9. Datapath Design

Jacob Abraham

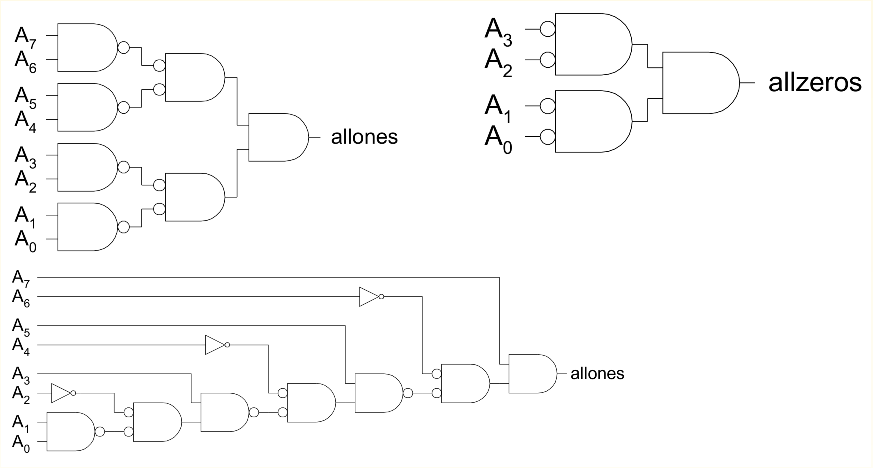
Department of Electrical and Computer Engineering
The University of Texas at Austin

VLSI Design
Fall 2020

September 24, 2020

1s and 0s Detectors

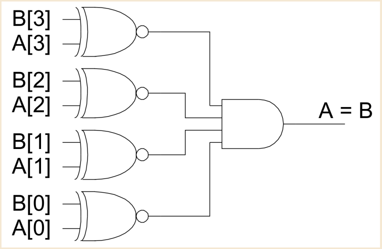
- 1s detector: N-input AND gate
- 0s detector: Inversions + 1s detector (N-input NOR)



Comparators

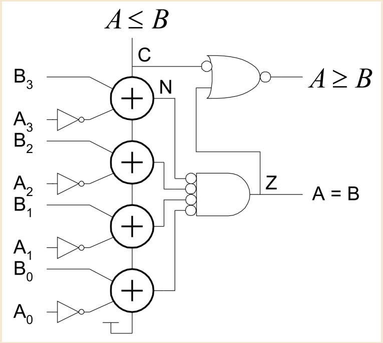
Equality Comparator

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



Magnitude Comparator

- Compute $B - A$ and look at sign
- $B - A = B + \overline{A} + 1$
- For unsigned numbers, carry out is sign bit



Signed Versus Unsigned Numbers

- 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)

Magnitude Comparison

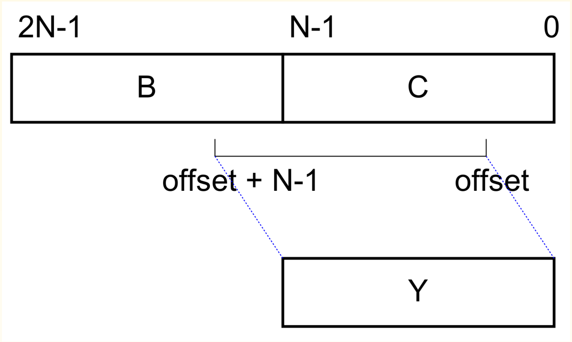
Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C + Z}$	$\overline{(N \oplus V) + Z}$
$A > B$	$\overline{C}$	$(N \oplus V)$
$A \leq B$	C	$\overline{(N \oplus V)}$
$A \geq B$	$\overline{C} + Z$	$(N \oplus V) + Z$

Shifters

- Logical Shift:
  - Shifts number left or right and fills with 0s
    - 1011 LSR 1 = 0101
    - 1011 LSL 1 = 0110
- Arithmetic Shift:
  - Shifts number left or right; right shift – sign extend
    - 1011 ASR 1 = 1101
    - 1011 ASL 1 = 0110
- Rotate:
  - Shifts number left or right and fills with lost bits
    - 1011 ROR 1 = 1101
    - 1011 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

