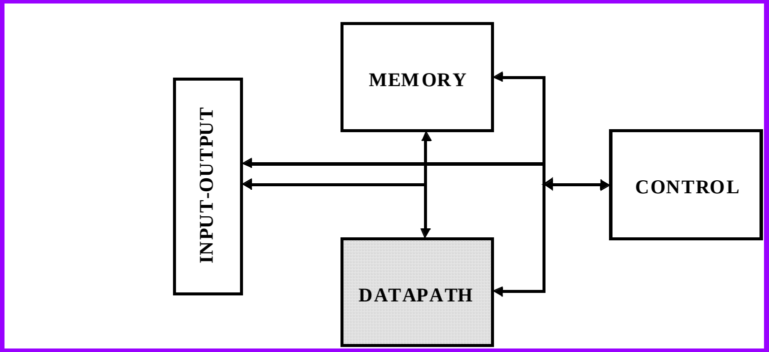
# Chapter 11. Arithmetic Building Blocks

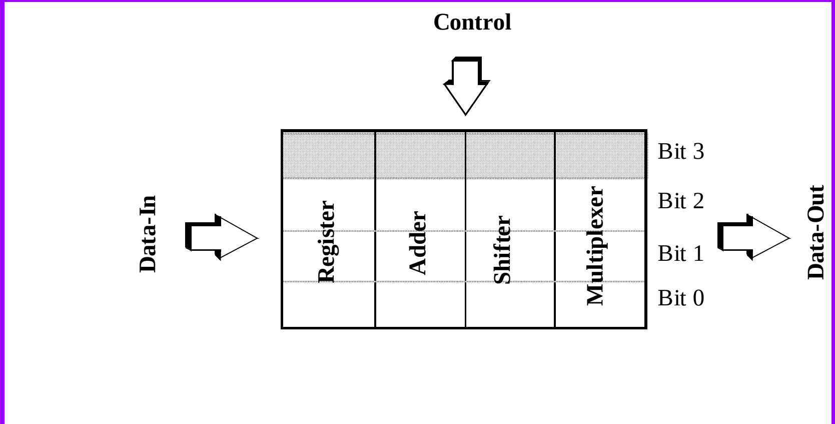
## A Generic Digital Processor



## Building Blocks for Digital Architectures

- ❑ **Arithmetic unit**
  - ✓ Bit-sliced datapath (adder, multiplier, shifter, comparator, etc.)
- ❑ **Memory**
  - ✓ RAM, ROM, Buffers, Shift registers
- ❑ **Control**
  - ✓ Finite state machine (PLA, random logic.)
  - ✓ Counters
- ❑ **Interconnect**
  - ✓ Switches
  - ✓ Arbiters
  - ✓ Bus

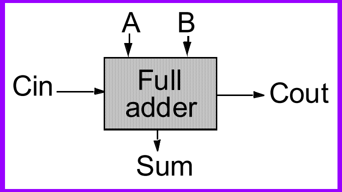
## Bit-Sliced Design



**Tile identical processing elements**

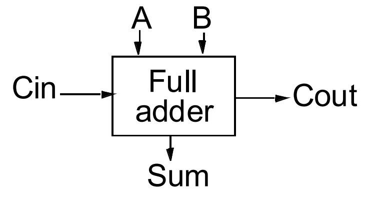
## Full-Adder

❑ Addition: most commonly used arithmetic operation, also speed-limiting element.



| $A$ | $B$ | $C_i$ | $S$ | $C_o$ | *Carry status* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | delete |
| 0 | 0 | 1 | 1 | 0 | delete |
| 0 | 1 | 0 | 1 | 0 | propagate |
| 0 | 1 | 1 | 0 | 1 | propagate |
| 1 | 0 | 0 | 1 | 0 | propagate |
| 1 | 0 | 1 | 0 | 1 | propagate |
| 1 | 1 | 0 | 0 | 1 | generate |
| 1 | 1 | 1 | 1 | 1 | generate |

## The Binary Adder



$$S = A \oplus B \oplus C_i$$

$$= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC$$

$$C_o = AB + BC_i + AC_i$$

## Express Sum and Carry as a function of P, G, D

❑ Define 3 new variable which only depend on A, B:

✓G=1 (D=1) ensures that a carry bit will be generated (deleted) at Co independent of Ci.

✓P=1 ensures that an incoming carry will propagate to Co.

***Generate (G) = AB***

***Propagate (P) = $A \oplus B$***

***Delete = $\bar{A}\ \bar{B}$***

✓ Sometimes P is also taken as P=A+B

❑ S and Co can be rewritten as functions of P and G (or D):

$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

## The Ripple-Carry Adder

❑ N-bit ripple-carry adder: cascading N full-adder circuits in series, connecting $C_{o,k-1}$ to $C_{i,k}$ (k=1~N-1), and set $C_{i,0}$=0.

