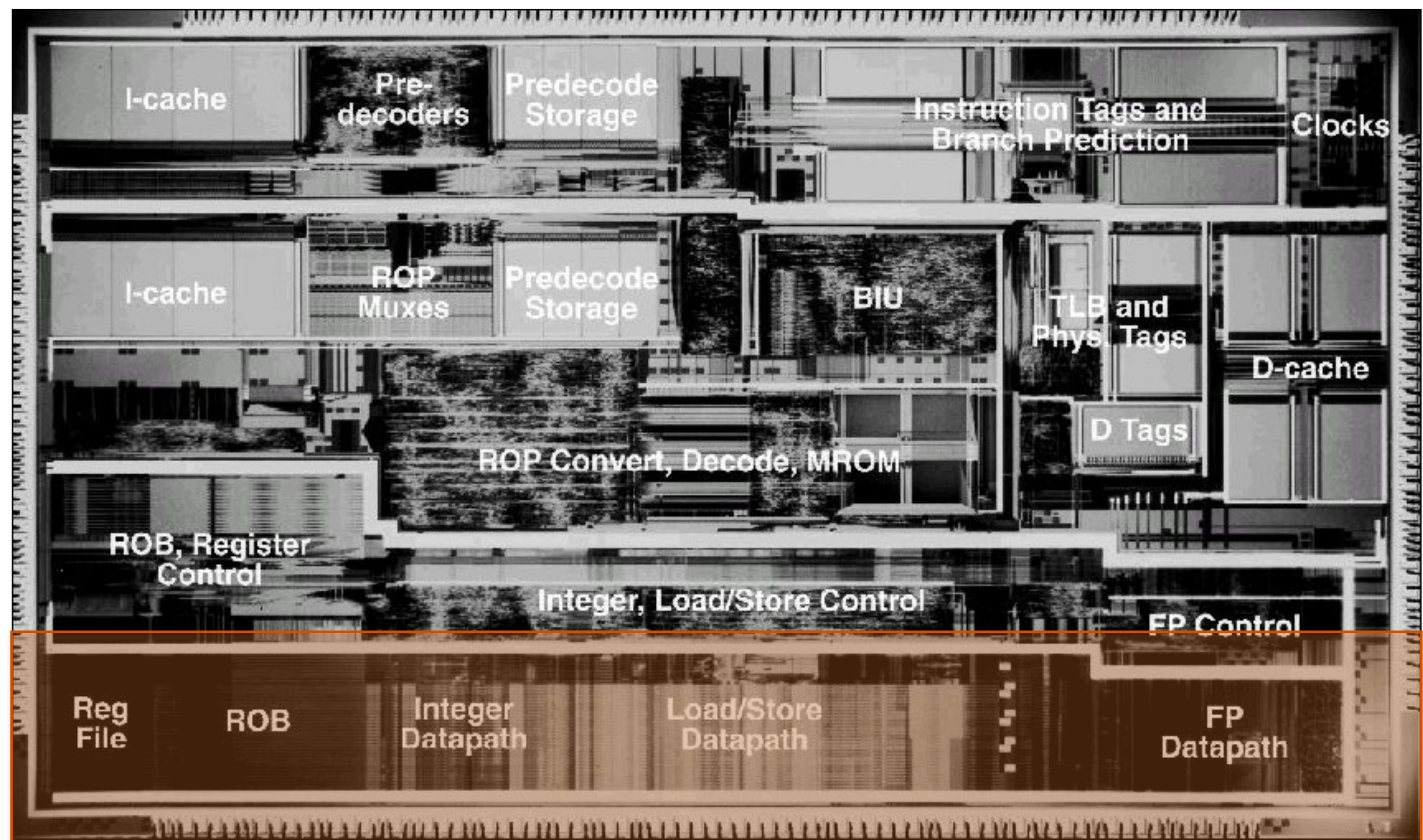
Lecture 12:
Datapath Design

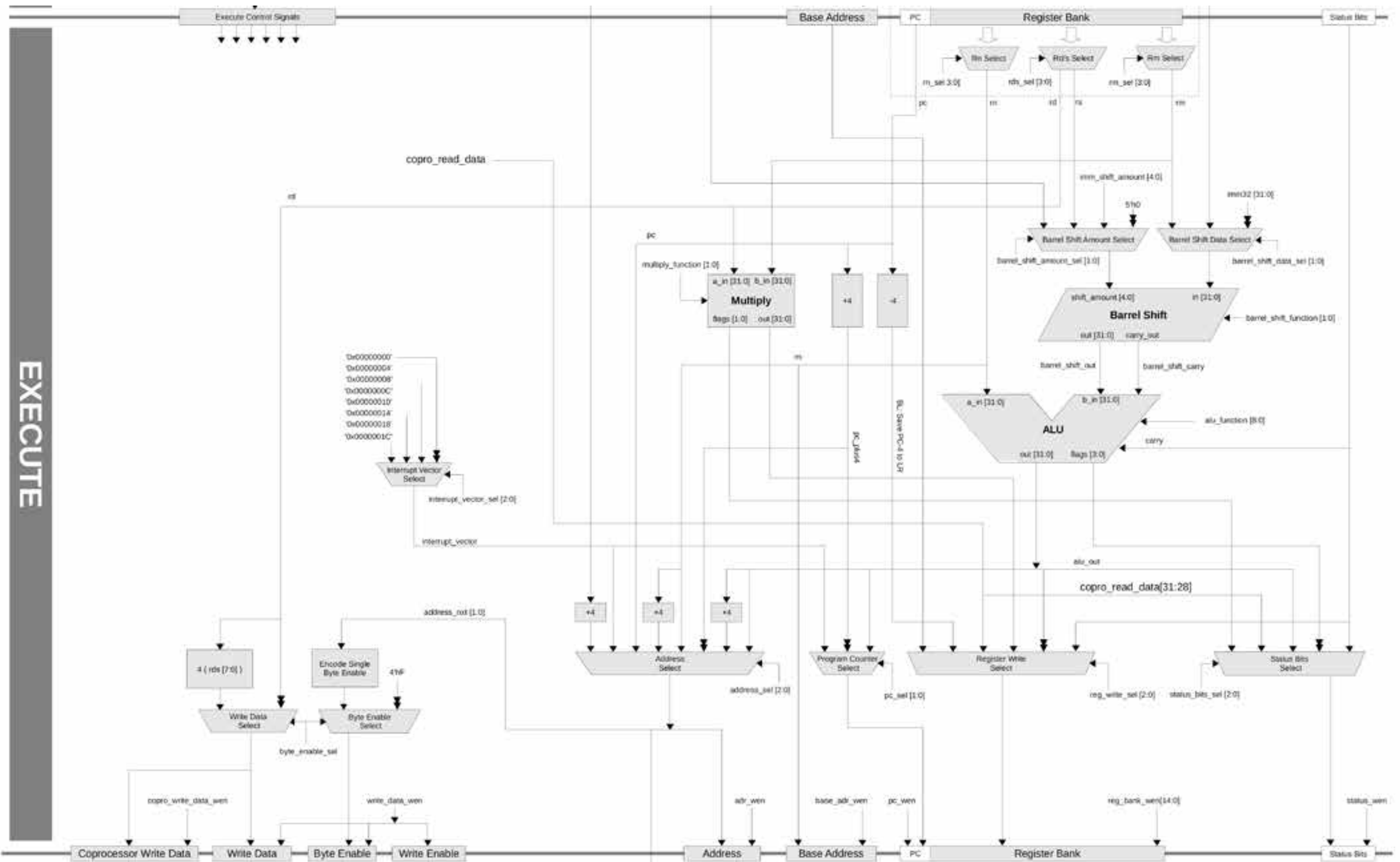
Mark McDermott
Electrical and Computer Engineering
The University of Texas at Austin

