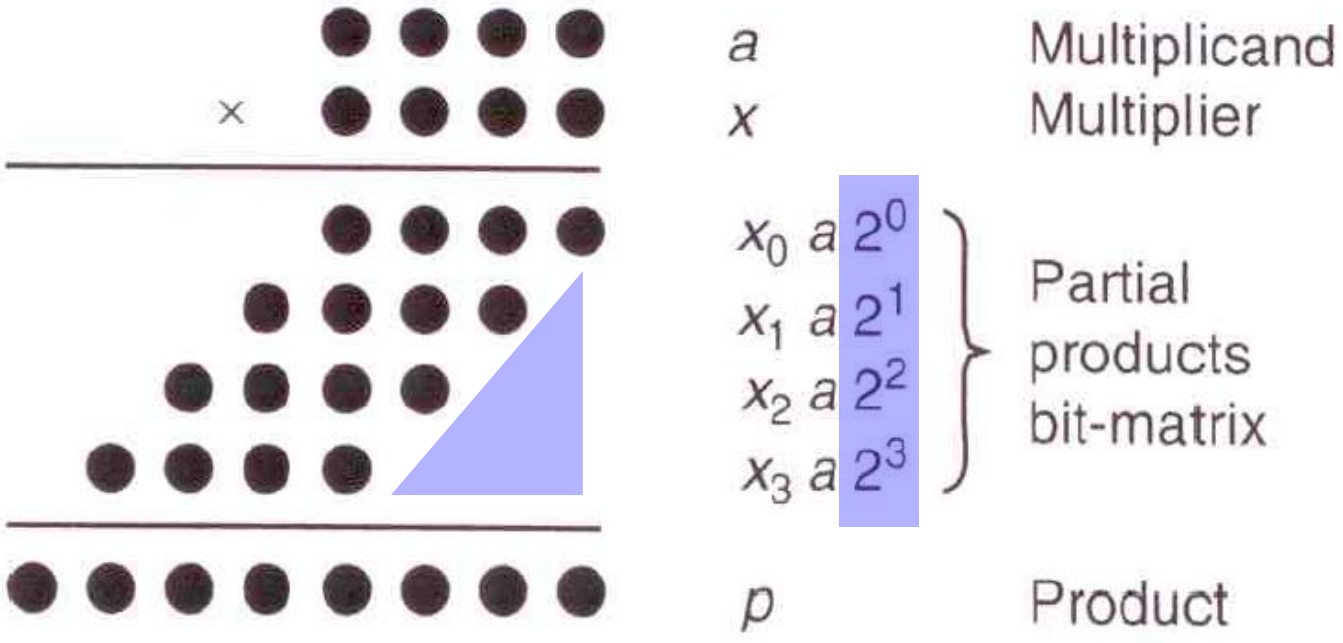
Multiplication and Shift Circuits

Shmuel Wimer
Bar Ilan University, Engineering Faculty
Technion, EE Faculty

Shift/Add Unsigned Multiplication Algorithms

a	Multiplicand	$a_{k-1}a_{k-2}\cdots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2}\cdots x_1x_0$
p	Product $(a \times x)$	$p_{2k-1}p_{2k-2}\cdots p_1p_0$



Shift partial products

In right shift multiplication the partial products $x_j a$, $0 <= j <= k-1$, are recursively accumulated from top to bottom.

$$p^{(j+1)} = \underbrace{\left(p^{(j)} + x_j a 2^k \right)}_{\text{add}} \times 2^{-1} \quad \text{with} \quad p^{(0)} = 0 \text{ and } p^{(k)} = p$$

shift right

$x_0 a 2^k$ will be multiplied by 2^{-k} after k iterations. a is pre multiplied by 2^k to offset the effect of right shifts.

After k iteration the recurrence yields $p^{(k)} = ax + p^{(0)} 2^{-k} = ax$

a is aligned to the left (MSB) k bits of a $2k$-bit .

How to obtain $p = ax + y$? Initialize $p^{(0)}$ to $y2^k$

(a) Right-shift algorithm

a			1	0	1	0			
x			1	0	1	1			
$p^{(0)}$			0	0	0	0			initialize
$+x_0 a$			1	0	1	0			**add** partial product aligned to left
	0	1	0	1	0				
$p^{(1)}$		0	1	0	1	0			multiply by 2^{-1} by **right-shift**
$+x_1 a$		1	0	1	0				**add** partial product aligned to left
	0	1	1	1	1	0			
$p^{(2)}$		0	1	1	1	1	0		multiply by 2^{-1} by **right-shift**
$+x_2 a$		0	0	0	0				**add** partial product aligned to left
	0	0	1	1	1	1	0		
$p^{(3)}$		0	0	1	1	1	1	0	
$+x_3 a$		1	0	1	0				
	0	1	1	0	1	1	1	0	
$p^{(4)}$		0	1	1	0	1	1	1	0

In left shift multiplication the partial products $x_{k-1-j}a$, $0 \le j \le k-1$, are recursively accumulated from bottom to top.

shift left

$$p^{(j+1)} = \underbrace{2\,p^{(j)} + x_{k-1-j}a}_{\text{add}} \quad \text{with} \quad p^{(0)} = 0 \text{ and } p^{(k)} = p$$

After k iteration the recurrence yields $p^{(k)} = ax + p^{(0)}2^k = ax$

How to obtain $p = ax + y$? Initialize $p^{(0)}$ to $y2^{-k}$

(b) Left-shift algorithm

a					1	0	1	0	
x					1	0	1	1	
$p^{(0)}$					0	0	0	0	initialize
				0	0	0	0	0	multiply by 2 by **left-shift**
$+x_3a$					1	0	1	0	**add** partial product aligned to right
$p^{(1)}$				0	1	0	1	0	multiply by 2 by **left-shift**
			0	1	0	1	0	0	
$+x_2a$					0	0	0	0	**add** partial product aligned to right
$p^{(2)}$			0	1	0	1	0	0	multiply by 2 by **left-shift**
		0	1	0	1	0	0	0	
$+x_1a$					1	0	1	0	**add** partial product aligned to right
$p^{(3)}$		0	1	1	0	0	1	0	
	0	1	1	0	0	1	0	0	
$+x_0a$					1	0	1	0	
$p^{(4)}$	0	1	1	0	1	1	1	0	

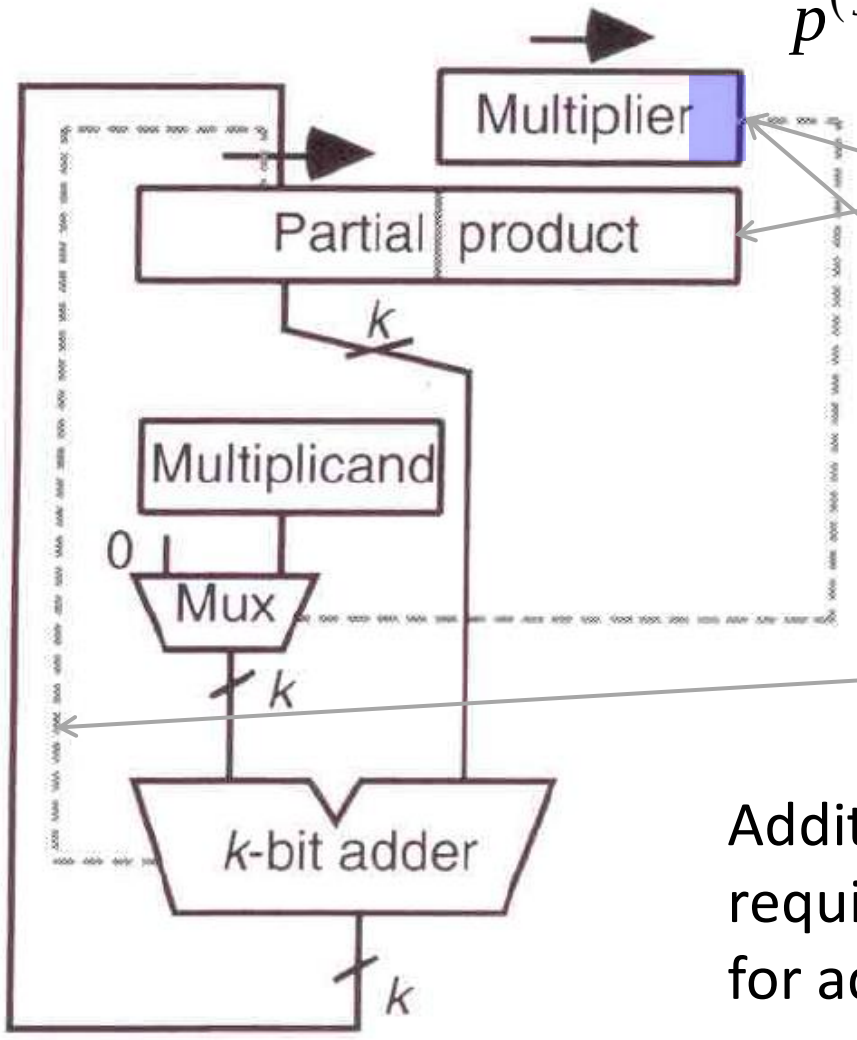
Serial multiplication by add and shift entails ***k*** additions and ***k*** shifts

Right shift is favored since the addition of partial product takes place at the MSB ***k***-bit part of the **2*k*** word. In left shift it takes place at the LSB ***k***-bit part and carry can propagate to MSB part.

Left shift algorithm requires therefore **2*k*** bit adder, while for right shift ***k*** bit suffice.

Hardware of right-shift multipliers (without control)

$$p^{(j+1)} = \left(p^{(j)} + x_j a 2^k \right) \times 2^{-1}$$



Shift registers

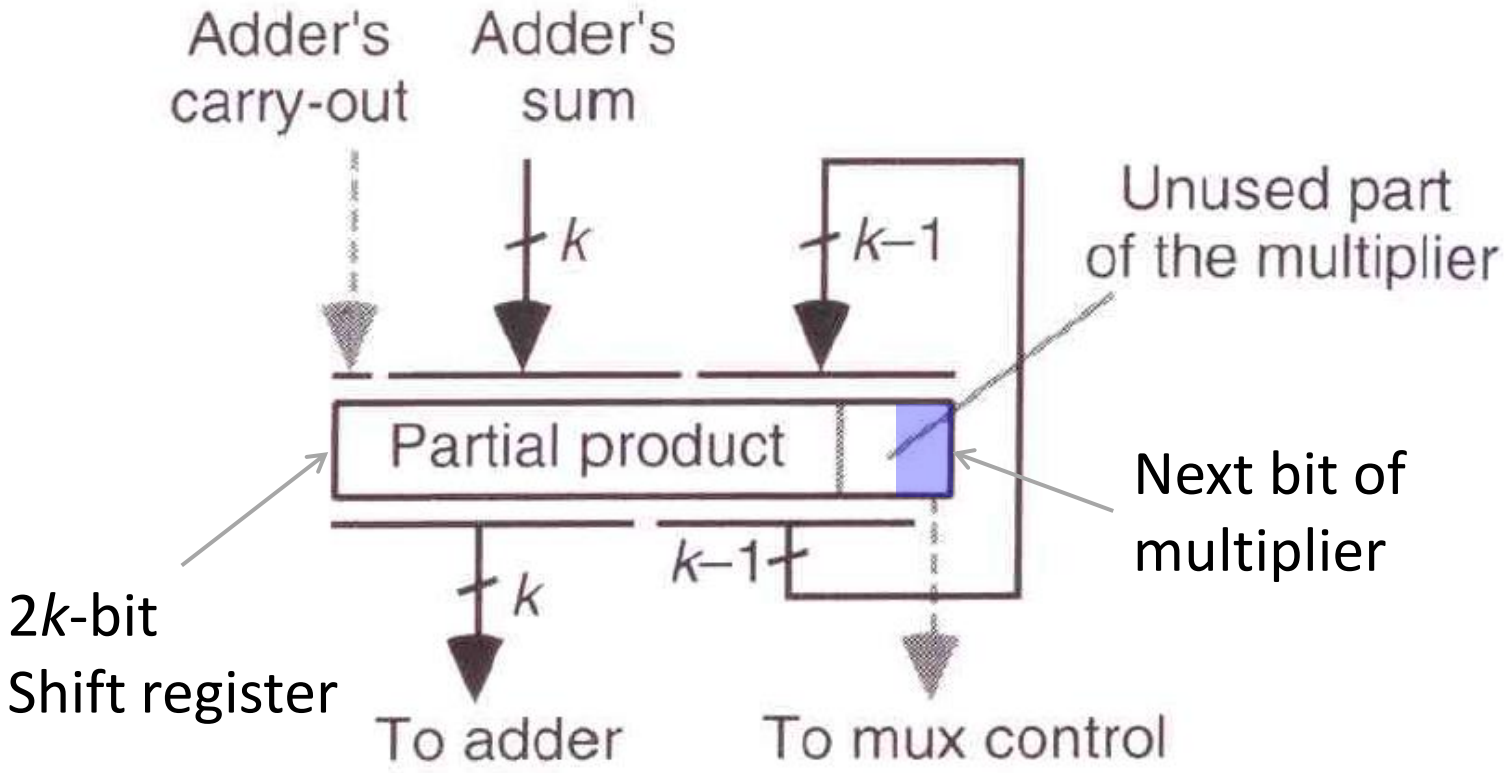
Next bit of multiplier

Control requires counter

Adder's carry out must be stored

Addition of a partial product requires two clock cycles. One for add and one for a shift.

