

Introduction to CMOS VLSI Design

Lecture 12: Datapath Functional Units

Outline

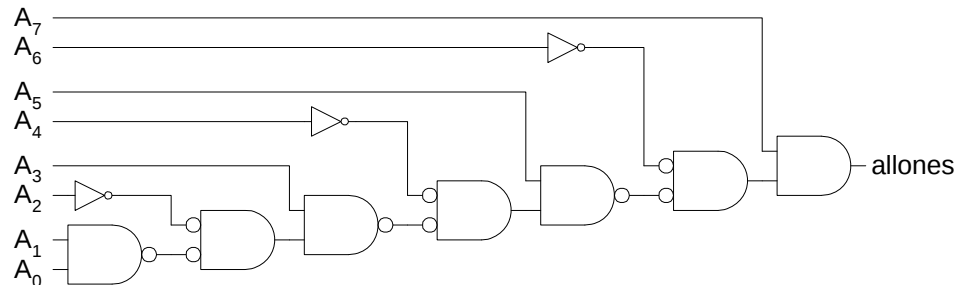
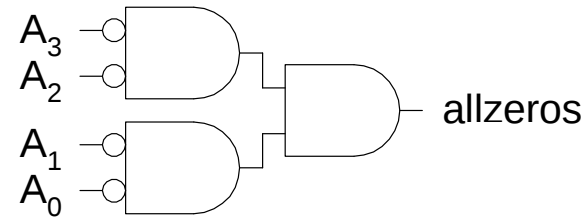
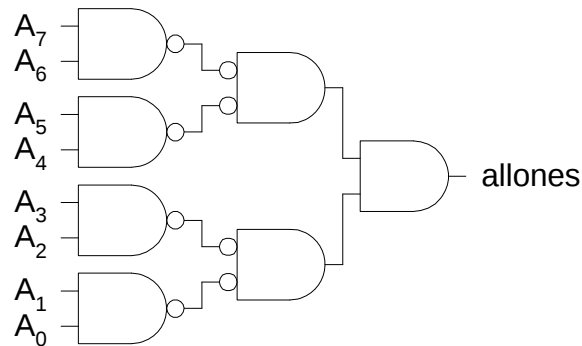
- ☐ Comparators
- ☐ Shifters
- ☐ Multi-input Adders
- ☐ Multipliers

Comparators

- ☐ 0's detector: $A = 00\dots000$
- ☐ 1's detector: $A = 11\dots111$
- ☐ Equality comparator: $A = B$
- ☐ Magnitude comparator: $A < B$

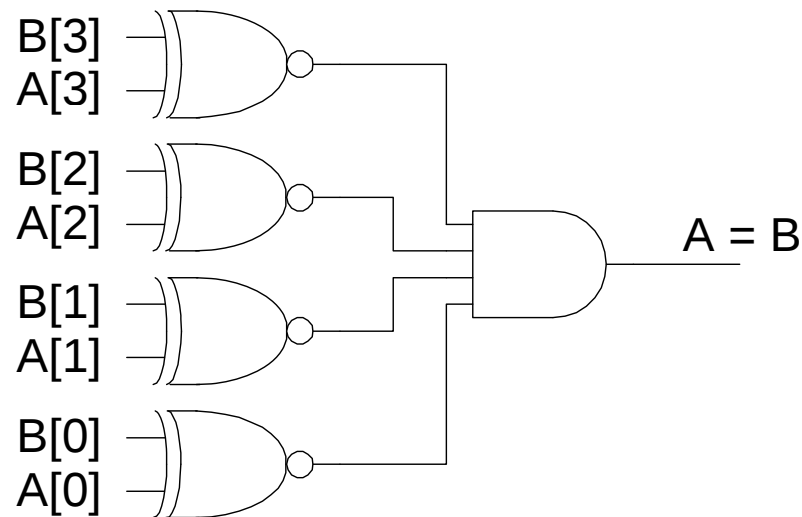
1's & 0's Detectors

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



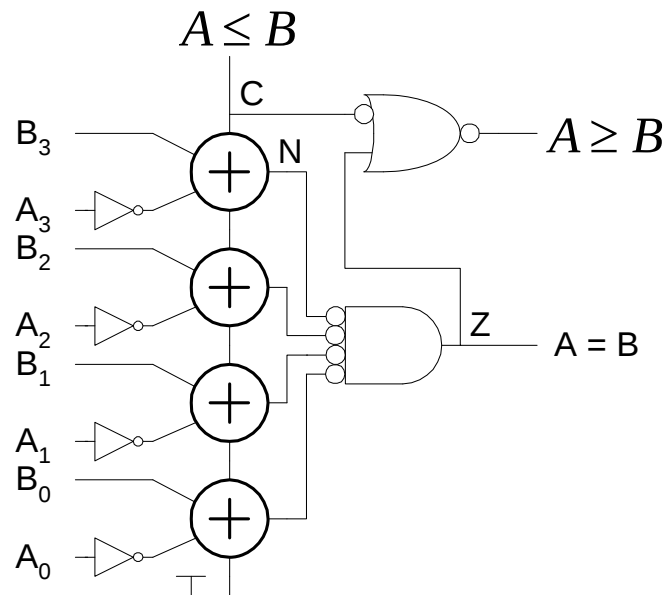
Equality Comparator

- ❑ 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
- ❑ $B-A = B + \sim A + 1$
- ❑ 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 $\neq$ 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. Rt shift sign extends

