

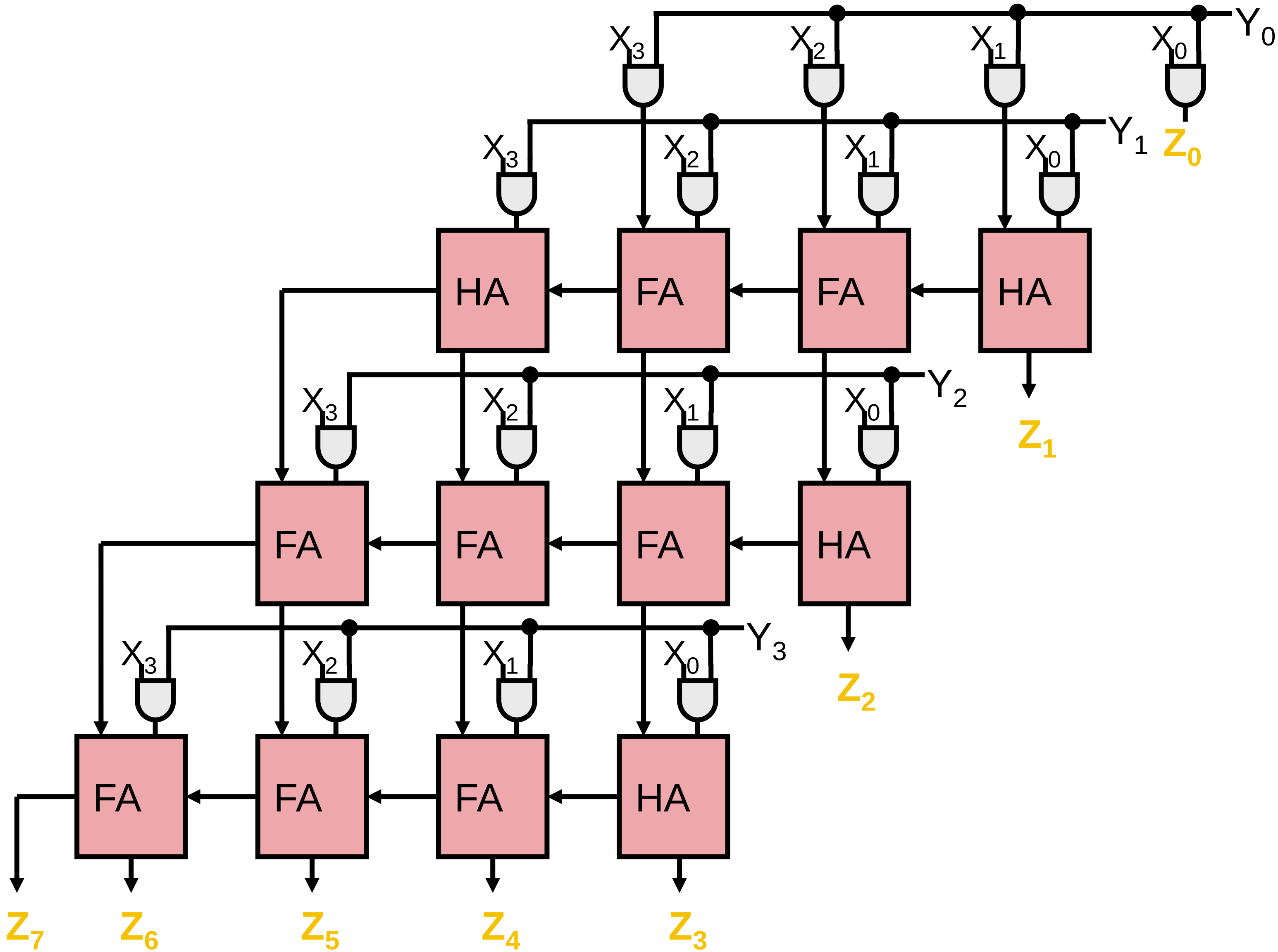
- What do we need to realize Array Multiplier?
- AND gates = ?
- FA = ?
- HA = ?

Array Multiplier: Idea

Use an array of AND gates to generate the partial products in parallel

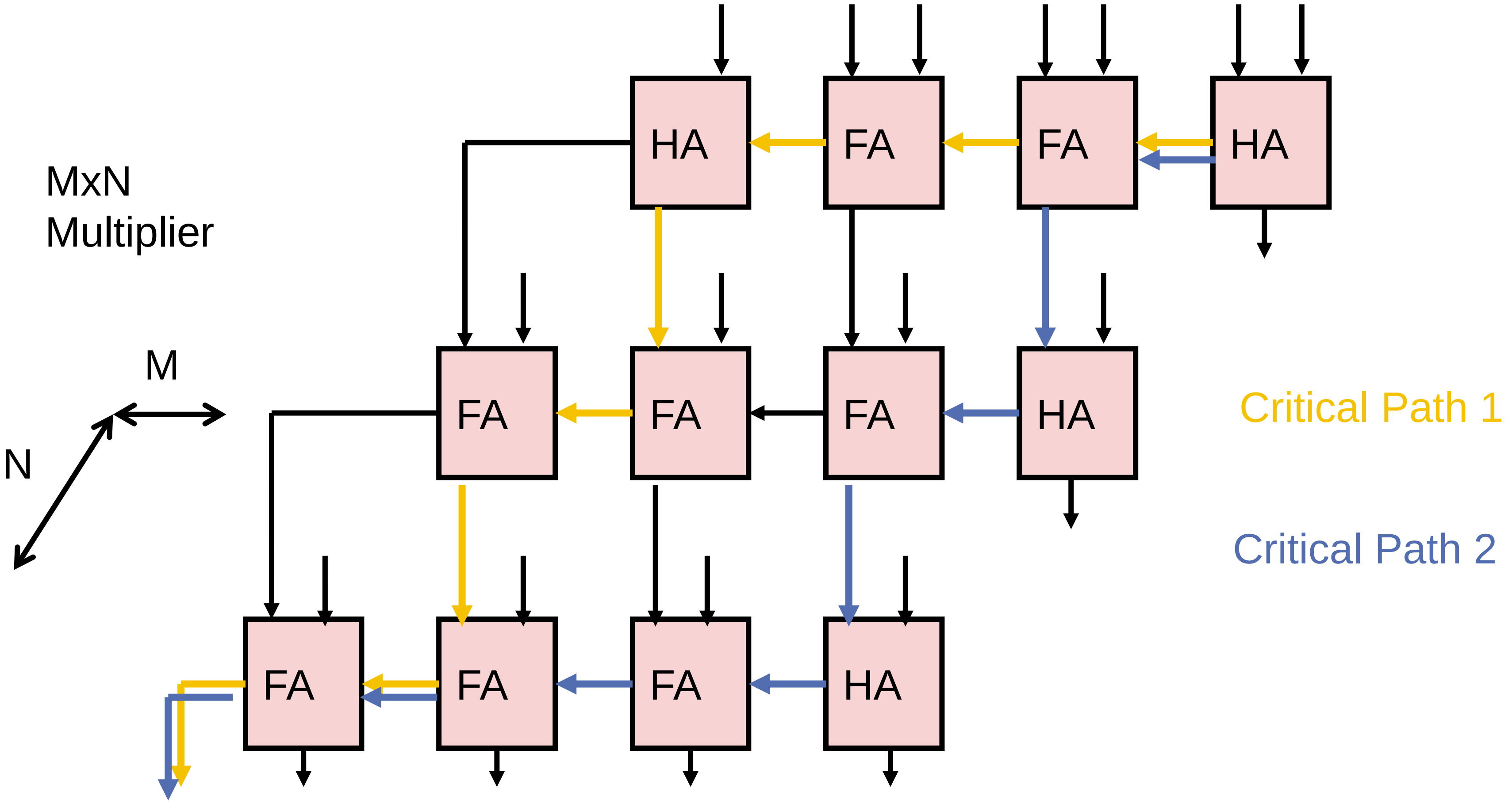


Array Multiplier: Adding Partial Products



Array Multiplier: Critical Path(s)