Shift Type	**B**	**C**	**Offset**
Logical Right	$0 \ldots 0$	$A_{N-1} \ldots A_0$	k
Logical Left	$A_{N-1} \ldots A_0$	$0 \ldots 0$	$N-k$
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	$A_{N-1} \ldots A_0$	$A_{N-1} \ldots A_0$	$N-k$

Computing N-k requires an adder

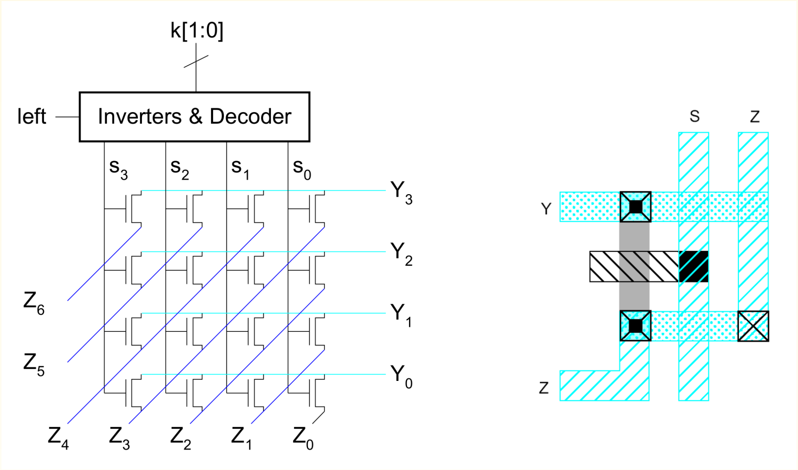
Simplified Funnel Shifter

Optimize down to 2N-1 bit input

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	$A_{N-1} \ldots A_0, A_{N-1} \ldots A_1$	$\overline{k}$

