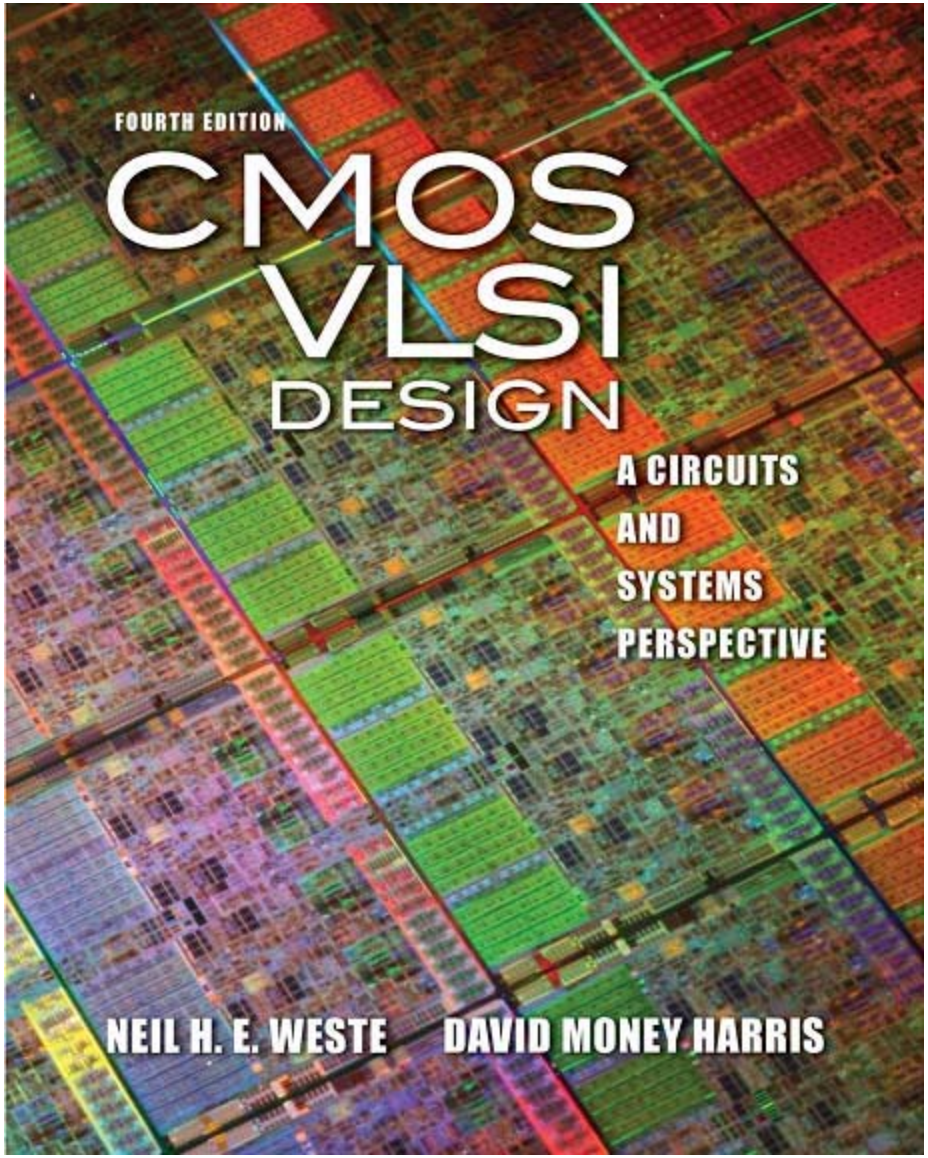



# Lecture 18: Datapath Functional Units

# Outline

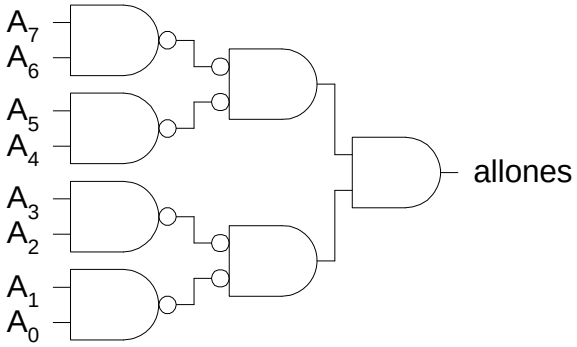
- Comparators
- Shifters
- Multi-input Adders
- Multipliers

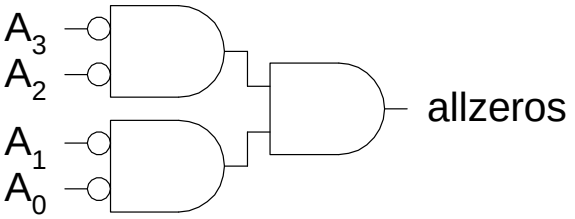
# Comparators

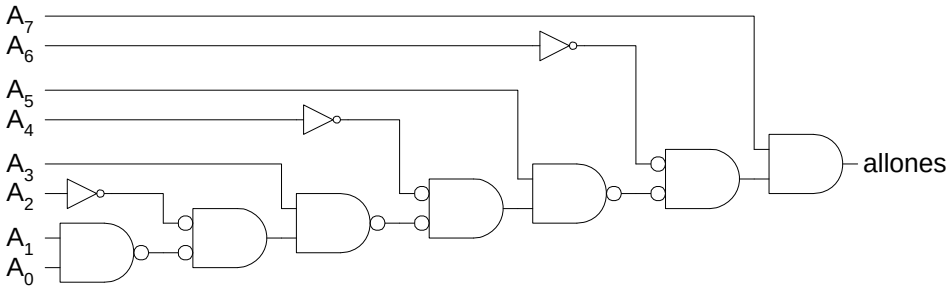
- 0's detector: A = 00…000
- 1's detector: A = 11…111
- Equality comparator: A = B
- Magnitude comparator: A < B