Datapath Design

- Comparators
- Shifters
- Adders
- Multipliers
- Registers



ARM Datapath

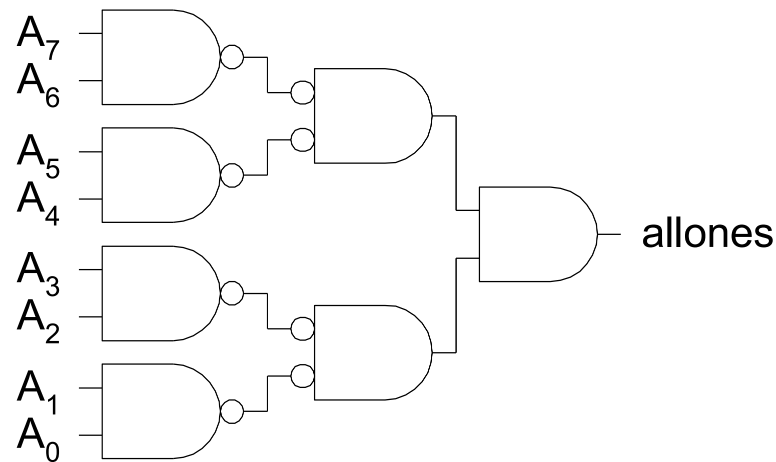


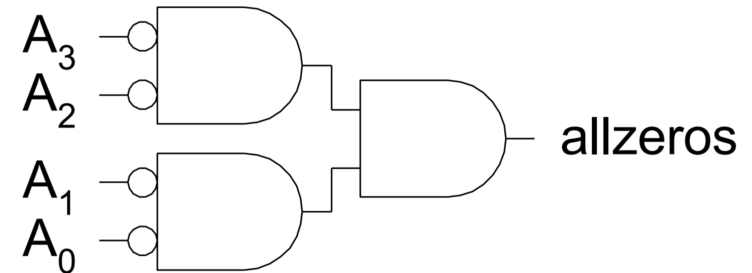
Comparators

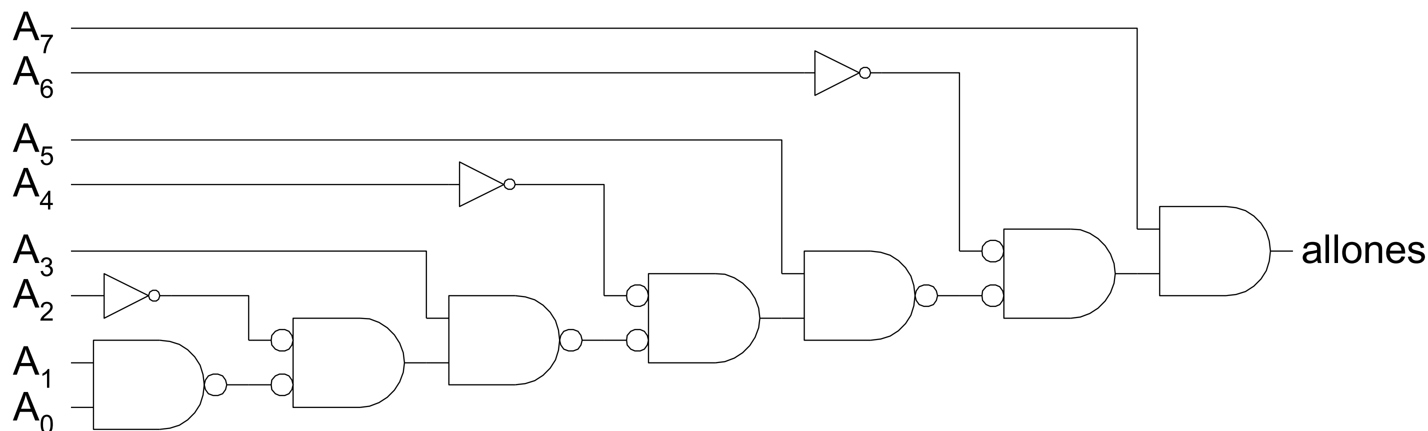
- 0' s detector: A = 00…000
- 1' s detector: A = 11…111
- Equality comparator: A = B
- Magnitude comparator: A < B

1's & 0's Detectors

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

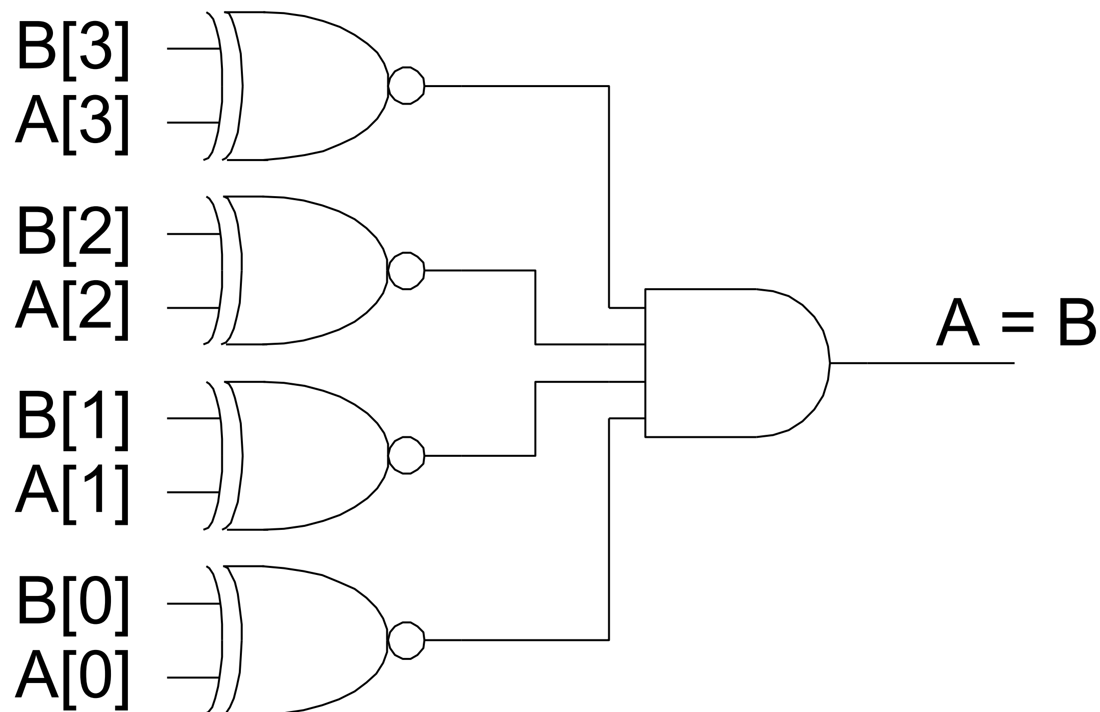






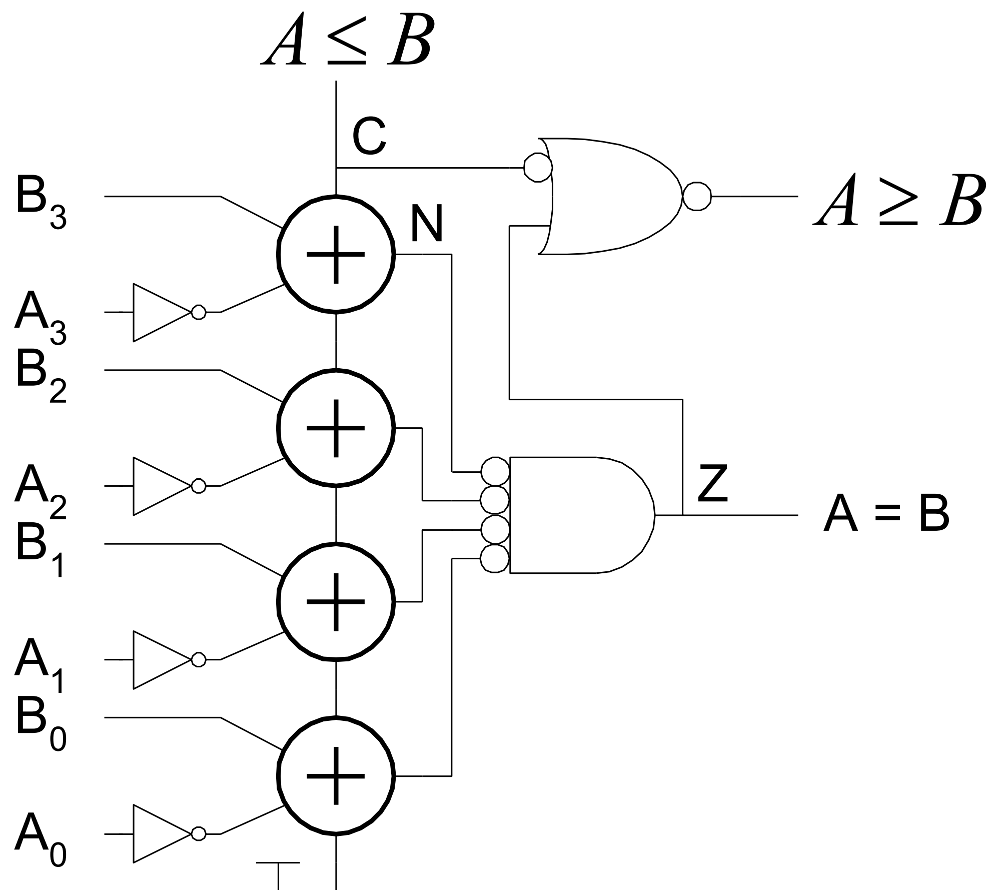
Equality Comparator

- Check if each bit is equal (XNOR, or “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

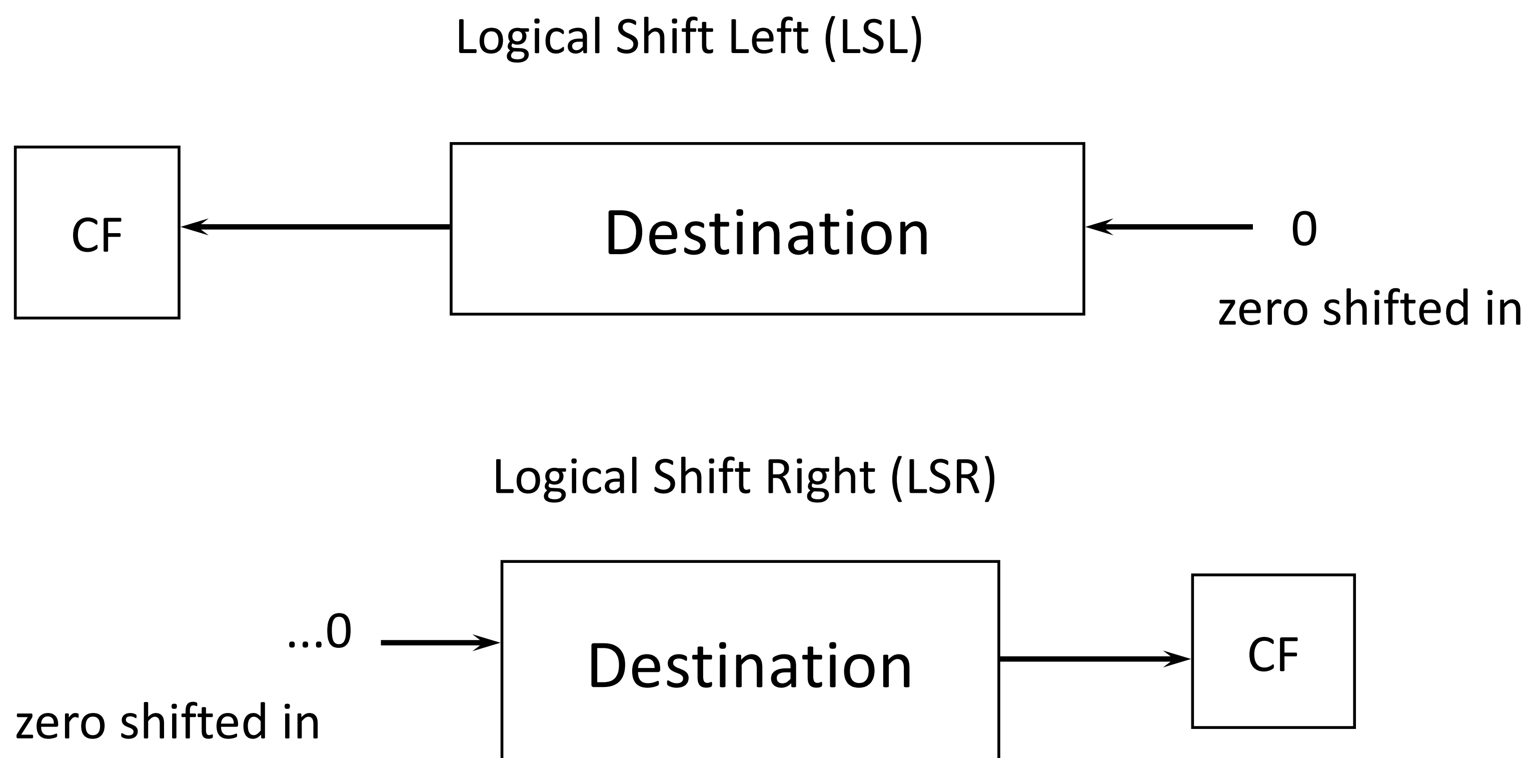
- **For signed numbers, comparison is harder**
  - C: carry out
  - Z: zero (all bits of A-B are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)