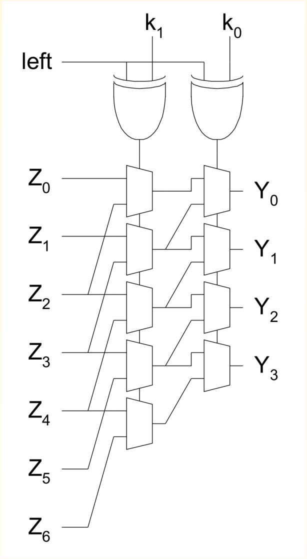
Funnel Shifter Design – 1

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



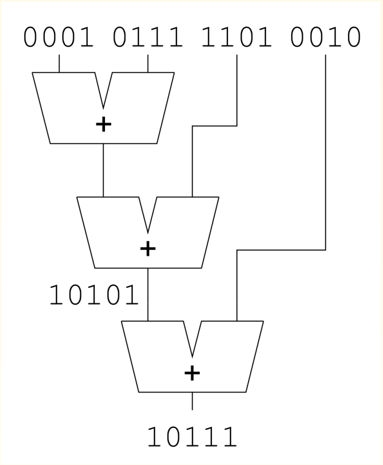
Funnel Shifter Design – 2

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



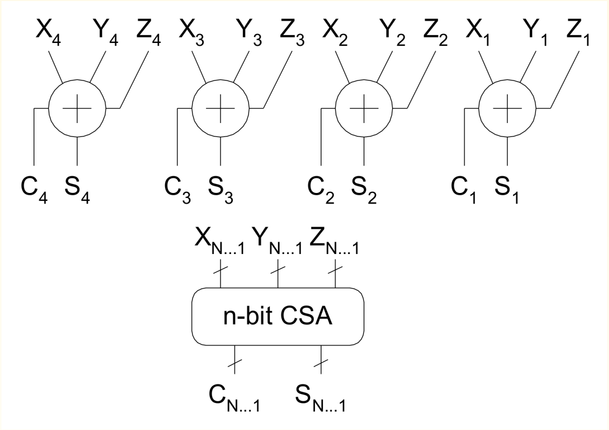
Multi-Input Adders

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



Carry Save Addition

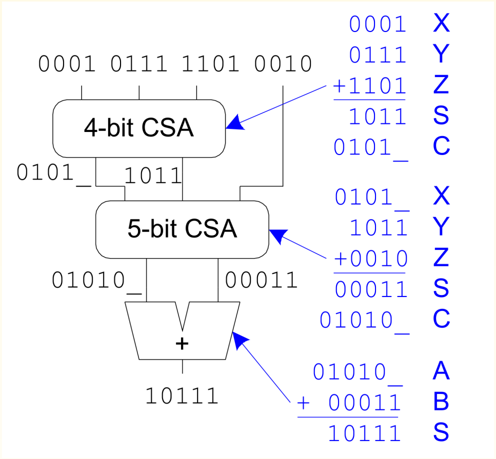
- Full adder sums 3 inputs, produces 2 outputs
  - Carry output has twice the **weight** of sum output
- N full adders in parallel: **carry save adder**
  - Produce N sums and N carry outs



X_4 Y_4 Z_4 X_3 Y_3 Z_3 X_2 Y_2 Z_2 X_1 Y_1 Z_1
C_4 S_4 C_3 S_3 C_2 S_2 C_1 S_1
$X_{N...1}$ $Y_{N...1}$ $Z_{N...1}$
n-bit CSA
$C_{N...1}$ $S_{N...1}$

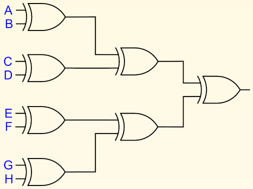
CSA Application

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

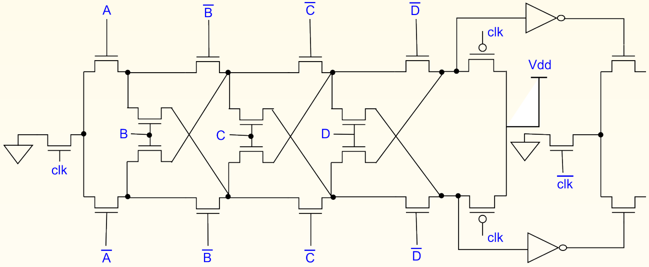


Parity Generators

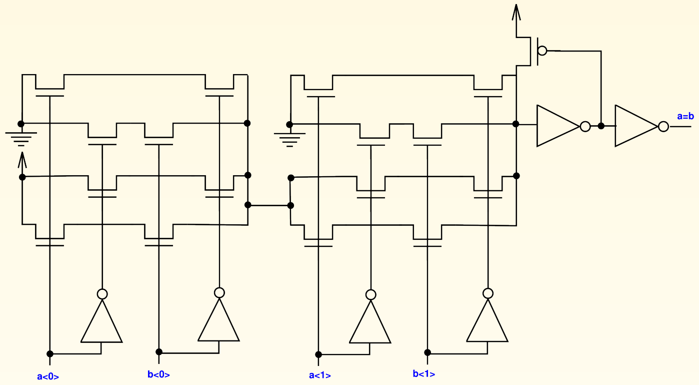
Static XOR Tree

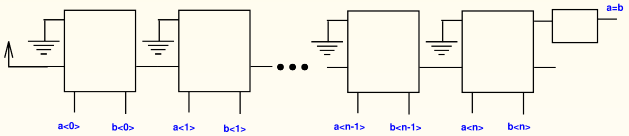


Dynamic XOR Circuit



Pass Gate Comparator





Multiplication

Example:

```
     1100  :  12₁₀        multiplicand
     0101  :  5₁₀         multiplier
     ----
     1100                ┐
    0000                 │  partial
   1100                  │  products
  0000                   ┘
  --------
  00111100 :  60₁₀        product
```