A lot of critical paths: same delay. (AND gates not shown)



Partial Product Generation in Verilog

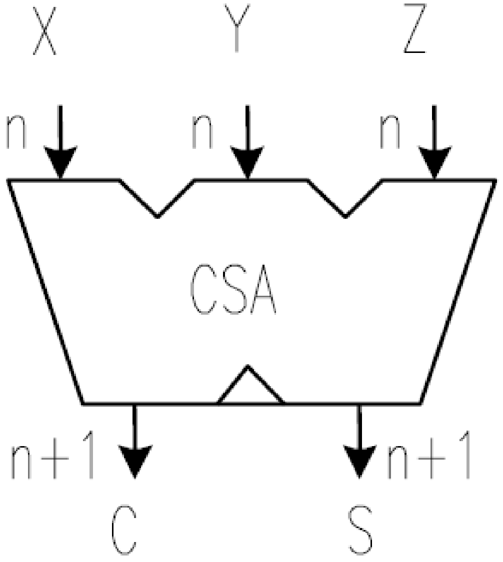
```
input [5:0] a,b;
output [11:0] prod;
integer i;

reg [5:0] pp [0:5];

always @ (a or b)
   begin
         for (i =0; i<6; i =  i+1)
               begin
                     pp[i] = a & {6{b[i]}};
               end
   end
```

Carry Save Adder (CSA)

		2^4	2^3	2^2	2^1	2^0
x		0	1	0	1	0
y		1	1	0	1	1
z		1	0	1	1	1
s		0	0	1	1	0
c	1	1	0	1	1	

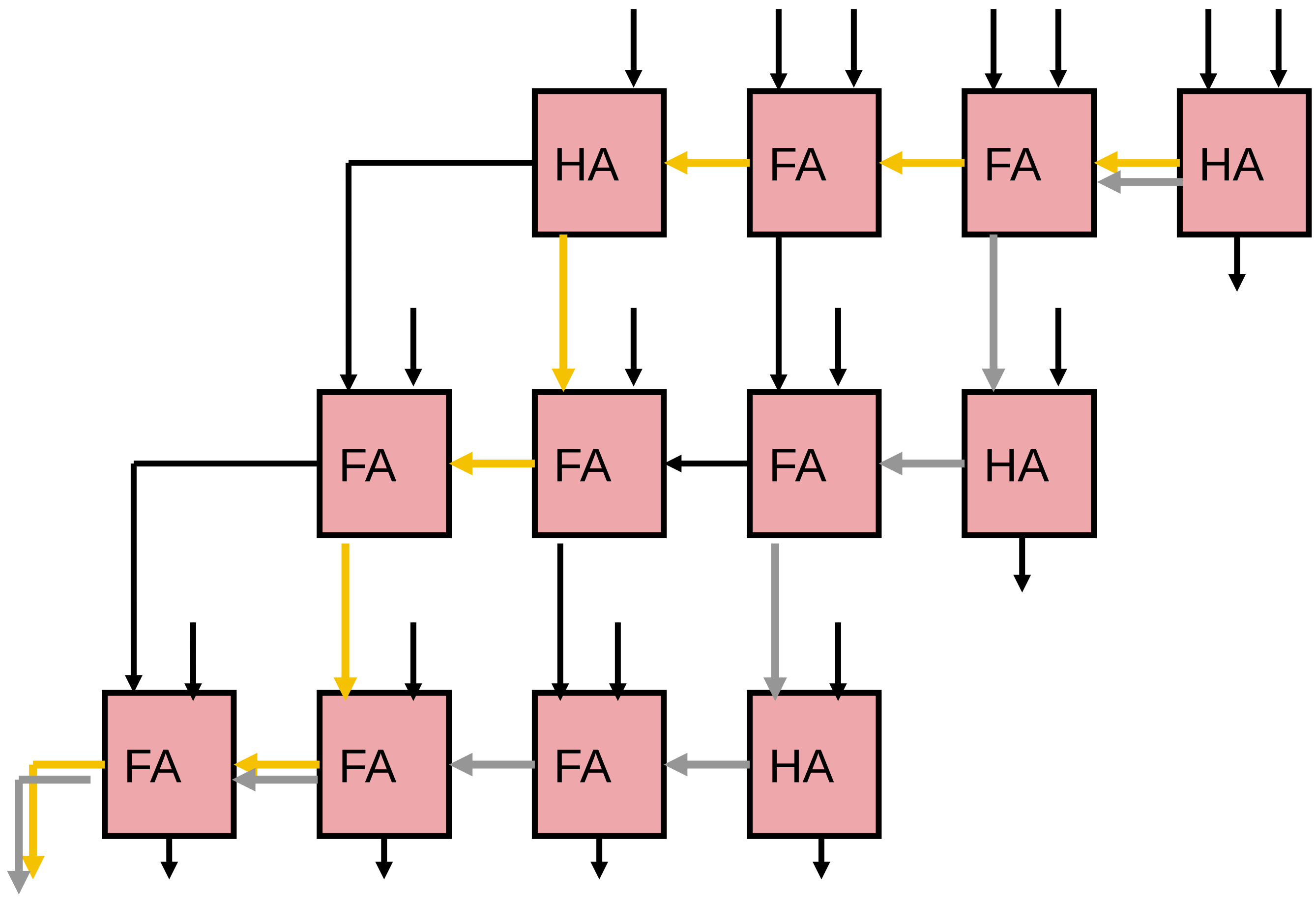


$$x+y+z = s + c$$

Carry-Save Adder: the Idea

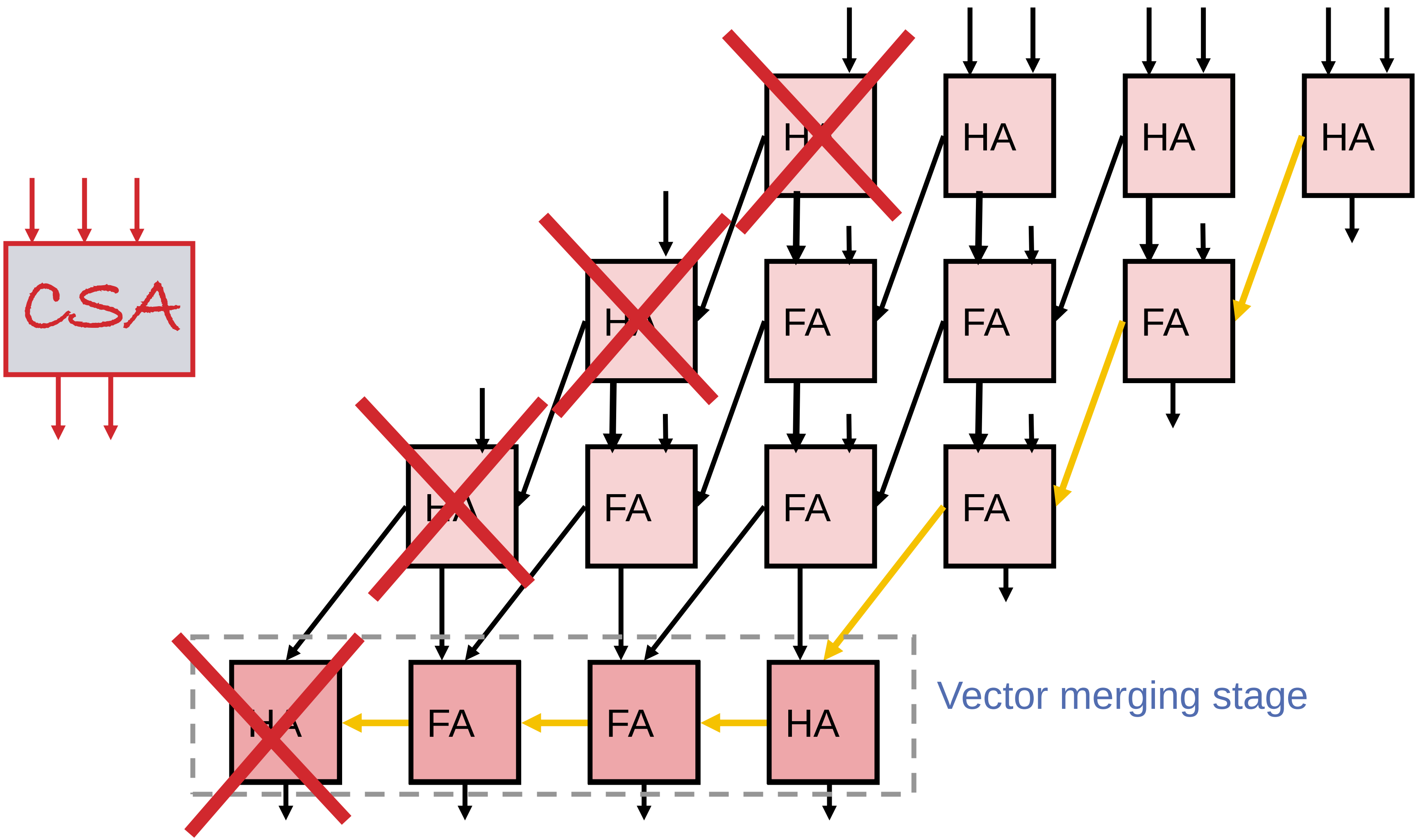
When adding k n-bit numbers, don't need to optimize the carry chain of each of the rows