• 1011 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

- $1011 \text{ LSR } 1 = 0101$ $1011 \text{ LSL } 1 = 0110$

❑ Arithmetic Shift:

- Shifts number left or right. Rt shift sign extends

- $1011 \text{ ASR } 1 = 1101$ $1011 \text{ ASL } 1 = 0110$

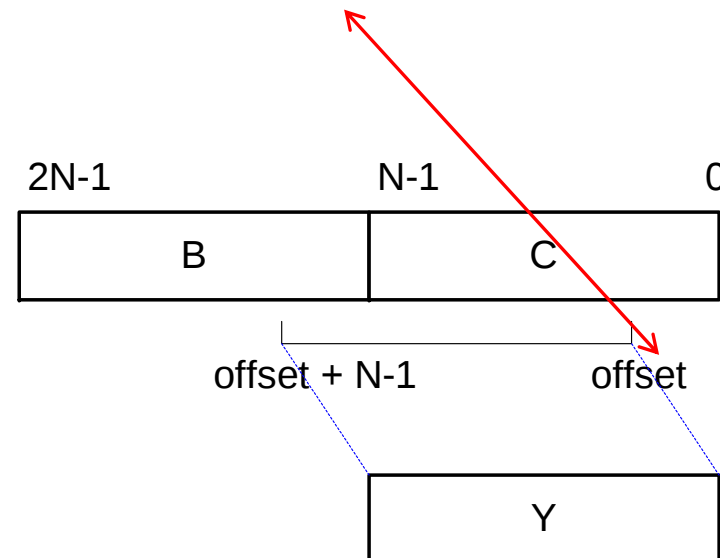
❑ Rotate:

- Shifts number left or right and fills with lost bits

- $1011 \text{ ROR } 1 = 1101$ $1011 \text{ ROL } 1 = 0111$

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	$0\dots 0$	$A_{N-1}\dots A_0$	k
Logical Left	$A_{N-1}\dots A_0$	$0\dots 0$	$N-k$
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	$0\dots 0$	$A_{N-1}\dots A_0$	k
Logical Left	$A_{N-1}\dots A_0$	$0\dots 0$	$N-k$
Arithmetic Right	$A_{N-1}\dots A_{N-1}$ (sign extension)	$A_{N-1}\dots A_0$	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	$0\dots 0$	$A_{N-1}\dots A_0$	k
Logical Left	$A_{N-1}\dots A_0$	$0\dots 0$	$N-k$
Arithmetic Right	$A_{N-1}\dots A_{N-1}$ (sign extension)	$A_{N-1}\dots A_0$	k
Arithmetic Left	$A_{N-1}\dots A_0$	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	$0\dots 0$	$A_{N-1}\dots A_0$	k
Logical Left	$A_{N-1}\dots A_0$	$0\dots 0$	$N-k$
Arithmetic Right	$A_{N-1}\dots A_{N-1}$ (sign extension)	$A_{N-1}\dots A_0$	k
Arithmetic Left	$A_{N-1}\dots A_0$	0	$N-k$
Rotate Right	$A_{N-1}\dots A_0$	$A_{N-1}\dots A_0$	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	$0\dots 0$	$A_{N-1}\dots A_0$	k
Logical Left	$A_{N-1}\dots A_0$	$0\dots 0$	$N-k$
Arithmetic Right	$A_{N-1}\dots A_{N-1}$ (sign extension)	$A_{N-1}\dots A_0$	k
Arithmetic Left	$A_{N-1}\dots A_0$	0	$N-k$
Rotate Right	$A_{N-1}\dots A_0$	$A_{N-1}\dots A_0$	k
Rotate Left	$A_{N-1}\dots A_0$	$A_{N-1}\dots A_0$	$N-k$

❑ Computing $N-k$ requires an adder

Simplified Funnel Shifter

- ❑ 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	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Arithmetic Right		
Arithmetic Left		
Rotate Right		
Rotate Left		

Simplified Funnel Shifter

- ❑ 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	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left		
Rotate Right		
Rotate Left		

Simplified Funnel Shifter

- ❑ 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	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Rotate Right		
Rotate Left		

Simplified Funnel Shifter

- ❑ 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	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Arithmetic Right	$A_{N-1}...A_{N-1}, A_{N-1}...A_0$	k
Arithmetic Left	$A_{N-1}...A_0, 0..0$	$\bar{k}$
Rotate Right	$A_{N-2}...A_0, A_{N-1}...A_0$	k
Rotate Left		