# 1's & 0's Detectors

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

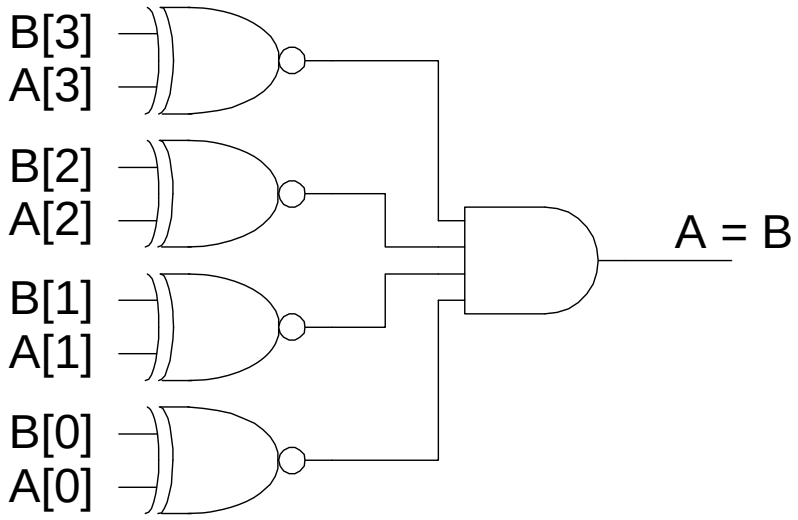


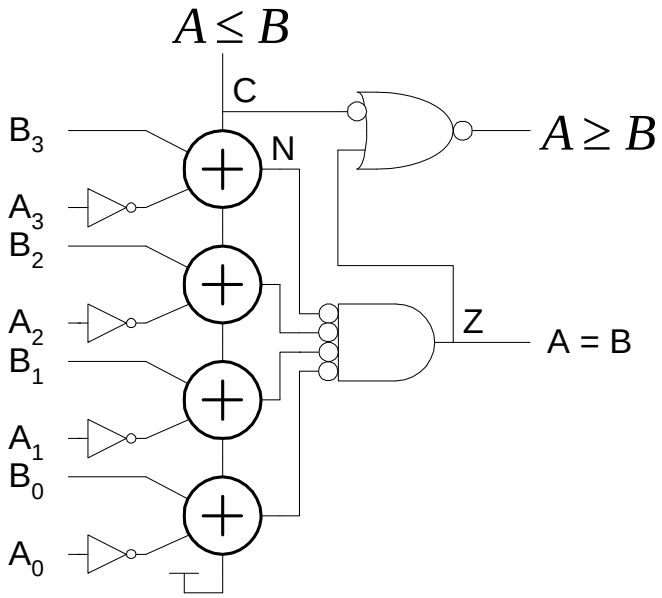
# Equality Comparator

- 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)
  - S: N xor V (sign of result)

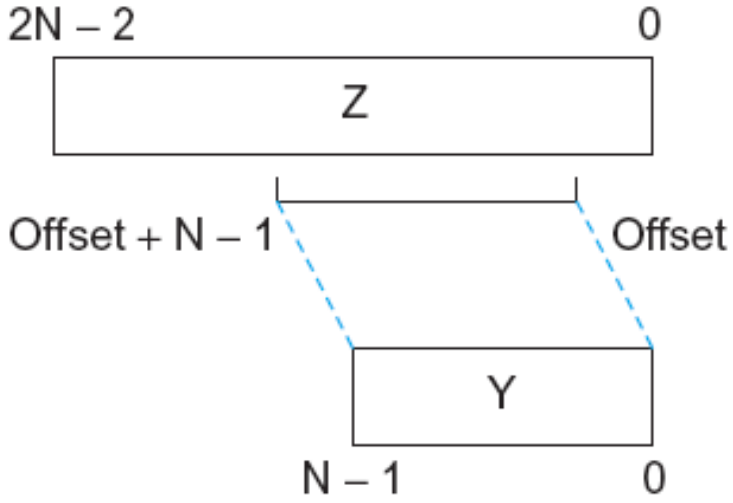
| Relation | Unsigned Comparison | Signed Comparison |
|---|---|---|
| $A = B$ | $Z$ | $Z$ |
| $A \neq B$ | $\overline{Z}$ | $\overline{Z}$ |
| $A < B$ | $C \cdot \overline{Z}$ | $\overline{S} \cdot \overline{Z}$ |
| $A > B$ | $\overline{C}$ | $S$ |
| $A \leq B$ | $C$ | $\overline{S}$ |
| $A \geq B$ | $\overline{C} + Z$ | $S + Z$ |

# 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 =

# Funnel Shifter

- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from 2N–1-bit input
  - Shift by k bits ($0 \leq k < N$)
  - Logically involves N N:1 multiplexers