- Below is the old-style ripple-adder

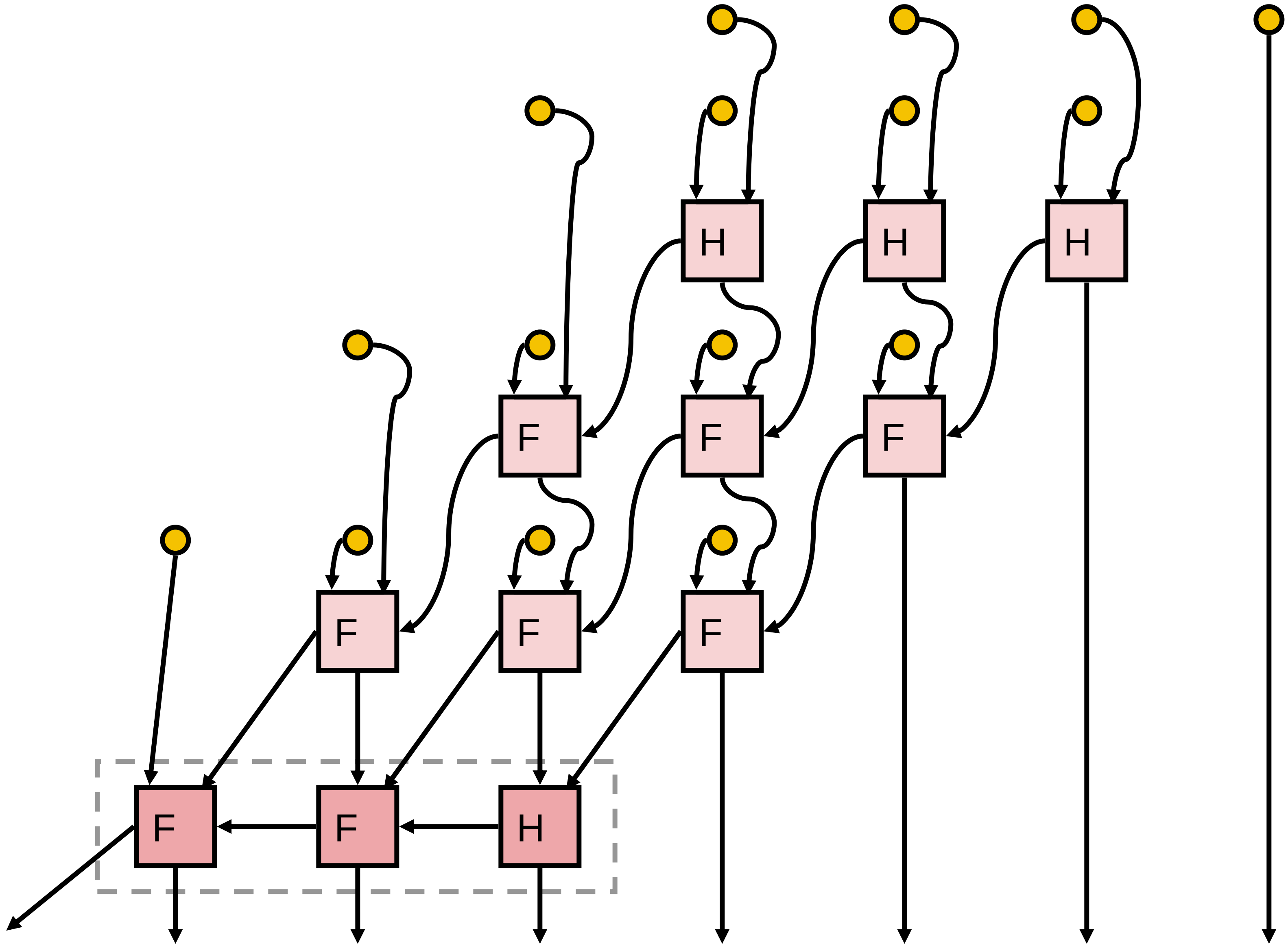


Carry-Save Adder: structure

Postpone the "carry propagation" operation to the last stage



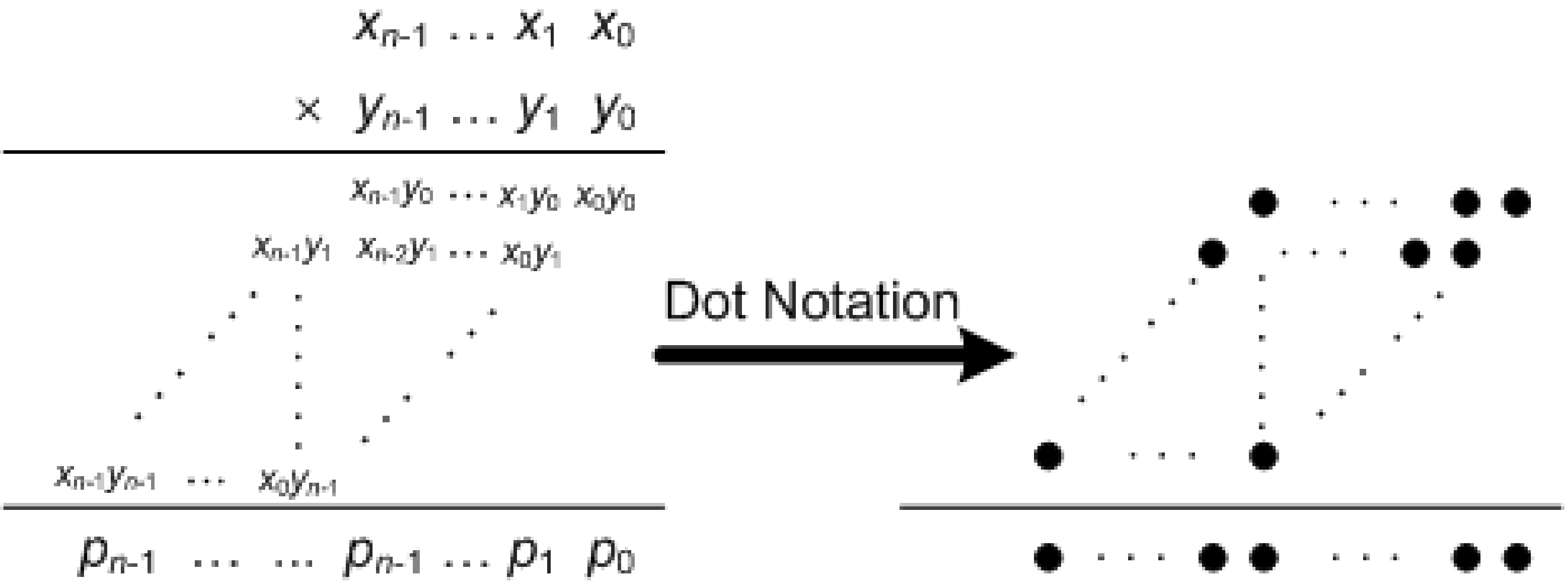
Carry-Save Adder: Details



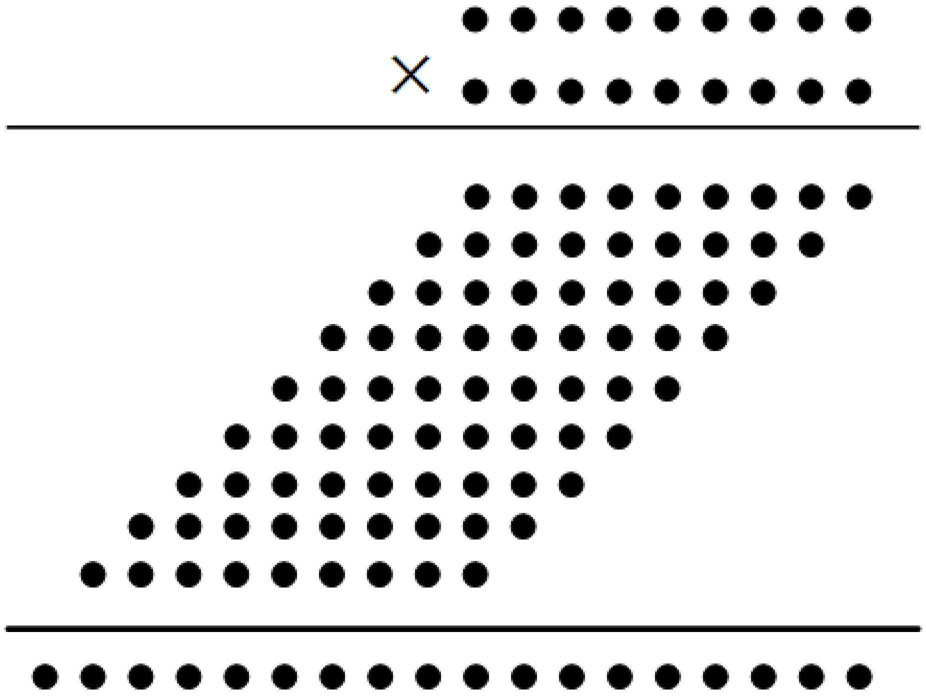
Carry-Save Adder: Four Operands

$$
\begin{array}{cccccc}
 & & \boxed{x_3} & \boxed{x_2} & \boxed{x_1} & \boxed{x_0} \\
 & & y_3 & y_2 & y_1 & y_0 \\
 & & z_3 & z_2 & z_1 & z_0 \\
 & & w_3 & w_2 & w_1 & w_0 \\
\hline
 & & s_{03} & s_{02} & s_{01} & s_{00} \\
 & c_{04} & c_{03} & c_{02} & c_{01} & \\
 & & w_3 & w_2 & w_1 & w_0 \\
\hline
 & C_{04} & s_{13} & s_{12} & s_{11} & s_{10} \\
 & c_{14} & c_{13} & c_{12} & c_{11} & \\
\hline
S_5 & S_4 & S_3 & S_2 & S_1 & S_0
\end{array}
$$