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 = 0101 1011 LSL1 = 0110



Shifters

- **Arithmetic Shift:**
  - **Shifts number left or right. Right shift sign extends**

    1011 ASR1 = 1101   1011 ASL1 = 0110

Arithmetic Shift Right (ASR)
Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations.
e.g.

ASR #5 = divide by 32

Arithmetic Shift Right

Sign bit shifted in

Shifters

- **Barrel Shift (Rotate):**
  - Shifts number left or right and fills with lost bits

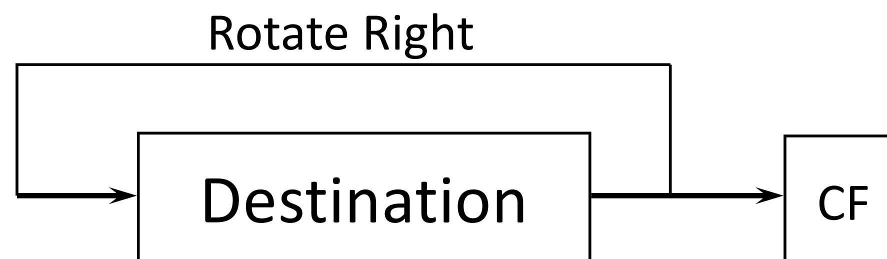
    1011 ROR1 = 1101   1011 ROL1 = 0111

Rotate Right (ROR)

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g.   ROR #5

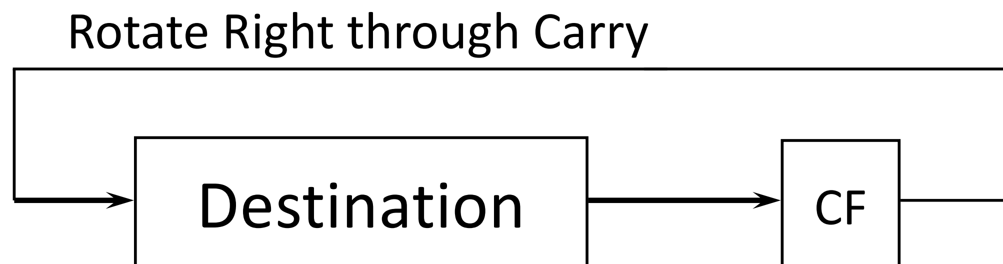
Note the last bit rotated is also used as the Carry Out.



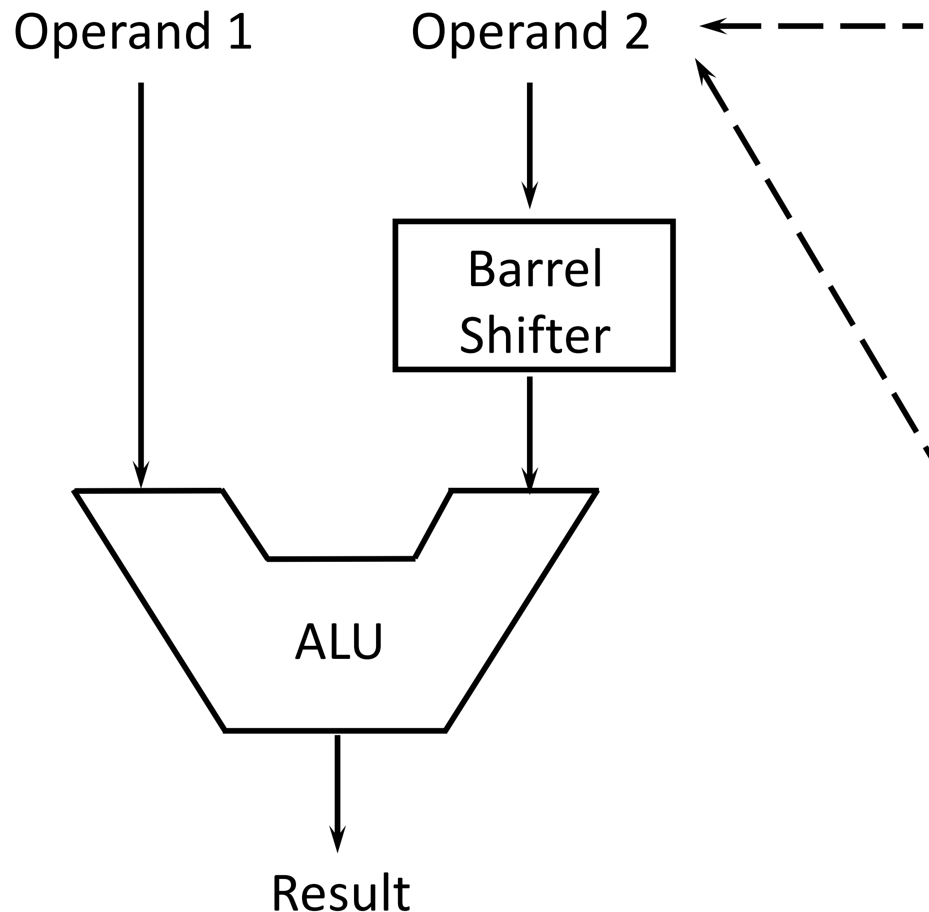
Rotate Right Extended (RRX)

This operation uses the CPSR C flag as a 33rd bit.

Rotates right by 1 bit. Encoded as ROR #0



ARM: Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation applied.
Shift value can be either be:
- **5 bit unsigned integer**
- **Specified in bottom byte of another register.**

Immediate value
- **8 bit number**
- **Can be rotated right through an even number of positions.**
- **Assembler will calculate rotate for you from constant.**

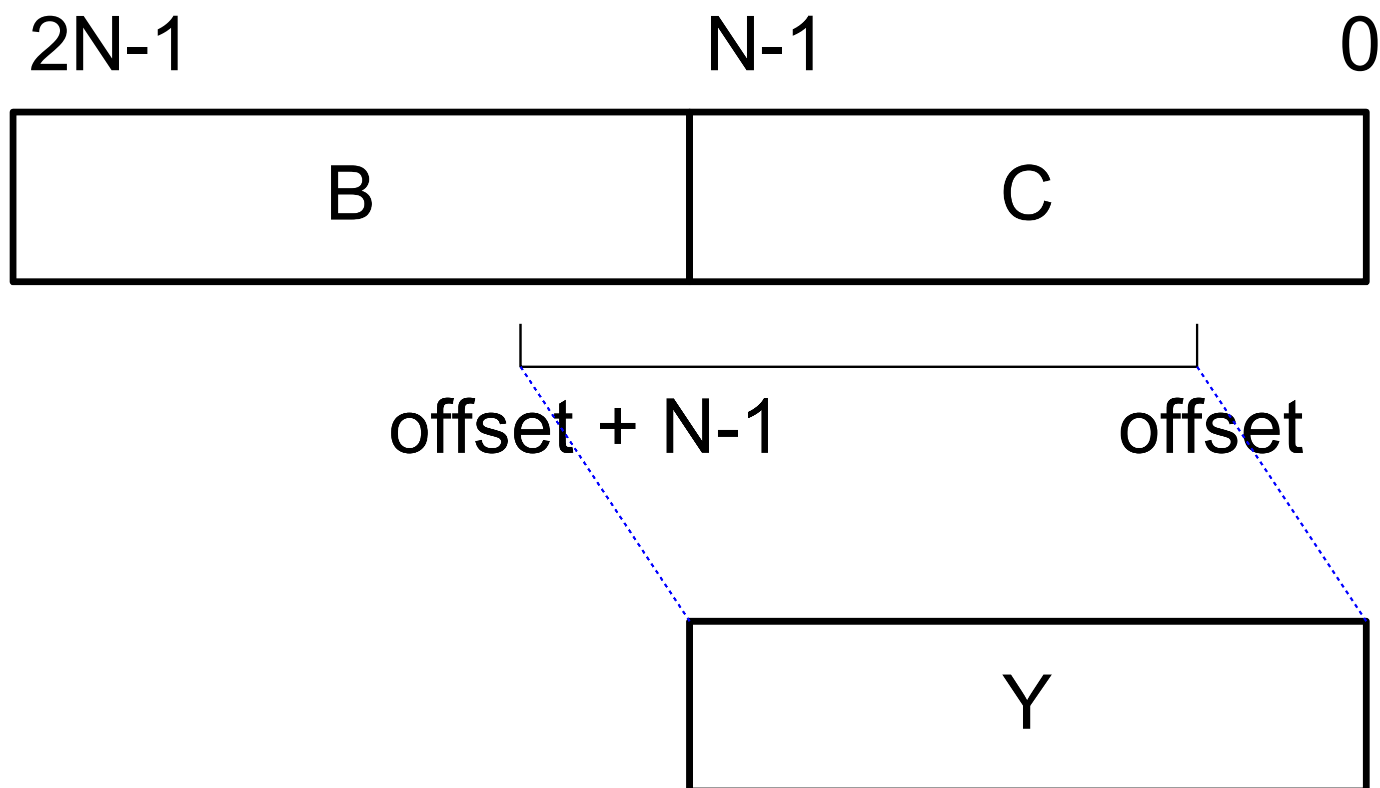
Second Operand: Using a Shifted Register