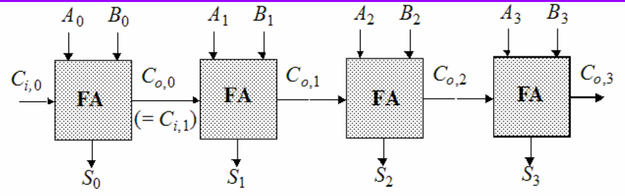
❑ Carry-bit “ripples” from one stage to the other

❑ Delay depends on input patterns: some input patterns has no rippling effect at all, while for others the carry has to ripple all the way from LSB to MSB (worst-case delay).

❑ Worst-case delay: $t_{adder} \approx (N-1)t_{carry} + t_{sum}$

where $t_{carry}$ and $t_{sum}$: propagation delays from $C_i$ to $C_o$ and S.

❑ Goal: make the fastest possible carry path circuit

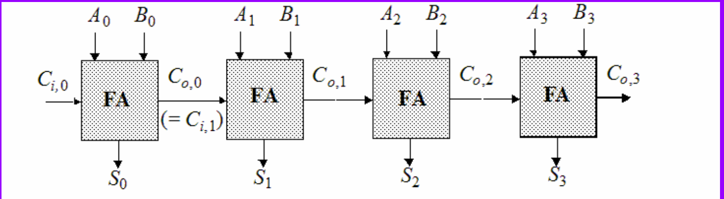


N-bit Ripple Carry Adder

**Worst case delay linear with the number of bits**: $t_d$ = O(N)

## The Ripple-Carry Adder

- ❑ Propagation delay of ripple-carry adder is linearly proportional to N.
- ❑ For a fast ripple-carry adder, it's far more important to optimize $t_{carry}$ than $t_{sum}$, since the latter has only a minor influence on the total value of $t_{adder}$.



N-bit Ripple Carry Adder

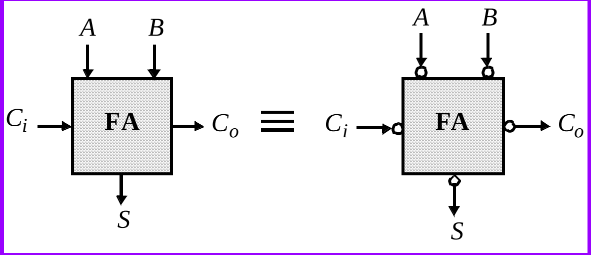
**Worst case delay linear with the number of bits**: $t_d = O(N)$

## Inversion Property

- ❑ Inversion property of full adder: inverting all inputs to a full adder results in inverted values for all outputs.

$$\overline{S}(A,B,C_i) = S(\overline{A},\overline{B},\overline{C_i})$$
$$\overline{C_o}(A,B,C_i) = C_o(\overline{A},\overline{B},\overline{C_i})$$

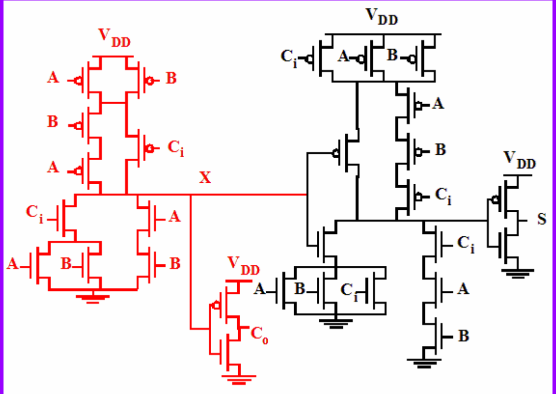
- ❑ Inversion property is useful for optimizing speed of ripple-carry adder
- ❑ The following two circuits are identical



## Complimentary Static CMOS Full Adder

- ❑ Static CMOS full adder is implemented as below :

  $C_o = AB + BC_i + AC_i$, $S = ABC_i + \overline{C_o}(A+B+C_i)$
- ❑ Design tricks: Ci are placed as close as possible to output→transistors on critical path should be placed as close as possible to gate output to reduce delay.