Reducing addition of partial product to one cycle



At each clock cycle adder's carry-out is written to MSB and LSB is used for multiplication (via a MUX)

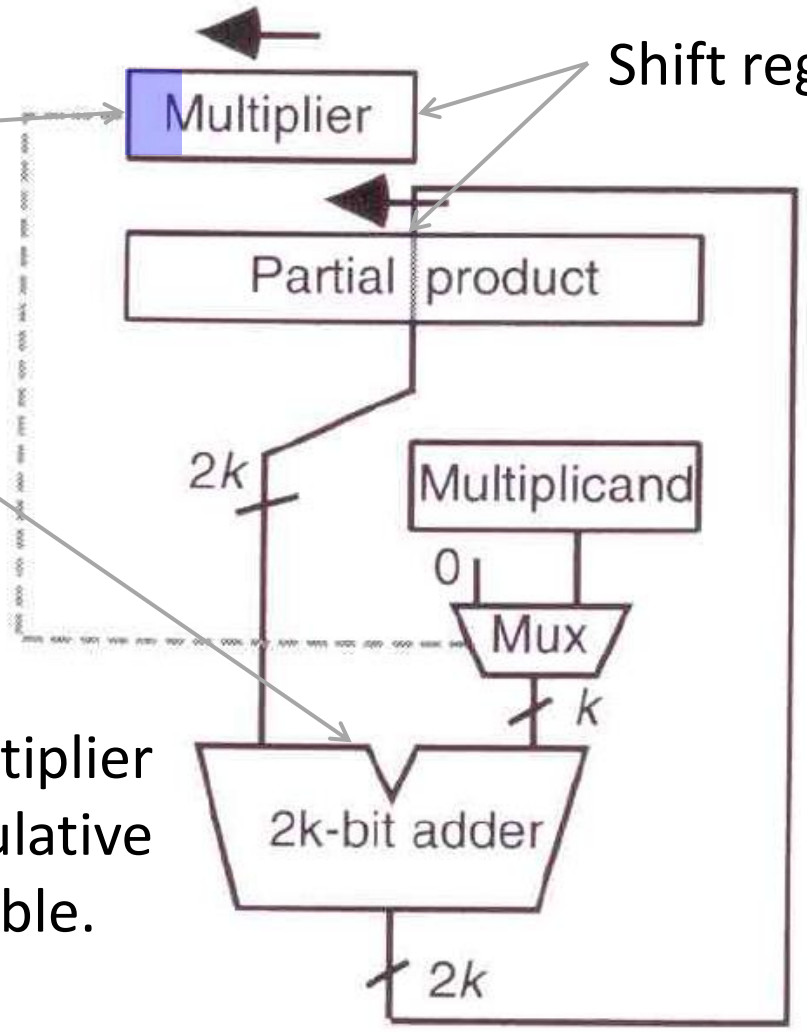
Hardware of left-shift multipliers (without control)

Next bit of multiplier

Shift registers

Adder is 2k bits rather than k bits in right-shift.

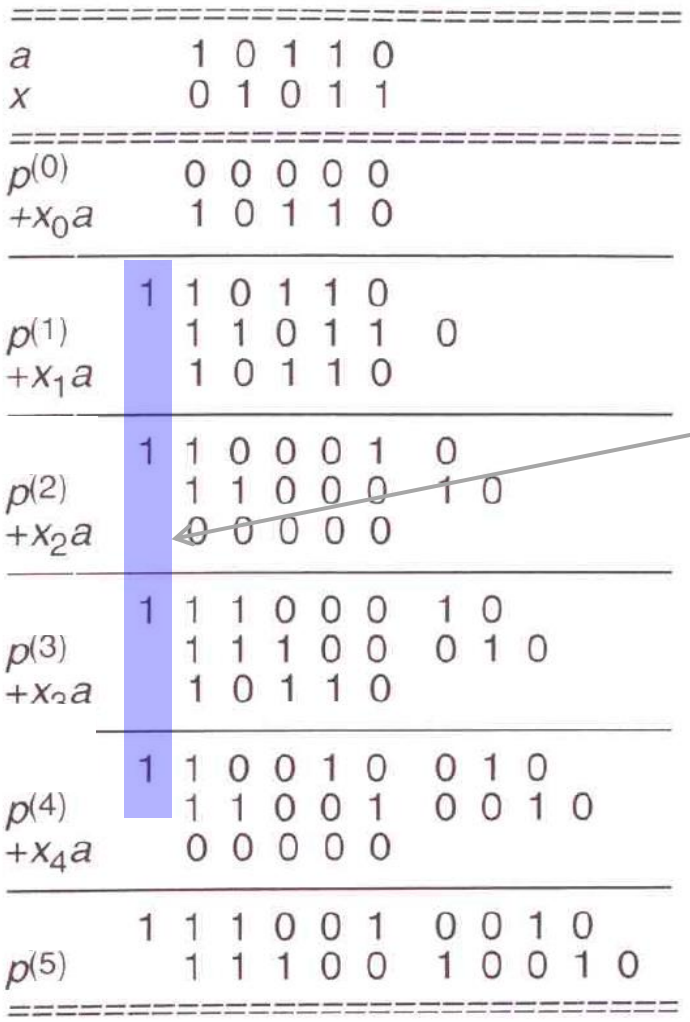
Register sharing of multiplier and MSB of cumulative partial product is possible.



Multiplication of Signed Numbers

- **Sign-magnitude** representation requires only XOR-ing of the operands' sign bits.
- In **1's-complement**, a negative operand is complemented and unsigned multiplication takes place. The result is complemented by XOR-ing of operands' sign bits.
- For **2's-complement**, right-shift multiplication is proper for negative multiplicand and positive multiplier.

Right-shift multiplication

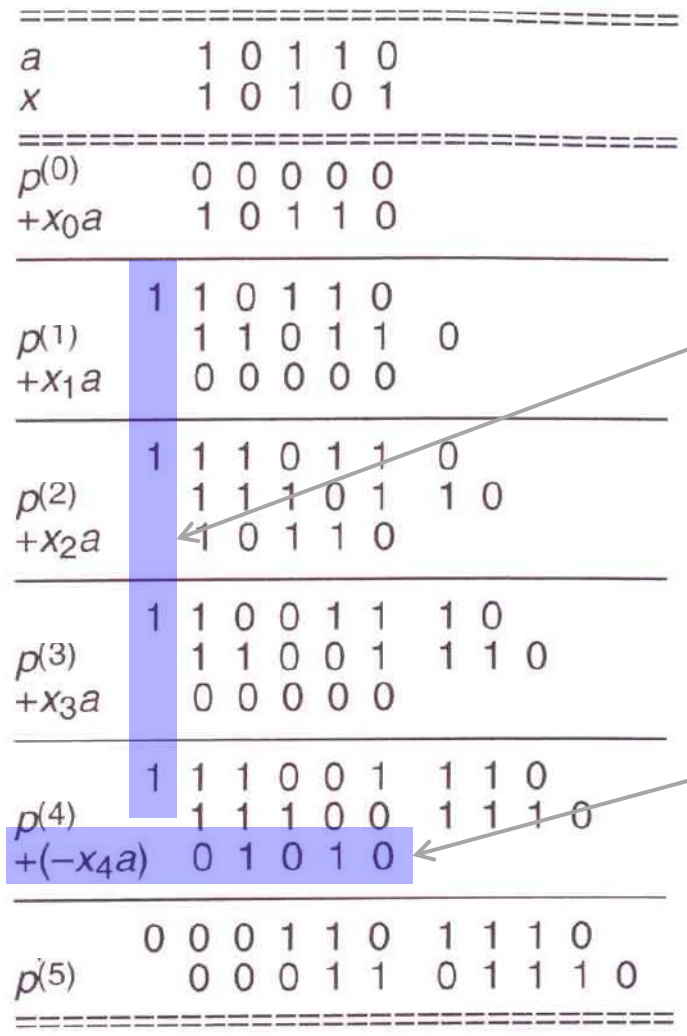


Negative multiplicand

Positive multiplier

Negative sign extensions

Negative multiplicand and 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			
	1	1	0	1	1	0				
$p^{(1)}$		1	1	0	1	1	0			
$+x_1a$		0	0	0	0	0				
	1	1	1	0	1	1	0			
$p^{(2)}$		1	1	1	0	1	1	0		
$+x_2a$		1	0	1	1	0				
	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				
	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				
	0	0	0	1	1	0	1	1	1	0
$p^{(5)}$		0	0	0	1	1	0	1	1	1	0

Negative multiplicand

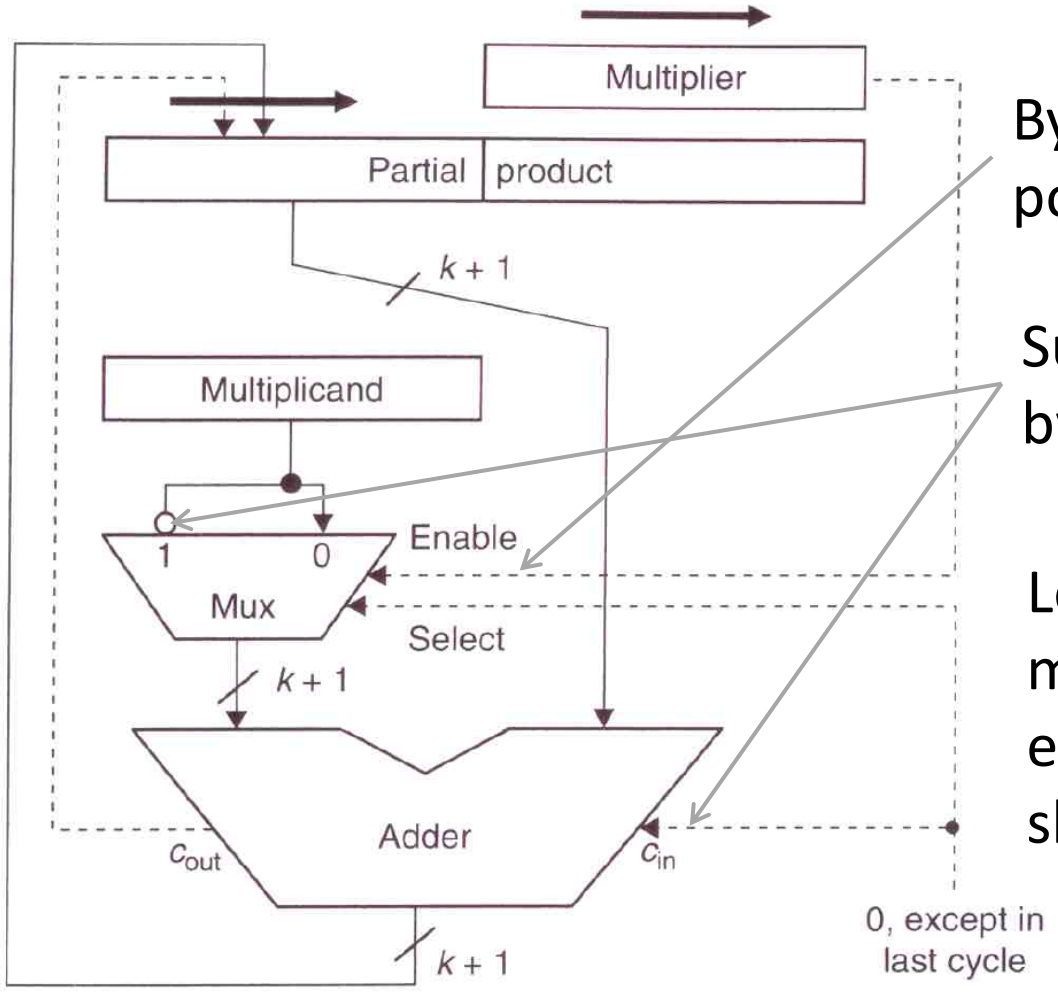
Negative multiplier

Negative sign extensions

Handle correctly by subtracting $x_{k-1}a$ rather than adding

Hardware complements multiplicand and adds 1 via carry-in

Hardware implementation (control logic not shown)



Bypassed MUX for positive multiplier

Subtracting $x_{k-1}a$ by negation plus 1

Left-shift requires more logic due to sign extension, and is also slower ($2k$ bit adder).

Parallel Multiplication Algorithms

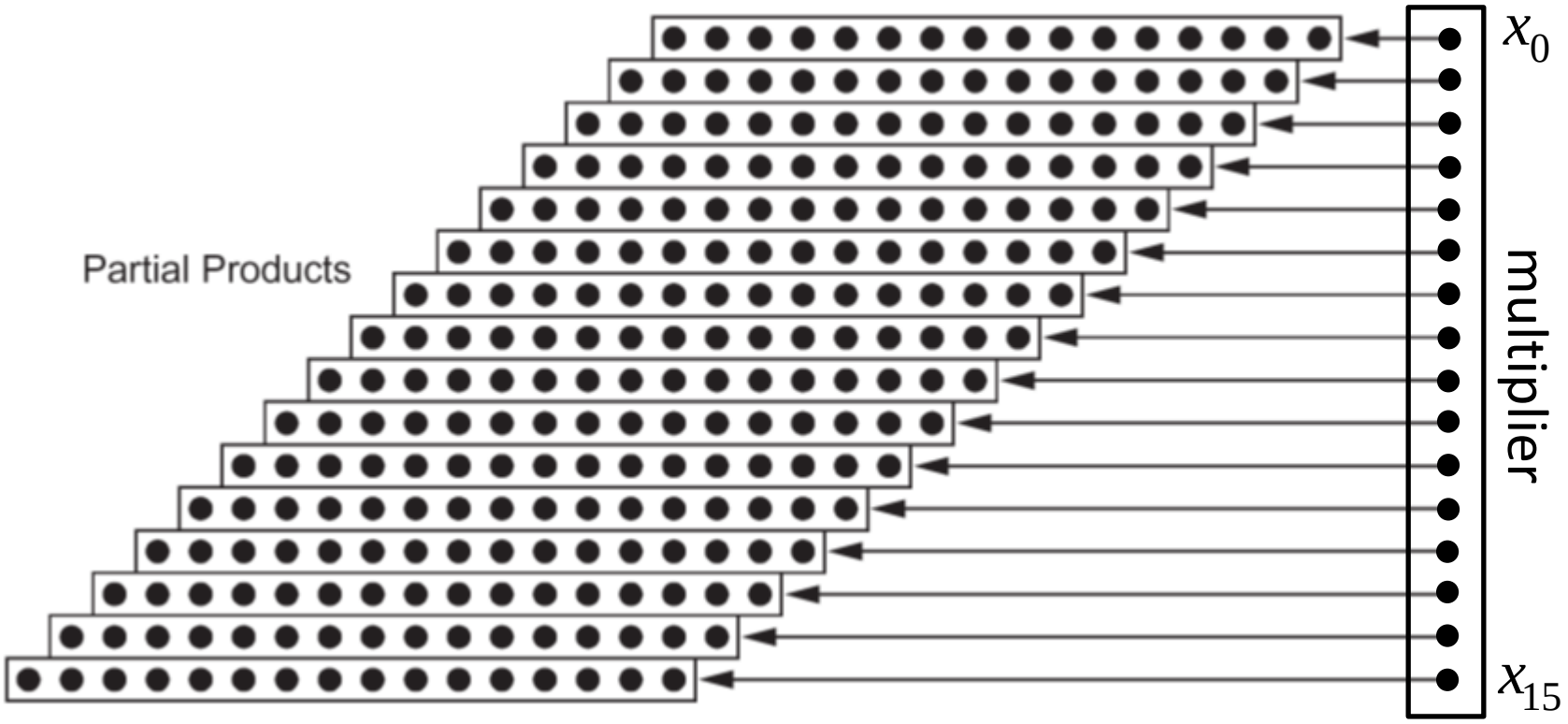
```
   011001  :  25₁₀      multiplicand
 × 100111  :  39₁₀      multiplier
  011001
 011001
011001                   partial
000000                   products
000000
+011001
001111001111  :  975₁₀   product
```

