CAL POLY
SAN LUIS OBISPO

EE 431: COMPUTER-AIDED DESIGN OF VLSI DEVICES

Datapath Functional Units

Nishith N. Chakraborty

November, 2024

OUTLINE

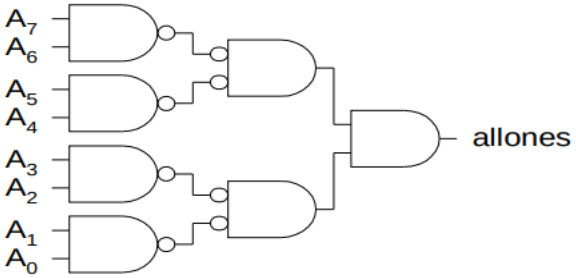
- Comparators
- Shifters
- Multi-input Adders
- Multipliers

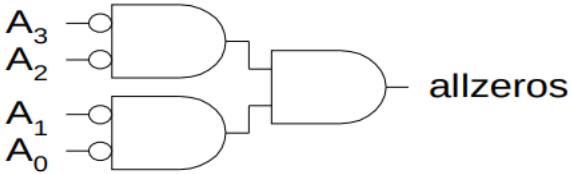
COMPARATORS

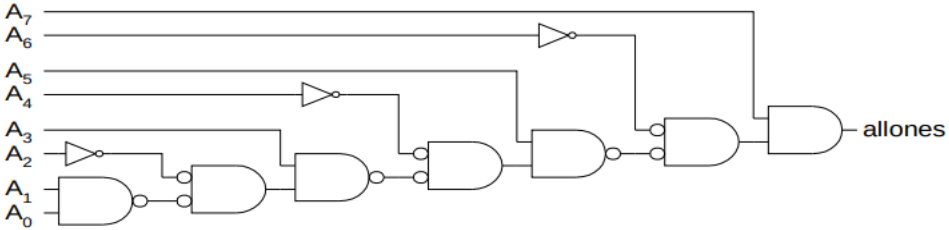
- 0's detector: A = 00…000
- 1's detector: A = 11…111
- Equality comparator: A = B
- Magnitude comparator: A < B

1'S & 0'S DETECTORS

- 1's detector: N-input AND gate
- 0's detector: NOTs + 1's detector (N-input NOR)

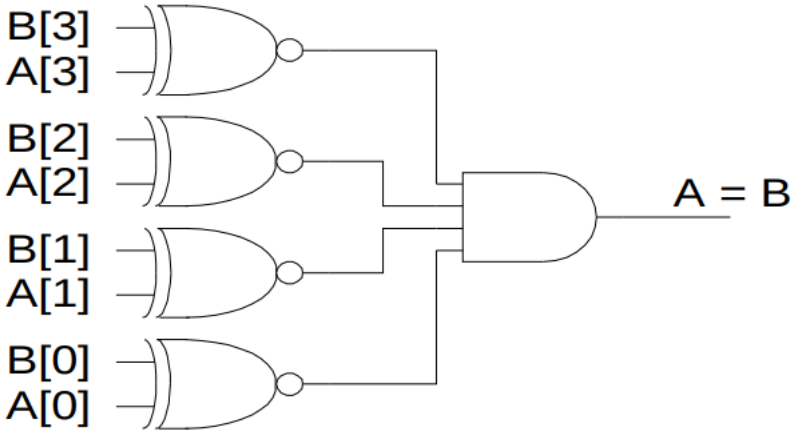






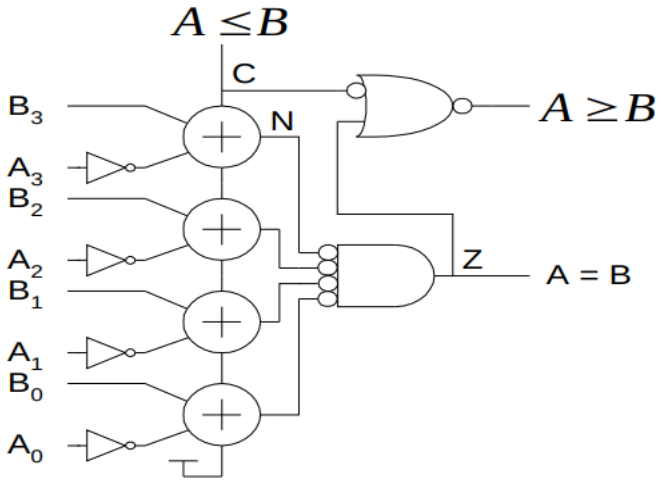
EQUALITY COMPARATOR

- Check if each bit is equal (XNOR, aka equality gate)
- 1's detect on bitwise equality