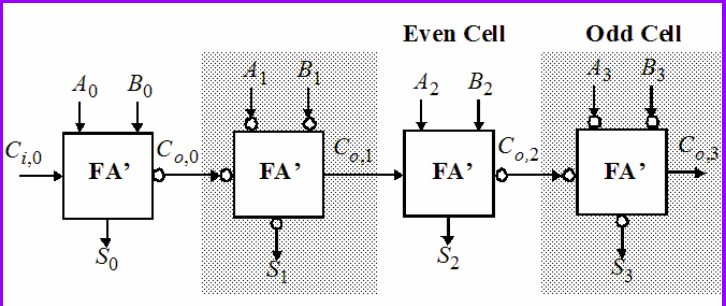
Static CMOS full adder: 28 transistors

## Complimentary Static CMOS Full Adder

- ❑ Static CMOS full adder has large area with slow speed
- ✓ Long chains of series PMOS in carry and sum circuits
- ✓ load capacitance of Co is large (consists of 2 diffusion and 6 gate capacitances plus wiring capacitance) (Co connects to Ci of next stage which takes 6 Ci inputs)
- ✓ carry circuit requires 2 inverting stages per bit (!Co plus inverter to get Co)

## Minimize Critical Path by Reducing Inverting Stages

❑ By cascading CMOS full adder and its inversion equivalent alternately, the extra inverters in carry path to get $C_o$ from $!C_o$ can be eliminated → worst case delay of adder is reduced.



**Exploit Inversion Property**

**Note: need 2 different types of cells**

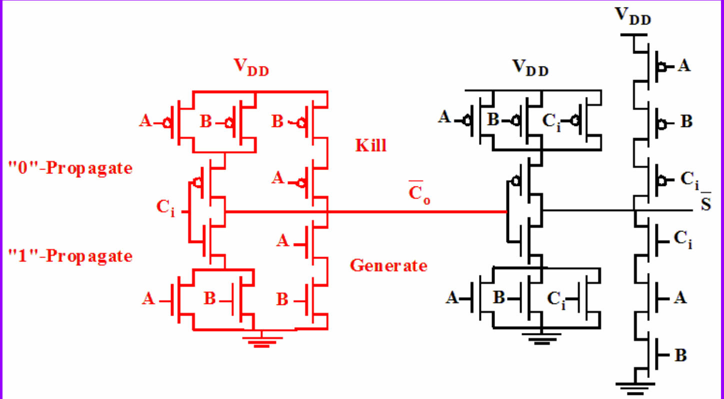
## The Better Structure: Mirror Adder

❑ Mirror adder: utilize propagate/generate/delete functions

✓ G=AB, D=$\overline{A}\,\overline{B}$, P=A+B

✓ When D=1 or G=1, $\overline{C_o}$=1 or 0.

✓ When P=1, $C_i$ is propagated (in inverted format) to $\overline{C_o}$.



Mirror adder (or symmetrical adder): 24 transistors

## The Mirror Adder

❑ The NMOS and PMOS chains are completely symmetrical. This guarantees identical rising and falling transitions if the NMOS and PMOS devices are properly sized. A maximum of two series transistors can be observed in the carry-generation circuitry.

❑ When laying out the cell, the most critical issue is the minimization of the capacitance at node $C_o$. The reduction of the diffusion capacitances is particularly important.

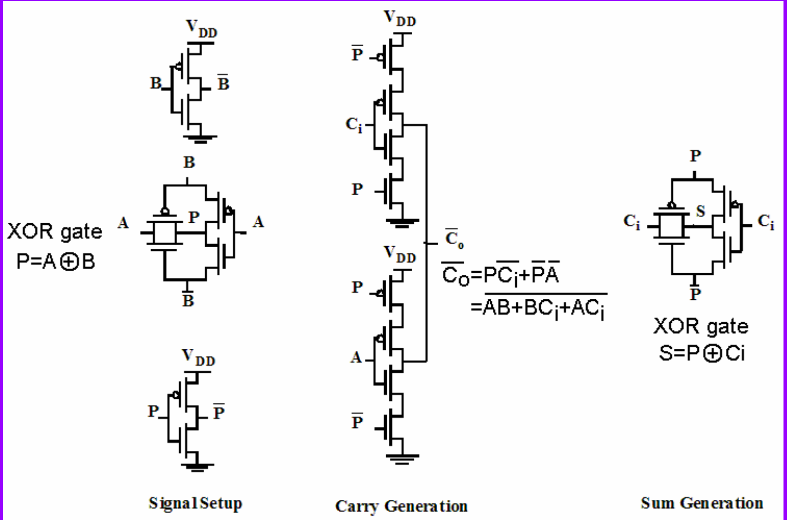
❑ The capacitance at node $C_o$ is composed of four diffusion capacitances, two internal gate capacitances, and six gate capacitances in the connecting adder cell .

❑ Transistors connected to $C_i$ are placed closest to output.