$$P = YX = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right)$$

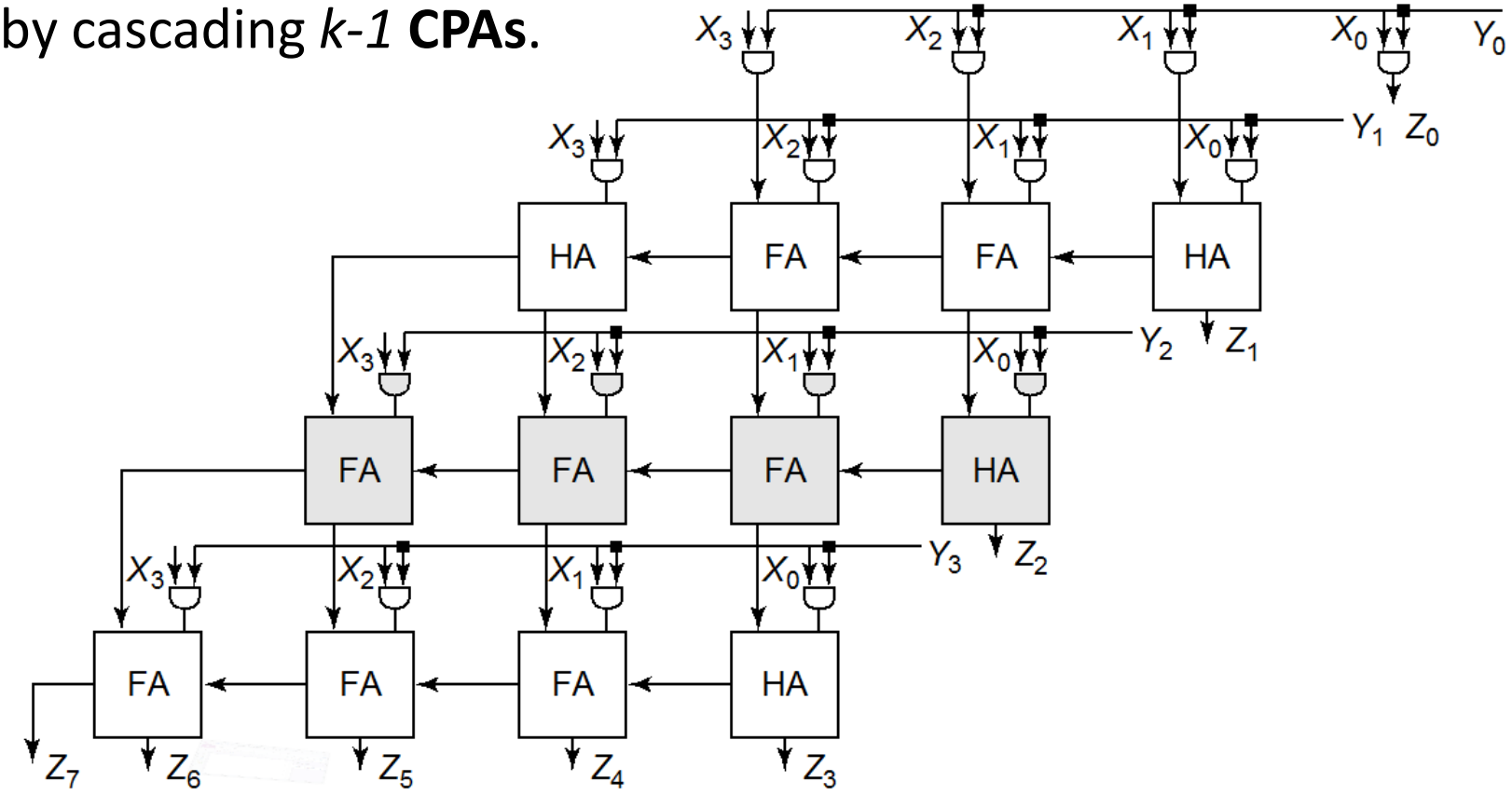
$$= \sum_{i=0}^{N-1}\sum_{i=0}^{M-1} x_i y_j 2^{i+j}$$

						y_5	y_4	y_3	y_2	y_1	y_0	Multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	Multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			Partial Products
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0					
	x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0						
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Product

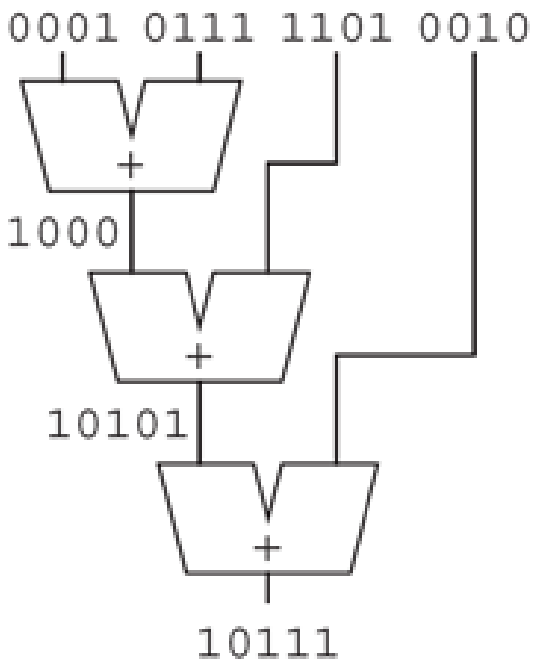
Dot diagram is convenient to illustrate large array multiplication.



The most obvious of adding *k* *N*-bit numbers is by cascading *k-1* **CPAs**.



The most obvious of adding *k* *N*-bit numbers is by cascading *k-1* **CPAs**.



This is slow and area consuming, taking *O*(*kN*) time and area (not really).

Observation:

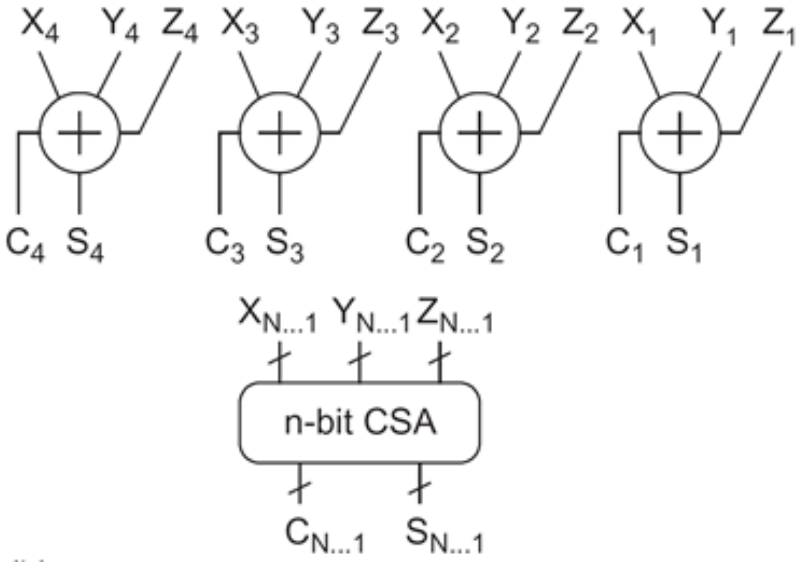
A Full-adder has three inputs *x*, *y* and *z*.

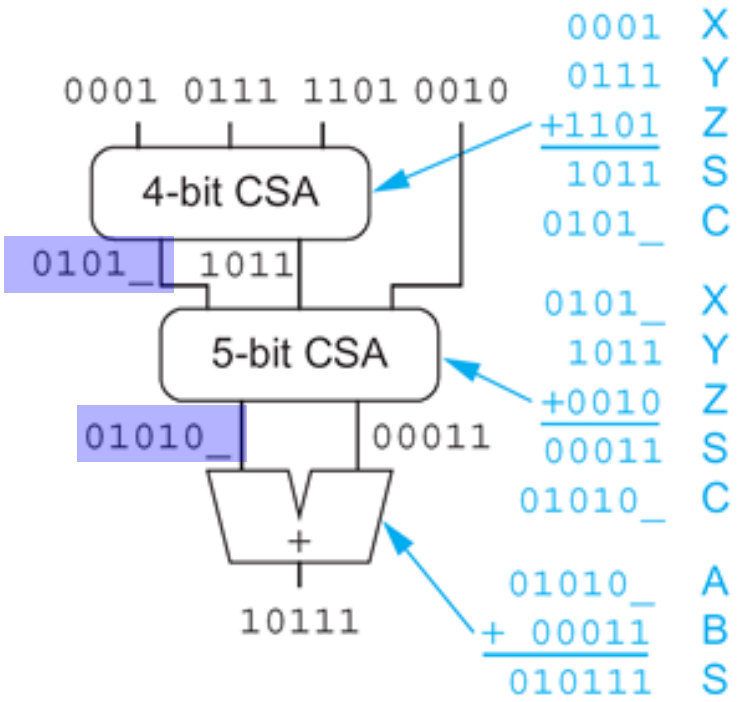
It is producing an output ***s*** of weight 1 and an output ***c*** of weight 2.

The inputs are symmetric with respect to ***s*** and ***c.***

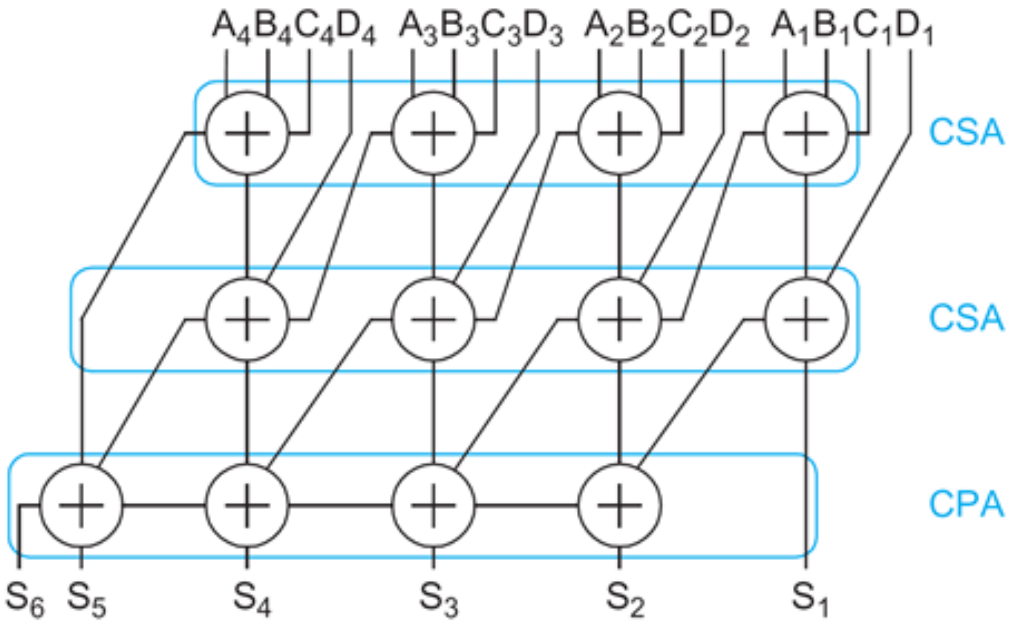
Carry-Save Adder

The sum $X+Y+Z$ can therefore be obtained by first summing $x_i+y_i+z_i$ in parallel, producing ***C*** and ***S***.





Then summing ***S*** and left shifted ***C*** by CPA. This is called ***Carry-Save Adder*** (CSA).