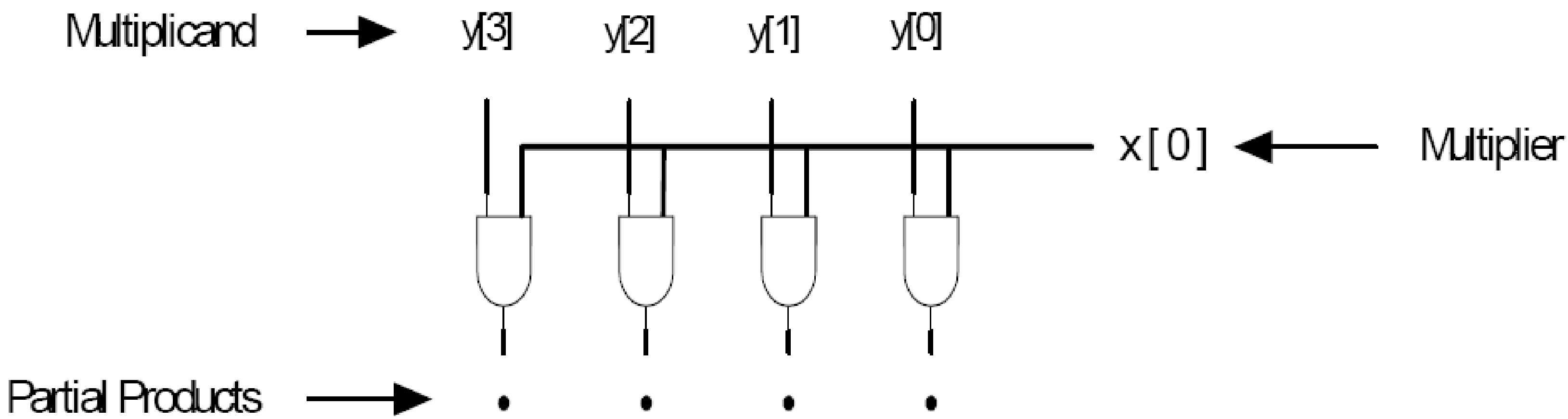
Dot Notation

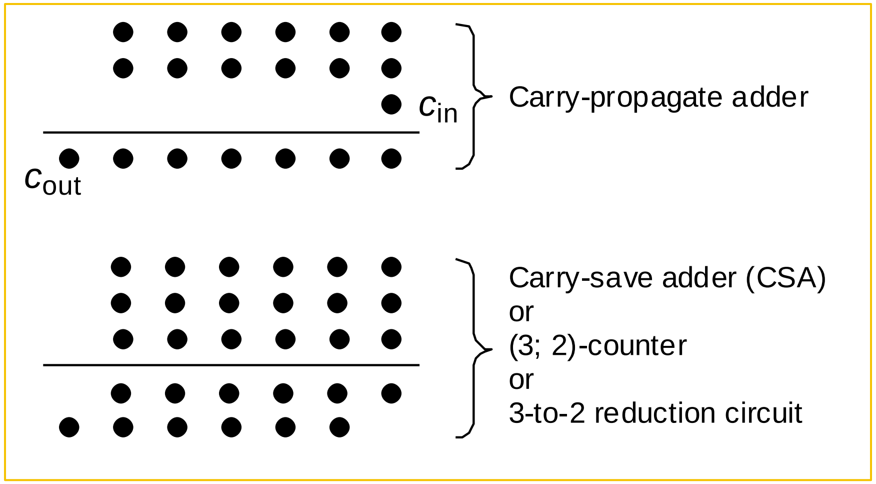


Dot Notation

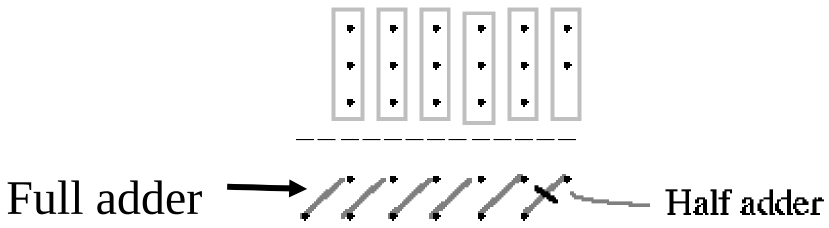


Dot Notation





Dot Notation: Specifying Full- and Half-Adder Blocks

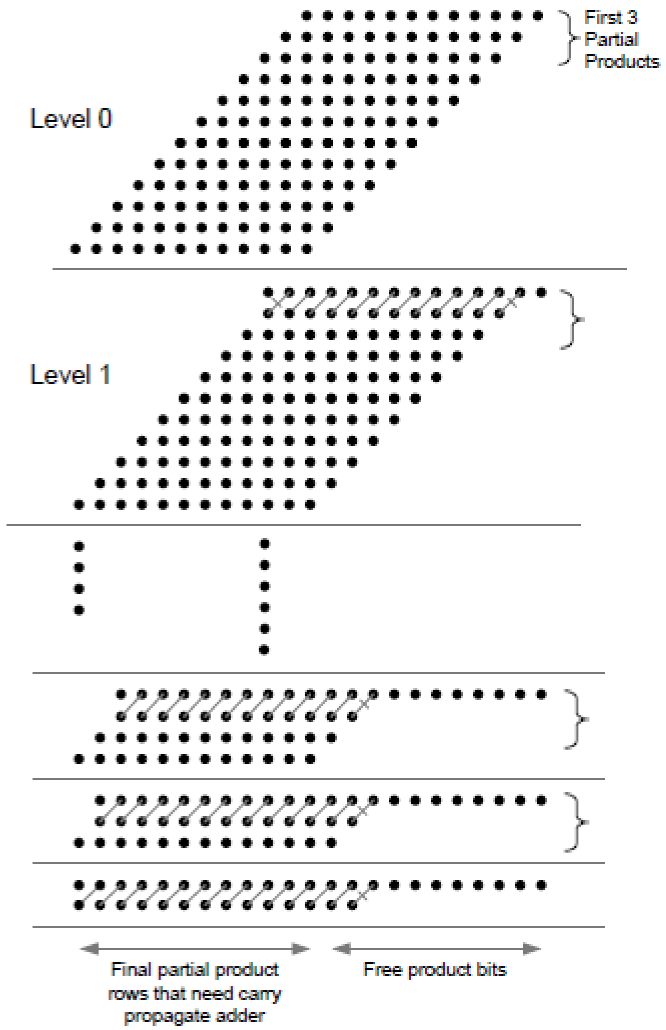


Partial Product Reduction Schemes

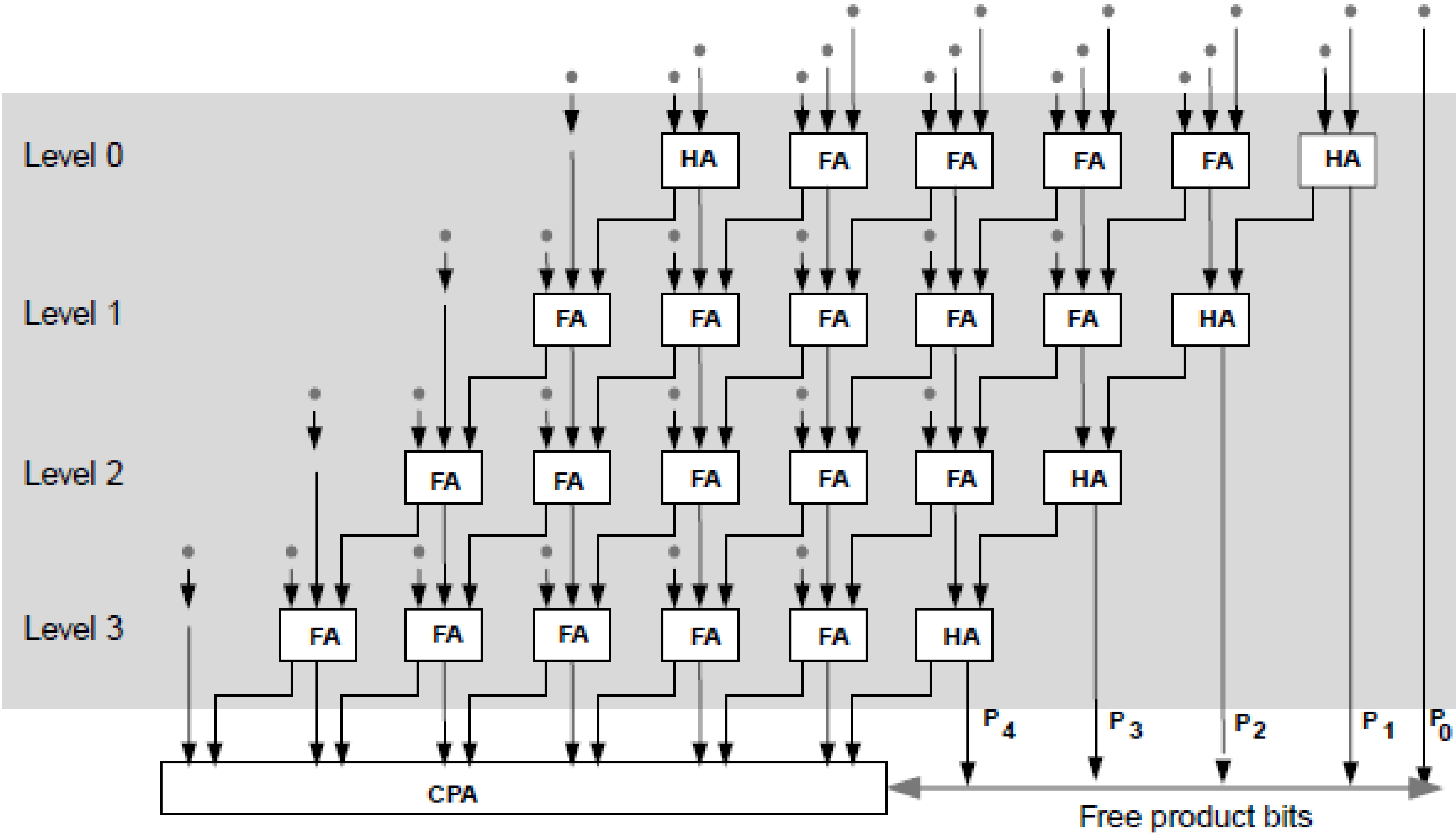
- Carry Save Reduction Scheme
- Dual Carry Save Reduction Scheme
- Wallace Tree Reduction Scheme
- Dadda Tree Reduction Scheme

Carry Save Reduction Scheme

- Reduce partial products by taking three at a time.
  - Reduce first three to two
  - Take the fourth partial product along with the reduced layer of the first three and then reduce them to two
  - And so on…
- Once you have only two layers, reduce them using any faster adder



Carry Save Reduction Scheme – (Example 6x6)