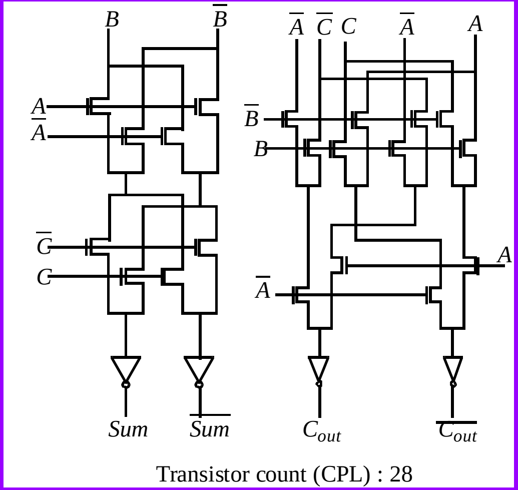
❑ Only the transistors in the carry stage have to be optimized for optimal speed. All transistors in the sum stage can be minimal size.

## Quasi-Clocked Adder

❑ Transmission-gate (TG) based quasi-clocked adder circuit

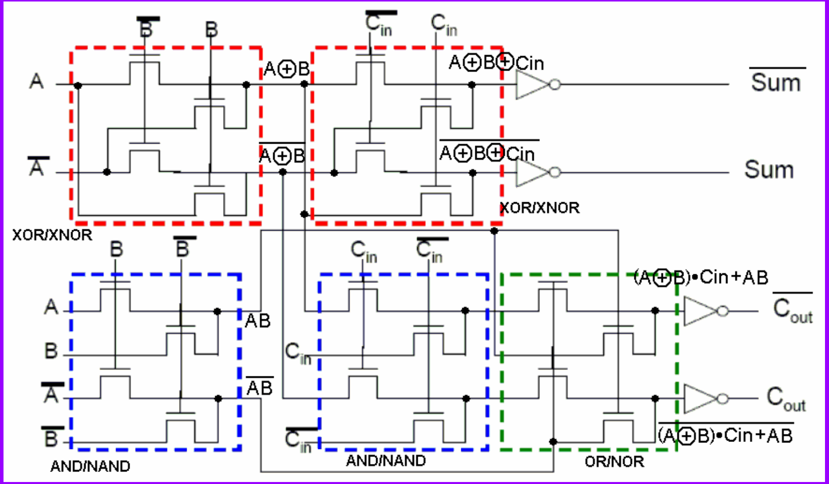


## NMOS-Only Pass Transistor Logic



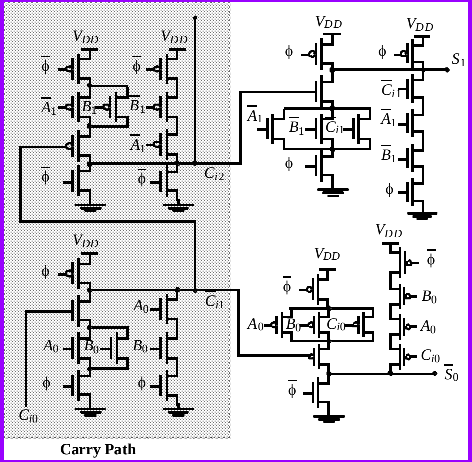
## CPL Full Adder

- ❑ 20+4×2=28 transistors
- ❑ Problems with (more than one) threshold drops due to chaining CPL blocks



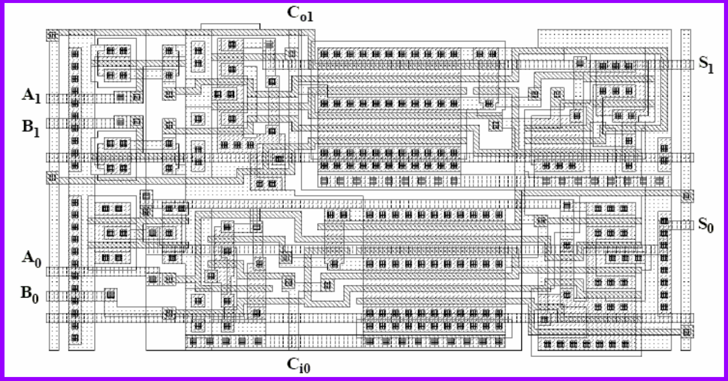
## NP-CMOS Adder

- ❑ Dynamic np-CMOS adder: only 17 transistors ignoring extra inverters required for input/output signals
- ❑ The alternating even and odd carry stages are realized using NMOS and PMOS networks respectively



## NP-CMOS Adder

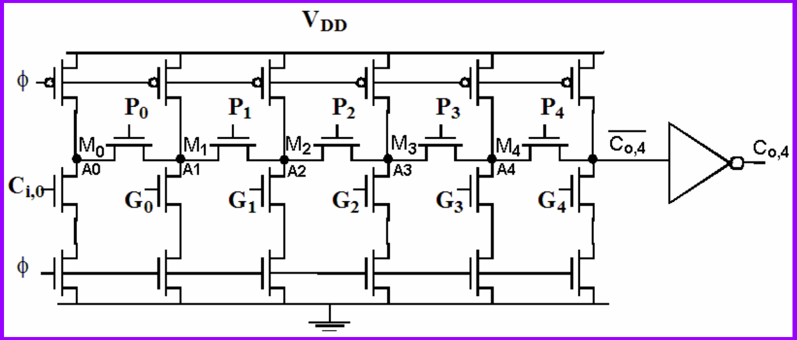
- ❑ Reduced capacitance of dynamic circuits results in substantial speed-up over static implementation
- ❑ Load capacitance on carry bit includes 3 diffusion and 4 gate capacitances.