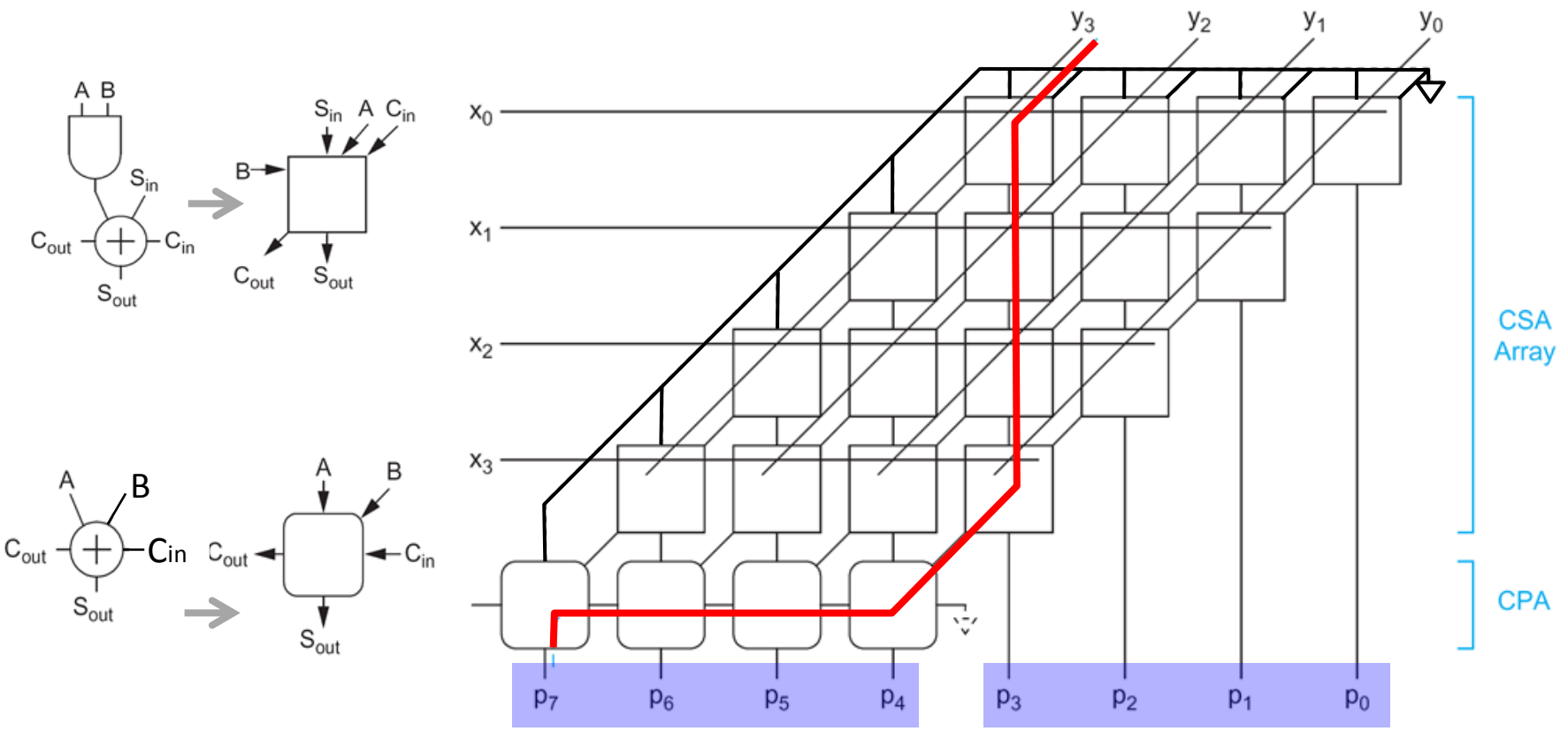


Summation of *k* numbers requires stacking *k-2* **CSA**s and a single **CPA**.

The resulting delay is *O*(*k+n*) rather than *O*(*kn*) if **CPA**s were used (not exactly…).

CSA was invented by von Neumann early digital computer (1946).

Unsigned Array Multiplication



Critical path has *N* CASs and *M*-bit CPAs, yielding *O*(*N*+*M*) delay.

The *N* LSBs are obtained directly from the sum outputs of CSAs.

The *M* MSBs are obtained by CPA.

It can be squashed in layout to occupy a rectangle.

2's Complement Array Multiplication

Same CSA array multiplication can be used.

$$P = yx = \left(-y_{M-1}2^{M-1} + \sum_{j=0}^{M-2} y_j 2^j \right)\left(-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i \right)$$ 2's complement

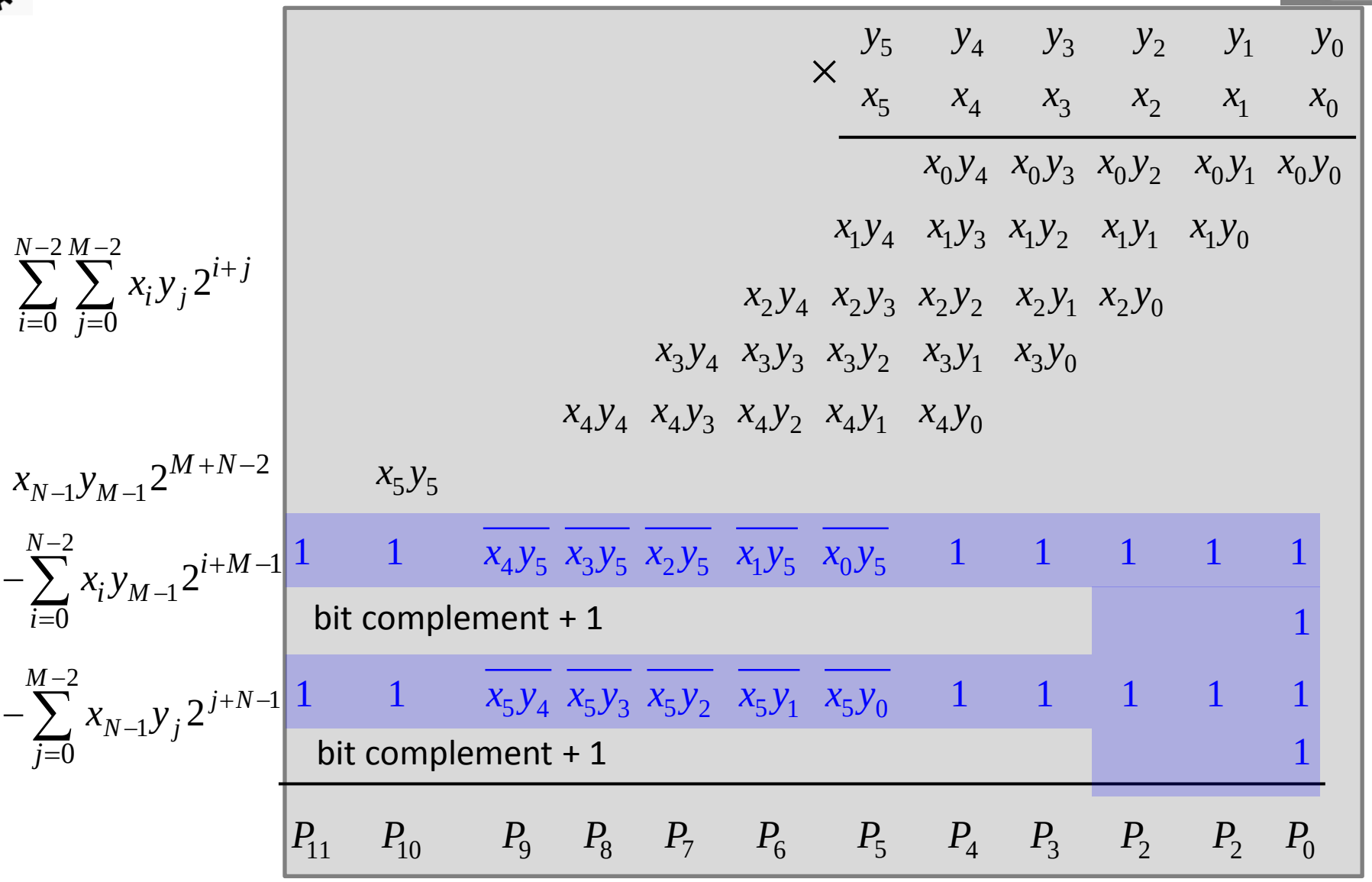
$$= \sum_{i=0}^{N-2}\sum_{j=0}^{M-2} x_i y_j 2^{i+j} + x_{N-1}y_{M-1}2^{M+N-2}$$ positive

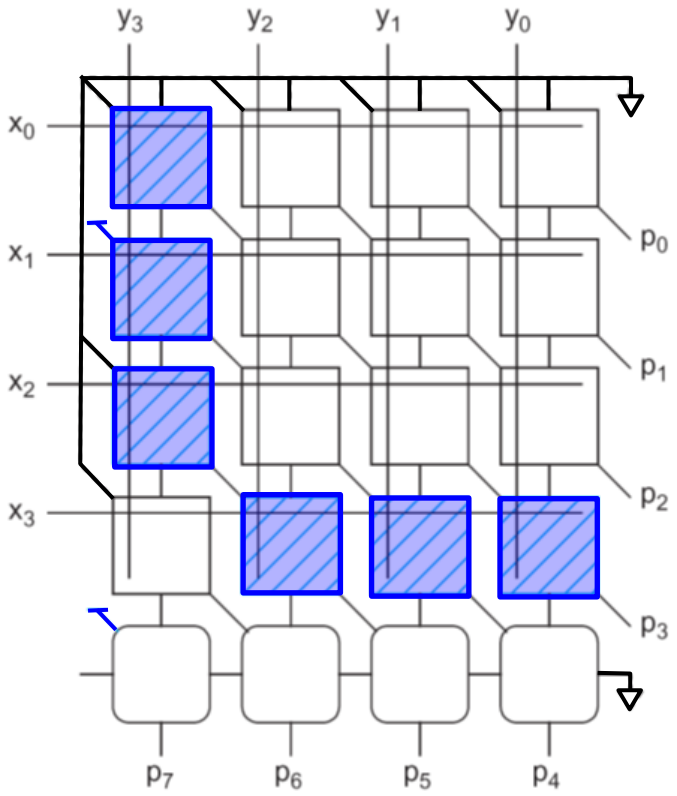
$$-\left(\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1} \right)$$ negative

To handle the negative part, 2's complement will be used.

Recall that 2's complement equals 1's complement plus 1.

1's complement is obtained by bit complement.


$\sum_{i=0}^{N-2}\sum_{j=0}^{M-2} x_i y_j 2^{i+j}$
$x_{N-1}y_{M-1}2^{M+N-2}$
$-\sum_{i=0}^{N-2} x_i y_{M-1} 2^{i+M-1}$
$-\sum_{j=0}^{M-2} x_{N-1} y_j 2^{j+N-1}$
$\times$
$y_5\ y_4\ y_3\ y_2\ y_1\ y_0$
$x_5\ x_4\ x_3\ x_2\ x_1\ x_0$
$x_0y_4\ x_0y_3\ x_0y_2\ x_0y_1\ x_0y_0$
$x_1y_4\ x_1y_3\ x_1y_2\ x_1y_1\ x_1y_0$
$x_2y_4\ x_2y_3\ x_2y_2\ x_2y_1\ x_2y_0$
$x_3y_4\ x_3y_3\ x_3y_2\ x_3y_1\ x_3y_0$
$x_4y_4\ x_4y_3\ x_4y_2\ x_4y_1\ x_4y_0$
x_5y_5
1 1 $\overline{x_4y_5}$ $\overline{x_3y_5}$ $\overline{x_2y_5}$ $\overline{x_1y_5}$ $\overline{x_0y_5}$ 1 1 1 1 1
bit complement + 1
1
1 1 $\overline{x_5y_4}$ $\overline{x_5y_3}$ $\overline{x_5y_2}$ $\overline{x_5y_1}$ $\overline{x_5y_0}$ 1 1 1 1 1
bit complement + 1
1
P_{11} P_{10} P_9 P_8 P_7 P_6 P_5 P_4 P_3 P_2 P_2 P_0



Physical layout

Notice how all 1s were summed and propagated leftward.

$$
\begin{array}{cccccccccccc}
 & & & & & & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 \\
 & & & & & & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
 & & & & & 1 & \overline{x_5y_0} & x_0y_4 & x_0y_3 & x_0y_2 & x_0y_1 & x_0y_0 \\
 & & & & & \overline{x_5y_1} & x_1y_4 & x_1y_3 & x_1y_2 & x_1y_1 & x_1y_0 & \\
 & & & & \overline{x_5y_2} & x_2y_4 & x_2y_3 & x_2y_2 & x_2y_1 & x_2y_0 & & \\
 & & & \overline{x_5y_3} & x_3y_4 & x_3y_3 & x_3y_2 & x_3y_1 & x_3y_0 & & & \\
 & & \overline{x_5y_4} & x_4y_4 & x_4y_3 & x_4y_2 & x_4y_1 & x_4y_0 & & & & \\
\hline
P_{11} & P_{10} & P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

Acceleration of Serial Multiplication

Observation: $2^j + 2^{j-1} + \ldots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$

Consequently, additions occurring by a string of 1s in the multiplier can be replaced by an addition and a subtraction.