MAGNITUDE COMPARATOR

- Compute B-A and look at sign
- B-A = B + ~A + 1
- For unsigned numbers, carry out is sign bit



SIGNED VS. UNSIGNED

- For signed numbers, comparison is harder
  - C: carry out
  - Z: zero (all bits of A-B are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)

Table 10.4 Magnitude comparison

Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C + Z}$	$\overline{(N \oplus V) + Z}$
$A > B$	$\overline{C}$	$(N \oplus V)$
$A \leq B$	C	$(\overline{N \oplus V})$
$A \geq B$	$\overline{C} + Z$	$(N \oplus V) + Z$

SHIFTERS

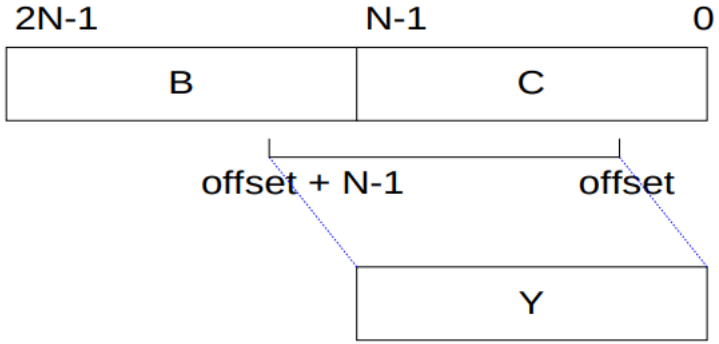
- Logical Shift:
  - Shifts number left or right and fills with 0's
    - 1011 LSR 1 =___ 1011 LSL1 = ___
- Arithmetic Shift:
  - Shifts number left or right. Right shift sign extends
    - 11011 ASR1 =___ 1011 ASL1 =___
- Rotate:
  - Shifts number left or right and fills with lost bits
    - 1011 ROR1 =___ 1011 ROL1 =___

SHIFTERS

- Logical Shift:
  - Shifts number left or right and fills with 0's
    - 1011 LSR 1 = 0101 1011 LSL1 = 0110
- Arithmetic Shift:
  - Shifts number left or right. Right shift sign extends
    - 1011 ASR1 = 1101 1011 ASL1 = 0110
- Rotate:
  - Shifts number left or right and fills with lost bits
    - 1011 ROR1 = 1101 1011 ROL1 = 0111

FUNNEL SHIFTER

- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from 2N-bit input
  - Shift by k bits ($0 \leq k < N$)



FUNNEL SHIFTER OPERATION

Table 10.10 Funnel shifter operation

Shift Type	*B*	*C*	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N-k$
Arithmetic Right			
Arithmetic Left			
Rotate Right			
Rotate Left			

- Computing N-k requires an adder

FUNNEL SHIFTER OPERATION

Table 10.10 Funnel shifter operation

Shift Type	*B*	*C*	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N-k$
Arithmetic Right	$A_{N-1}…A_{N-1}$ (sign extension)	$A_{N-1}…A_0$	k
Arithmetic Left			
Rotate Right			
Rotate Left			

- Computing N-k requires an adder

FUNNEL SHIFTER OPERATION

Table 10.10 Funnel shifter operation

Shift Type	*B*	*C*	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N-k$
Arithmetic Right	$A_{N-1}…A_{N-1}$ (sign extension)	$A_{N-1}…A_0$	k
Arithmetic Left	$A_{N-1}…A_0$	0	$N-k$
Rotate Right			
Rotate Left			

- Computing N-k requires an adder

FUNNEL SHIFTER OPERATION

Table 10.10 Funnel shifter operation

Shift Type	*B*	*C*	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N-k$
Arithmetic Right	$A_{N-1}…A_{N-1}$ (sign extension)	$A_{N-1}…A_0$	k
Arithmetic Left	$A_{N-1}…A_0$	0	$N-k$
Rotate Right	$A_{N-1}…A_0$	$A_{N-1}…A_0$	k
Rotate Left			