Layout of np-CMOS full adder

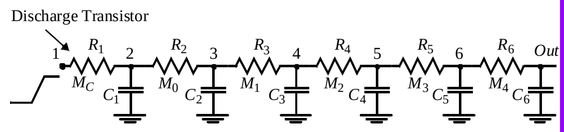
## Manchester Carry Chain

- ❑ Manchester carry-chain adder: uses a cascade of pass-transistors to implement carry chain.
- ❑ In precharge phase ($\Phi$=0), all intermediate nodes A0~A4 of pass-transistor chain are precharged to “1”.
- ❑ In evaluation phase ($\Phi$=1), $M_k$ node is discharged to 0 when incoming carry=1 and propagate signal $P_k$=1, or when generate signal for stage k ($G_k$) is 1.

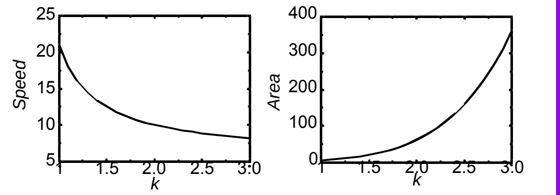


## Sizing Manchester Carry Chain

- ❑ Carry-chain is a distributed RC-network→$t_d$=O($N^2$)
- ❑ Reducing delay of carry-chain:
- ✓ insert signal-buffering inverters with optimum stages/sizing
- ✓ Sizing transistors progressively ($I_{M0}$>$I_{M1}$>…>$I_{M4}$, thus size of $M_4$ to $M_0$ should be increased progressively.)



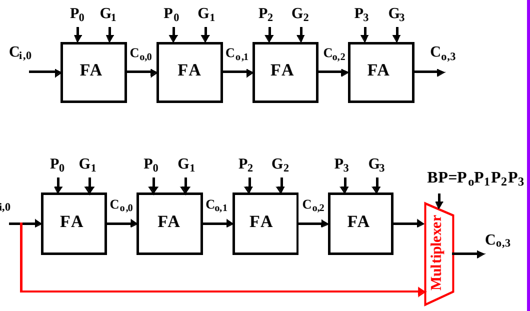
$$t_p = 0.69\sum_{i=1}^{N} C_i\left(\sum_{j=1}^{i} R_j\right)$$



Speed (normalized by 0.69*RC*) Area (in minimum size devices)

## Carry-Bypass Adder

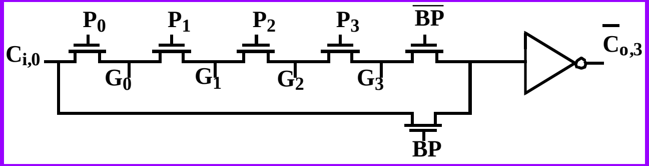
- ❑ For ripple-carry adder, if ($P_0P_1P_2P_3$=1) then $C_{o,3}$=$C_{i,0}$ else either DELETE or GENERATE occurred.
- ❑ Thus if ($P_0P_1P_2P_3$=1), we can directly bypass carry-in $C_{i0}$ to $C_{o,3}$ to reduce the delay caused by carry propagation. → carry-bypass adder.



Idea: If (P0 and P1 and P2 and P3 = 1) then $C_{o3}$ = $C_0$, else “kill” or “generate”.

## Manchester-Carry Implementation

- ❑ Manchester-carry implementation: either carry propagates through bypass path, or carry is generated somewhere in the chain
- ❑ In both cases, delay is smaller than normal ripple configuration.
- ❑ Area overhead due to adding bypass path: 10~20%.



## Delay of Carry-Bypass Adder

❑ Assume total adder is divided in (N/M) equal by-pass stages, each contains M bits. Total propagation time

$t_p \approx t_{setup} + Mt_{carry} + (N/M-1)t_{bypass} + Mt_{carry} + t_{sum}$

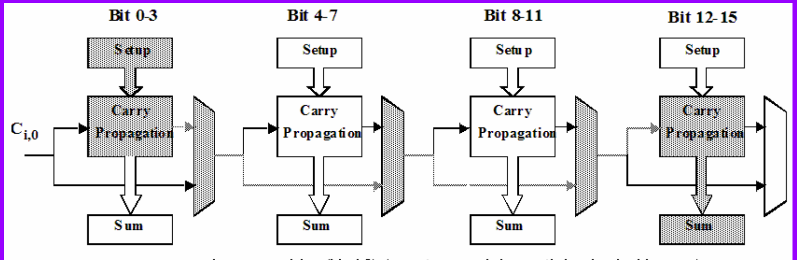
where: $t_{setup}$: fixed overhead time to create G and P signals,

$t_{carry}$: propagation delay through a single bit. The worst-case carry-propagation delay through a single stage is $Mt_{carry}$

$t_{bypass}$: propagation delay through bypass MUX of a single stage

$t_{sum}$: time to generate sum of final stage.