Bits x_{i-1} and x_i of the multiplier are encoded in y_i as follows:

$(x_i,x_{i-1}) = (00) \Rightarrow y_i = 0$; No string of 1s in sight

$(x_i,x_{i-1}) = (01) \Rightarrow y_i = 1$; End of string of 1s

$(x_i,x_{i-1}) = (10) \Rightarrow y_i = \bar{1}$; Beginning of string of 1s

$(x_i,x_{i-1}) = (11) \Rightarrow y_i = 0$; Continuation of string of 1s

Example: radix-2 encoding of a 16-bit word artifact

1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	0	x
$\bar{1}$	0	1	0	0	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	1	0	0	$\bar{1}$	0		y

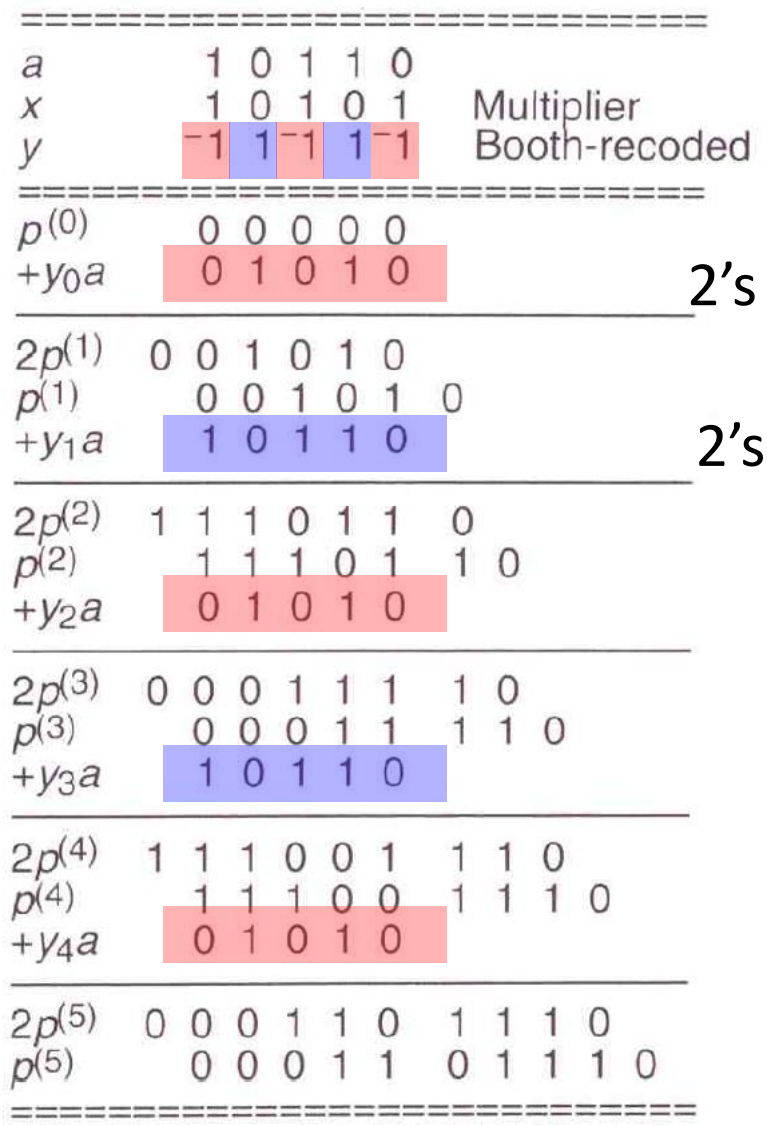
	1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	0	*x*
(1)	$\bar{1}$	0	1	0	0	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	1	0	0	$\bar{1}$	0		*y*

Above is a proper interpretation for signed multiplication. A MSB string 111…111 of 1s will be encoded into a string of 000…00$\bar{1}$, resulting in appropriate subtraction.

Problem: Assume that the unsigned value of X is intended.

Booth encoding results in -2^{15} rather than $+2^{15}$.

Solution: Add 2^{16} by extending *y* with 1 MSB.



2's Comp. addition (negative)

2's Comp. addition (positive)

Booth Encoding

Proposed by Booth in 1951 to accelerate serial multiplication (series of shift and add).

$P = Y \times X = Y \times 00111110$ requires 5 shifts and additions.

$$Y \times 00111110 = Y \times \left(2^5 + 2^4 + 2^3 + 2^2 + 2^1\right)$$

$$= Y \times \left(2^6 - 2^1\right) = Y \times \left(01000000 - 00000010\right)$$

requires 1 add, 1 subtract (add 2's complement) and 2 shifts.

$$Y \times 00111010 = Y \times \left(2^5 + 2^4 + 2^3 + 2^1\right) = Y \times \left[\left(2^6 - 2^3\right) + 2^1\right]$$

$$= Y \times \left(01000000 - 00001000 + 00000010\right)$$

Multiplication can be considerably accelerated by turning sequences of 1s into leading and trailing 1s.

$$P = \overbrace{011001}^{Y} \times \overbrace{100111}^{X} = \overbrace{011001}^{Y} \times \overbrace{\underbrace{10}_{2}\underbrace{01}_{1}\underbrace{11}_{3}}^{X}$$

Instead of multiplying *Y* and adding bit-by-bit of *X* we look at groups of 2 bits, hence working in **radix-4**.

In radix-4 each partial product has 4 times the weight of its predecessor one.

Radix-4 multiplication will **reduce to half** the number of partial products, with 2-bit left shift at each one. The partial products are {0, *Y*, 2*Y*, 3*Y*}.

3*Y* is a problem since it cannot be obtained by a shift but rather requires addition $3Y=2Y+Y$.

Radix-4 algorithm implements $3Y= 4Y -Y$ and $2Y= 4Y -2Y$.

Weight of LSB in current pair is twice the MSB in previous.

Weight of MSB in current pair is 4 times the MSB in previous.

$$P = \overbrace{011001}^{Y} \times \overbrace{100111}^{X}$$

X=3. PP=–*Y*. 4*Y* will be discovered in next step.

X=1. PP=2*Y*. 4*Y* is carried from previous, which is 1 in current.

X=2. PP=–2*Y*. 4*Y* will be discovered in next step.

X=0. PP=*Y*. No need for sign. Always *Y* or 0.

Sign ext. Y's 2's comp.

1 1 1 1 1 1 1 0 0 1 1 1

Sign ext.

0 0 0 0 1 1 0 0 1 0

Sign ext.

1 1 1 0 0 1 1 1 0

0 1 1 0 0 1

0 0 1 1 1 1 0 0 1 1 1 1

0	artifact
1	LSB
1	
1	
0	*X*
0	
1	MSB
0	artifact
0	

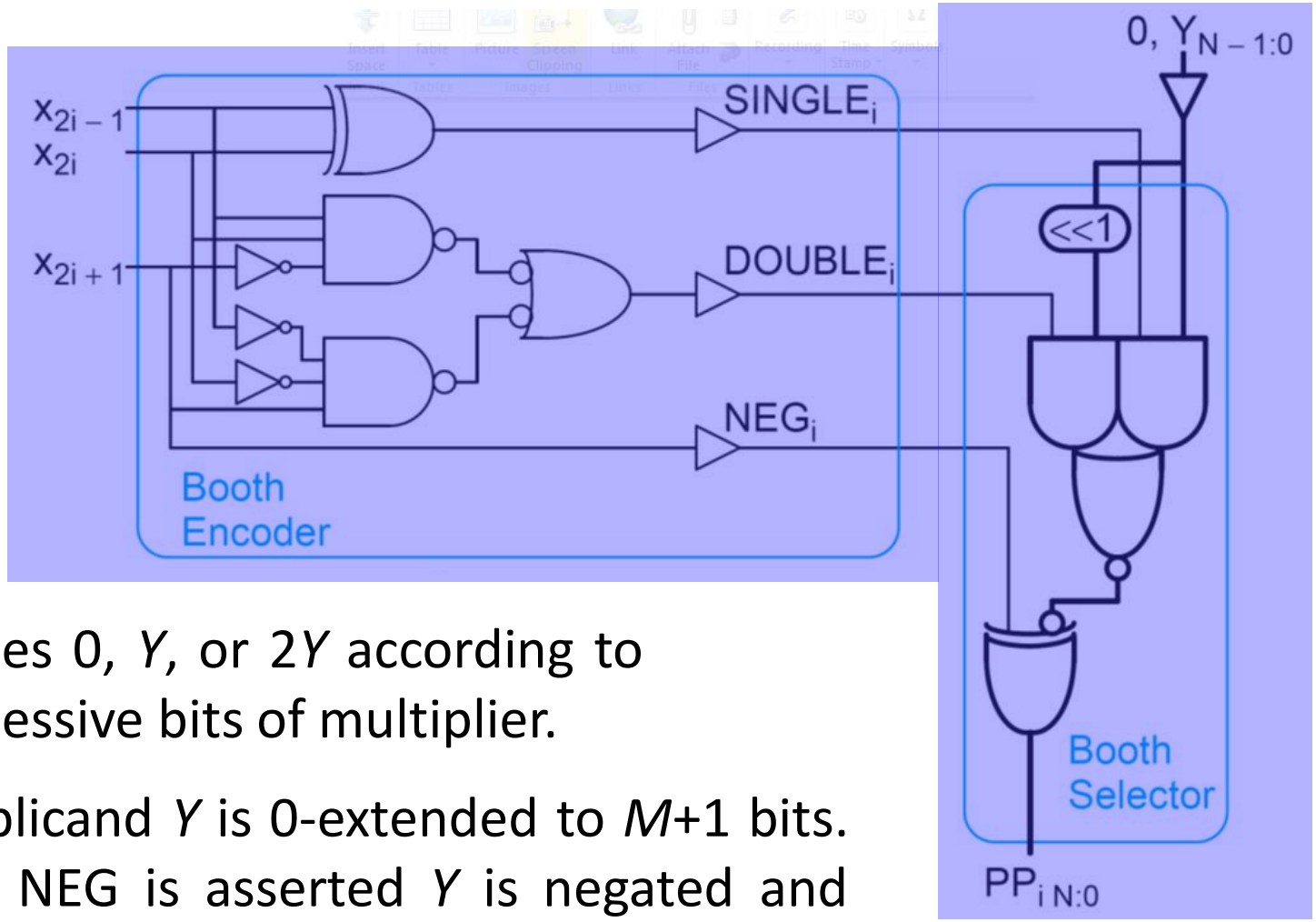
PP table defines the appropriate encoding of multiplicand: 0, *Y*, - *Y*, 2*Y* or -2*Y.*

Partial Product (PP) Selection Table		
Multiplier	Selection	Explanations

Radix-4 modified Booth encoding values

Inputs			Partial Products	Booth Selects		
X_{2i+1}	X_{2i}	X_{2i-1}	*PP i*	SINGLE *i*	DOUBLE *i*	NEG *i*
0	0	0	0	0	0	0
0	0	1	*Y*	1	0	0
0	1	0	*Y*	1	0	0
0	1	1	*2Y*	0	1	0
1	0	0	*-2Y*	0	1	1
1	0	1	*-Y*	1	0	1
1	1	0	*-Y*	1	0	1
1	1	1	-0(=0)	0	0	1

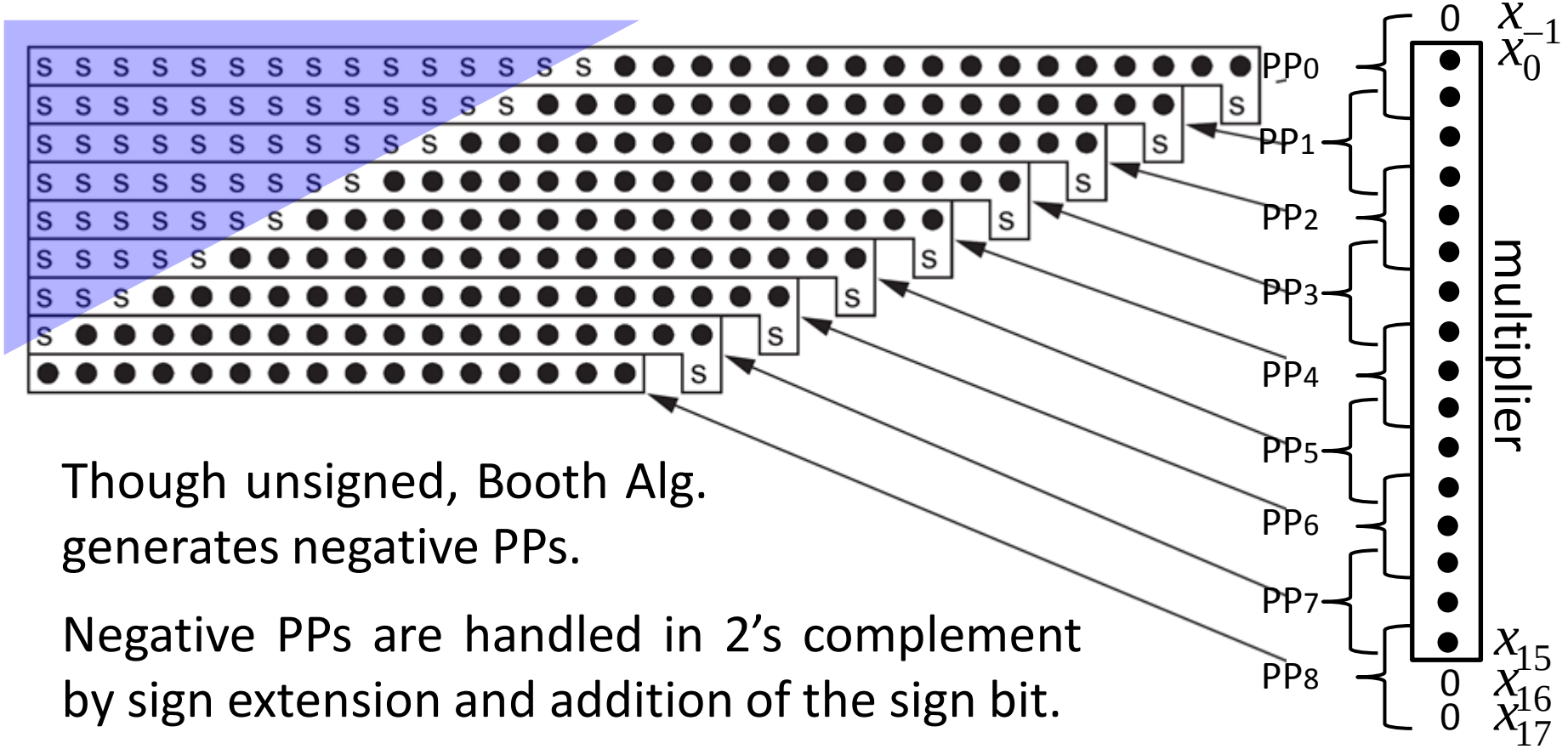
Radix-4 Booth encoder and selector



Encodes 0, *Y*, or 2*Y* according to 3 successive bits of multiplier.

Multiplicand *Y* is 0-extended to *M*+1 bits. If the NEG is asserted *Y* is negated and extra 1 is added in the next row.