- **Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.**
- **A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.**
  - **Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.**

```
MOV R2, R0, LSL #2    ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD R9, R5, R5, LSL #3      ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB R9, R5, R5, LSL #3      ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB R10, R9, R8, LSR #4     ; R10 = R9 - R8 / 16
MOV R12, R4, ROR R3         ; R12 = R4 rotated right by value of R3
```

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

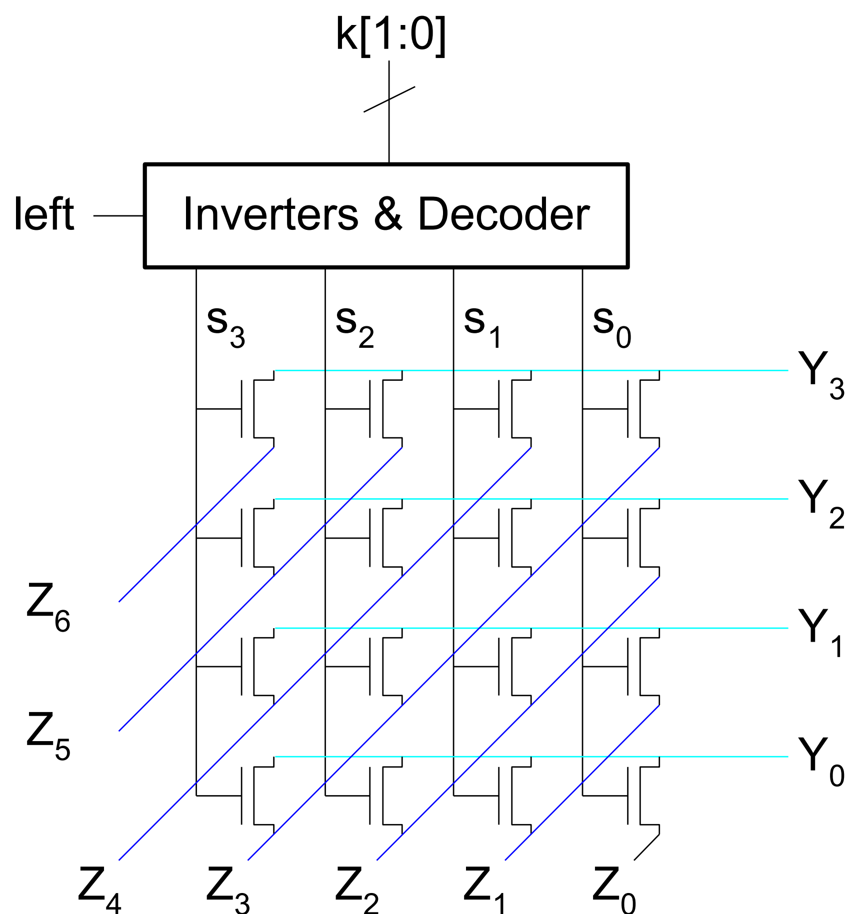
Simplified Funnel Shifter

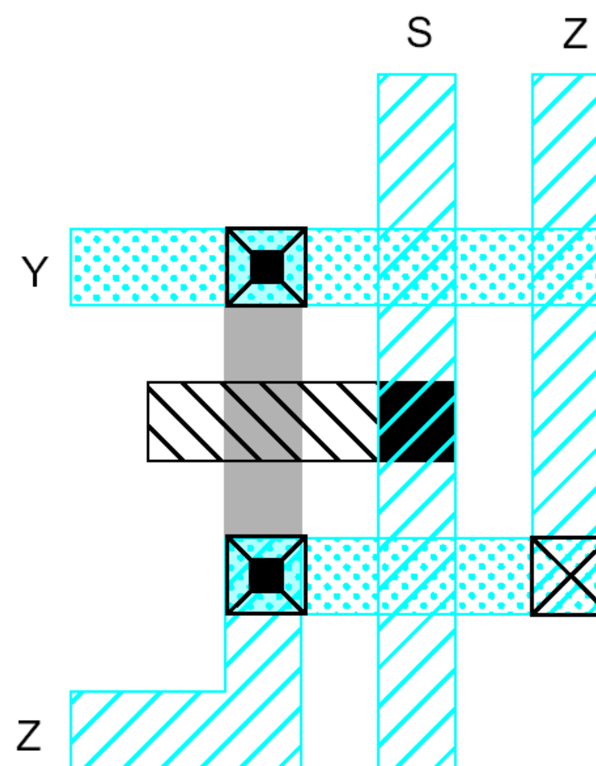
- **Optimize down to 2N-1 bit input**

Table 10.11	Simplified funnel shifter	
Shift Type	**Z**	**Offset**
Logical Right	$0..0, A_{N-1} \ldots A_0$	k
Logical Left	$A_{N-1} \ldots A_0, 0..0$	$\overline{k}$
Arithmetic Right	$A_{N-1} \ldots A_{N-1}, A_{N-1} \ldots A_0$	k
Arithmetic Left	$A_{N-1} \ldots A_0, 0..0$	$\overline{k}$
Rotate Right	$A_{N-2} \ldots A_0, A_{N-1} \ldots A_0$	k
Rotate Left	$A_{N-1} \ldots A_0, A_{N-1} .. A_1$	$\overline{k}$

Funnel Shifter Design 1

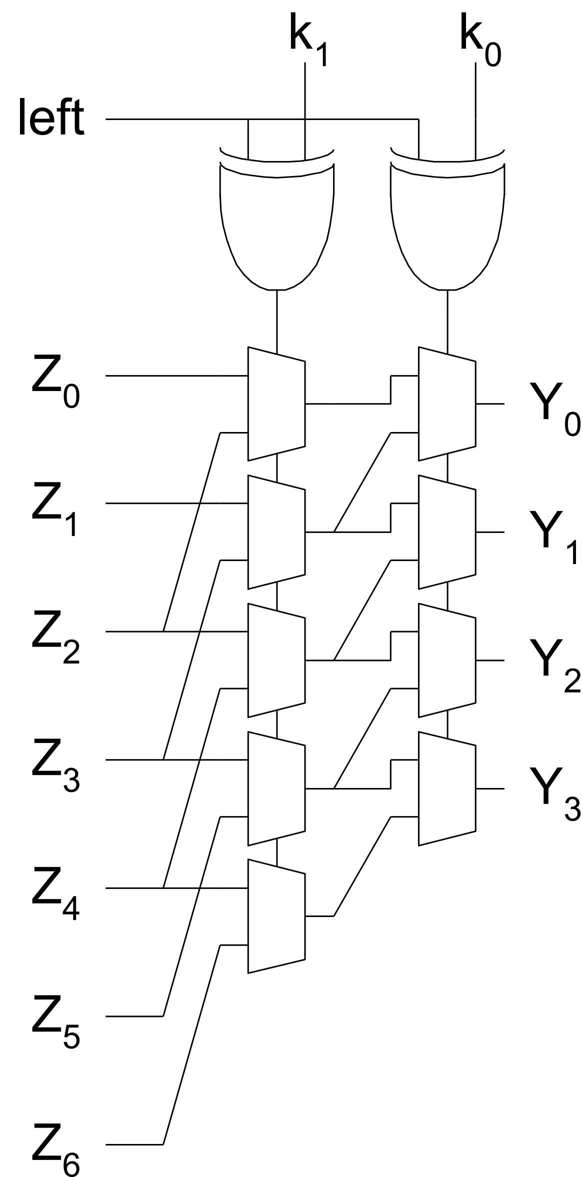
- **N N-input multiplexers**
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design (Vt drops!)



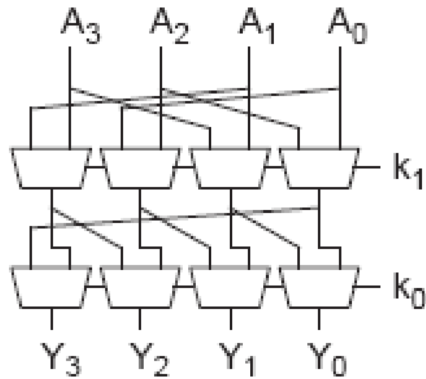


Funnel Shifter Design 2

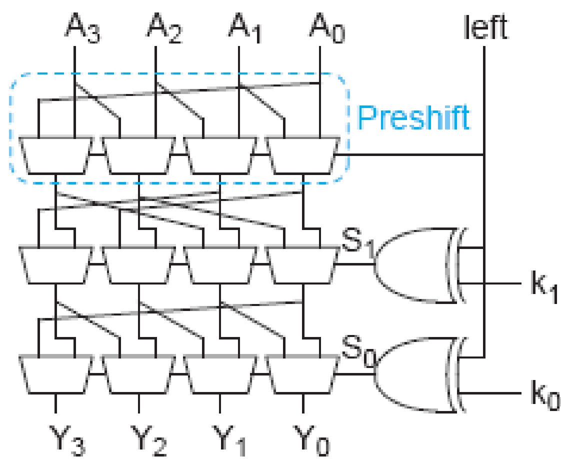
- **Log N stages of 2-input MUXes**
  - No select decoding needed