- Computing N-k requires an adder

FUNNEL SHIFTER OPERATION

Table 10.10 Funnel shifter operation

Shift Type	*B*	*C*	Offset
Logical Right	0…0	$A_{N-1}…A_0$	k
Logical Left	$A_{N-1}…A_0$	0…0	$N-k$
Arithmetic Right	$A_{N-1}…A_{N-1}$ (sign extension)	$A_{N-1}…A_0$	k
Arithmetic Left	$A_{N-1}…A_0$	0	$N-k$
Rotate Right	$A_{N-1}…A_0$	$A_{N-1}…A_0$	k
Rotate Left	$A_{N-1}…A_0$	$A_{N-1}…A_0$	$N-k$

- Computing N-k requires an adder

SIMPLIFIED SHIFTER OPERATION

- Optimize down to 2N-1 bit input

Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right		
Arithmetic Left		
Rotate Right		
Rotate Left		

SIMPLIFIED SHIFTER OPERATION

- Optimize down to 2N-1 bit input

Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left		
Rotate Right		
Rotate Left		

SIMPLIFIED SHIFTER OPERATION

- Optimize down to 2N-1 bit input

Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Rotate Right		
Rotate Left		

SIMPLIFIED SHIFTER OPERATION

- Optimize down to 2N-1 bit input

Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Rotate Right	$A_{N-2}...A_0, A_{N-1}...A_0$	k
Rotate Left		

SIMPLIFIED SHIFTER OPERATION

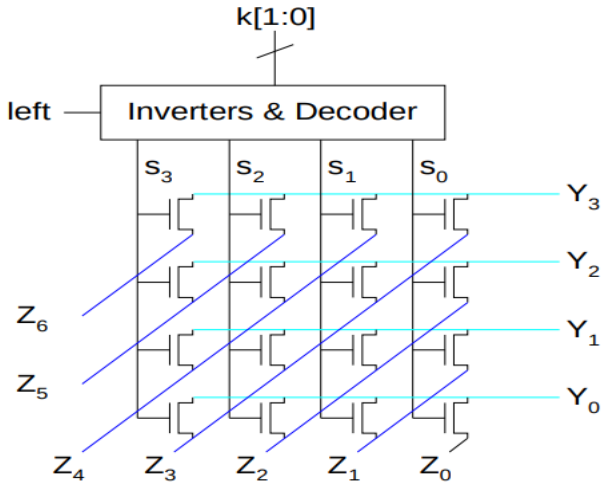
- Optimize down to 2N-1 bit input

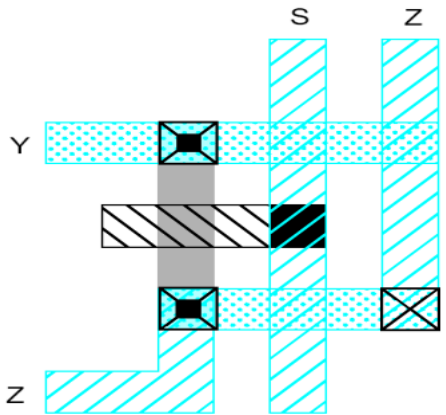
Table 10.11 Simplified funnel shifter

Shift Type	Z	Offset
Logical Right	$0..0, A_{N-1}...A_0$	k
Logical Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\overline{k}$
Rotate Right	$A_{N-2}...A_0, A_{N-1}...A_0$	k
Rotate Left	$A_{N-1}...A_0, A_{N-1}..A_1$	$\overline{k}$

FUNNEL SHIFTER DESIGN 1

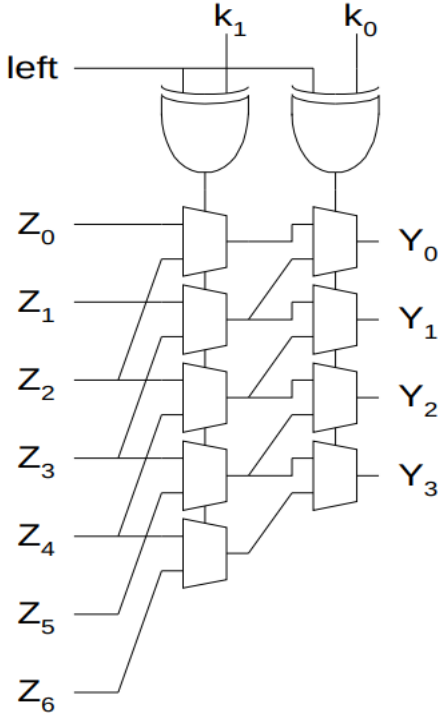
- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design (V_t drops!)