Radix-4 Booth-encoded partial products with sign extension for unsigned multiplication

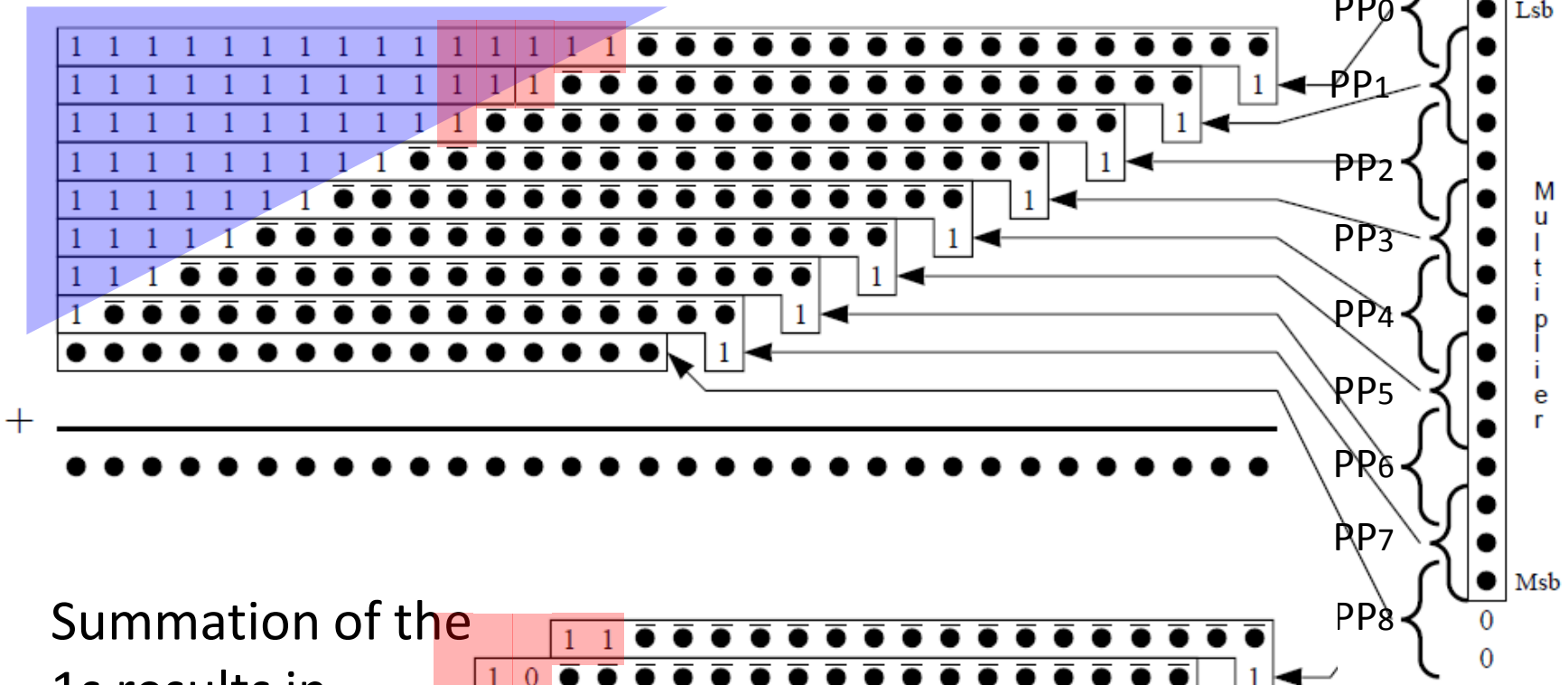


Though unsigned, Booth Alg. generates negative PPs.

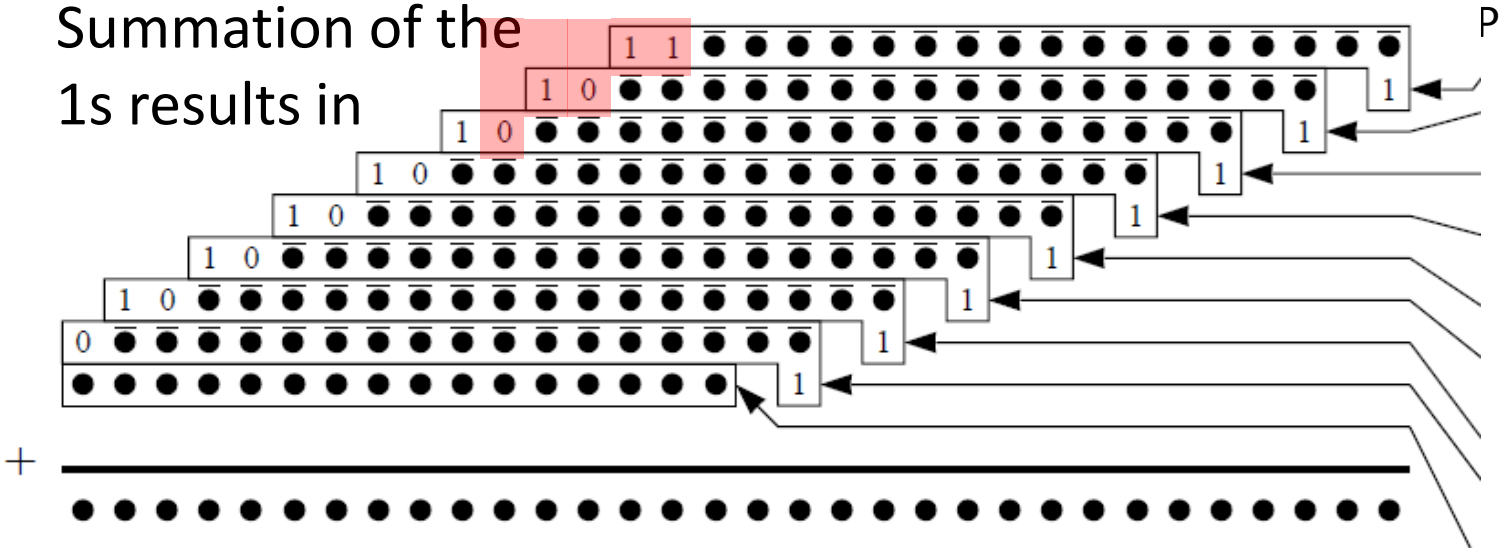
Negative PPs are handled in 2's complement by sign extension and addition of the sign bit.

To squash into rectangular floor plan the sign bit triangle should better be out.

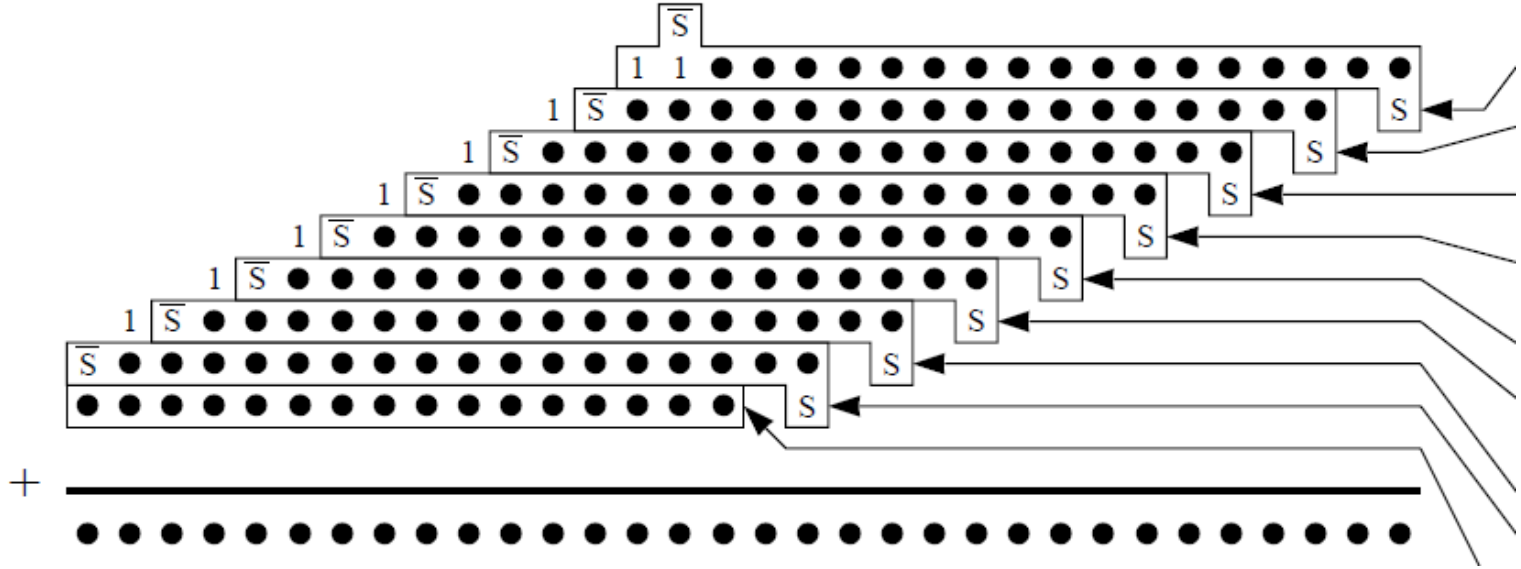
Suppose that all partial products are negative.



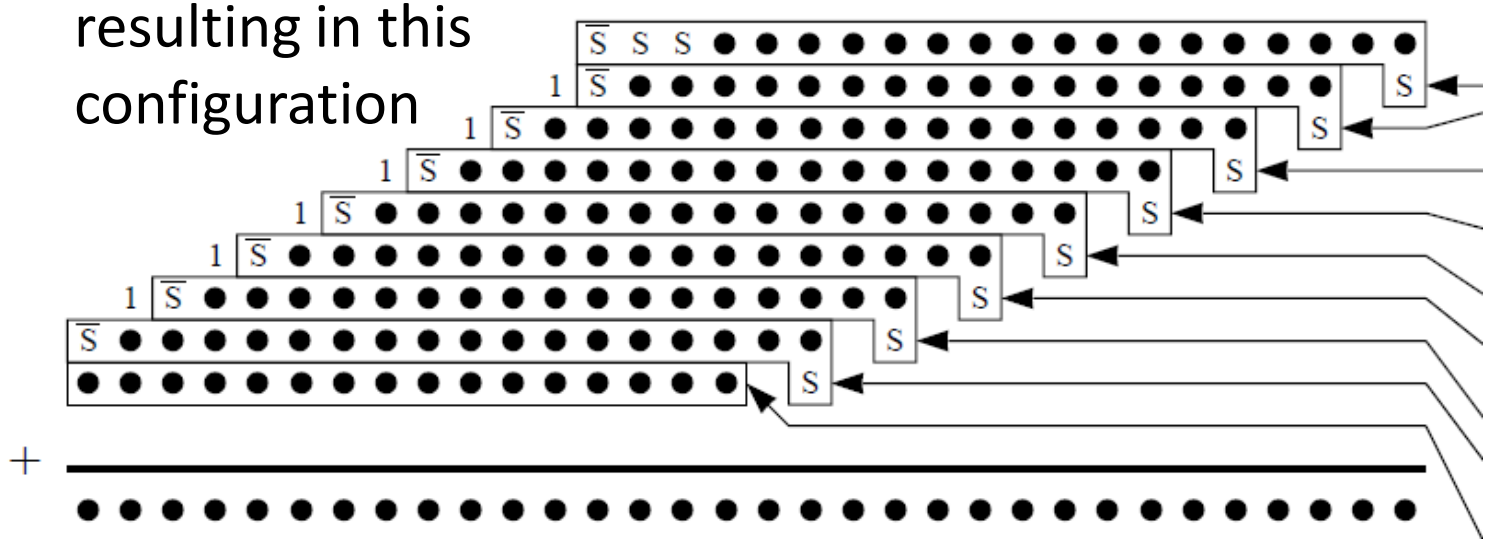
Summation of the 1s results in

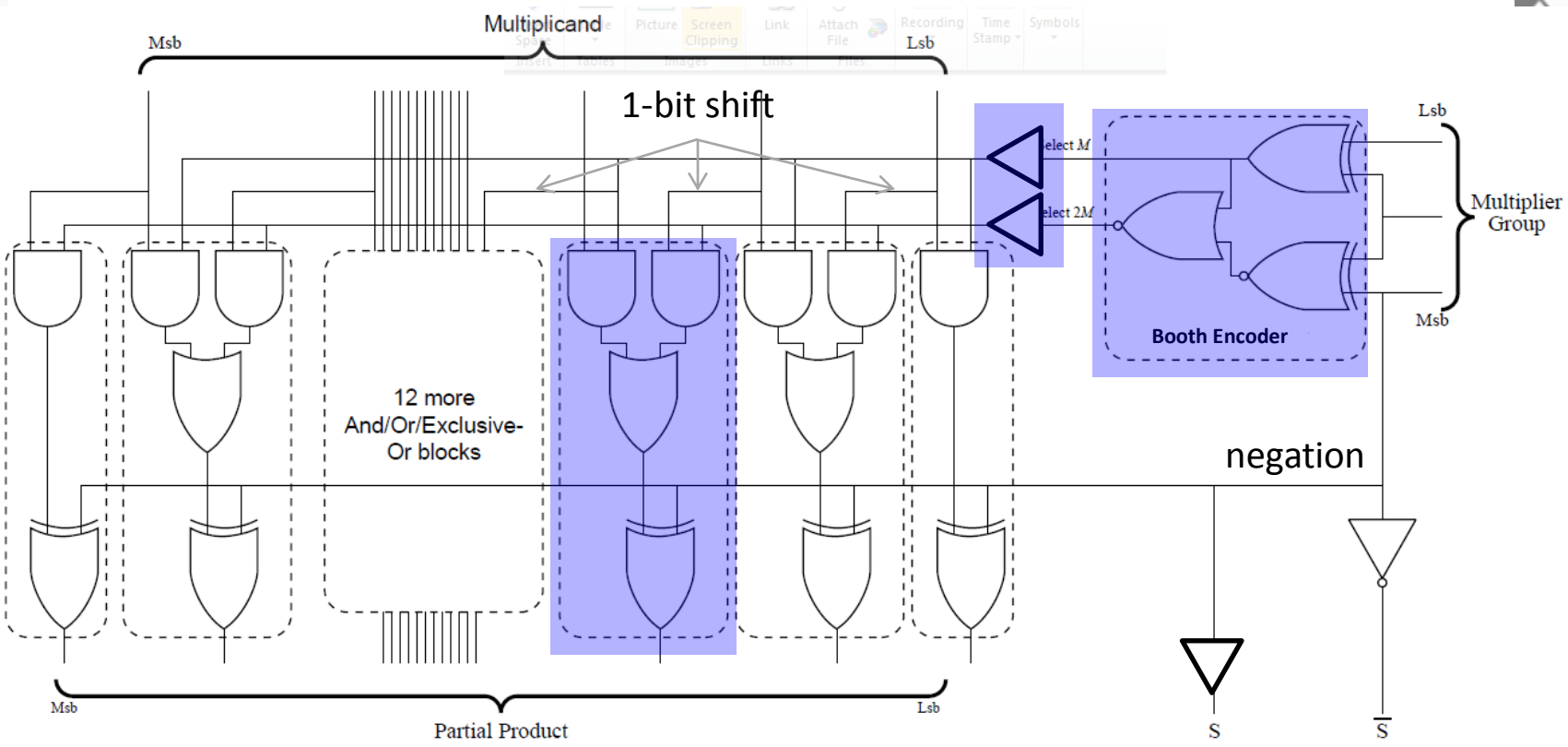


If a particular PP is positive, the negation can be reverted by adding 1 to the LSB of the original 1s string.