❑ Carry-bypass adder delay $t_p = O(N)$



carry-bypass adder (N=16) (worst-case delay path is shaded in gray)

## Carry Ripple versus Carry Bypass

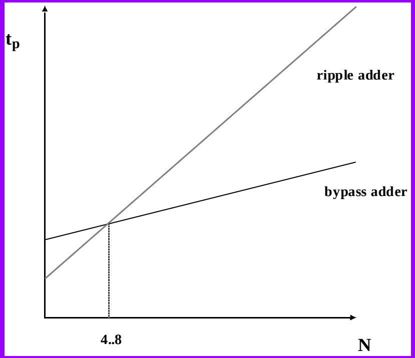
❑ Propagation delay of carry ripple adder vs carry-bypass adder

✓ Difference is substantial for larger adders (large N)

✓ Ripple carry adder is actually faster for smaller N

✓ Overhead of extra bypass MUX makes bypass structure not interesting for small N

✓ Crossover point depends on technology considerations and is normally situated between 4 and 8 bits



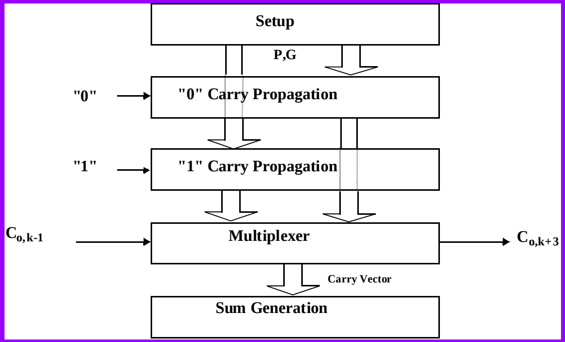
## Linear Carry-Select Adder

❑ Ripple carry adder: every full adder cell waits for incoming carry before outgoing carry can be generated

❑ To avoid the waiting, anticipate both possible values (0 and 1) of carry input and evaluate result for both cases in advance

❑ Once the real value of incoming carry is known, the correct output is selected with a simple MUX → carry-select adder

❑ Hardware overhead: 30% (due to additional carry path and a MUX



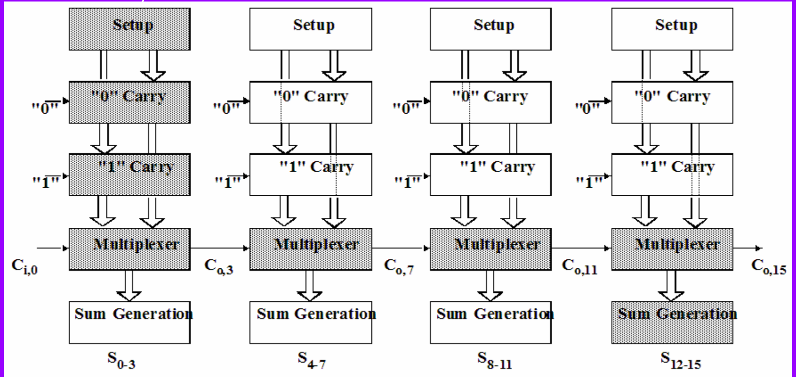
## Carry Select Adder: Critical Path

❑ Assume total adder is divided in (N/M) equal by-pass stages, each contains M bits. Total propagation time

$t_{add} = t_{setup} + Mt_{carry} + (N/M)t_{mux} + t_{sum}$

where $t_{carry}$: carry delay through a single bit,

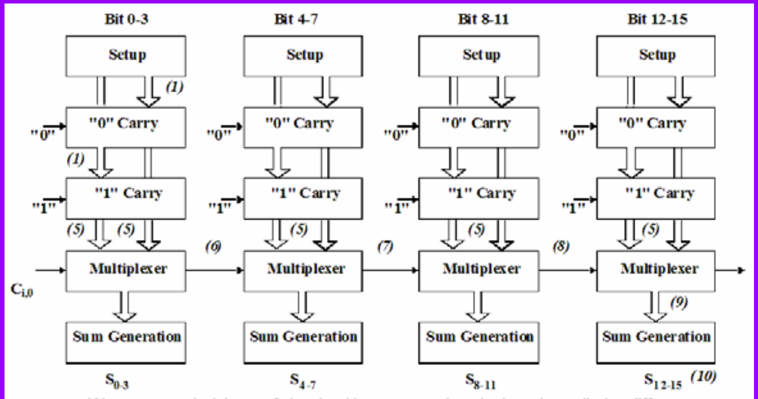
$Mt_{carry}$: carry delay through a single block