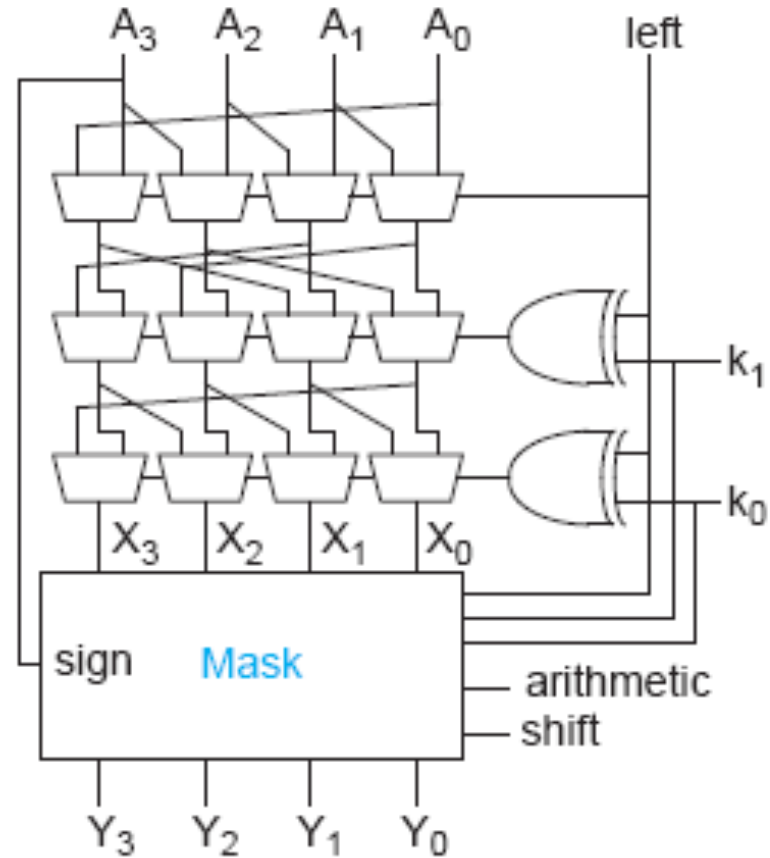
Logarithmic Barrel Shifter



Right shift only



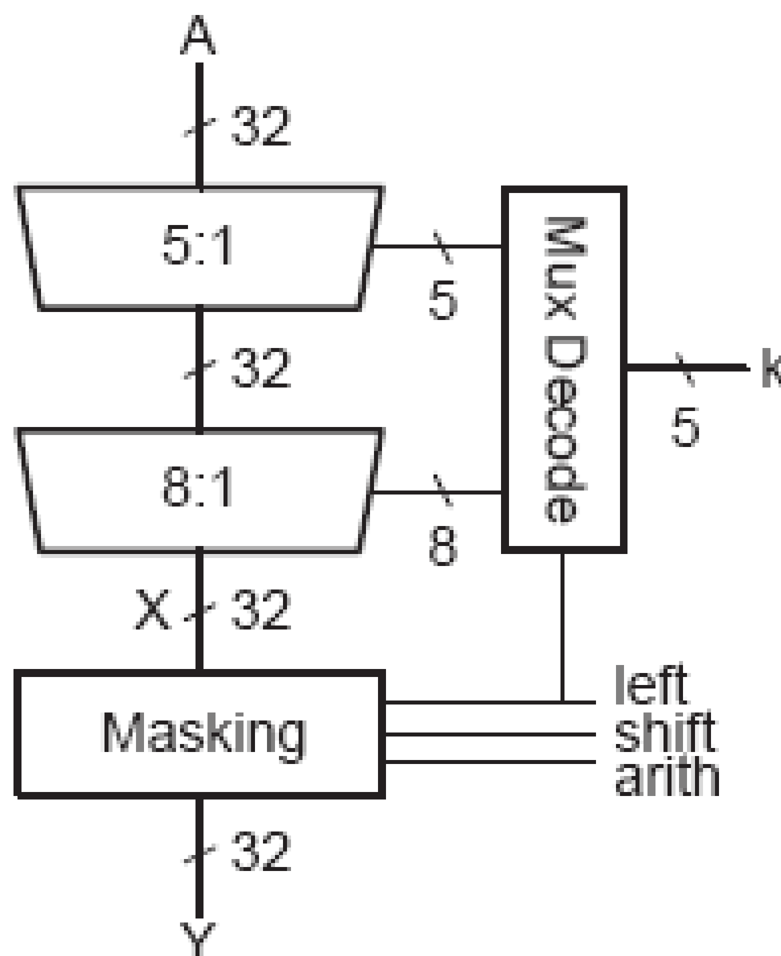
Right/Left shift



Right/Left Shift & Rotate

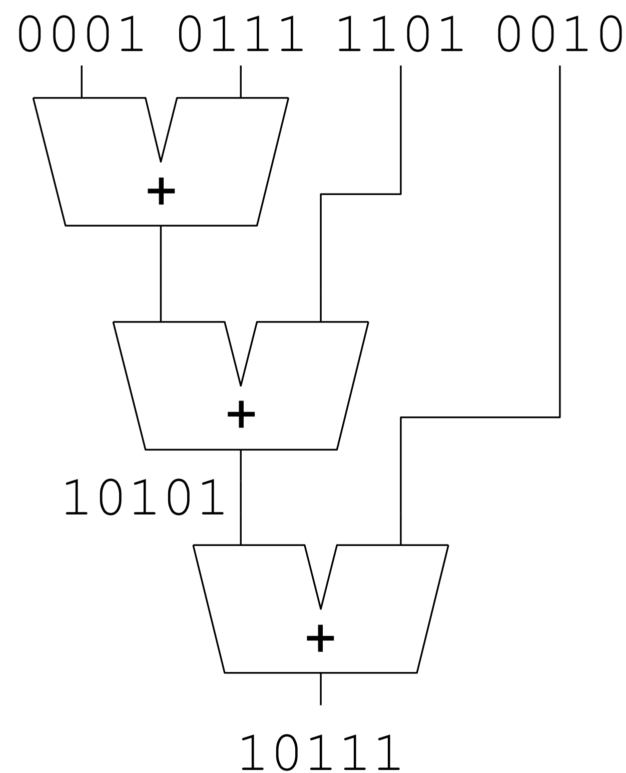
32-bit Logarithmic Barrel

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



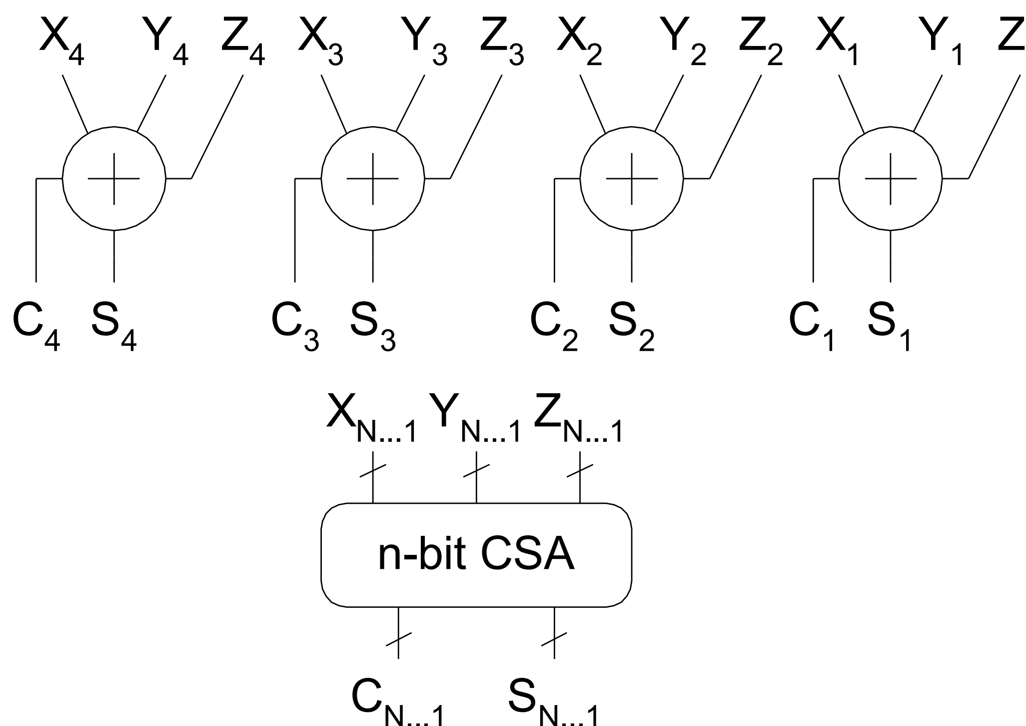
Multi-input Adders

- **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



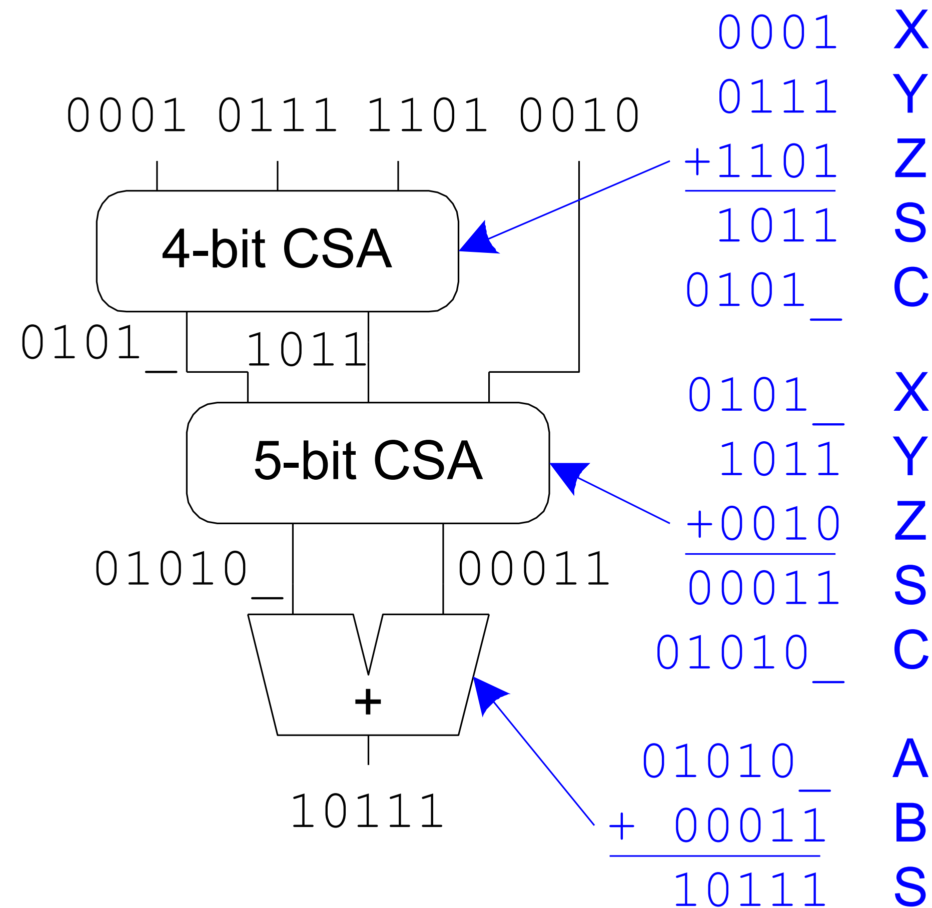
Carry Save Addition

- **Full adder sums 3 inputs, produces 2 outputs**
  - **Carry output has twice *weight* of sum output**
- **N full adders in parallel: *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:**

```
         1100  :  12₁₀
         0101  :  5₁₀
         ----
         1100
        0000
       1100
      0000
     --------
     00111100  :  60₁₀
```

