## 9. Datapath Design

- Last module:
  - Adder circuits
  - Simple adders
  - Fast addition
- This module
  - Comparators
  - Shifters
  - Multi-input Adders
  - Multipliers



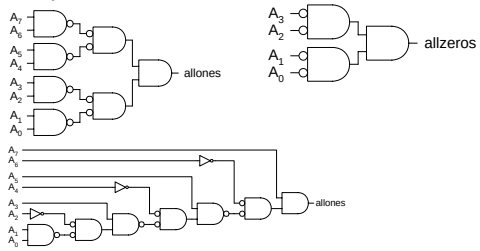
## Comparators

- 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

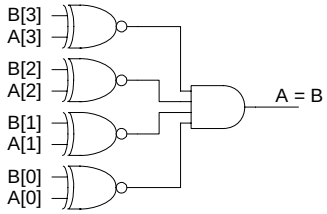
- 1's detector: N-input AND gate
- 0's detector: NOTs + 1's detector (N-input NOR)






## Equality Comparator

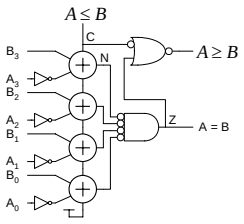
- Check if each bit is equal (XNOR, or "equality gate")
- 1's detect on bitwise equality






## 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 |
|---|---|---|
| $A = B$ | $Z$ | $Z$ |
| $A \neq B$ | $\overline{Z}$ | $\overline{Z}$ |
| $A < B$ | $\overline{C} + Z$ | $(\overline{N \oplus V}) + Z$ |
| $A > B$ | $\overline{C}$ | $(N \oplus V)$ |
| $A \leq B$ | $C$ | $(\overline{N \oplus V})$ |
| $A \geq B$ | $\overline{C} + Z$ | $(N \oplus V) + Z$ |

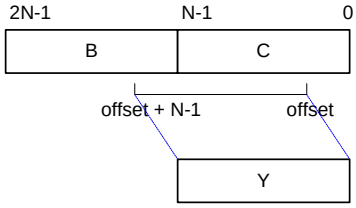
## Shifters

- Logical Shift:
  - Shifts number left or right and fills with 0's
    - 1011 LSR 1 = 0101  1011 LSL1 = 0110
- Arithmetic Shift:
  - Shifts number left or right. Rt shift sign extends
    - 1011 ASR1 = 1101  1011 ASL1 = 0110
- Barrel Shift (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 ($0 \le k < N$)






## 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 | $A_{N-1}…A_{N-1}$ (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 Funnel Shifter

- Optimize down to 2N-1 bit input

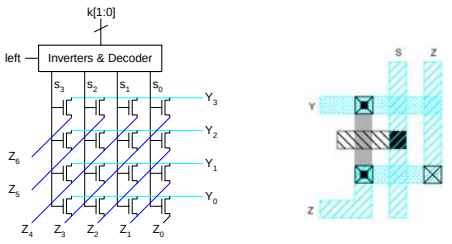
**Table 10.11 Simplified funnel shifter**

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

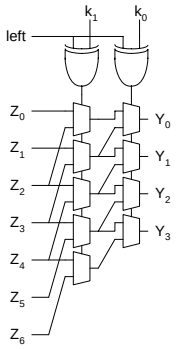
- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design ($V_t$ drops!)





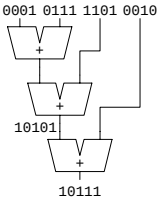

## Funnel Shifter Design 2

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



## Multi-input Adders

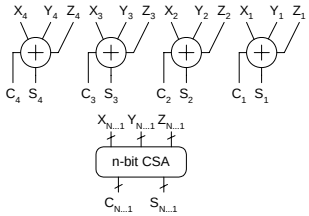
- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111
- Straightforward solution: k-1 N-input CPAs
  - Large and slow






## Carry Save Addition

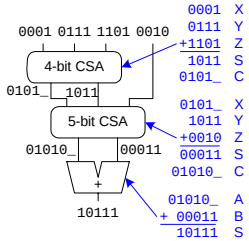
- Full adder sums 3 inputs, produces 2 outputs
  - Carry output has twice *weight* of sum output
- N full adders in parallel: *carry save adder*
  - Produce N sums and N carry outs






## CSA Application

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




```
0001  X
0111  Y
+1101 Z
1011  S
0101_ C

0101_ X
1011  Y
+0010 Z
00011 S
01010_ C

01010_ A
+ 00011 B
10111   S
```



## Multiplication

- Example:

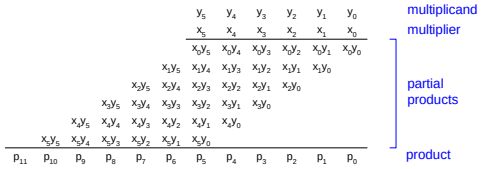
```
   1100 : 12_10      multiplicand
   0101 : 5_10       multiplier
   1100
  0000
 1100               } partial products
0000
00111100 : 60_10     product
```