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

General Form for Multiplication

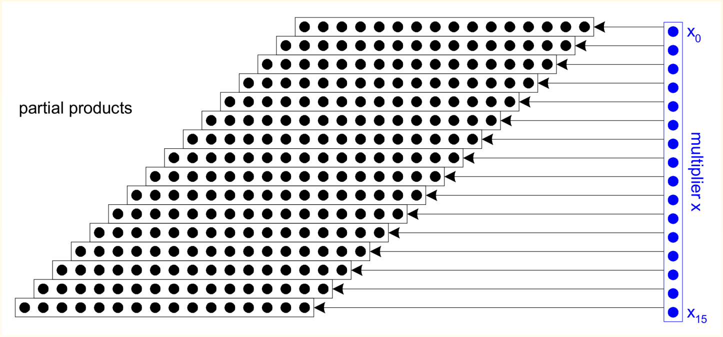
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
Product:

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

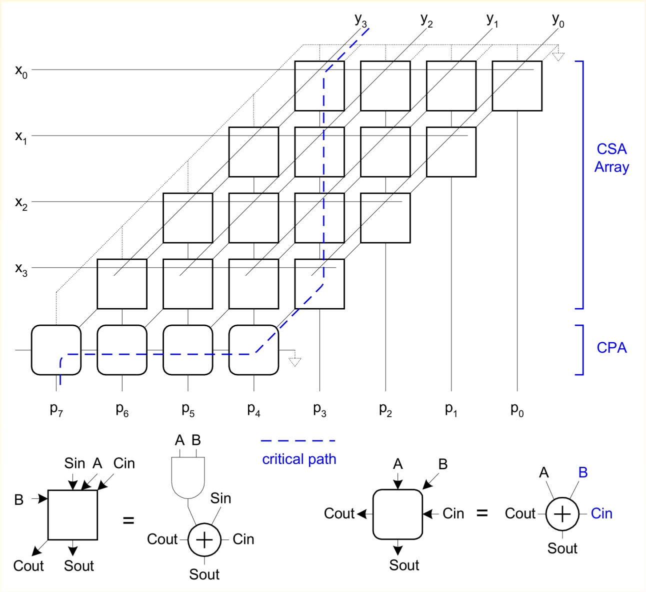
						y_5	y_4	y_3	y_2	y_1	y_0	multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			partial products
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4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

Each dot represents a bit

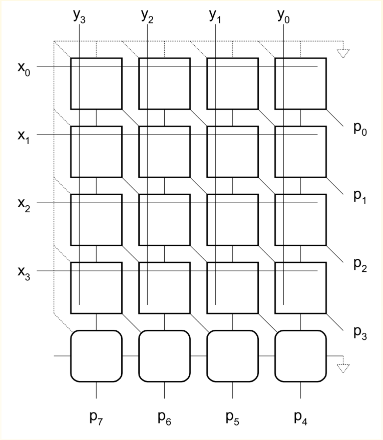


Array Multiplier



Rectangular Array

Squash array to fit rectangular floorplan



Fewer Partial Products – Booth Encoding

- 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
- Example, for r $=$ 2, look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder
- Is there a way to get 3Y without an addition step?

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

Radix-4 modified Booth encoding value

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

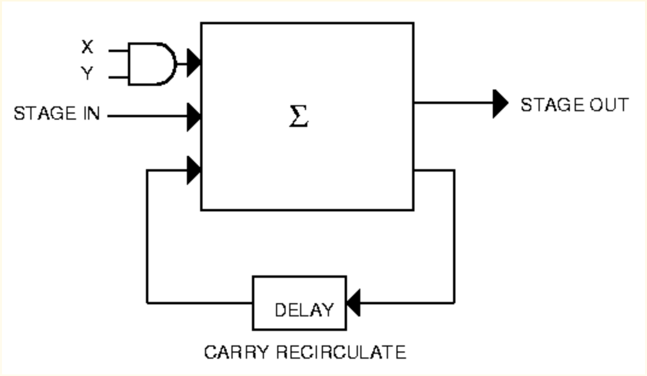
Advanced Multiplication

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

Serial Multiplication

- Lower area at expense of speed
  - Example, signal processing on bit streams
- Delay for $n \times n$ multiply
  - 2n bit product with 2n bit delay
  - Additional n-bit delay to shift n bits
  - Total delay of 3n bits
- Pipelined multiplier: possible to produce a new 2n bit product every 2n bit times after initial n bit delay
  - Only interest in high-order bits: n bit *latency* for n bit product
  - Can design for desired *throughput*