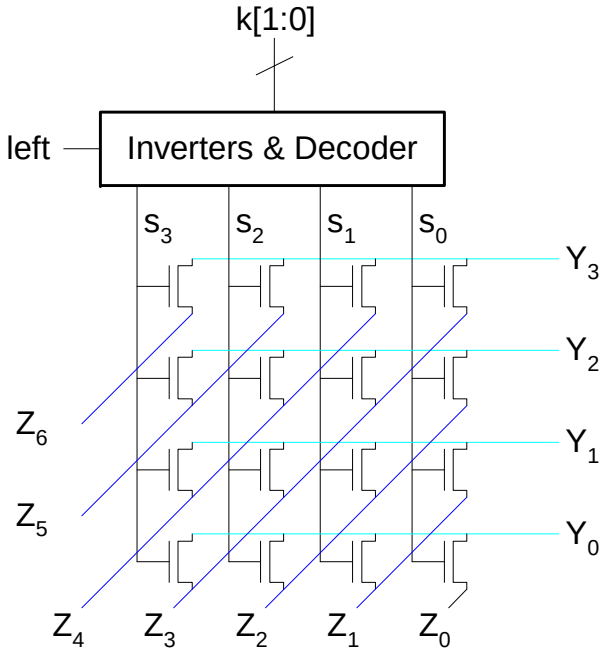


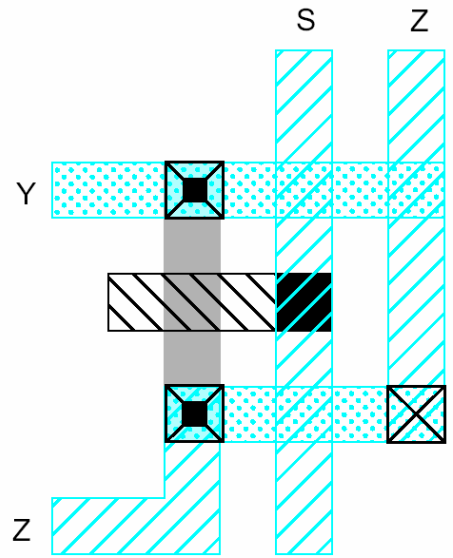
# Funnel Source Generator

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

# Array Funnel Shifter

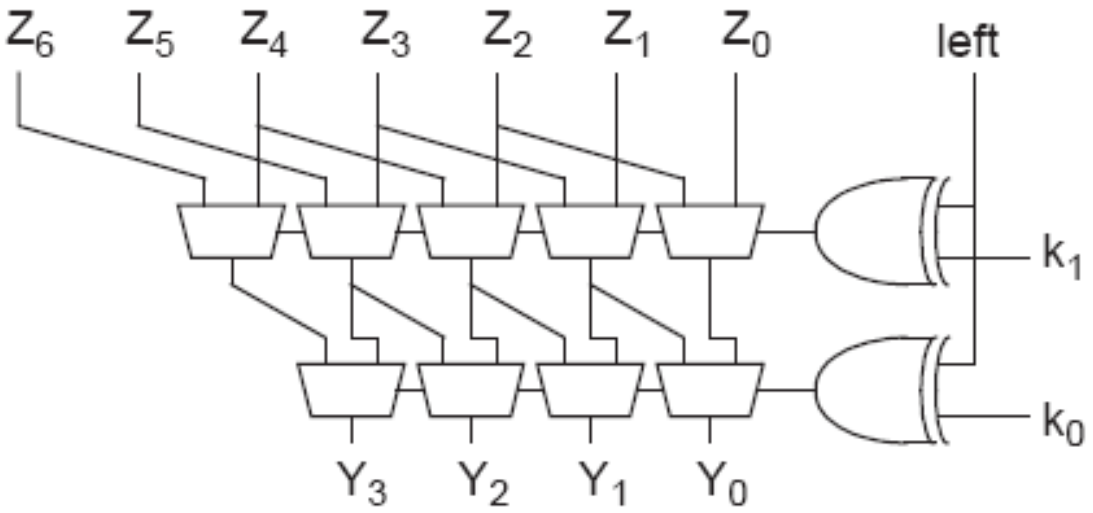
- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design ($V_t$ drops!)



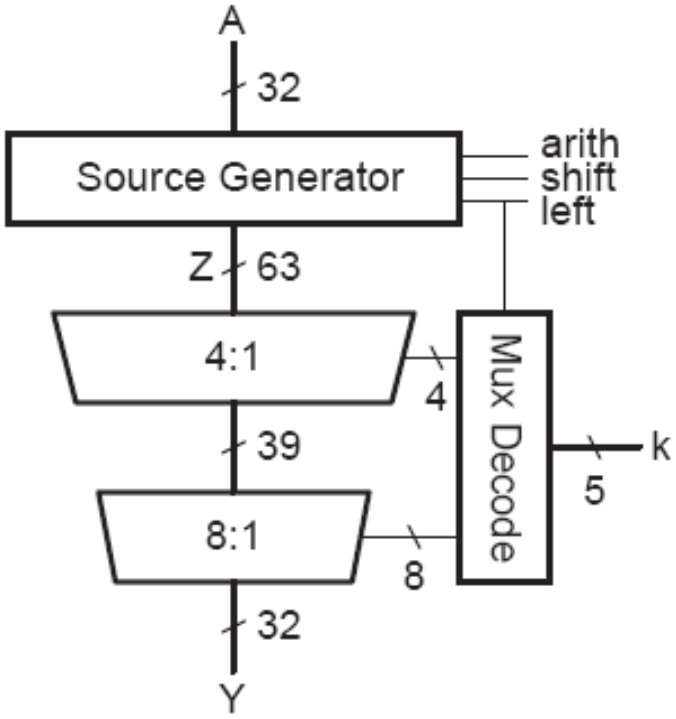
# Logarithmic Funnel Shifter

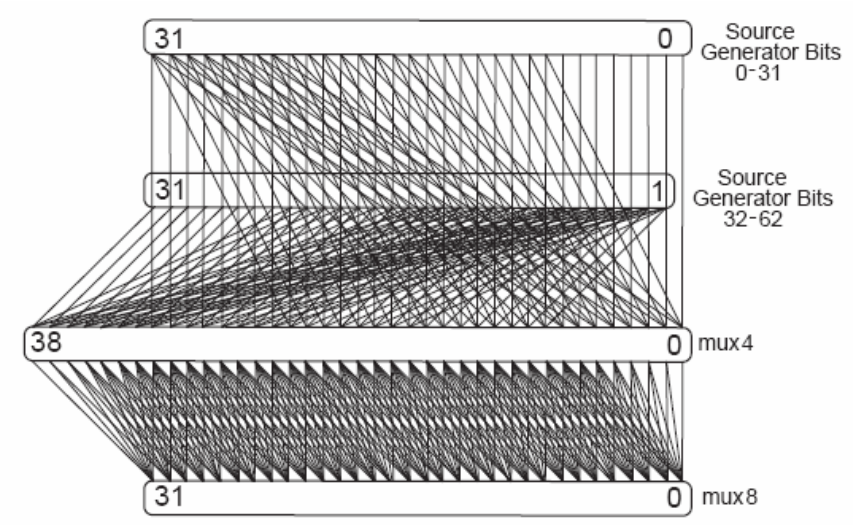
- Log N stages of 2-input muxes
  - No select decoding needed



# 32-bit Logarithmic Funnel

- Wider multiplexers reduce delay and power
- Operands > 32 bits introduce datapath irregularity