resulting in this configuration

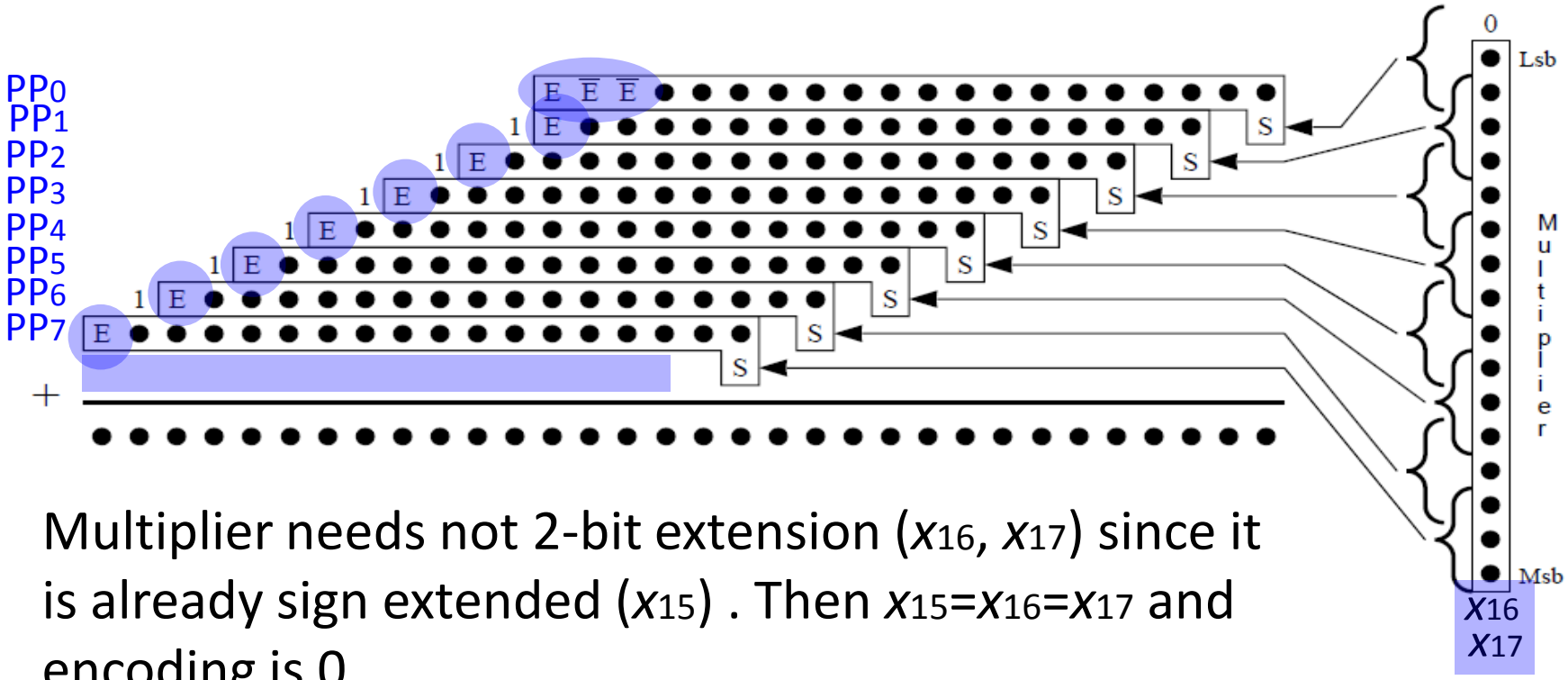




Critical path involves: Booth encoder, select line driver, Booth selector, $N/2$ CSAs and final CPA.

Selector resides in every bit of the array, consuming significant area. Good area/power/performance tradeoff is to downsize it as much as possible. (why?)

Booth Encoding Signed Multiplier



Multiplier needs not 2-bit extension (x_{16}, x_{17}) since it is already sign extended (x_{15}) . Then $x_{15}=x_{16}=x_{17}$ and encoding is 0.

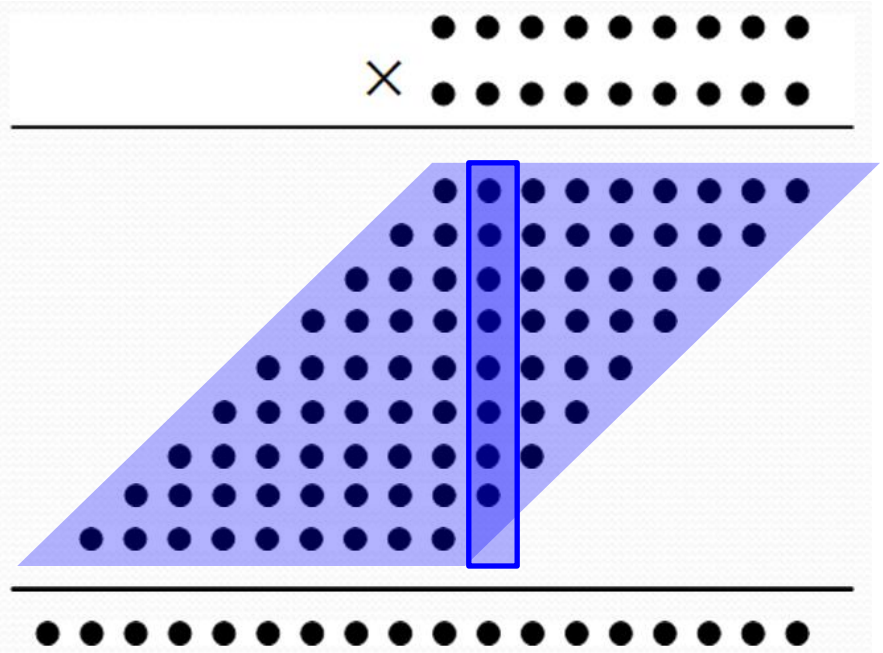
PP_8 is therefore not required.

Wallace Tree Multiplication

Consider the following 9-bit unsigned multiplication

Every dot of the array represents a partial product.

Partial products are vertically summed by half and full adders (CSA).



Multiplication time complexity is $O(n)$, There are n-2 sequential CSAs additions.

Wallace tree accelerates CSAs time complexity to $O(\log n)$ by different organization of CSAs sums.

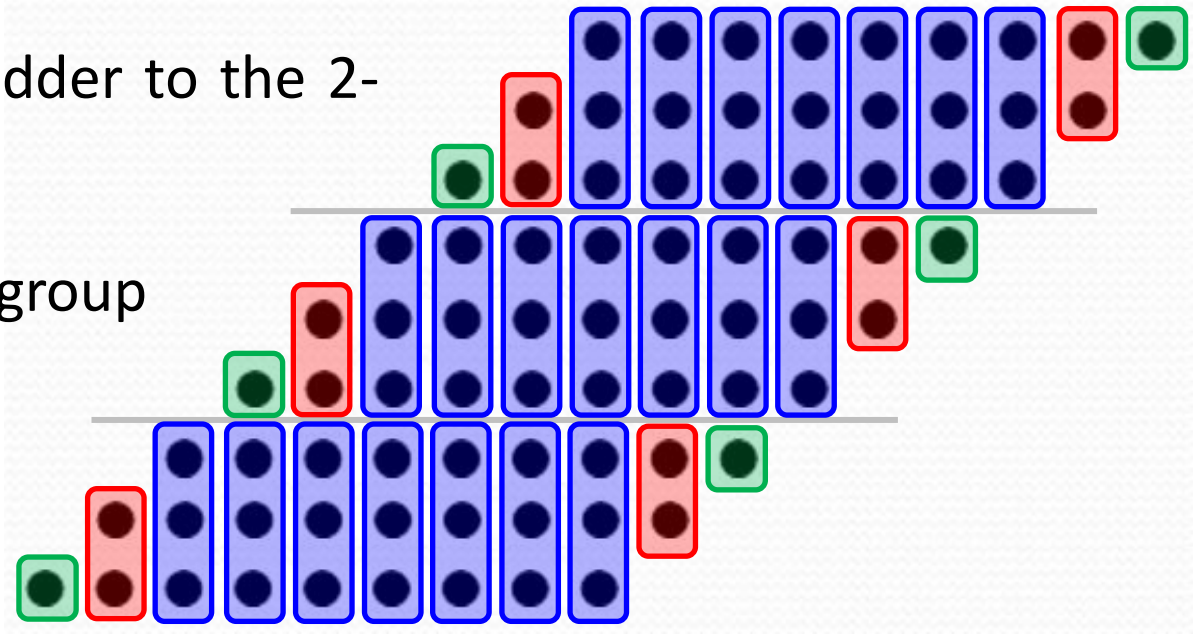
In each column of partial products, every three adjacent rows construct a group.

Reduction in each group is done by one of the following cases:

Applying a full adder (CSA) to the 3-bit groups

Applying a half adder to the 2-bit groups

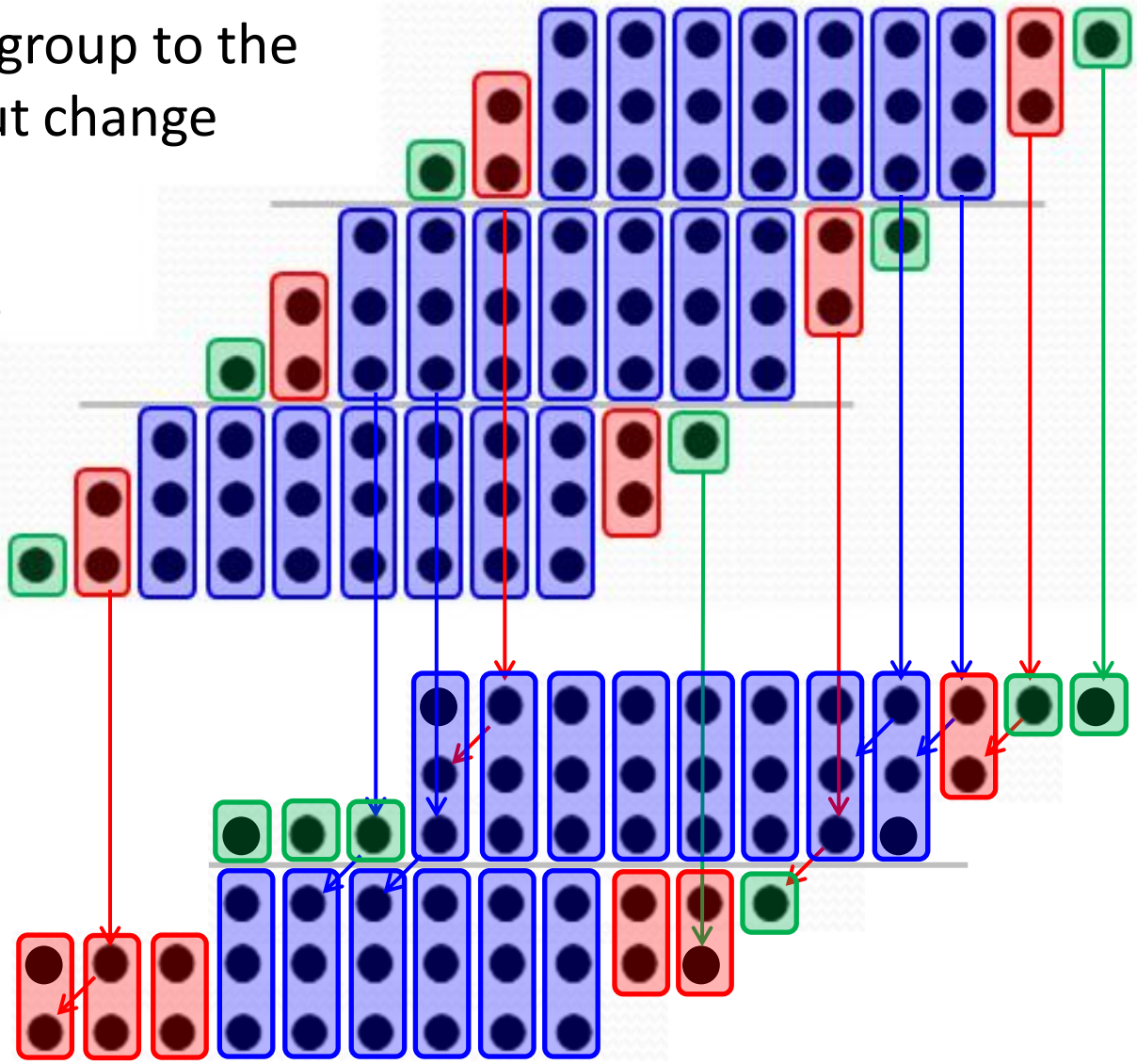
Passing any 1-bit group to the next stage without change