FUNNEL SHIFTER DESIGN 2

- Log N stages of 2-input muxes
- No select decoding needed



MULTI-INPUT ADDERS

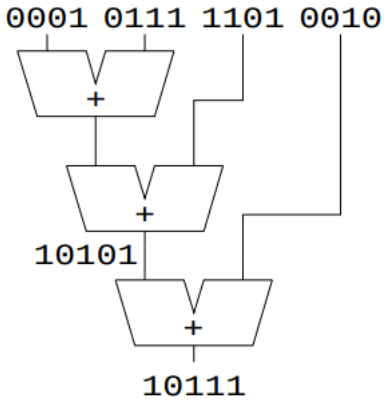
- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 =____

MULTI-INPUT ADDERS

- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111

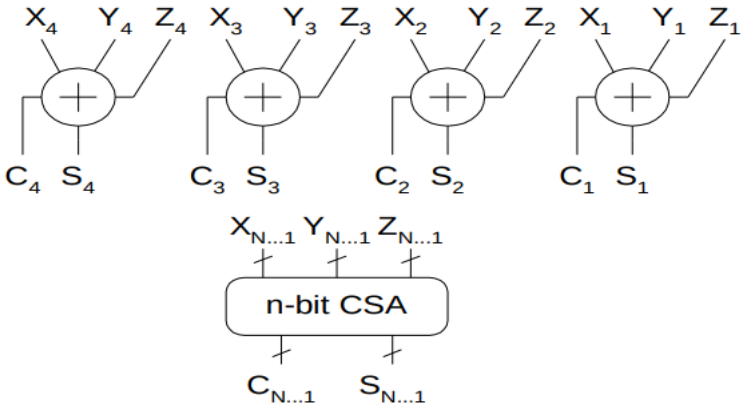
MULTI-INPUT ADDERS

- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111
- Straightforward solution: k-1 N-input CPAs
  - Large and slow



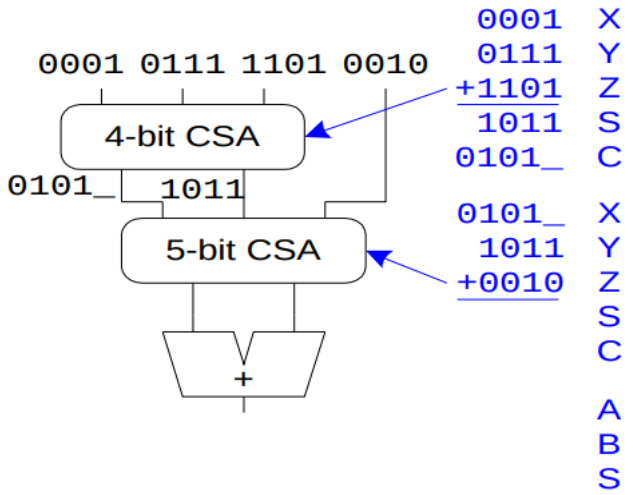
CARRY-SAVE ADDITION (CSA)

- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice weight of sum output
- N full adders in parallel are called carry save adder
  - Produce N sums and N carry outs