Dual Carry Save Reduction Scheme

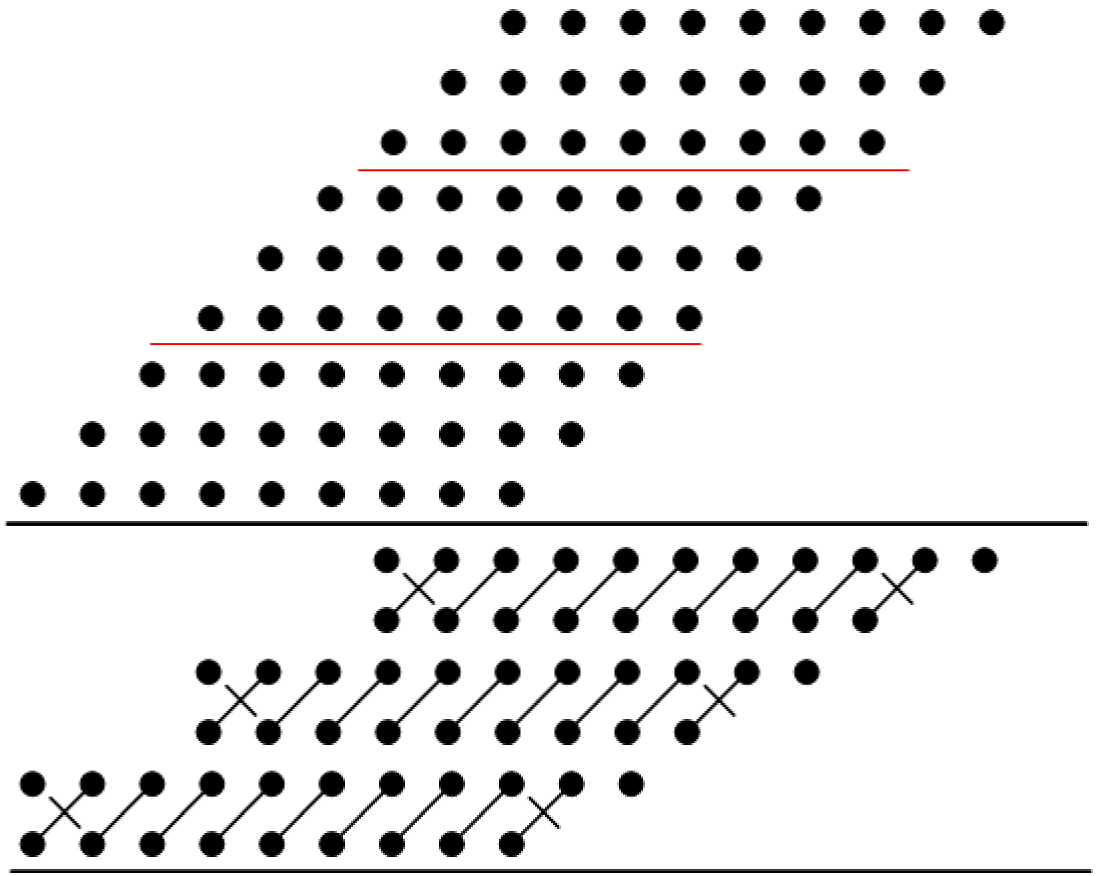
- Divide partial products into two equal groups
- Apply carry save reduction scheme on both of the groups simultaneously.
- This finally results in four layers of partial products. These four are then reduced to two using carry save reduction scheme.

- This results in speeding up the reduction operation.

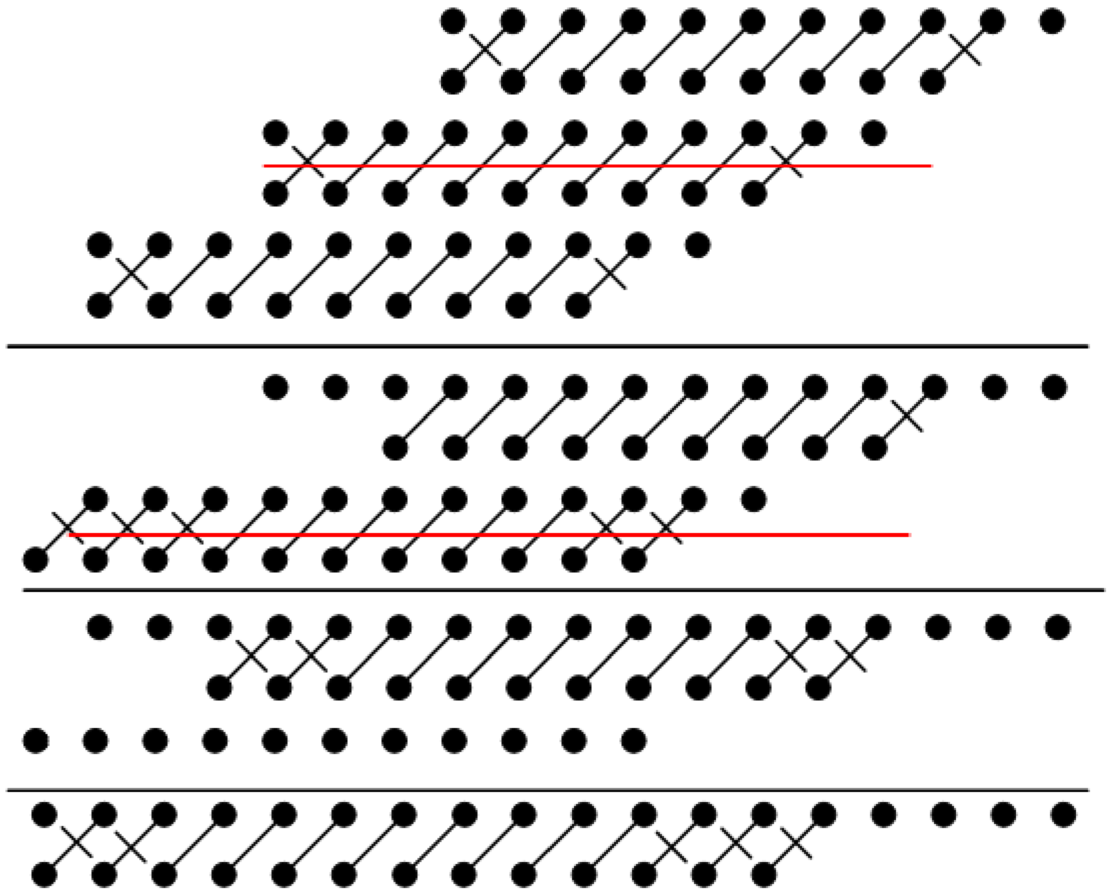
Wallace Tree Reduction Scheme

- Idea: divide & conquer
- In each column of partial products, every three adjacent rows construct a group. These groups should be non-overlapped.
- Then reduction in each group is done by one of the following cases:
  - Applying a full adder to the 3-bit groups
  - Applying a half adder to the 2-bit groups and
  - Passing any 1-bit group to the next stage without change
- CSA is used for reduction