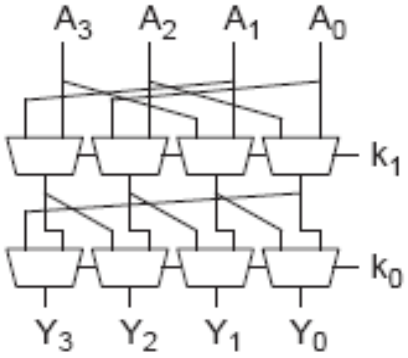


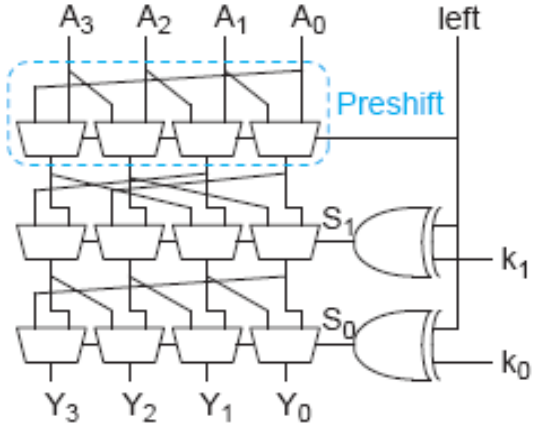
# Barrel Shifter

- Barrel shifters perform right rotations using wrap-around wires.
- Left rotations are right rotations by $N - k = \overline{k} + 1$ bits.
- Shifts are rotations with the end bits masked off.

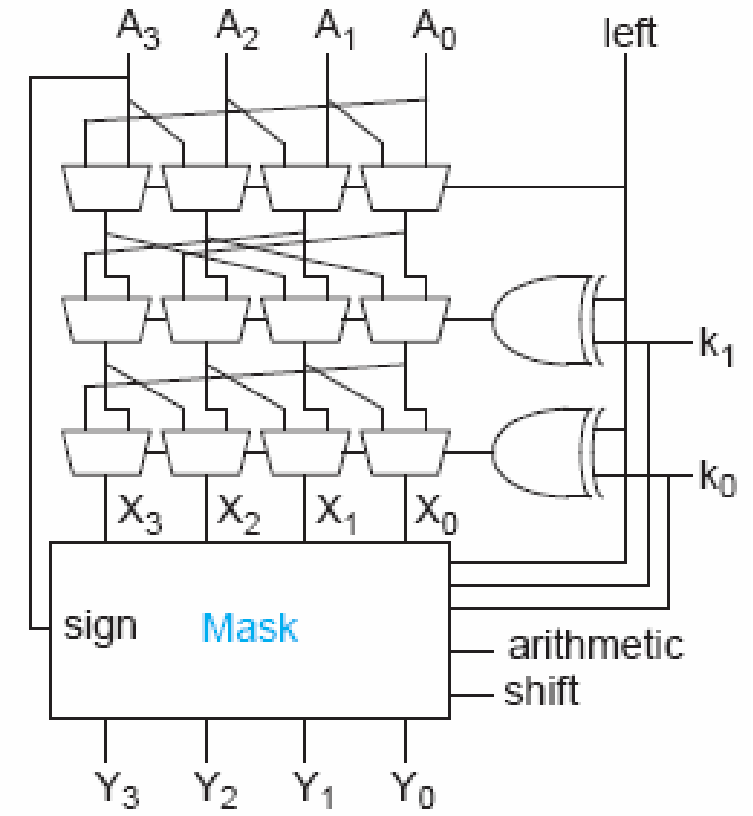
# Logarithmic Barrel Shifter




Right shift only



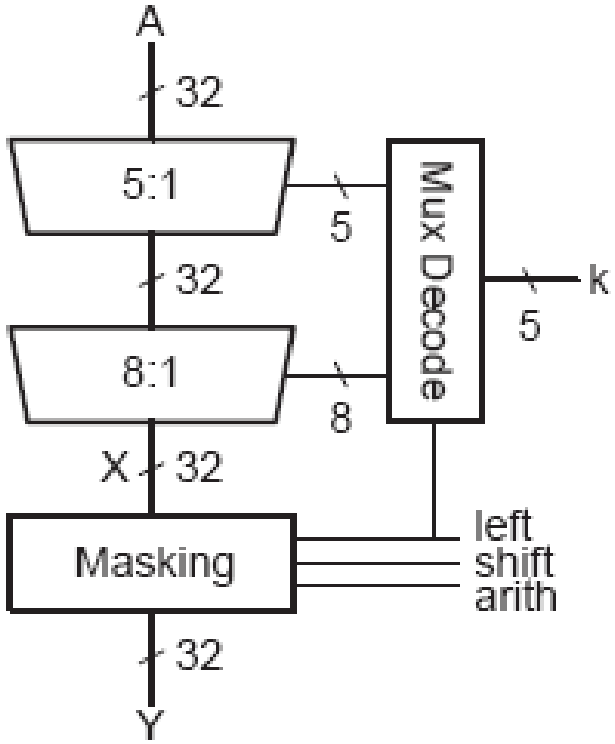

Right/Left shift




Right/Left Shift & Rotate

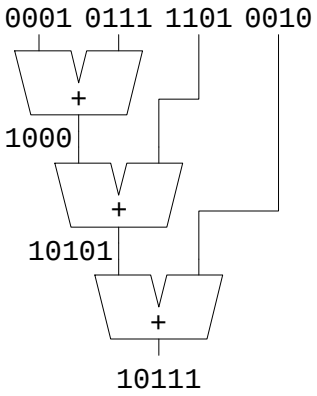
# 32-bit Logarithmic Barrel

- Datapath never wider than 32 bits
- First stage preshifts by 1 to handle left shifts