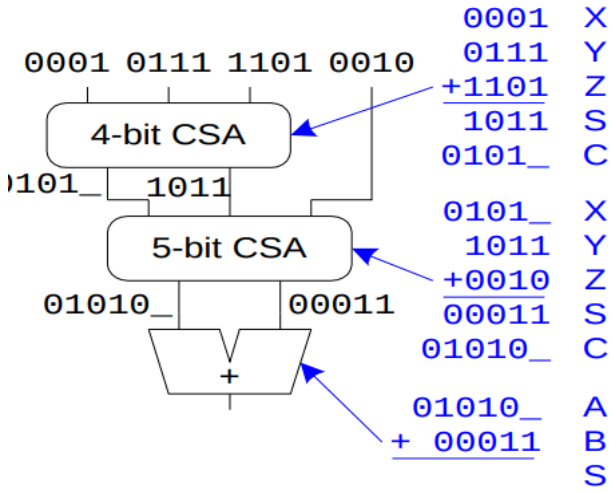
CSA APPLICATION

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result



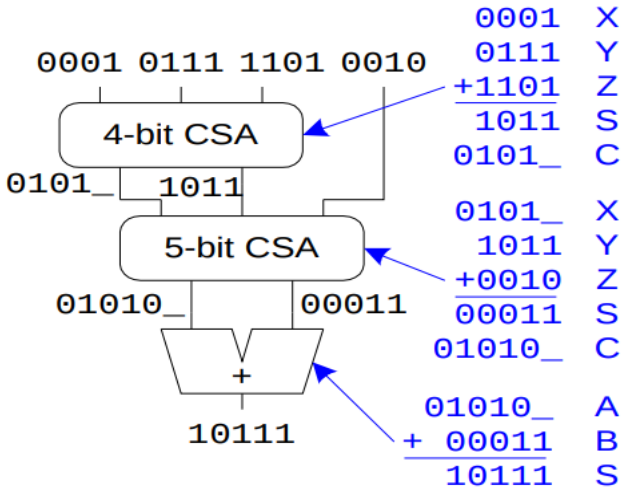
CSA APPLICATION

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result



CSA APPLICATION

- Use k-2 stages of CSAs
  - Keep result in carry-save redundant form
- Final CPA computes actual result



MULTIPLICATION

- Example:

$$\begin{array}{rcl} 1100 & : & 12_{10} \\ \underline{0101} & : & 5_{10} \end{array}$$

MULTIPLICATION

- Example:

$$\begin{array}{rcl} 1100 & : & 12_{10} \\ \underline{0101} & : & 5_{10} \\ 1100 & & \end{array}$$

MULTIPLICATION

- Example:

```
  1100  :  12(10)
  0101  :  5(10)
  1100
 0000
```

MULTIPLICATION

- Example:

$$
\begin{array}{rl}
1100 & : 12_{10} \\
0101 & : 5_{10} \\
\hline
1100 & \\
0000 & \\
1100 & \\
\hline
\end{array}
$$

MULTIPLICATION

- Example:

```
  1100  :  12(10)
  0101  :  5(10)
  1100
 0000
1100
0000
```

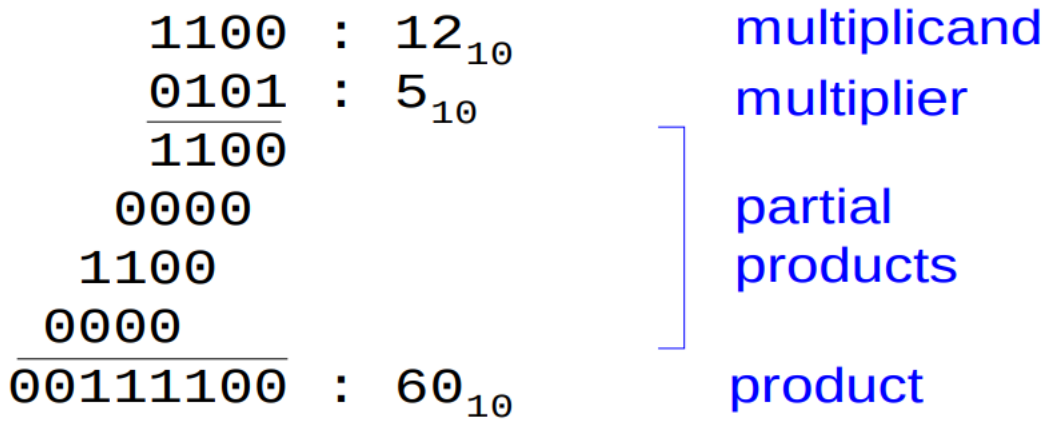
MULTIPLICATION

- Example:

$$
\begin{array}{rcl}
1100 & : & 12_{10} \\
0101 & : & 5_{10} \\
\hline
1100 & & \\
0000 & & \\
1100 & & \\
0000 & & \\
\hline
00111100 & : & 60_{10}
\end{array}
$$

MULTIPLICATION

- Example:



- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

GENERAL FORM

- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product:

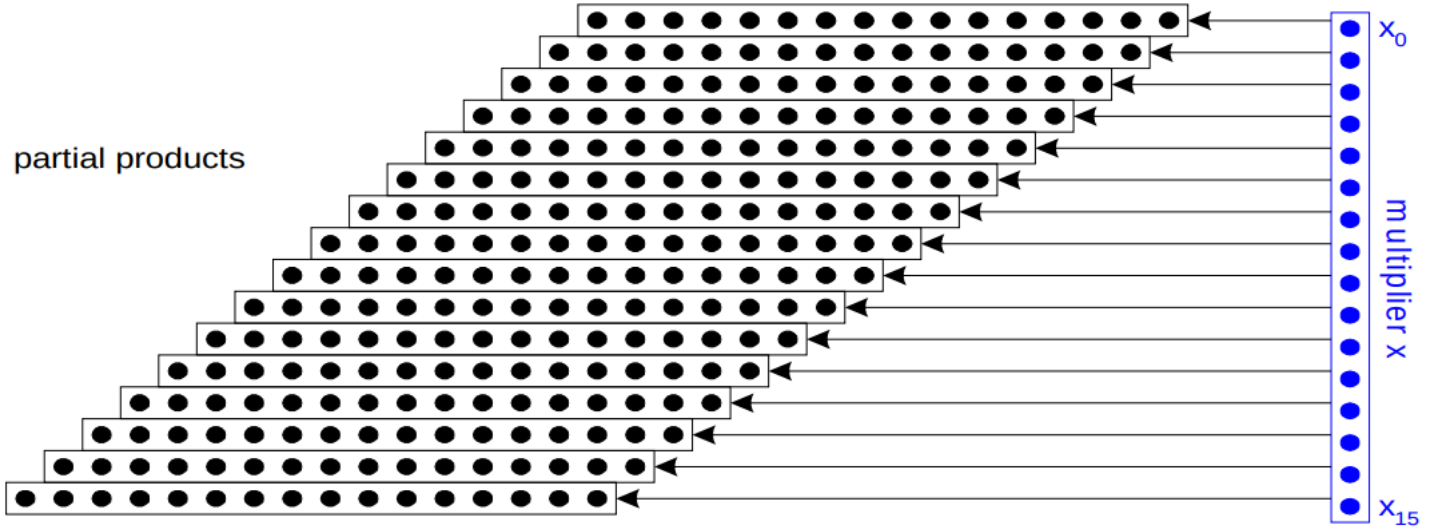
$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

						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_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		
			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

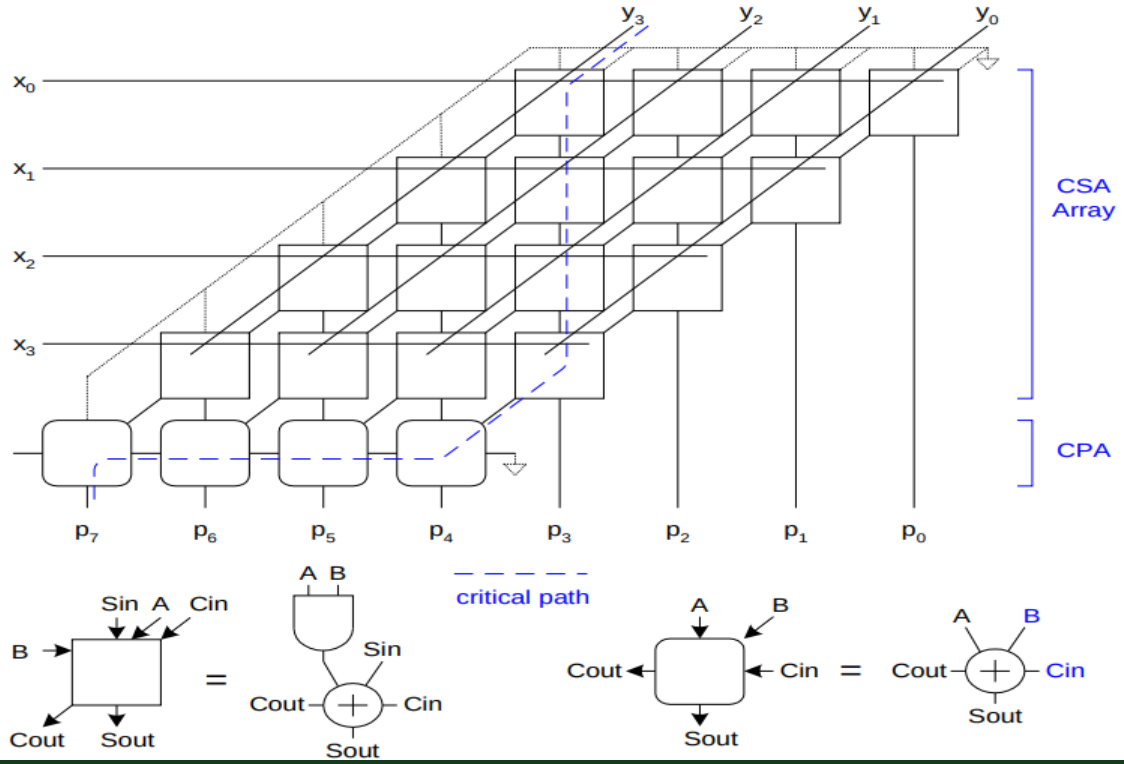
multiplicand
multiplier
partial products
product