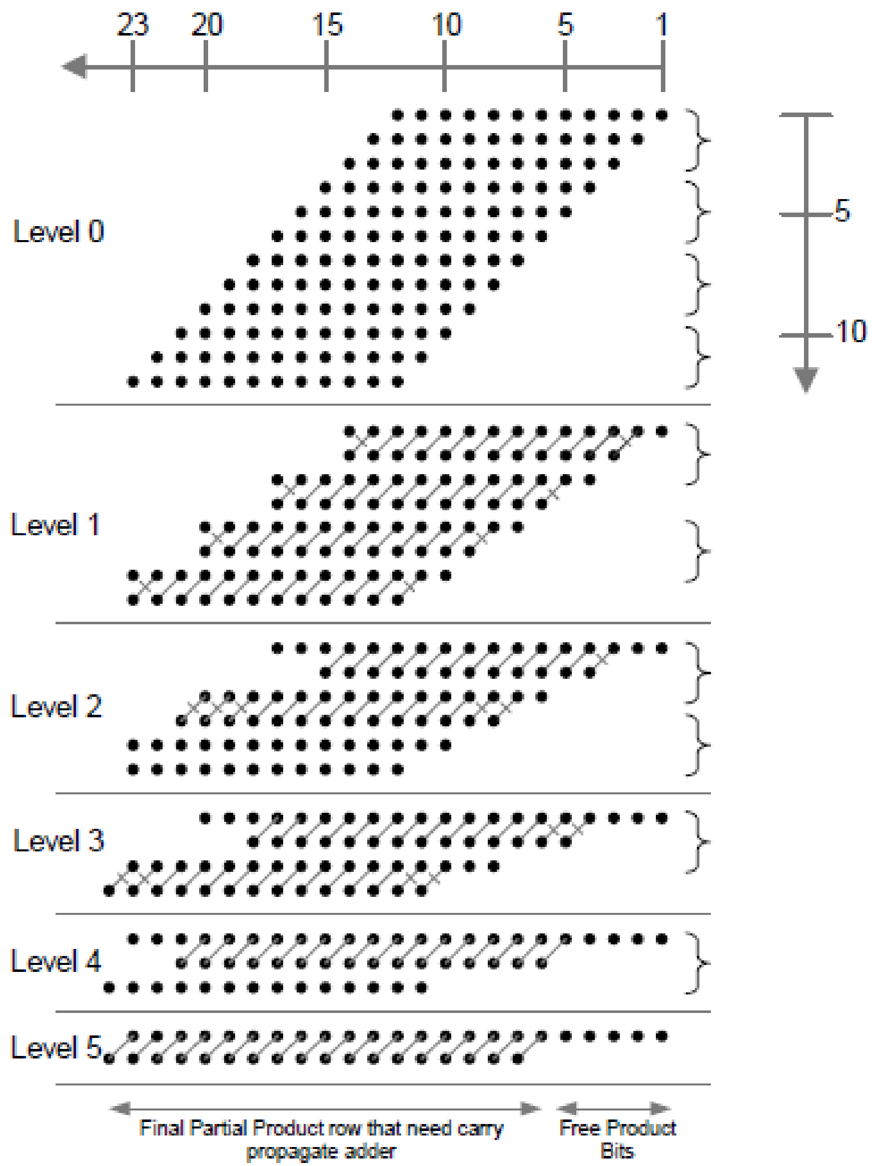
9x9 Wallace Tree Reduction Scheme



9x9 Wallace Tree Reduction Scheme



12x12 Wallace Tree Reduction Scheme



Adder Levels in Wallace Tree Reduction Scheme

Number of partial Products	Number of full adder Levels
3	1
4	2
$5 \leq n \leq 6$	3
$7 \leq n \leq 9$	4
$10 \leq n \leq 13$	5
$14 \leq n \leq 19$	6
$20 \leq n \leq 28$	7
$29 \leq n \leq 42$	8
$43 \leq n \leq 63$	9

- Same number of full adder delays as of number of full adder levels

Dadda Tree Reduction Scheme

- This method does as few reductions as possible.

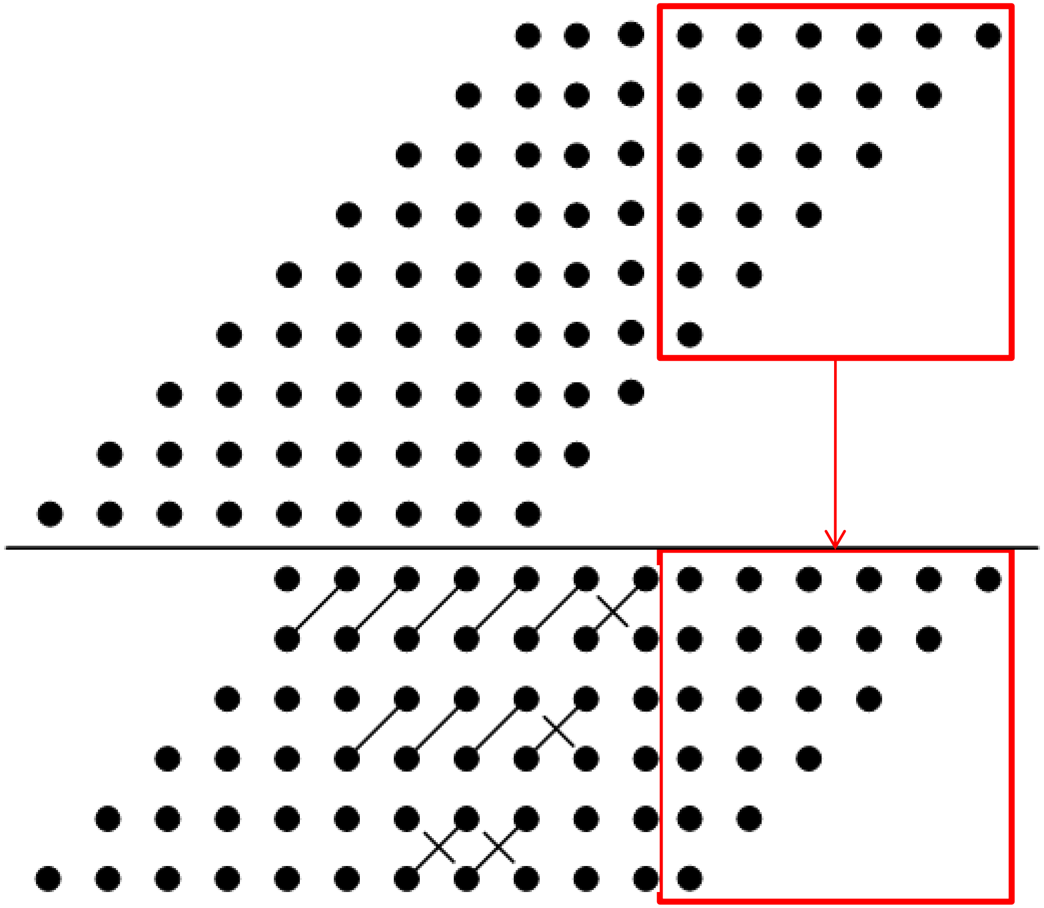
- To determine how much reduction is required, the maximum height of each stage is calculated by working back from the final stage (i.e., 2 rows).

    - 2, 3, 4, 6, 9, 13, 19, 28, 42, 63, etc.

- In our example, the first stage contains 9 rows; therefore, we would have 4 reduction stages as (9, 6, 4, 3, 2)