16-bit linear carry-select adder (critical path is shaded in gray)

## Linear Carry Select

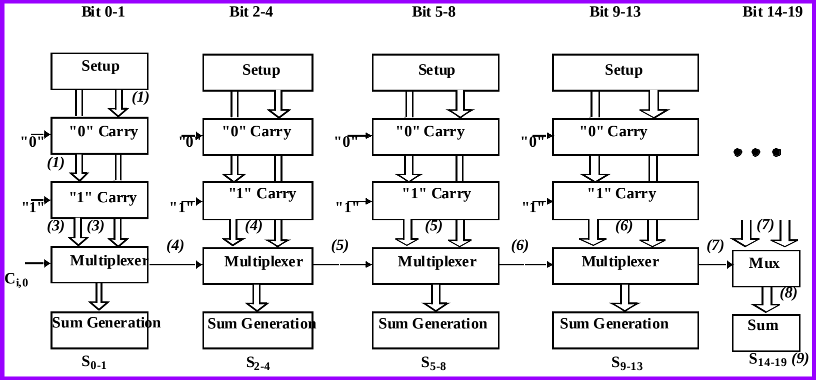
- ❑ Assume full-adder and MUX cells have 1 unit delay each
- ❑ A major mismatch between the signal arrival times is observed in the MUX gate of last adder stage
- ❑ Total delay can be reduced if we equalize the delay through both inputs of the MUX gate → square-root carry select adder



Worst-case arrival times of signals with respect to time the input is applied at different network nodes of linear carry-select adder

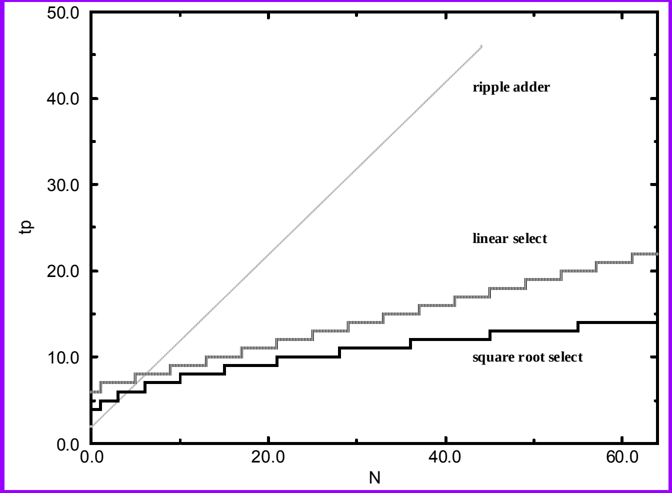
## Square Root Carry Select

- ❑ To equalize the inputs to MUX gates of all stages: progressively adding more bits to subsequent stages in the adder, requiring more time for the generation of carrysignals
- ❑ E.g. 1st stage add 2 bits, 2nd stage add 3 bits, 3rd stage add 4 bits, etc.



$$t_{add} = t_{setup} + Mt_{carry} + \left(\sqrt{2N}\right)t_{mux} + t_{sum} \propto \sqrt{N}$$

## Adder Delays - Comparison



## Carry-Lookahead Adder - Basic Idea

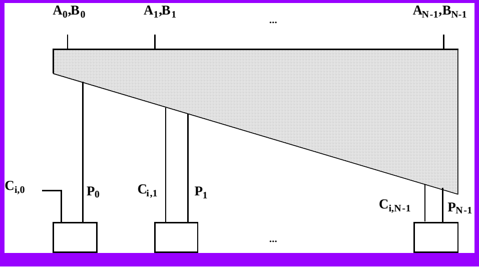
- ❑ Carry-lookahead adder: avoid rippling effect of carry in both carry-bypass and carry-select adders
- ❑ For each bit position in an N-bit adder:

$C_{o,k}=f(A_k, B_k, C_{o,k-1})=G_k+P_kC_{o,k-1}$

Dependency between $C_{o,k}$, $C_{o,k-1}$ can be avoided by expanding $C_{o,k-1}$: $C_{o,k}=G_k+P_k(G_{k-1}+P_{k-1}C_{o,k-2})$