Simplified Funnel Shifter

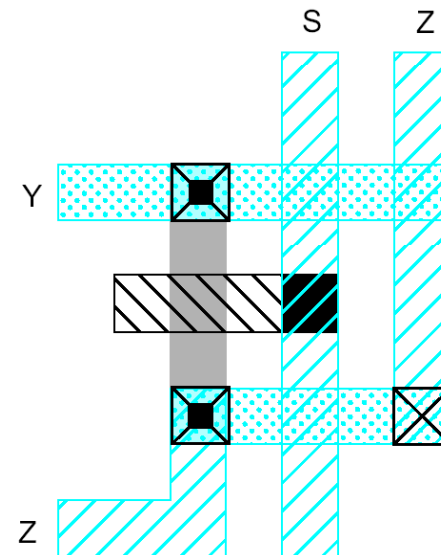
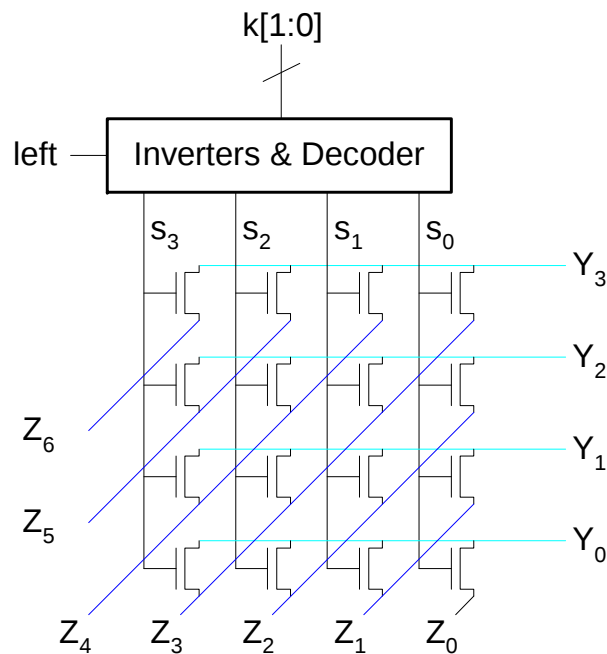
- ❑ Optimize down to $2N-1$ bit input

Table 10.11 Simplified funnel shifter

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

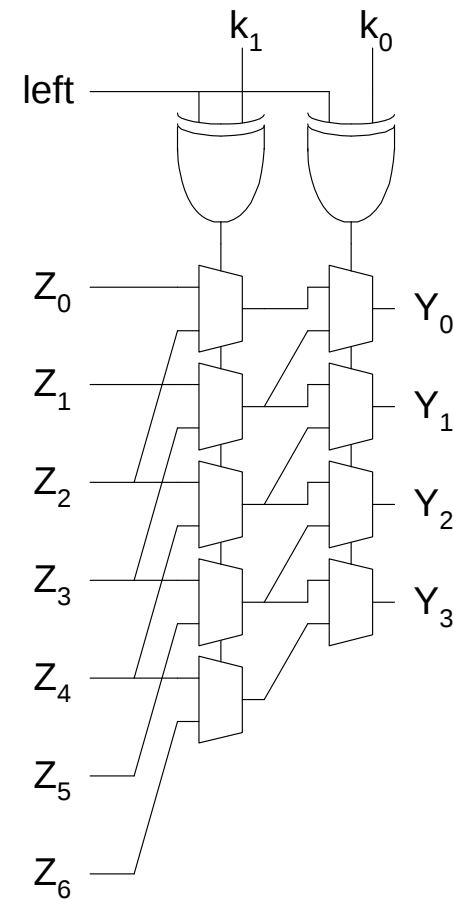
Funnel Shifter Design 1

- ❑ 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

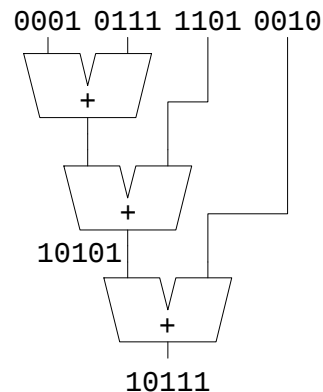
- Suppose we want to add k N-bit words
 - Ex: $0001 + 0111 + 1101 + 0010 = \underline{\hspace{2cm}}$