multiplicand

multiplier

partial products

product

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

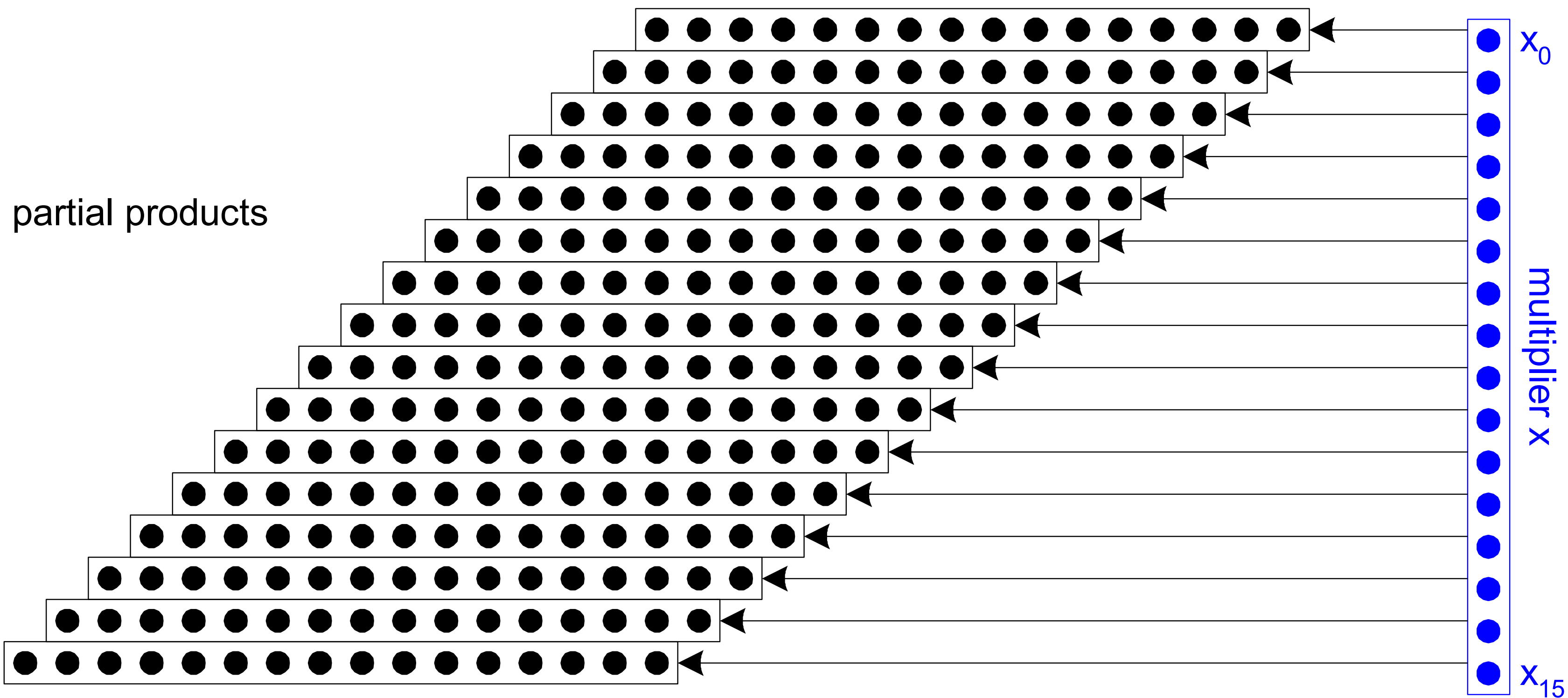
General Form

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

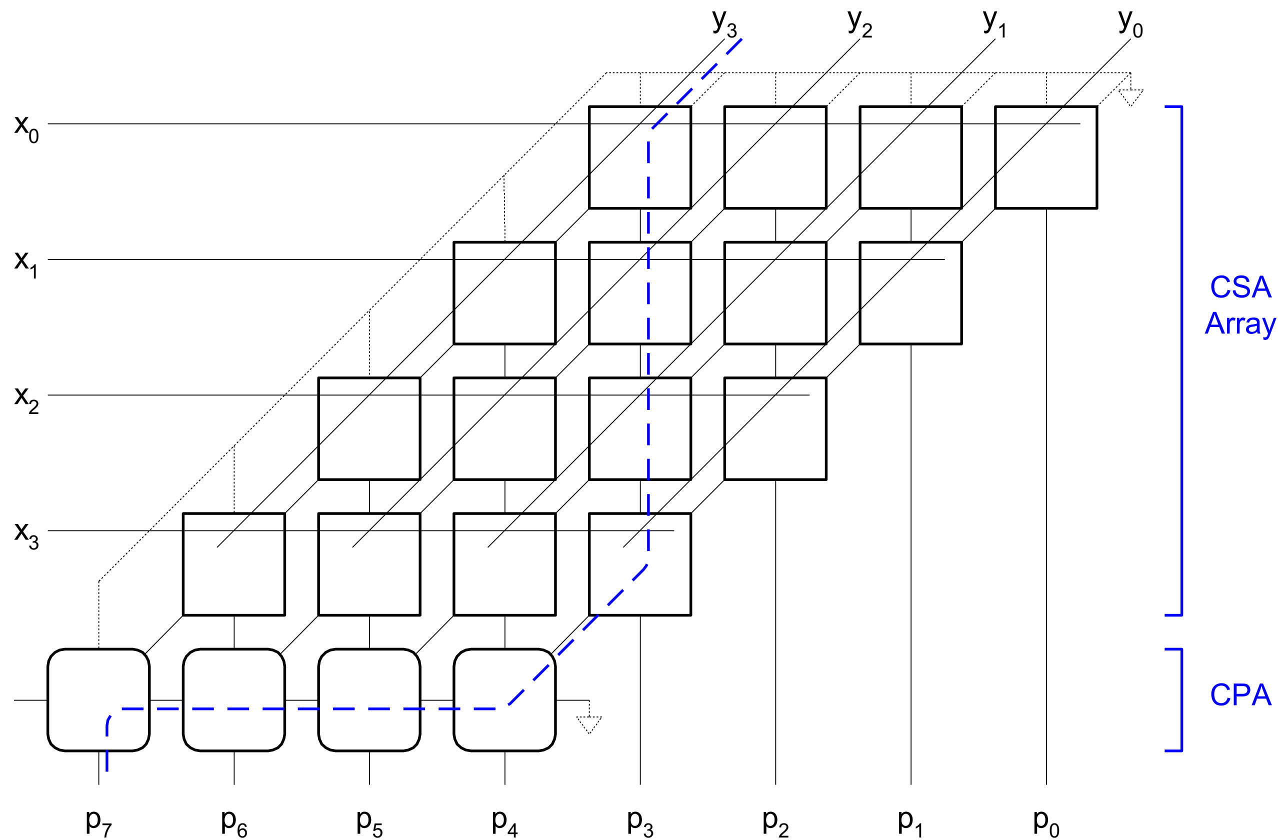
						y_5	y_4	y_3	y_2	y_1	y_0	multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	partial products
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4	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						
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