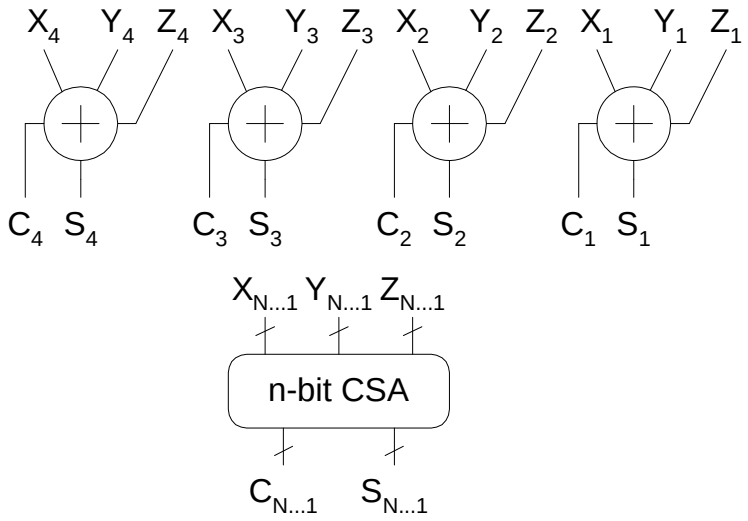
# Multi-input Adders

- 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



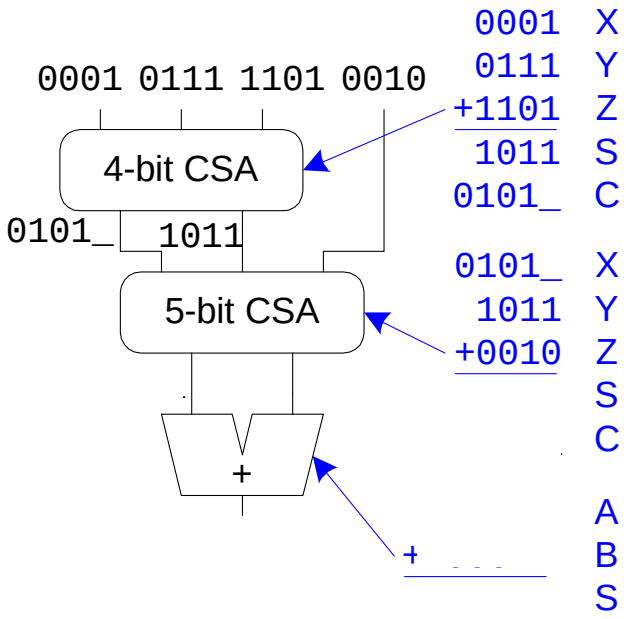
# Carry Save Addition

- 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



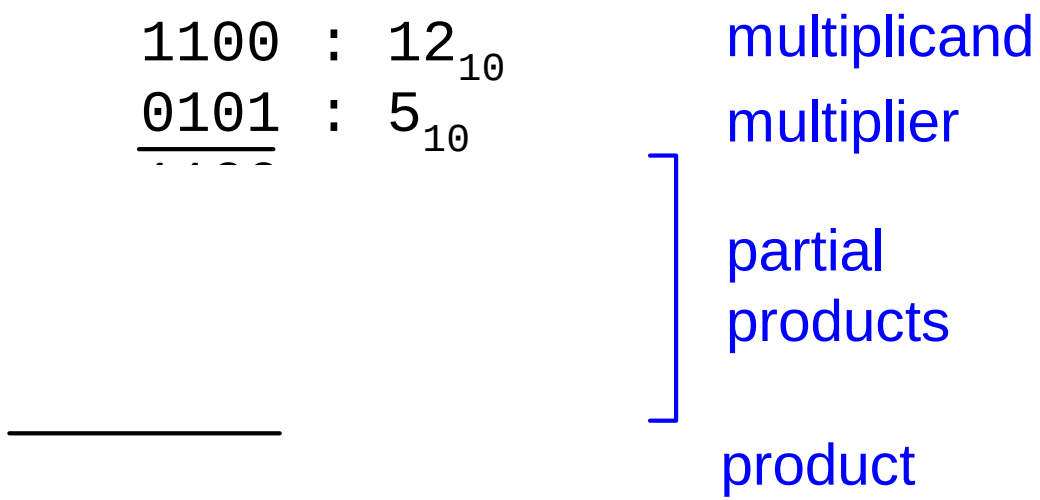
# CSA Application

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



# Multiplication

- Example:

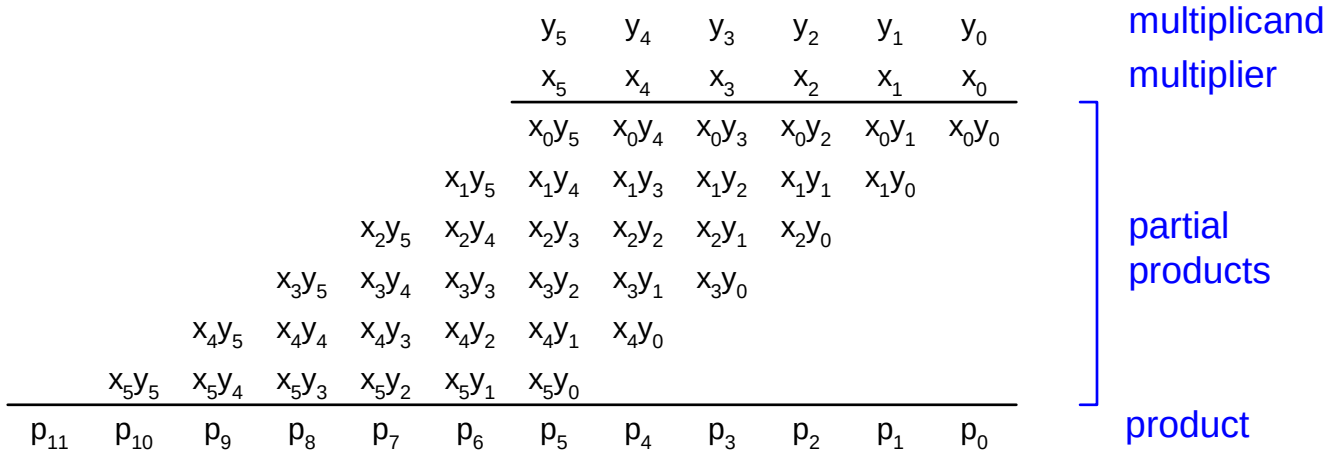



- M x N-bit multiplication
  - Produce N M-bit partial products
  - Sum these to produce M+N-bit product