Multi-input Adders

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

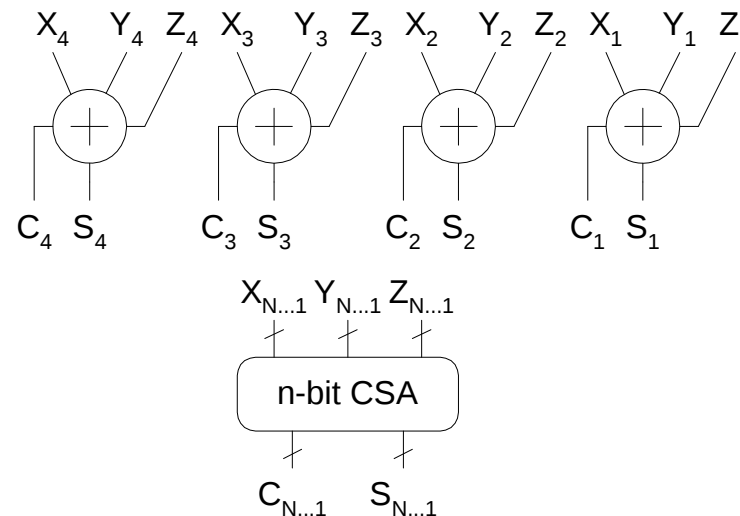
Multi-input Adders

- ❑ 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



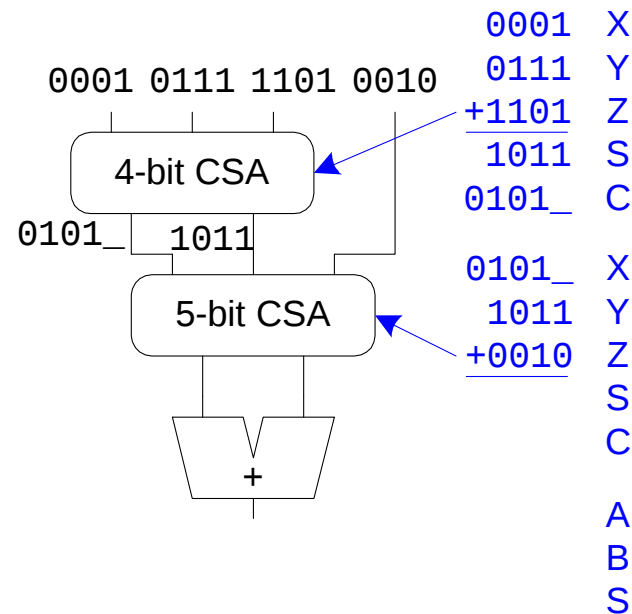
Carry Save Addition

- ❑ 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



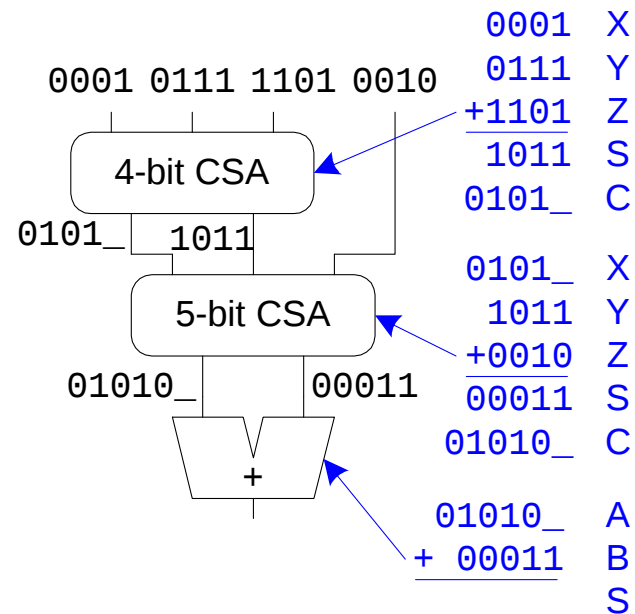
CSA Application

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



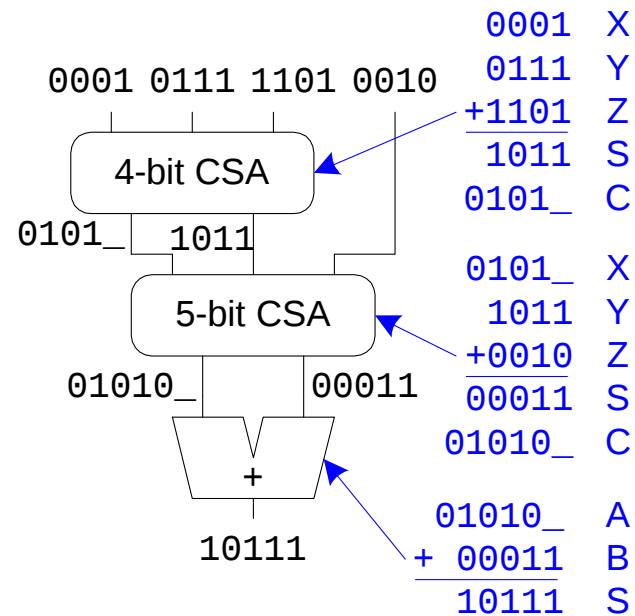
CSA Application

- ❑ 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:

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

Multiplication

□ Example:

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

Multiplication

□ Example:

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

Multiplication

□ Example:

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

Multiplication

□ Example:

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

Multiplication

□ Example:

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

multiplicand

multiplier

partial
products

product

□ M x N-bit multiplication

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

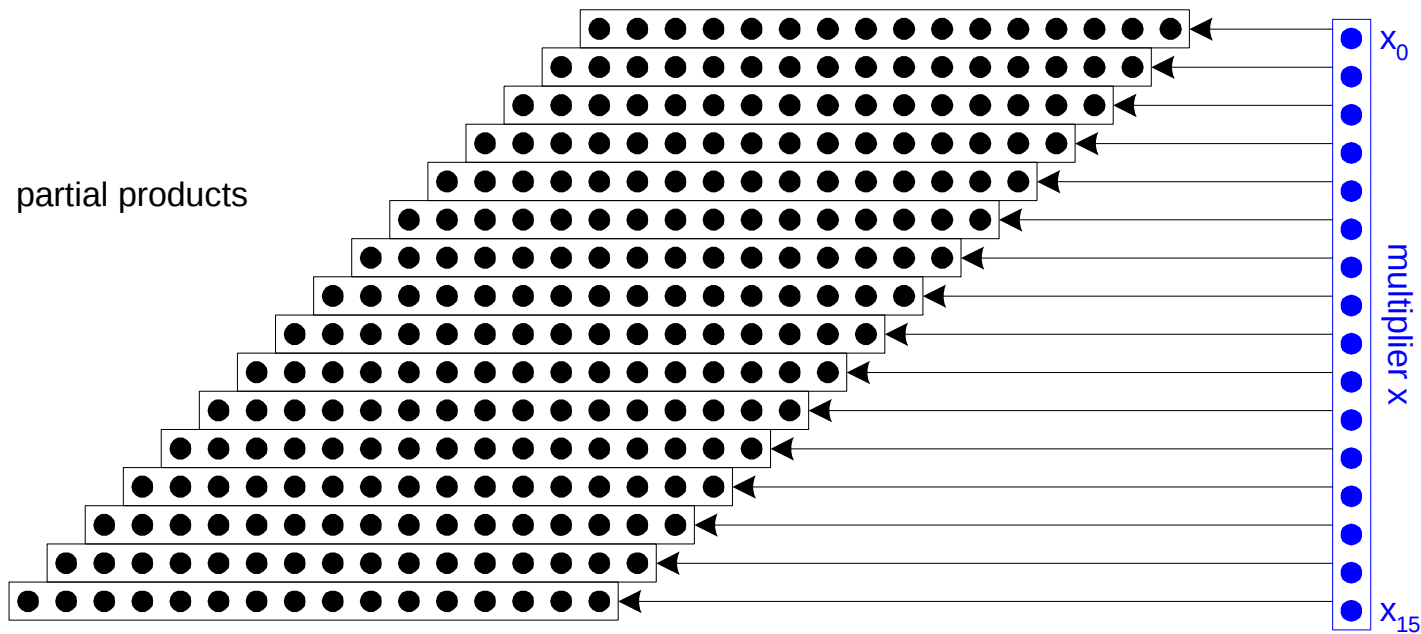
General Form

- ❑ Multiplicand: $Y = (y_{M-1}, y_{M-2}, \dots, y_1, y_0)$
- ❑ Multiplier: $X = (x_{N-1}, x_{N-2}, \dots, x_1, x_0)$
- ❑ Product:
$$P = \left(\sum_{j=0}^{M-1} y_j 2^j \right) \left(\sum_{i=0}^{N-1} x_i 2^i \right) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

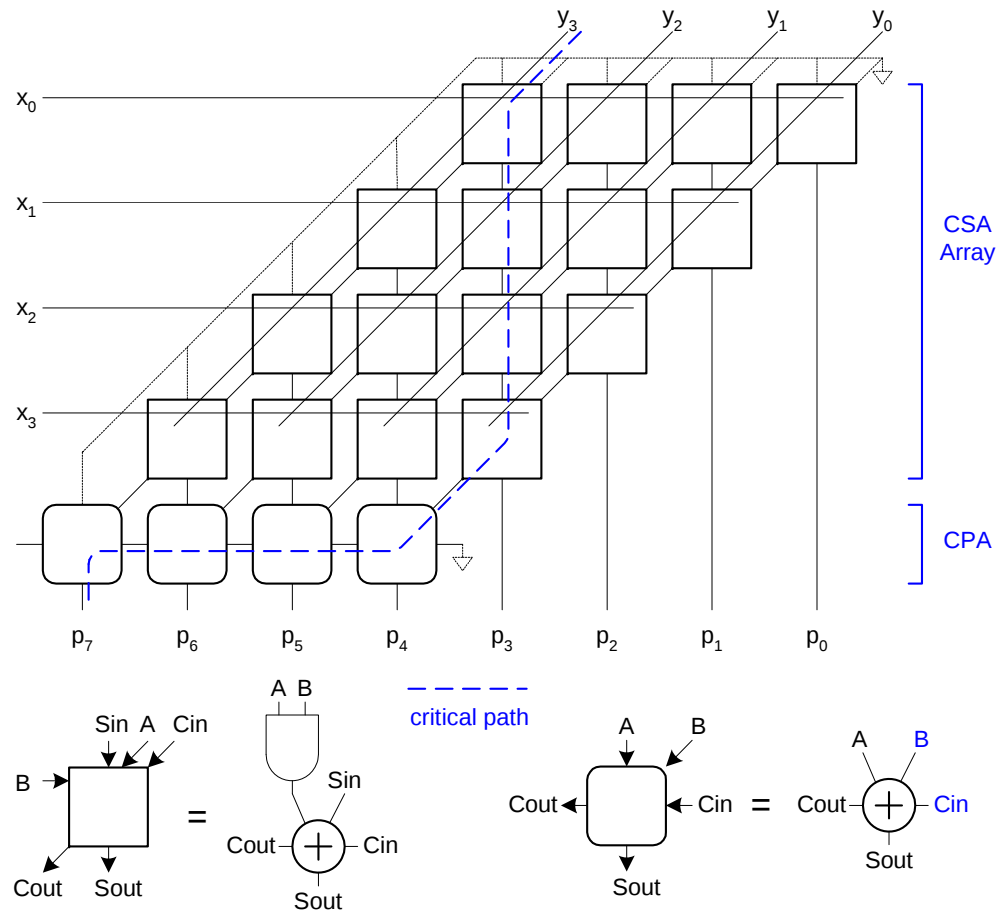
							y_5	y_4	y_3	y_2	y_1	y_0	multiplicand multiplier
							x_5	x_4	x_3	x_2	x_1	x_0	
							x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	partial products
						x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
					x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0					
		x_4y_5	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							product
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0		