9x9 Dadda Tree Reduction Scheme

9, 6, 4, 3, 2



9, **6**, 4, 3, 2

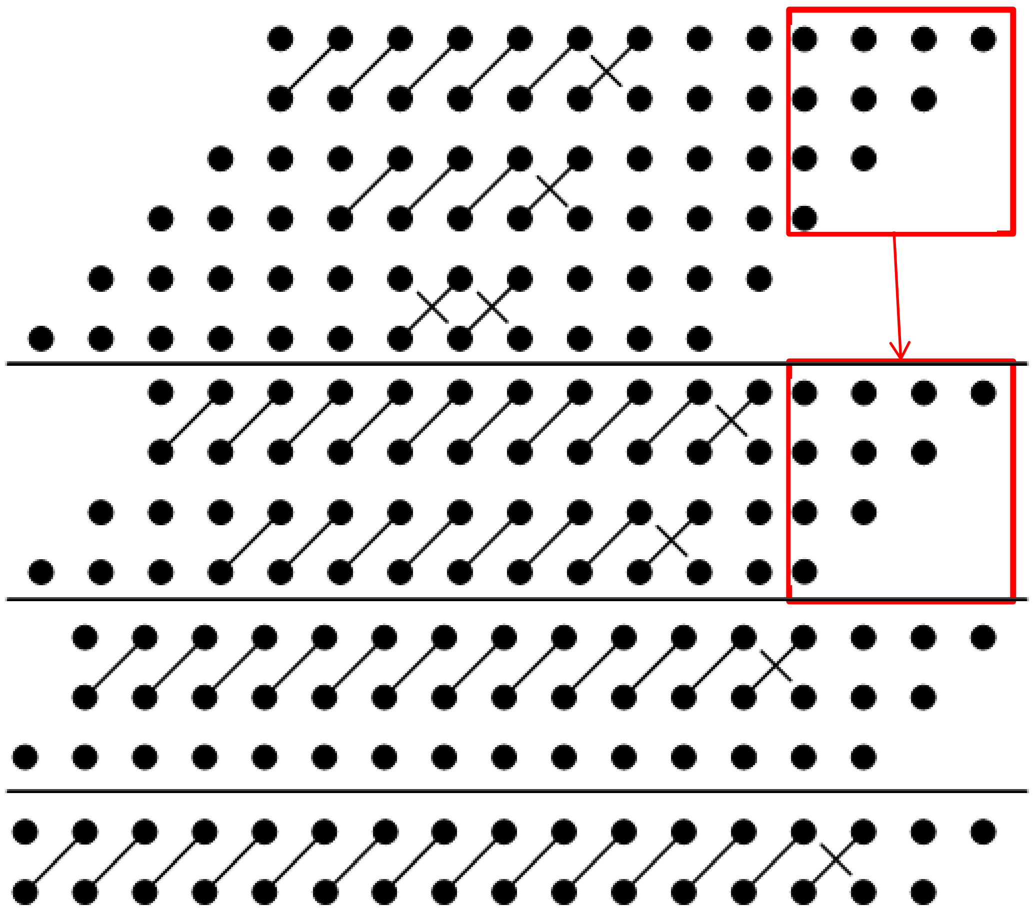
9x9 Dadda Tree Reduction Scheme

2, 3, 4, **6**, 9

2, 3, **4**, 6, 9

2, **3**, 4, 6, 9

2, 3, 4, 6, 9



Comparison

INPUT SIZE (N)	8	16	24	32	64
STAGES (S)	4	6	7	8	10
WALLACE					
FULL ADDERS	38	200	488	906	3,850
HALF ADDERS	15	52	100	156	430
TOTAL GATES	402	2,008	4,801	8,778	36,388
MODIFIED WALLACE					
FULL ADDERS	39	201	490	907	3,853
HALF ADDERS	3	9	16	23	53
TOTAL GATES	363	1,845	4,474	8,263	34,889
DADDA					
FULL ADDERS	35	195	483	899	3,843
HALF ADDERS	7	15	23	31	63
TOTAL GATES	343	1,815	4,439	8,215	34,839

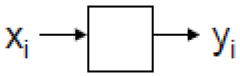
Multiplication Revisited

- Our discussion so far has assumed both the multiplicand (A) and the multiplier (X) can vary at runtime.

- What if one of the two is a constant?

$$Y = C * X$$

  - "Constant Coefficient" multiplication comes up often in signal processing and other hardware. e.g.



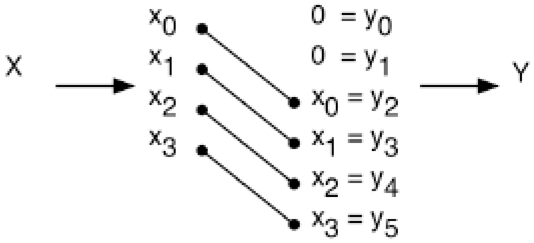
$$y_i = \alpha y_{i-1} + x_i$$

where α is an application dependent constant that is hard-wired into the circuit.

• How do we build and array style (combinational) multiplier that takes advantage of the constancy of one of the operands?

Multiplication by a Constant

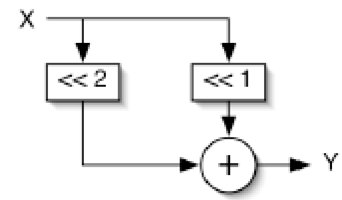
- Remember, for any number X, one bit left shift provides us 2X, two bit left shift gives 4X and so on.
  - $1_2 = 1_{10}$ (<<1) $\rightarrow$ $10_2 = 2_{10}$ (<<1) $\rightarrow$ $100_2 = 4_{10}$ (<<1) $\rightarrow$ $1000_2 = 8_{10}$ …..
- Thus, if the constant C in C*X is a power of 2, then the multiplication is simply a shift of X. e.g. 4*X $\rightarrow$ (X<<2)

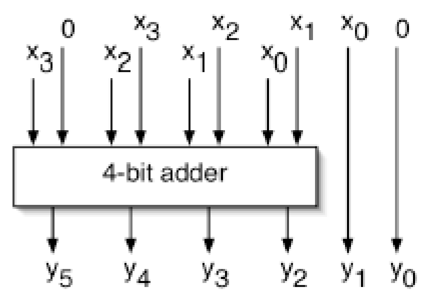


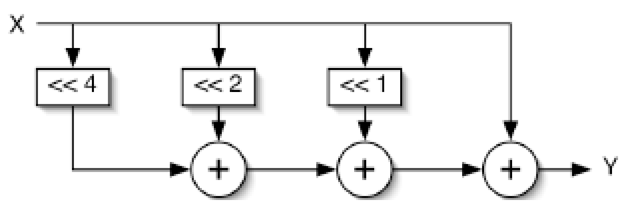
- What about division???
  - Right Shift
- What about multiplication by non- powers of 2?

Multiplication by non – powers of 2

- A combination of fixed shifts and addition
- Shift by weights at which bit = 1, then add.
  - $6*X = 0110 * X = (2^2 + 2^1)*X$
  - $23*X = 010111 * X = (2^4 + 2^2 + 2^1 + 2^0)*X$





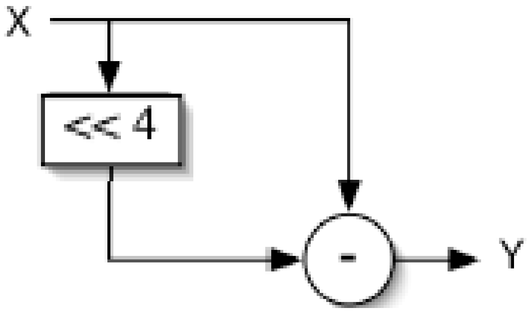


- In general, the number of additions equals one minus the number of 1's in the constant, C.
- Is there a way to further reduce the number of adders (and thus the cost and delay)?

Multiplication using Subtraction

- Subtraction is the same cost and delay as addition.
- Consider C*X where C is the constant value 15_{10} = 01111.
  - C*X requires 3 adders.
  - We can “recode” 15
    - from 01111 = $(2^3 + 2^2 + 2^1 + 2^0)$
    - to 1000 – 0001 = $(2^4 - 2^0)$

- Therefore, 15*X can be implemented with only one subtractor.



Canonic Signed Digit Representation

- CSD represents numbers using 1, $\bar{1}$(= –1), & 0
  - Contains least possible number of non-zero digits.
  - Strings of 2 or more non-zero digits are replaced. Thus no two consecutive bits in a CSD representation are non–zero.
  - Leads to a unique representation.
  - The bit position with $\bar{1}$ carries negative weightage

- To form CSD representation:
  - Starting from LSB, find a string of 1s.
  - Replace the first 1 by $\bar{1}$
  - All other ones in the string are changed to 0s
  - The 0 marking the end of the string is changed to 1.
  - The process is repeated for any succeeding strings of 1s.

$$0\boxed{111111}0 \rightarrow 100000\bar{1}0$$

$$(64 + 32 + 16 + 8 + 4 + 2) = 126 = (128 – 2)$$

CSD Representation of $Q_{n.m}$ Format Numbers

- Same rules apply as of integers.

- Consider a $\mathbf{Q_{1.15}}$ number

0111 0101 0011 1111$= (2^{-1}+2^{-2}+2^{-3}+2^{-5}+2^{-7}+2^{-10}+2^{-11}+2^{-12}+2^{-13}+2^{-14}+2^{-15}) = 0.916_{10}$

0111 0101 0100 000$\bar{1}$

100$\bar{1}$ 0101 0100 000$\bar{1}$ $= (2^{0} - 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} - 2^{-15}) = 0.916_{10}$

Class Task

- Convert the following numbers into equivalent CSD representation.

  - 011101_2 29_{10} | 0010111_2 23_{10} | 0110110_2 54_{10} | 01101111_2 111_{10}

  - $100\bar{1}01_2$ | $010\bar{1}00\bar{1}_2$ | $100\bar{1}0\bar{1}0_2$ | $100\bar{1}000\bar{1}_2$

  - 01101111_2 in $\mathbf{Q_{1.7}}$ format
    0.8671875

  - $100\bar{1}000\bar{1}$

CSD Multiplier

- CSD multiplier makes use of CSD property to implement multiply by 2 multiplier

- Consider C = 01101111_2 in $\mathbf{Q_{1.7}}$ format
  - Equivalent CSD representation $100\bar{1}000\bar{1}$

- Remember, right shift by 1, divides the number by 2.