# General Form

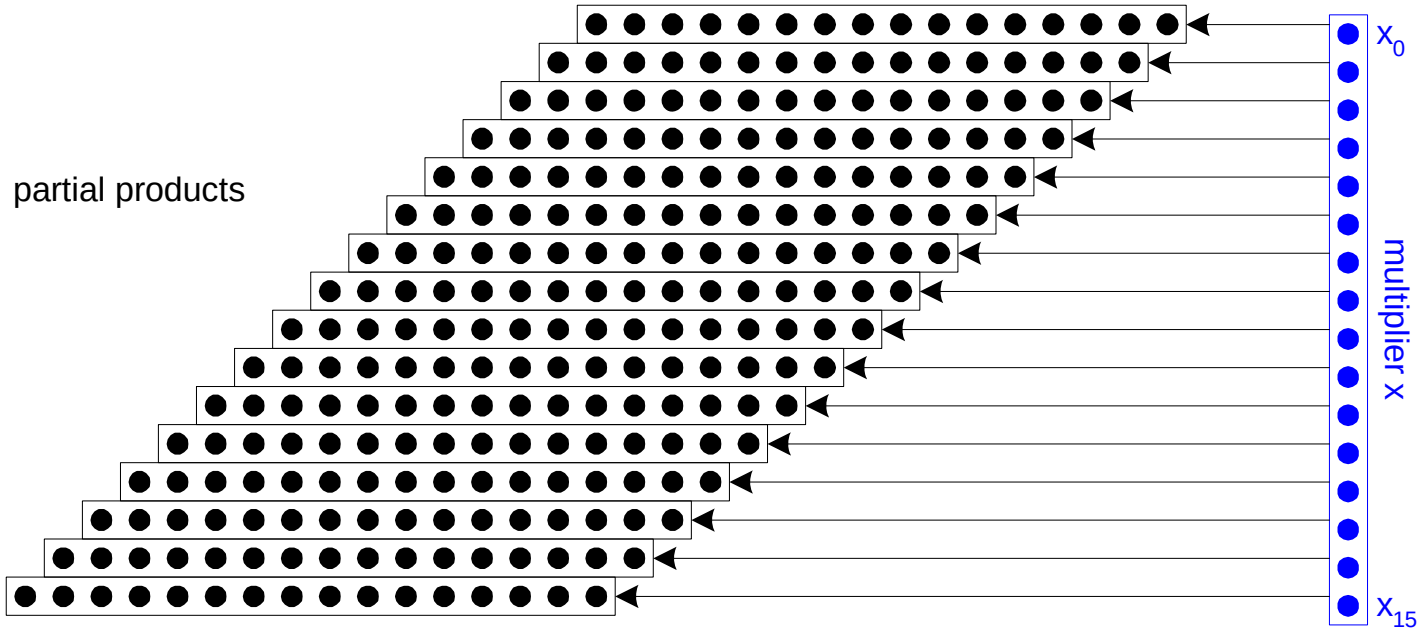
- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product:

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

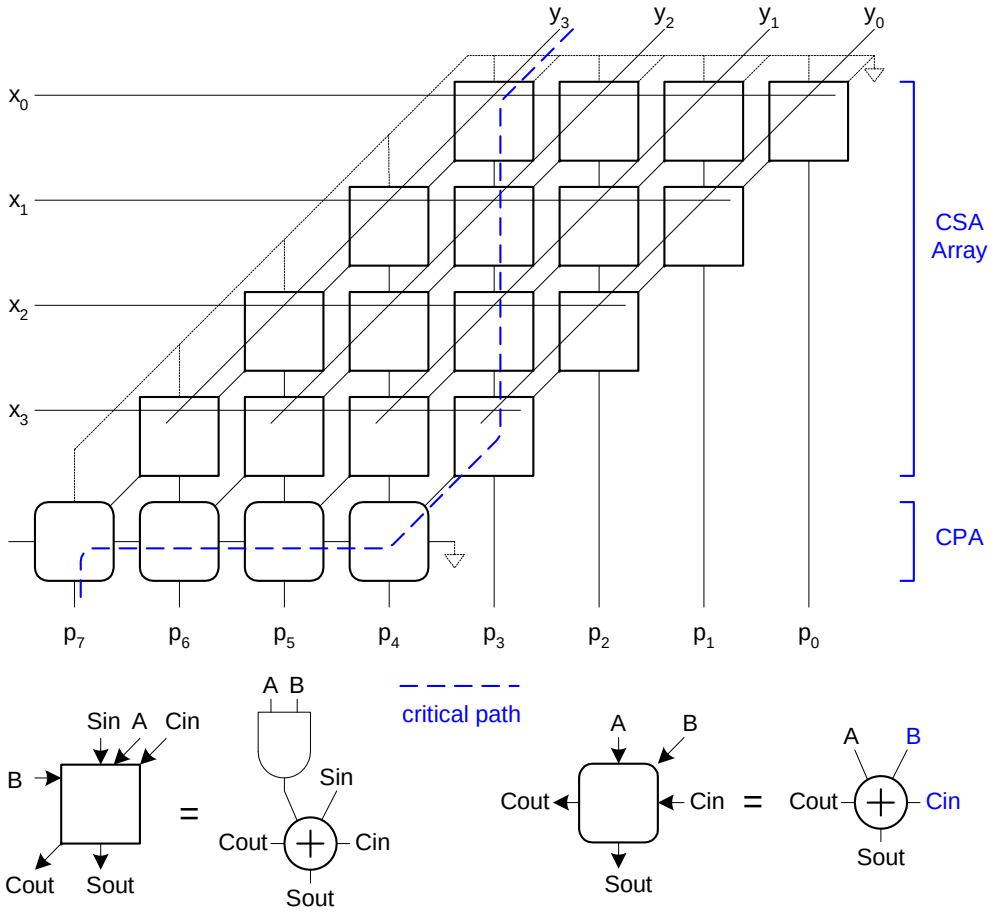


# Dot Diagram

- Each dot represents a bit

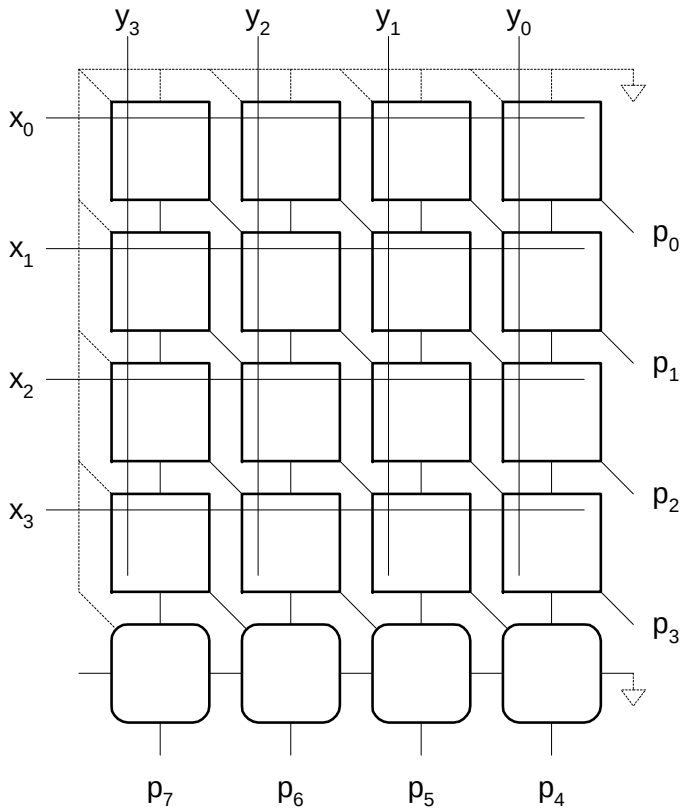


# Array Multiplier



# 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 ☹

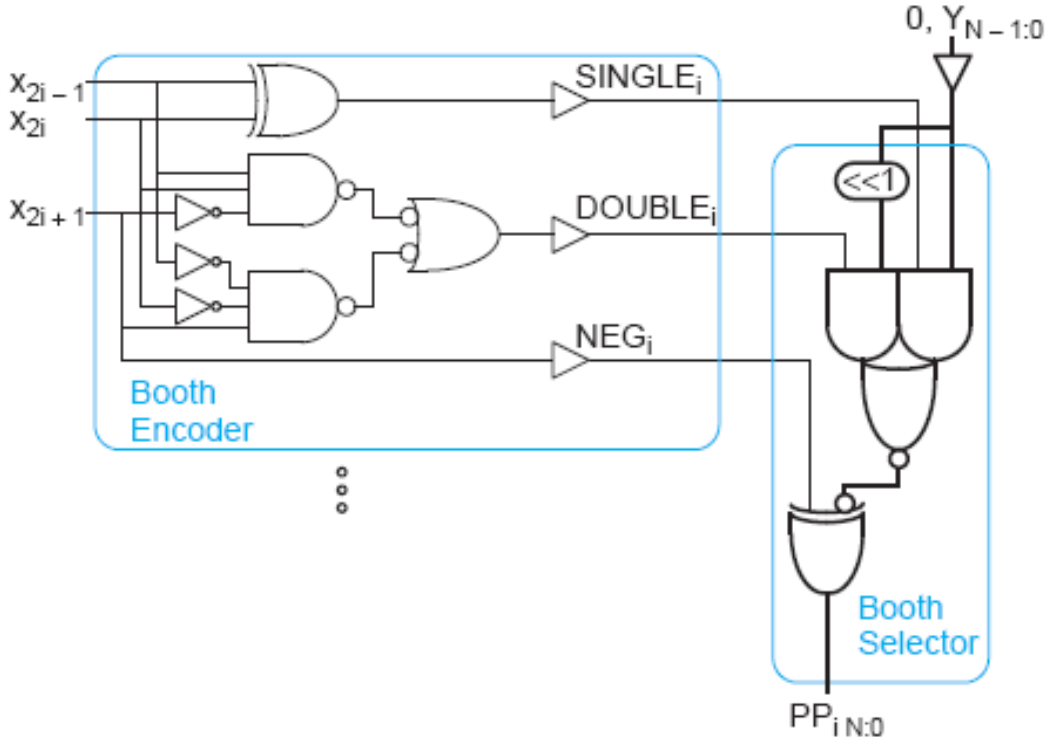
# Booth Encoding

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

| Inputs | | | Partial Product | Booth Selects | | |
|---|---|---|---|---|---|---|
| $x_{2i+1}$ | $x_{2i}$ | $x_{2i-1}$ | $PP_i$ | $\text{SINGLE}_i$ | $\text{DOUBLE}_i$ | $\text{NEG}_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | $Y$ | 1 | 0 | 0 |
| 0 | 1 | 0 | | | | |
| 0 | 1 | 1 | | | | |
| 1 | 0 | 0 | | | | |
| 1 | 0 | 1 | | | | |
| 1 | 1 | 0 | | | | |
| 1 | 1 | 1 | | | | |