Dot Diagram

- Each dot represents a bit

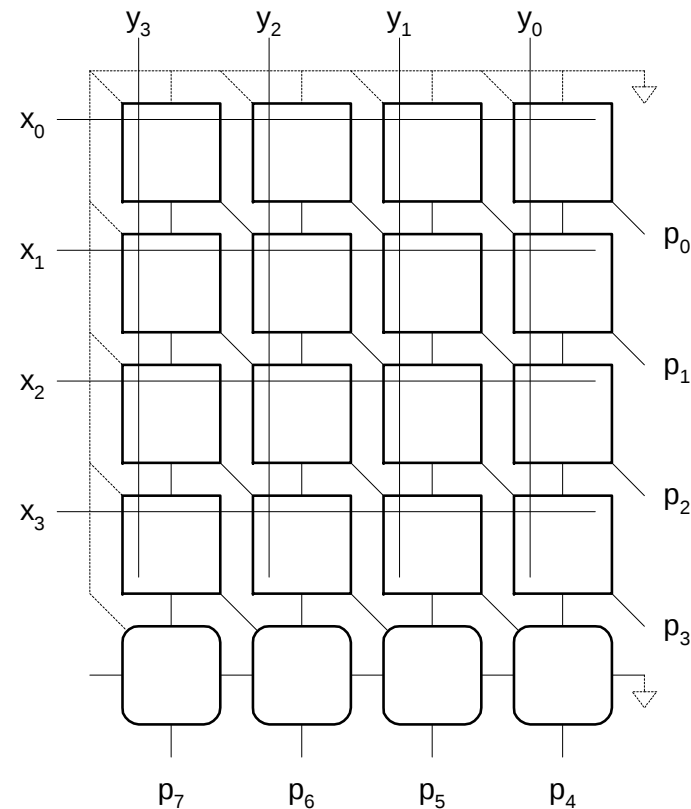


Array Multiplier



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 (radix 4)
 - 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	1	Y			
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1	$2Y$			
1	0	0				
1	0	1				
1	1	0				
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1	$2Y$			
1	0	0	$-2Y$			
1	0	1				
1	1	0				
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1	$2Y$			
1	0	0	$-2Y$			
1	0	1	$-Y$			
1	1	0				
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1	$2Y$			
1	0	0	$-2Y$			
1	0	1	$-Y$			
1	1	0	$-Y$			
1	1	1				

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	1	Y			
0	1	0	Y			
0	1	1	$2Y$			
1	0	0	$-2Y$			
1	0	1	$-Y$			
1	1	0	$-Y$			
1	1	1	$-0 (= 0)$			