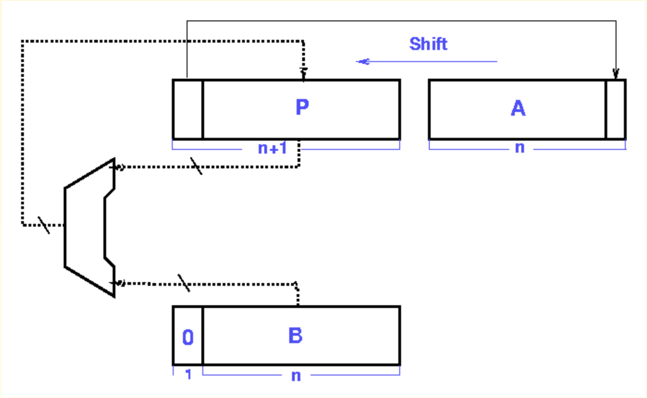
Serial Multiplier Architecture



Area for structure increases linearly with number of bits, n

Pipeline multiplier accumulates partial product sums starting with the least significant partial product (result is n-bit number which is truncated to n-1 bits before the next partial product)

Division



- To divide A by B
  - Shift P and A one bit left
  - Subtract B from P, put the result back
  - If result is negative, additional steps, set low order bits of A to 0, otherwise to 1
  - "restoring" or "non-restoring" division to fix negative result

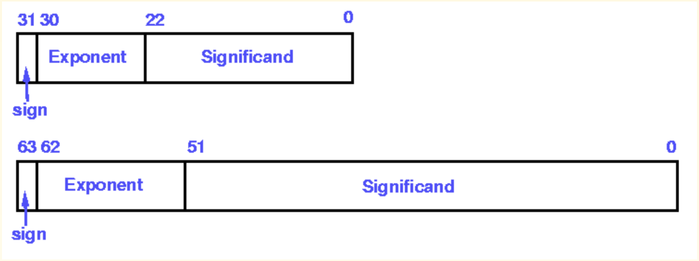
SRT Division

Divide A by B (n-bits)(view numbers as fractions between $1/2$ and 1)

1. If B has k leading 0s when expressed using n bits, shift all registers by k bits
2. For i = 0 to (n-1)
    1. If top 3 bits of P equal, set $q_i = 0$, shift (P,A) one bit left
    2. If top 3 bits of P not all equal, and P negative, set $q_i = -1$, (written as $\overline{1}$, shift (P,A) one bit left and add B
    3. Otherwise, set $q_i = 1$, shift (P,A) one bit left, subtract B
3. If the final remainder is negative, correct by adding B, correct quotient by subtracting 1; finally, shift remainder k bits right

Radix-4 SRT algorithm used in Pentium chip

Floating Point (IEEE 754-1985)



Single precision:

$N = (-1)^{\text{sign}} \times 1.\text{Significand} \times 2^{\text{Exponent}-127}, 1 \leq \text{Exponent} \leq 254$

$N = (-1)^{\text{sign}} \times 0.\text{Significand} \times 2^{\text{Exponent}-126}, \text{Exponent} = 0$

- "Hidden 1", Bias, Special values **NaN, ∞, $-\infty$**
- Rounds to nearest by default, but three other rounding modes
- "Halfway" result rounded to nearest even FP number
- "Denormal" numbers to represent results $< 1.0 \times 2^{Emin}$
- Sophisticated facilities for handling exceptions

Iterative Division

Newtons iteration: finding the 0 of a function

- Starting from a guess for the 0, approximate function by its tangent at the guess, form new guess based on where tangent has a 0

Goldschmidt's method

To compute a/b, iteratively, multiply both numerator and denominator by r where $r \times b = 1$

Find r iteratively

- Scale the problem so $b < 1$
- Set $x_0 = a$, $y_0 = b$ and write $b = 1 - \delta$ where $|\delta| < 1$
- If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$
- Next, pick $r_1 = 1 + \delta^2$, etc., and $y_i \to 1$

Used in the TI 8847 chip and AMD Athlon CPUs