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



## General Form

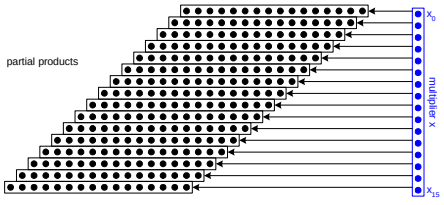
- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product: $P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$



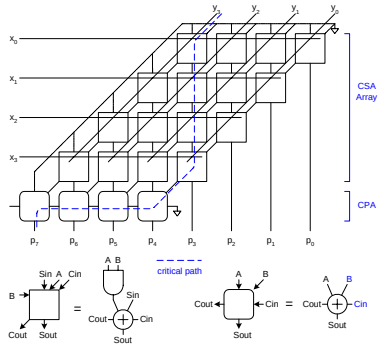



## Dot Diagram

- Each dot represents a bit



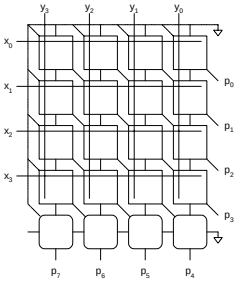
## Array Multiplier






## 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

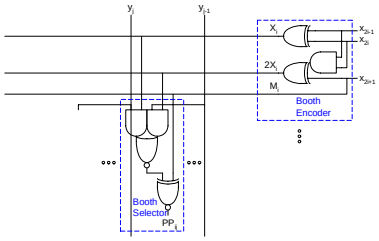
**Table 10.12** Radix-4 modified Booth encoding values

| 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 Hardware

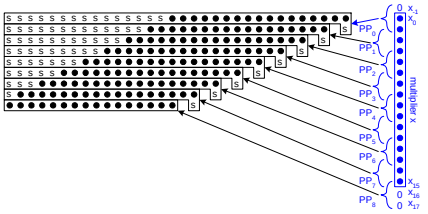
- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits





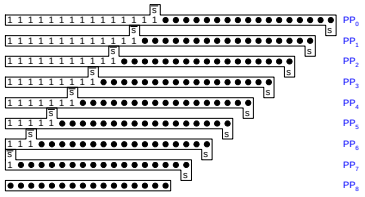

## Sign Extension

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



## Simplified Sign Extension

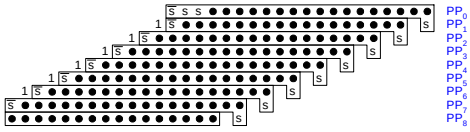
- Sign bits are either all 0's or all 1's
  - Note that all 0's is all 1's + 1 in proper column
  - Use this to reduce loading on MSB





## 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
- Serial Multiplication

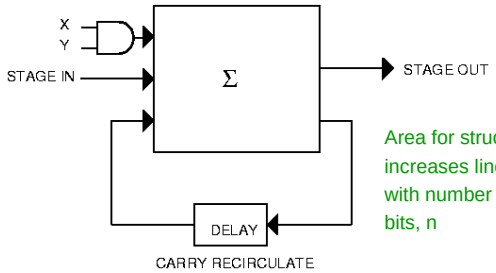


## Serial Multiplication

- Lower area at expense of speed
  - signal processing on bit streams
- Delay for n × 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 delay for n bit product



## Serial Multiplier Architecture

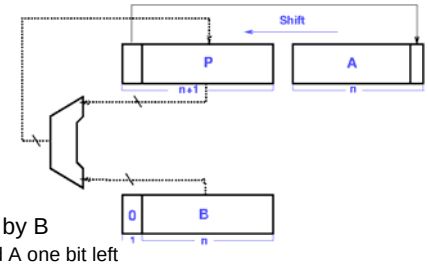



Area for structure increases linearly with number of bits, n

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

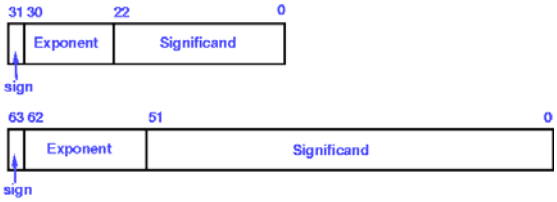
## SRT Division

- Divide A by B (n-bits)(view numbers as fractions between ½ 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 $\bar{1}$, 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
4. Radix-4 SRT algorithm used in Pentium chip



## Floating Point (IEEE 754-1985)




Special values NaN, ∞, -∞

Rounds to nearest by default, but three other rounding modes

"Halfway" result founded to nearest even floating point number

"Denormal" numbers to represent results of computations

$< 1.0 \times 2^{Emin}$

Sophisticated facilities for handling exceptions



## Iterative Division

- Newton's 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 with r.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 $y_i \rightarrow 1$
  - Used in the TI 8847 chip