Booth Encoding

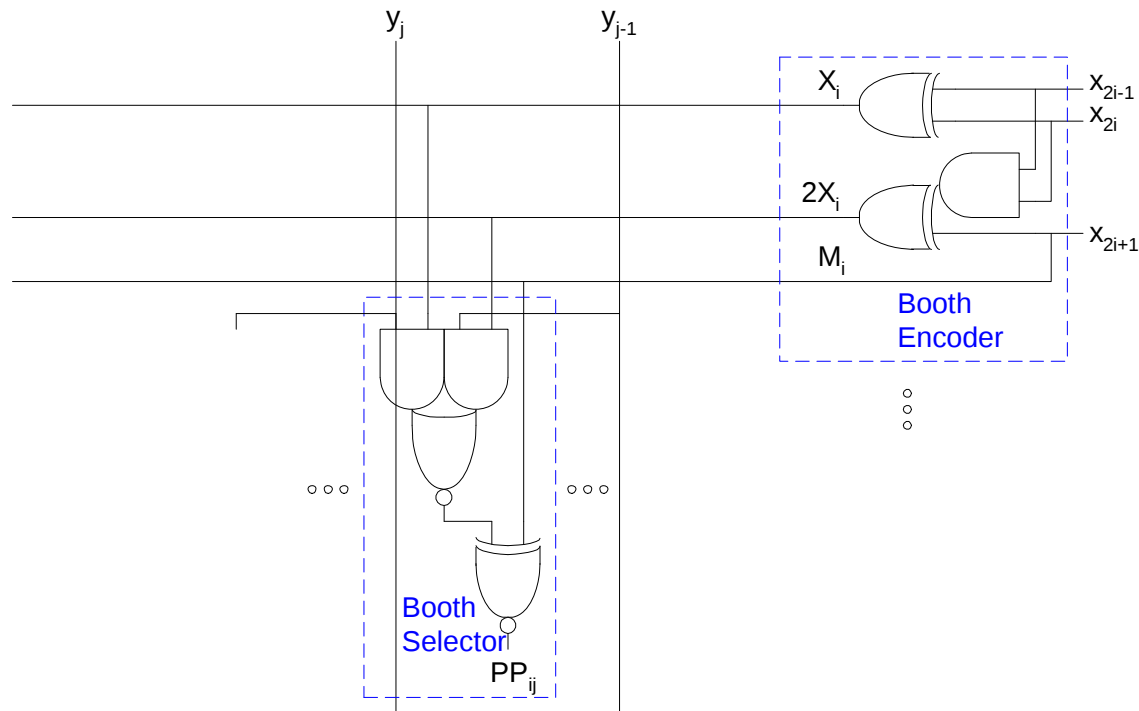
- ❑ Instead of $3Y$, try $-Y$, then increment next partial product to add $4Y$
- ❑ Similarly, for $2Y$, try $-2Y + 4Y$ in next partial product