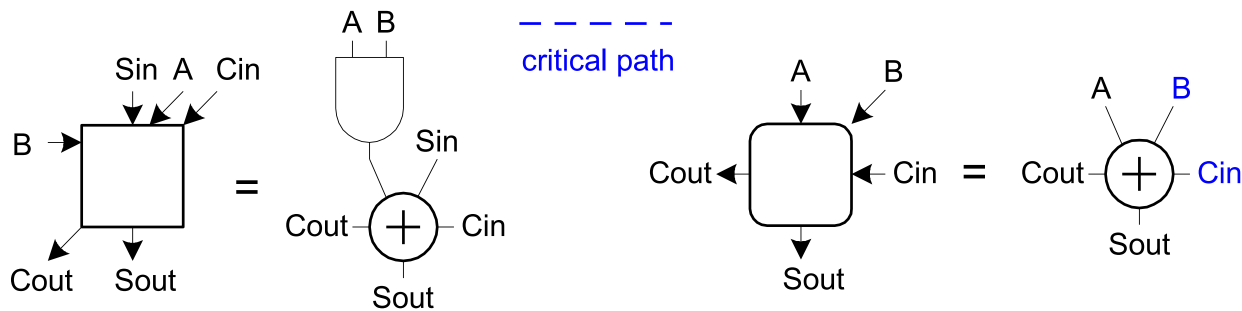
Dot Diagram

- **Each dot represents a bit**



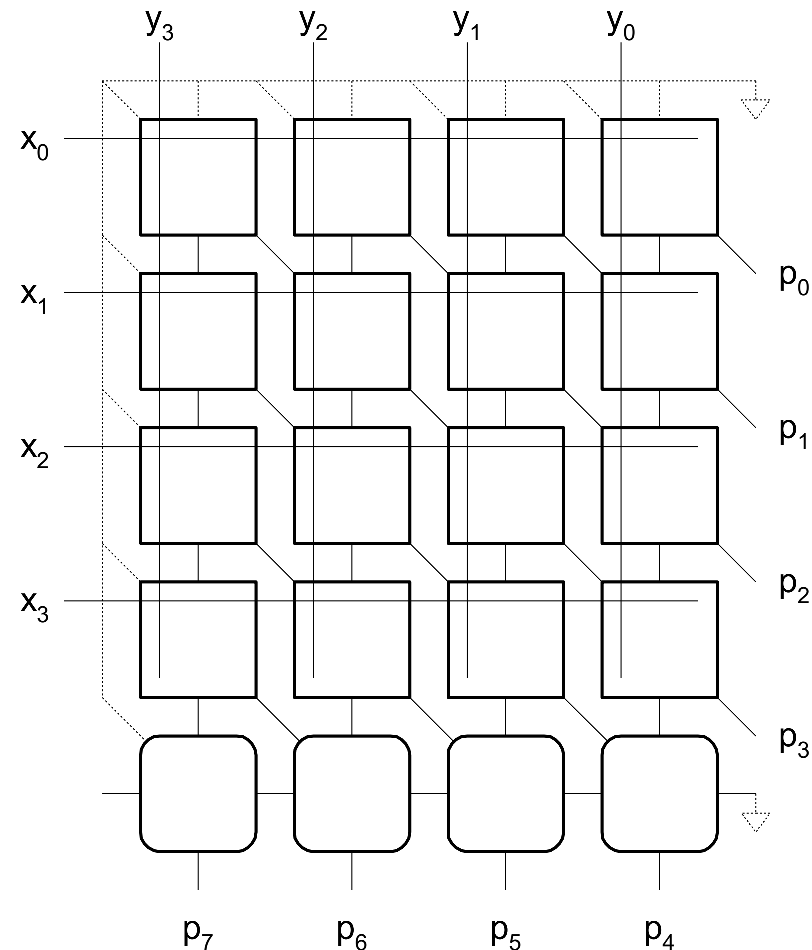
Array Multiplier





Rectangular Array

- Squash array to fit rectangular floorplan



Fewer Partial Products

UT ECE

- **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

UT ECE

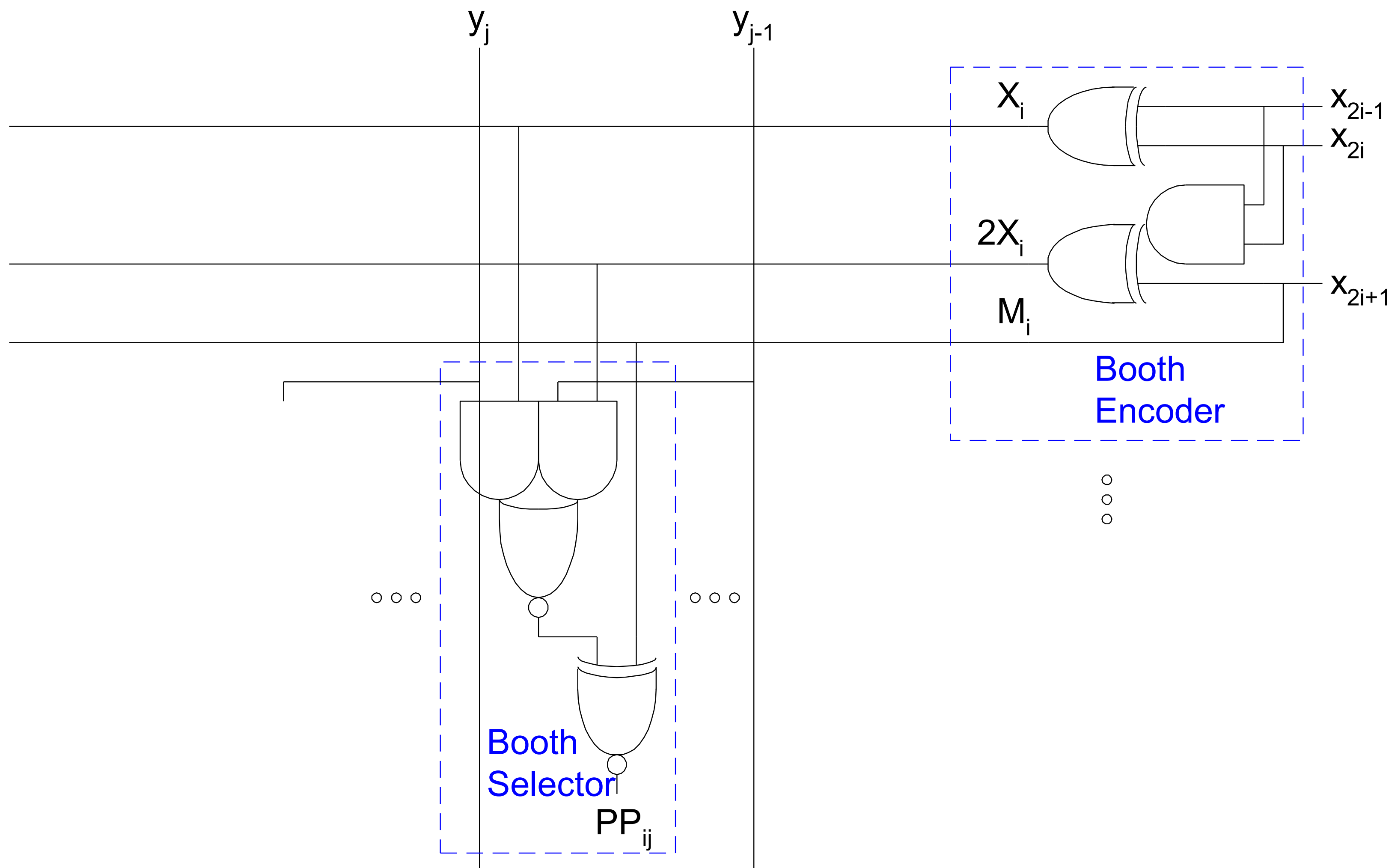
- Instead of 3Y, try –Y, then increment next partial product to add 4Y
- Similarly, for 2Y, try –2Y + 4Y in next partial product

Table 10.12 Radix-4 modified Booth encoding values

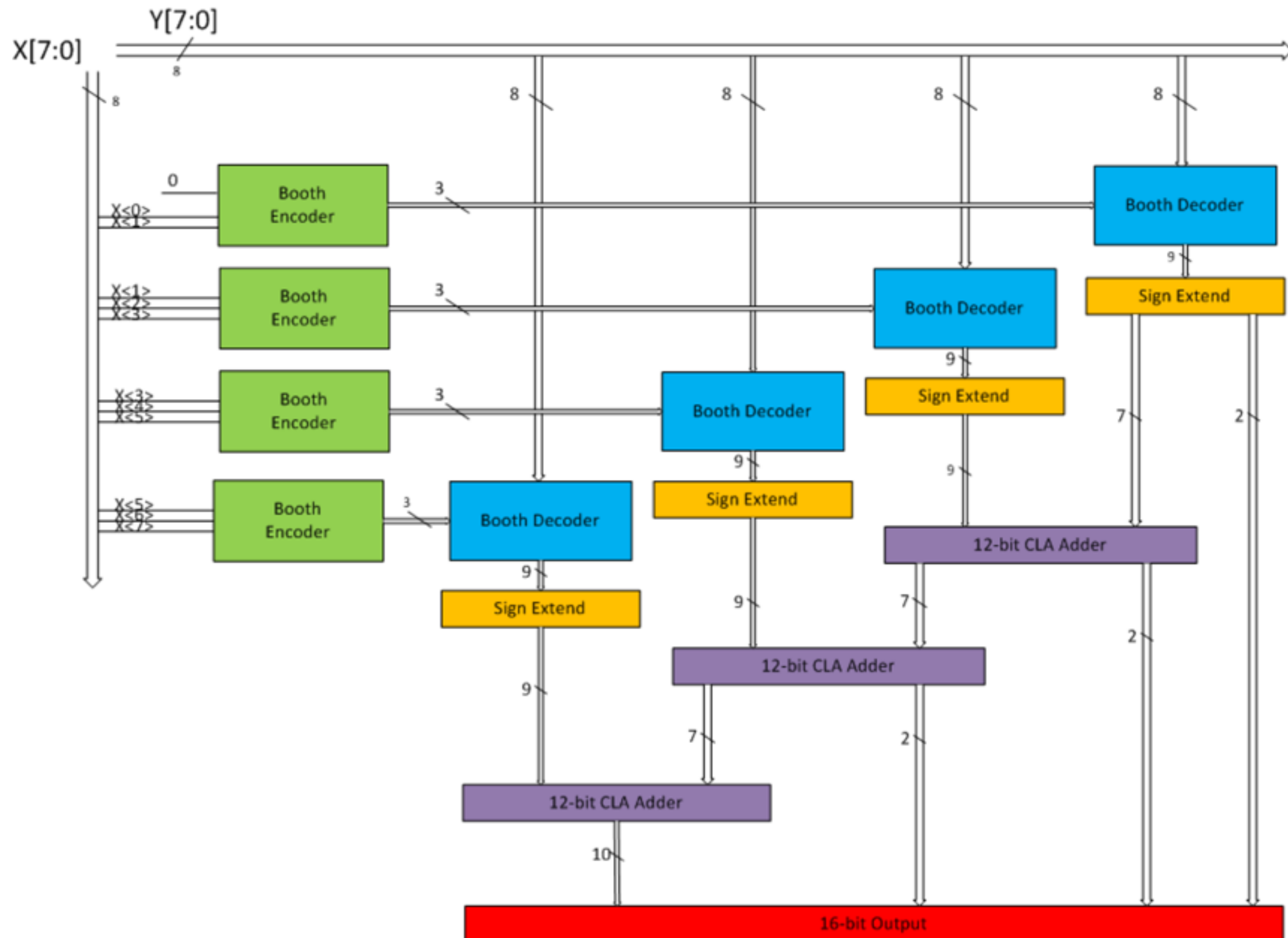
Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	X_i	$2X_i$	M_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	-0 (= 0)	0	0	1