DOT DIAGRAM

- Each dot represents a bit

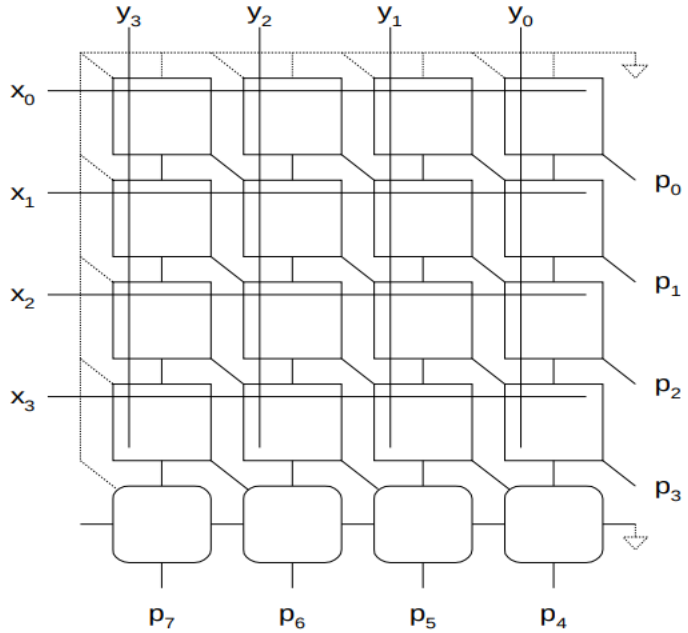


ARRAY MULTIPLIER



RECTANGULAR ARRAY

- Squash array to fit rectangular floorplan



FEWER PARTIAL PRODUCTS

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-2^r encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder

BOOTH ENCODING

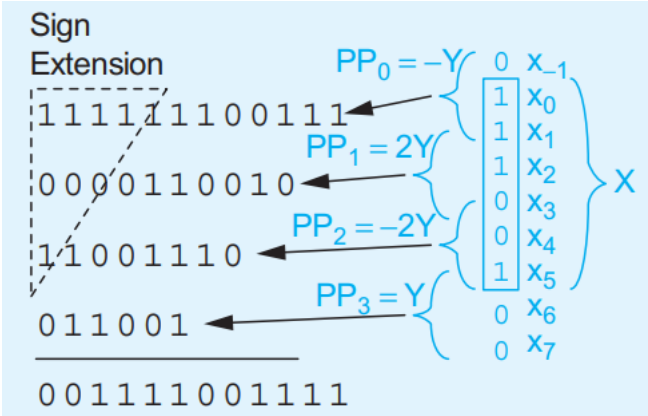
- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

Table 10.12 Radix-4 modified Booth encoding values

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	X_i	$2X_i$	M_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