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

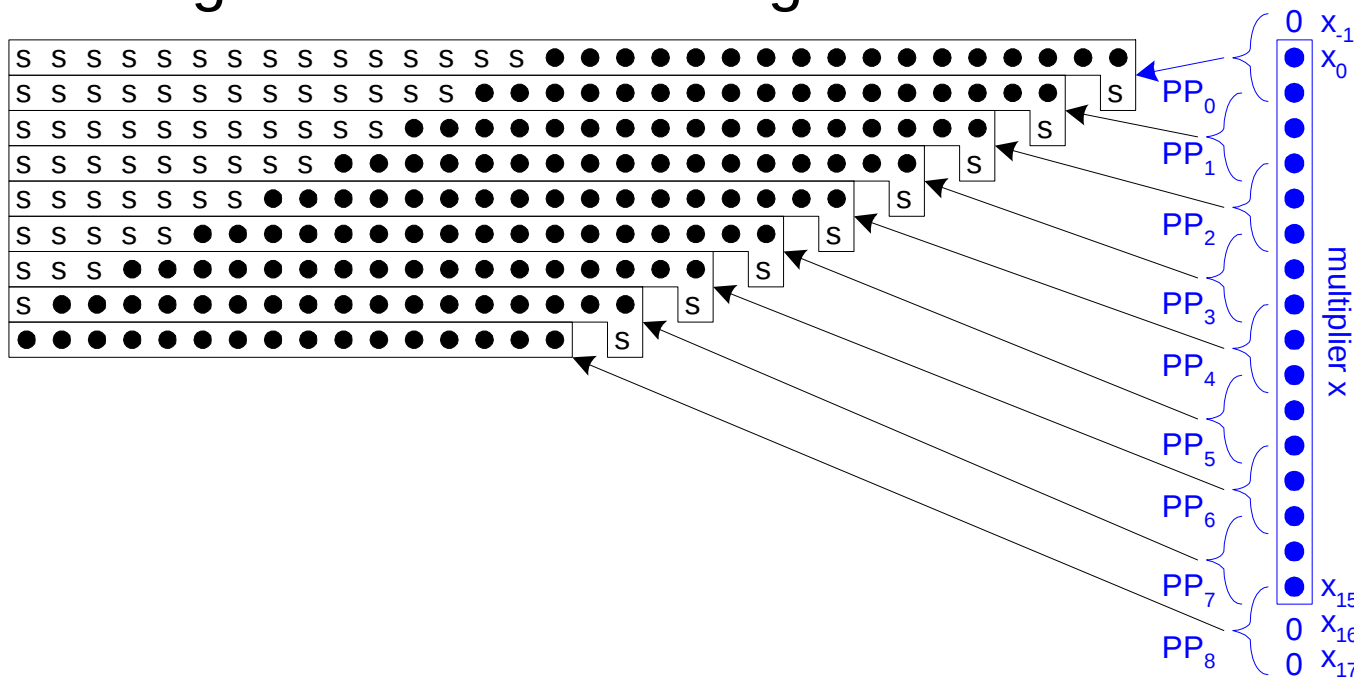
Booth Hardware

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



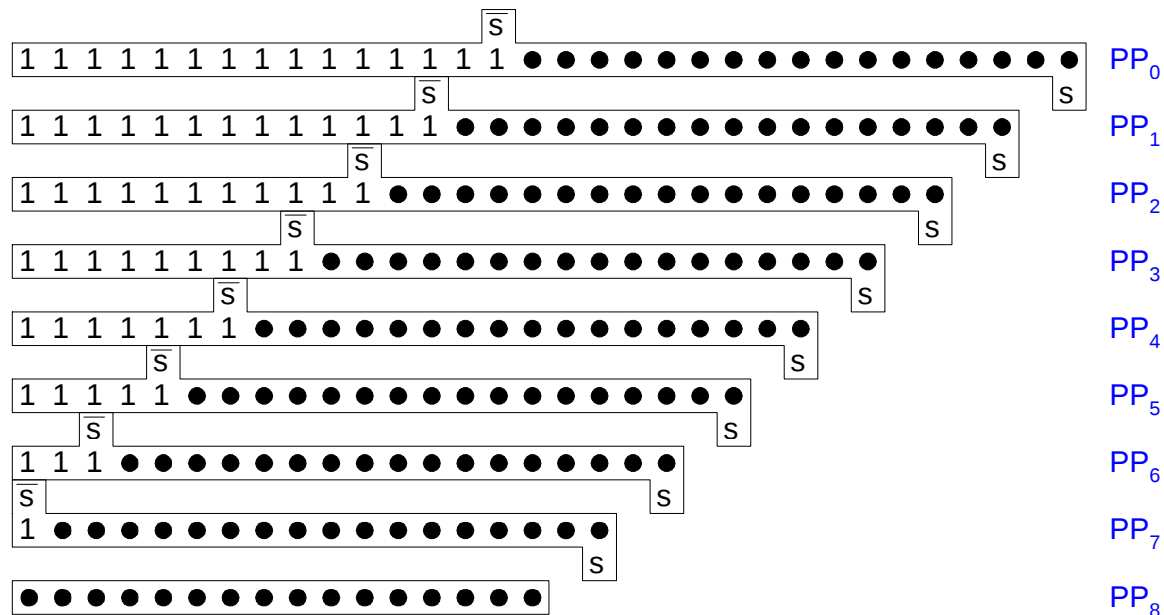
Sign Extension

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



Simplified Sign Ext.

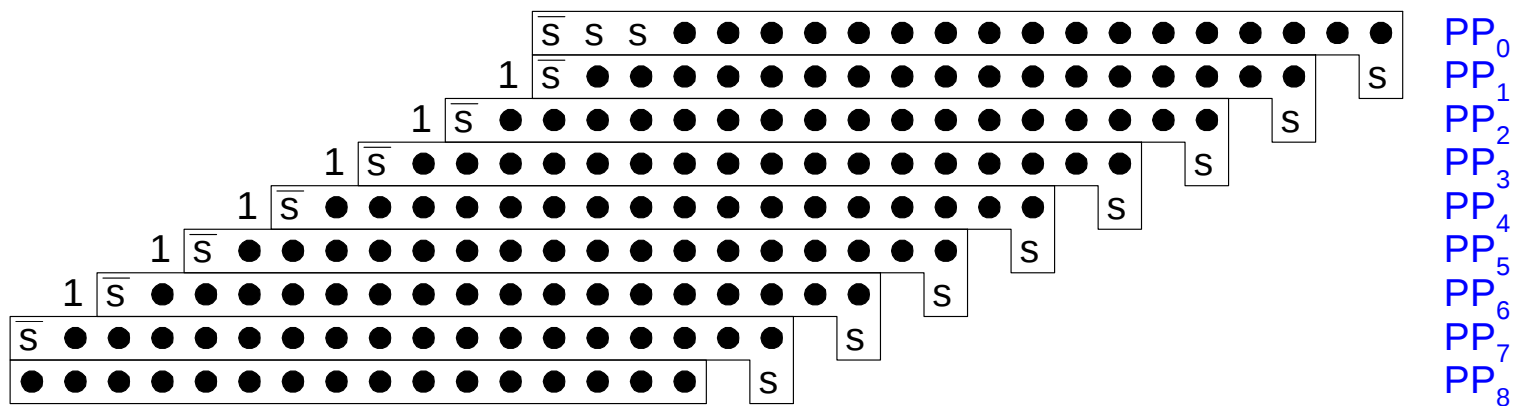
- ❑ Sign bits are either all 0's or all 1's
 - Note that all 0's is all 1's + 1 in proper column
 - Use this to reduce loading on MSB



Skip

Even Simpler Sign Ext.

- ❑ No need to add all the 1's in hardware
 - Precompute the answer!



Skip

Advanced Multiplication

- ☐ Signed vs. unsigned inputs
- ☐ Higher radix Booth encoding
- ☐ Array vs. tree CSA networks