Booth Hardware

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

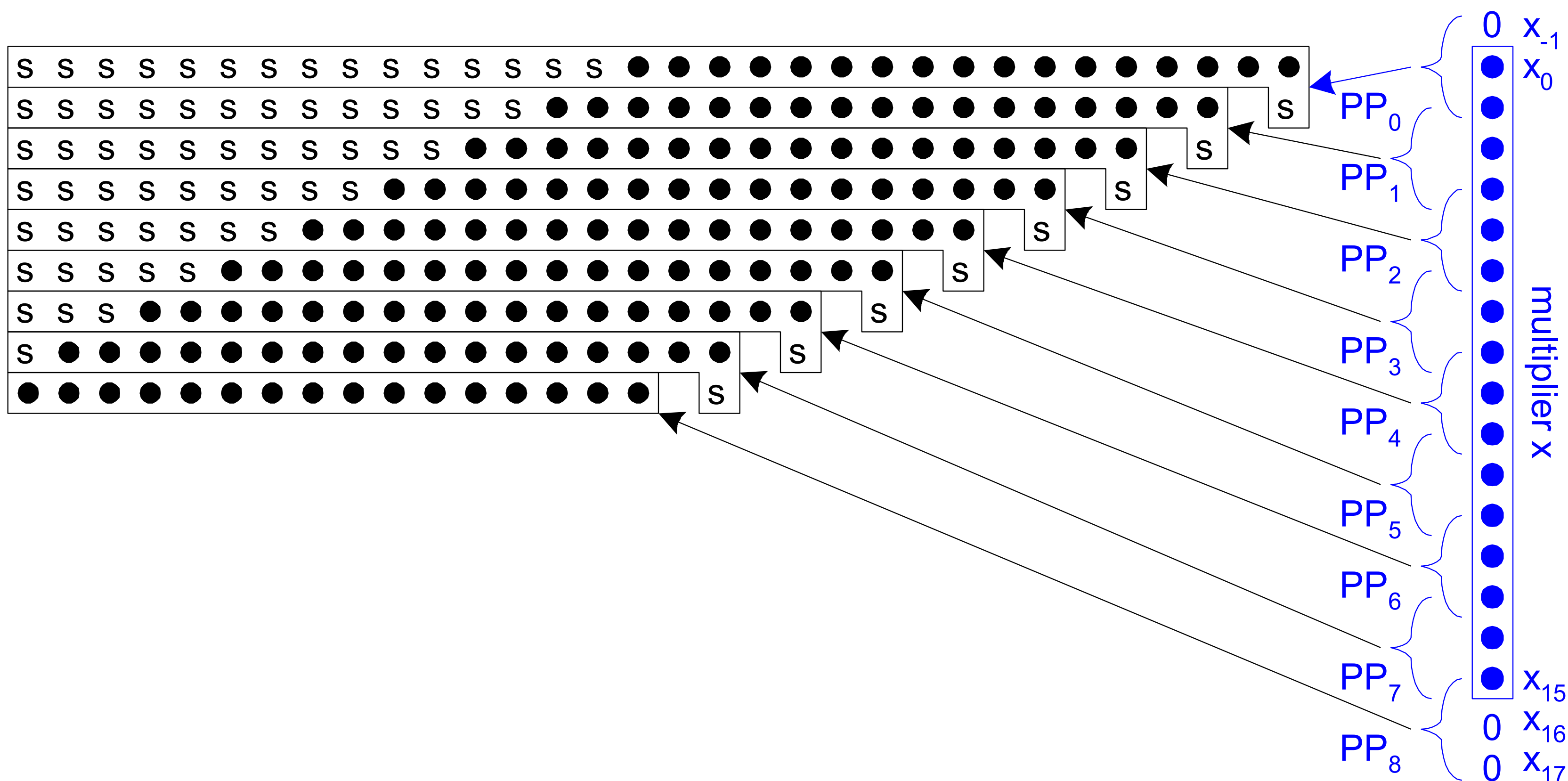


Booth hardware



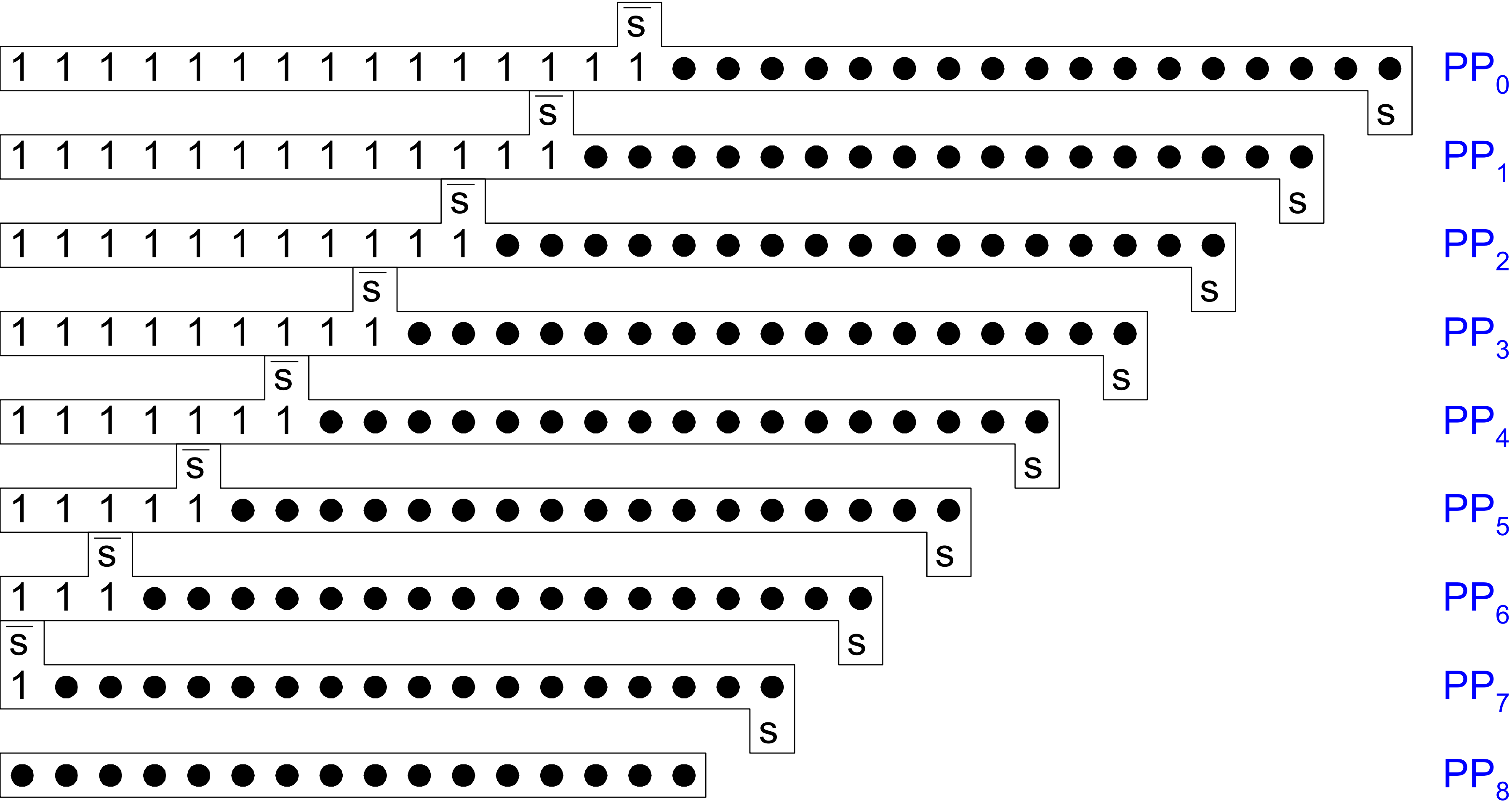
Sign Extension

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



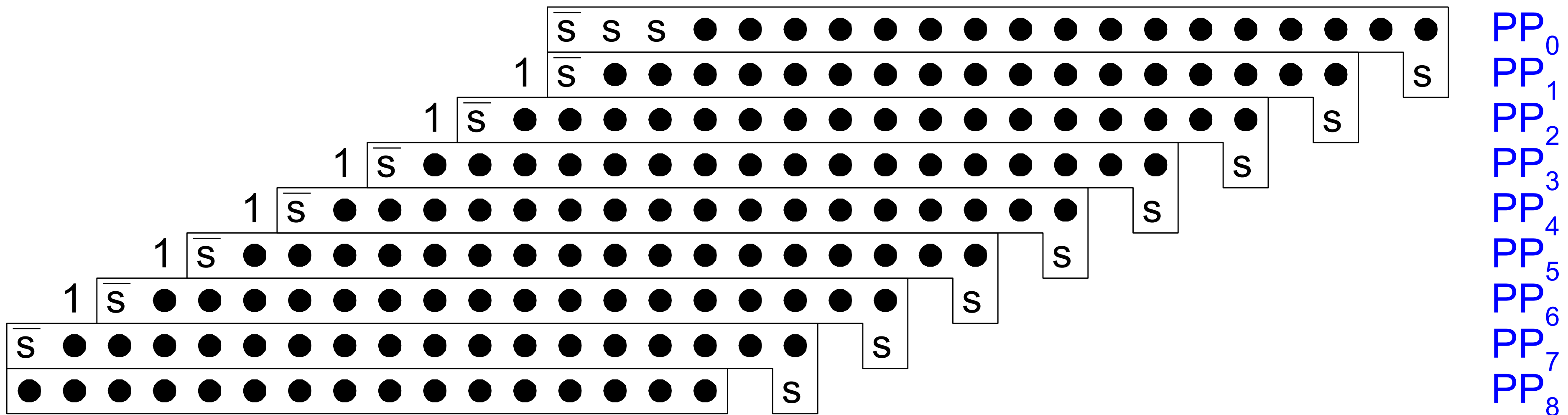
Simplified Sign Extension

- 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 Extension

- **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