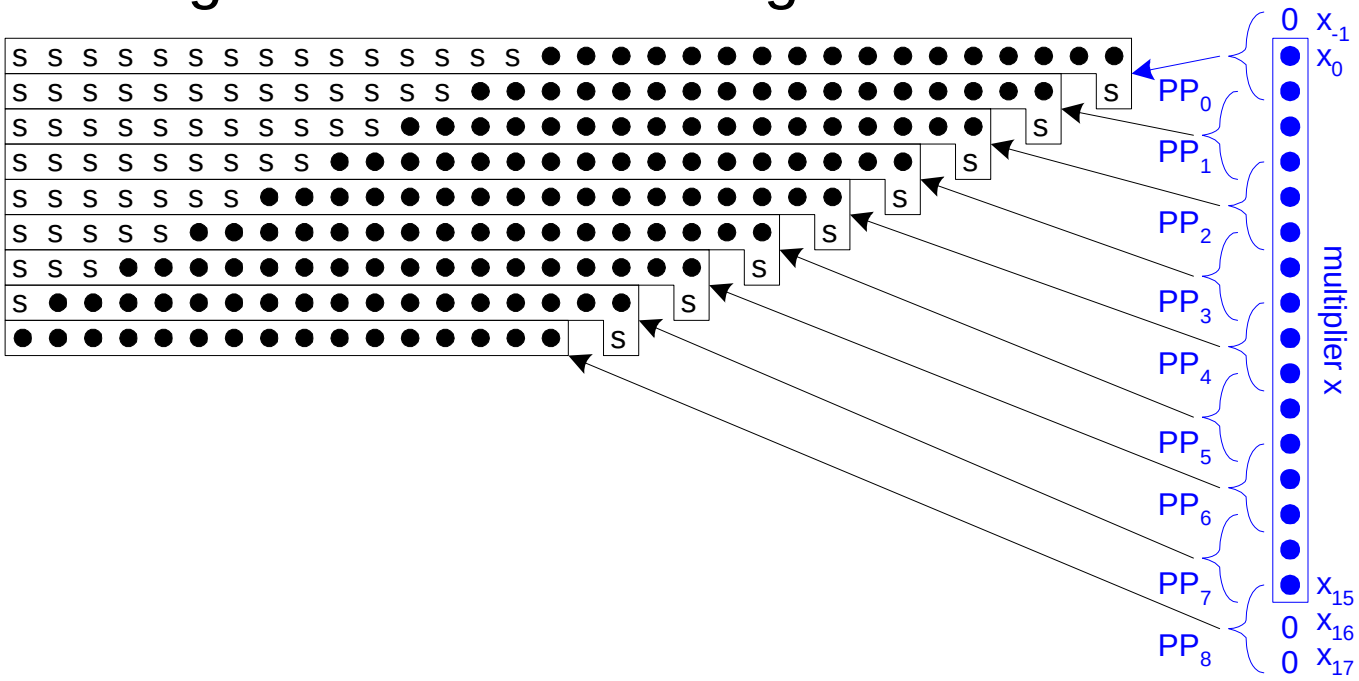
# Booth Hardware

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



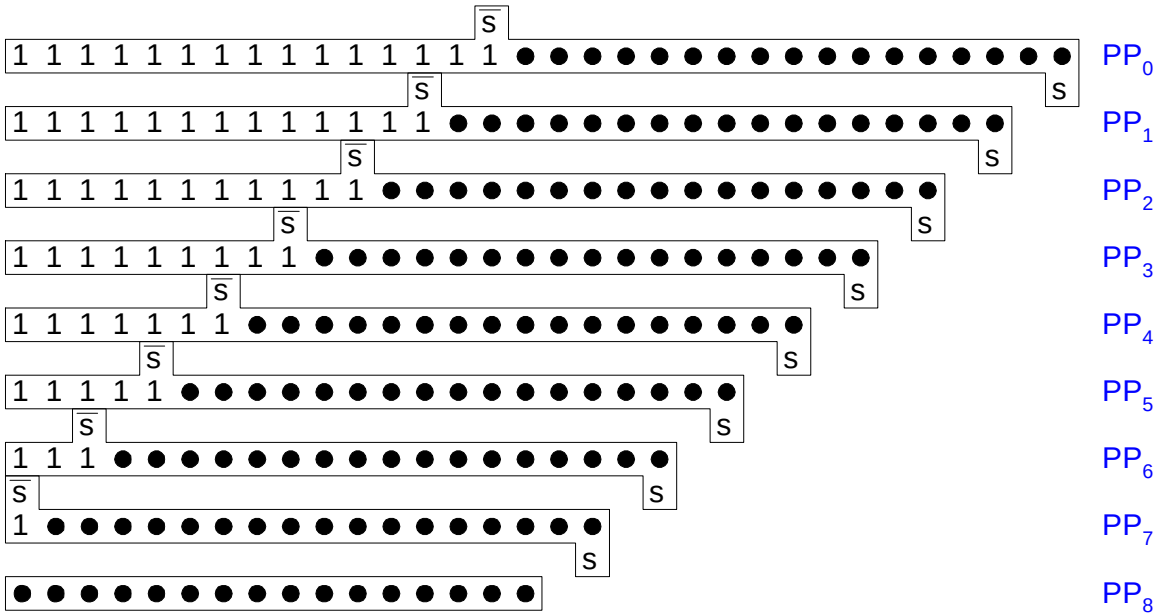
# Sign Extension

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



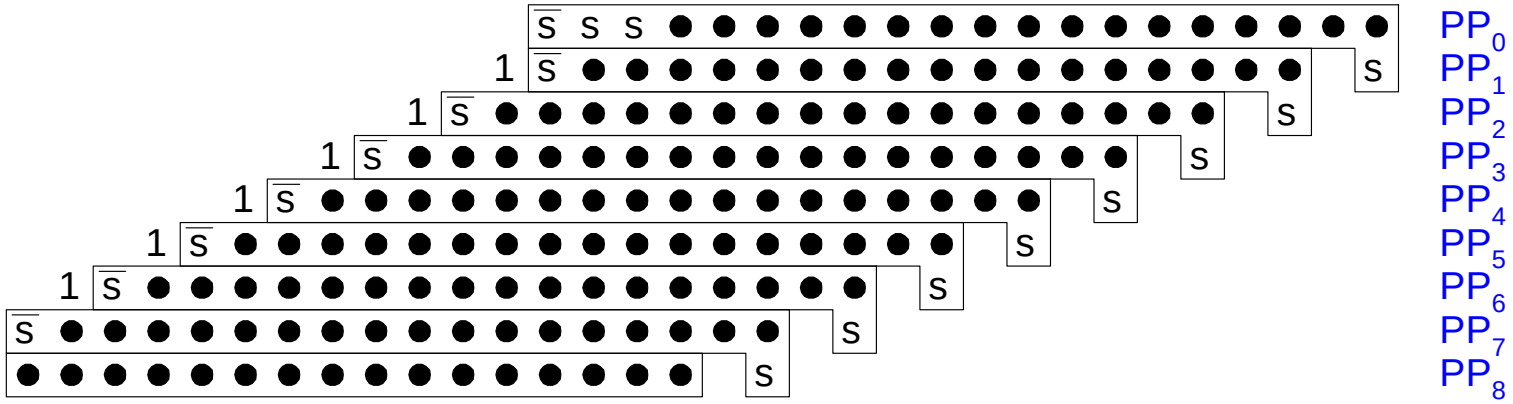
# Simplified Sign Ext.

- 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



# Even Simpler Sign Ext.

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



# Advanced Multiplication

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