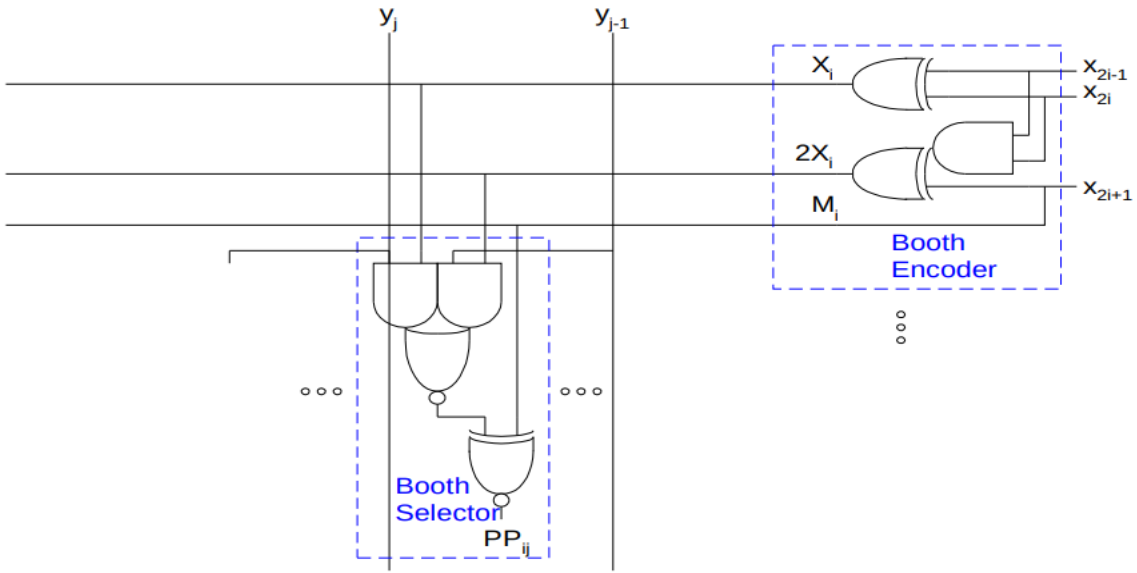
BOOTH ENCODING EXAMPLE

- Do the multiplication of $P = Y \times X = 011001_2 \times 100111_2$, applying Booth encoding to reduce the number of partial products.



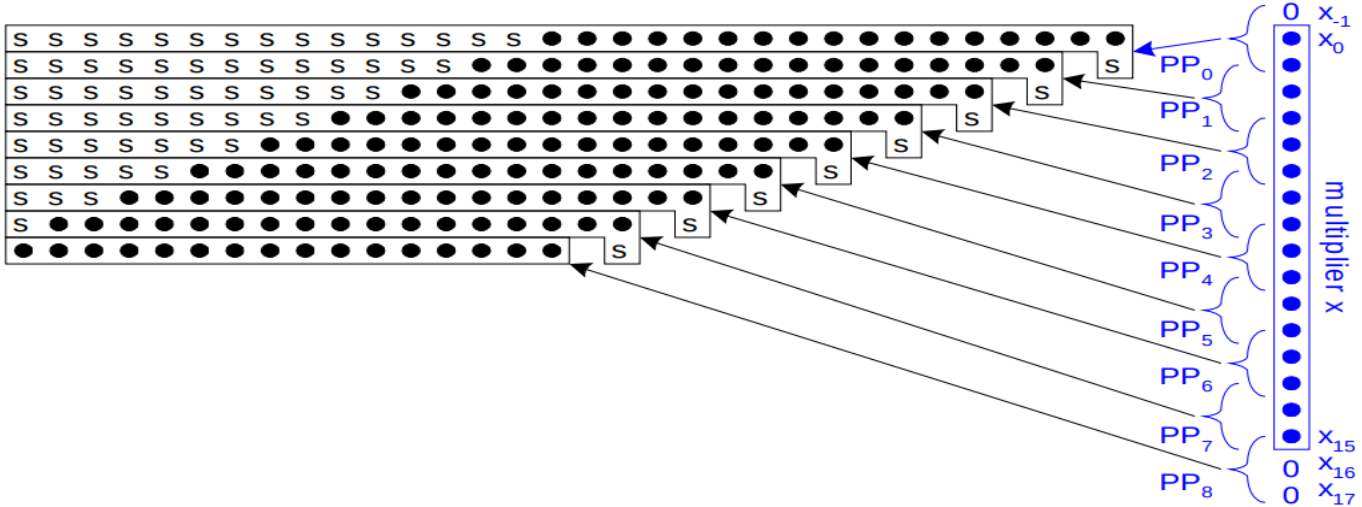
BOOTH HARDWARE

- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits



SIGN EXTENSION

- Partial products can be negative
  - Require sign extension, which is cumbersome
  - High fanout on most significant bit



Thank you!