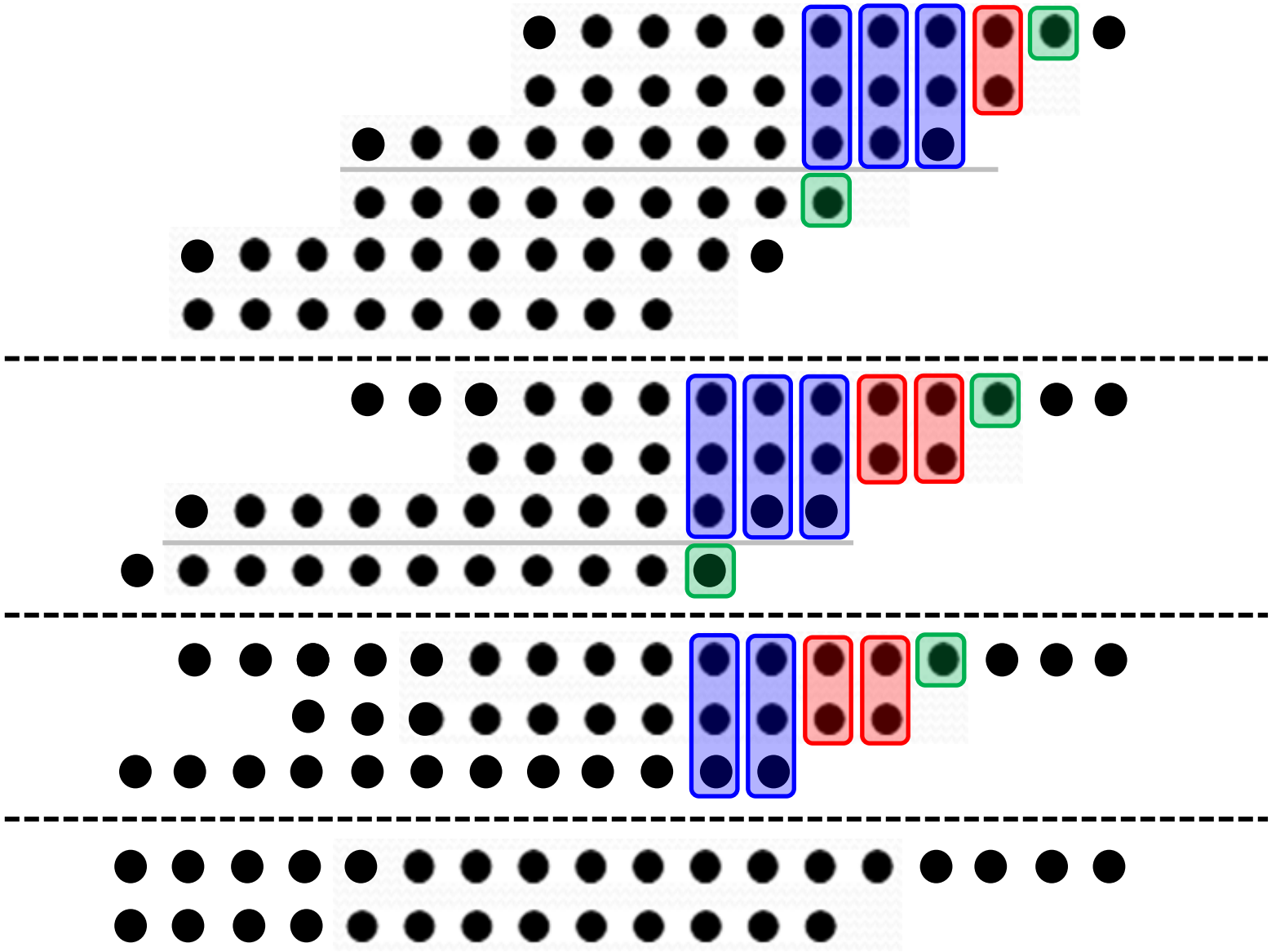


Passing any 1-bit group to the next stage without change

Sum of half adder stays in column, carry sent to next column.

Sum of full adder stays in column, carry sent to next column.





All the full-adder (CSA) and half-adder additions in a stage are performed simultaneously.

Every stage has its own adders.

Data is progressing through $O(\log_{3/2} n)$ stages (proven below).

The final two rows are summed by CPA.

Other groups organizations called ***Modified Wallace*** and ***Dadda*** reductions, yielding slight area improvement (number of circuits), are possible. Asymptotically all are similar.

Time and Area Complexity

At each stage of the computation each group of 3 bits is reduced to 2 bits, with at most 2 bits left over.

The depth of Wallace tree $D(n)$ satisfies

$$D(n) = \begin{cases} 0 & \text{if } n \le 2 \\ 1 & \text{if } n = 3 \\ 1 + D\left(\lceil 2n/3 \rceil\right) & \text{if } n \ge 4 \end{cases}$$

This is a recursive equation solved to $D(n) = \Theta(\log n)$.

The final addition is implemented by CPA.

Carry-lookahead adder takes $\Theta(\log n)$, so using CLA for final addition yields $\Theta(\log n)$ overall time complexity.

The number of adders $C(n)$ is $\Theta\left(n^2\right)$.

The number of bits in a row is between n and $2n$. There are n rows so $2/3n^2$ full and half adders are required in the first stage.

The number of rows is reducing by factor 2/3 from stage to stage, hence the total sums to $\Theta\left(n^2\right)$ as well.

Shifters

Logical shifter: Shifts the number to left or right and fills the empty spots with 0s. Specified by << or >> in Verilog.

1011 LSR 1 = 0101; 1011 LSL 1 = 0110

Arithmetic shifter: Similar to logical, but on right shift fills empty spots with sign bit. Specified by <<< or >>> in Verilog.

1011 ASR 1 = 1101; 1011 ASL 1 = 0110

Barrel shifter (rotator): Rotates numbers cyclically.

1011 ROR 1 = 1101; 1011 ROL 1 = 0111

Conceptually, rotation of *N* -bit word involves array of *N* *N*-input MUXes to select each of the outputs from each of the possible input positions. This is called ***array shifter***.

Array shifter requires a decoder to produce 1-of-N shift.

MUXes of more than 8 inputs have excessive parasitic capacitance, so it is faster to construct shifters from $\log_v N$ *v*-input MUXes. This is called ***logarithmic shifter***.

Left rotate by *k* bits is equivalent to right rotate by *N*-*k* bits. Computing *N*-*k* requires subtracter in the critical path.

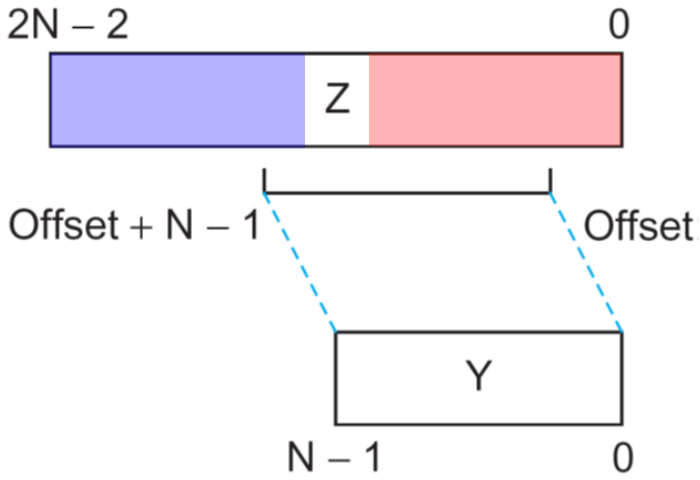
We take advantage of 2's complement and the fact that rotation is cyclic modulo *N* $N - k = N + \bar{k} + 1 = \bar{k} + 1$.

Left shift can therefore be done by first pre shifting right by 1 and then right shifting by the complement.

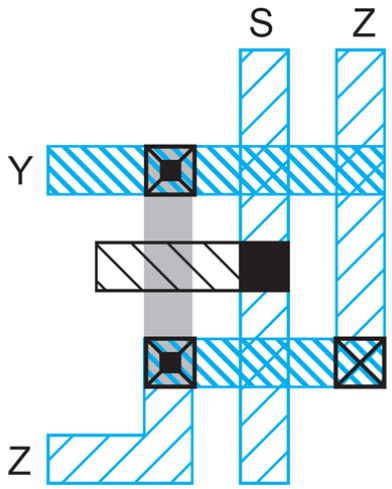
Logical and arithmetic shifts are similar to rotate except that the bits at one end or the other are replaced by 0 or sign bit.

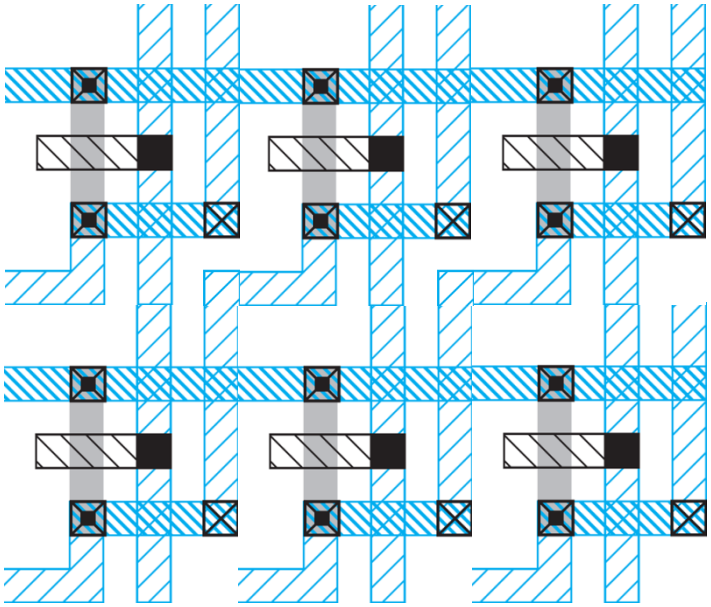
Funnel Shifter

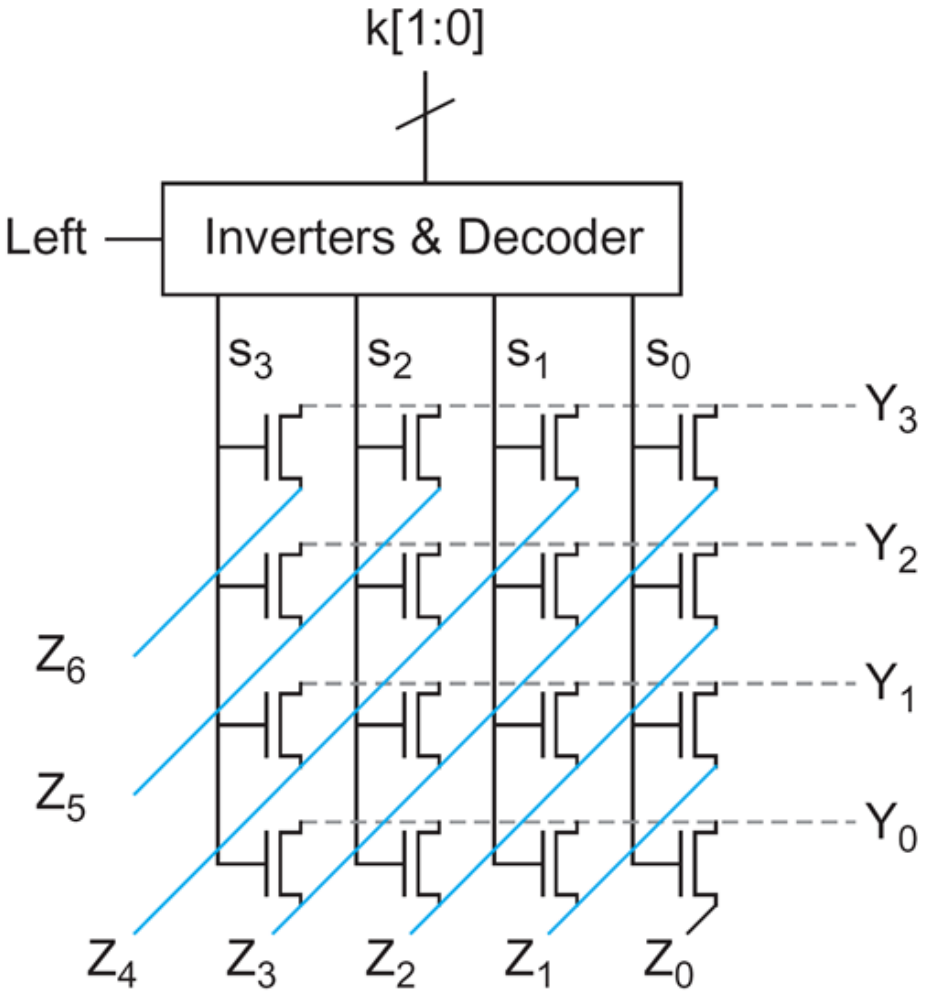
Creates a 2N-1 bit input word Z from A and kill variables. It then selects N-bit field from Z according to shift amount.



Shift Style	$Z_{2N-2:N}$	Z_{N-1}	$Z_{N-2:0}$	Offset
Logical Right	0	A_{N-1}	$A_{N-2:0}$	k
Logical Left	$A_{N-1:1}$	A_0	0	$\overline{k}$
Arithmetic Right	A_{N-1}	A_{N-1}	$A_{N-2:0}$	k
Arithmetic Left	$A_{N-1:1}$	A_0	0	$\overline{k}$
Rotate Right	$A_{N-2:0}$	A_{N-1}	$A_{N-2:0}$	k
Rotate Left	$A_{N-1:1}$	A_0	$A_{N-1:1}$	$\overline{k}$


S
Z
Y
Z




k[1:0]
Left
Inverters & Decoder
s_3
s_2
s_1
s_0
Y_3
Y_2
Y_1
Y_0
Z_6
Z_5
Z_4
Z_3
Z_2
Z_1
Z_0