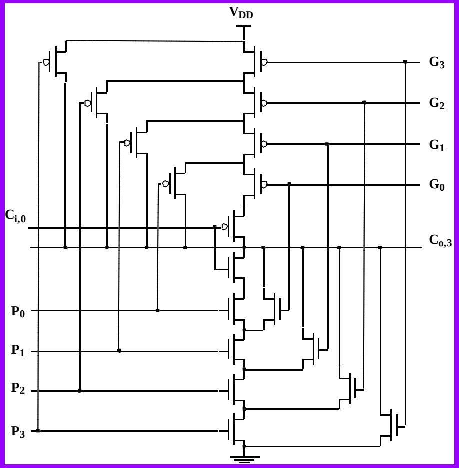
Finally: $C_{o,k}=G_k+P_k(G_{k-1}+P_{k-1}(\ldots+P_1(G_0+P_0C_{i,0})))$ $(C_{i,0}=0)$

- ❑ For every bit, carry and sum outputs are independent of previous bit. The ripple effect has been eliminated. → addition time should be independent of N.



## Carry Lookahead Adder: Topology

- ❑ Example carry lookahead adder for N=4
- ✓ It contains some hidden dependencies
- ✓ Large fan-in makes it prohibitively slow for large N



## Logarithmic Look-Ahead Adder

- ❑ Consider a generic associative operator – dot operation (•)
- ✓ associatiivity property: (a•b)•c=a•(b•c)
- ✓ Dot operation of N arguments can be executed with critical path of $(\log_2 N)t_•$  ($t_•$: propagation delay of dot operation)
- ❑ Ex: For N=8, both designs have same number of operators.
- ✓ Linear (ripple) topology: critical path delay=$7t_•$
- ✓ Logarithmic (tree-like fashion): critical path delay=$3t_•$

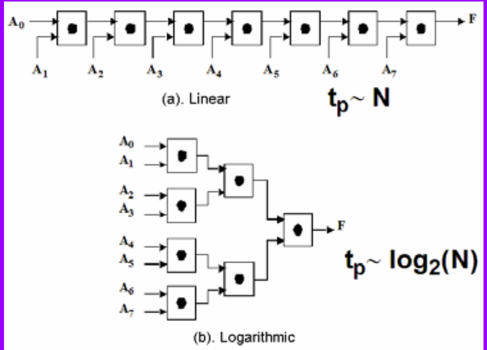
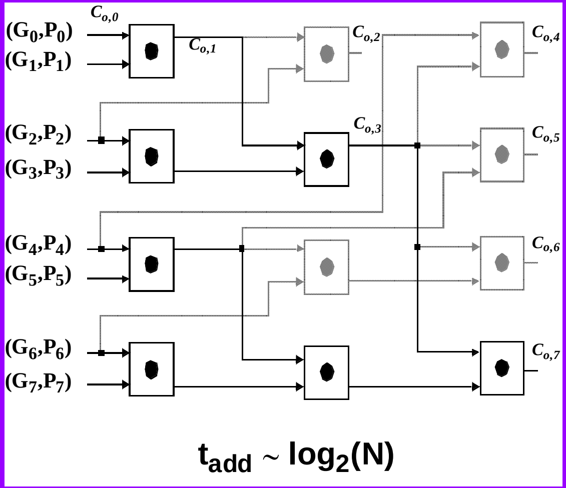


Figure. Configurations of associative operators

## Logarithmic Lookahead Adder: Brent-Kung Adder

- ❑ Addition operation is associative



## The Binary Multiplication

- ❑ Multiplication: expensive and slow operation
- ❑ Multipliers are in effect complex adder arrays
- ❑ Consider 2 unsigned binary numbers X(M bits) and Y(N bits)

$$Z = X \times Y = \sum_{k=0}^{M+N-1} Z_k 2^k$$

$$= \left(\sum_{i=0}^{M-1} X_i 2^i\right)\left(\sum_{j=0}^{N-1} Y_j 2^j\right) \quad \text{with} \quad X = \sum_{i=0}^{M-1} X_i 2^i$$

$$= \sum_{i=0}^{M-1}\left(\sum_{j=0}^{N-1} X_i Y_j 2^{i+j}\right) \qquad Y = \sum_{j=0}^{N-1} Y_j 2^j$$

## The Binary Multiplication

❑ Common implementation of binary multiplier: similar to manually computing a multiplication

✓ Multiplicand is consecutively multiplied (AND operation) with every bit of multiplier → partial products

✓ Intermediate results are added after proper shifting

```
          1 0 1 0 1 0      X:Multiplicand (M bit)
  ×           1 0 1 1      Y:Multiplier (N bit)
  ------------------
          1 0 1 0 1 0  ─┐  AND operation for multiplication of each bit
        1 0 1 0 1 0     │
      0 0 0 0 0 0       │  Partial Products
  + 1 0 1 0 1 0        ─┘
  ------------------
    1 1 1 0 0 1 1 1 0      Result
```

## The Array Multiplier

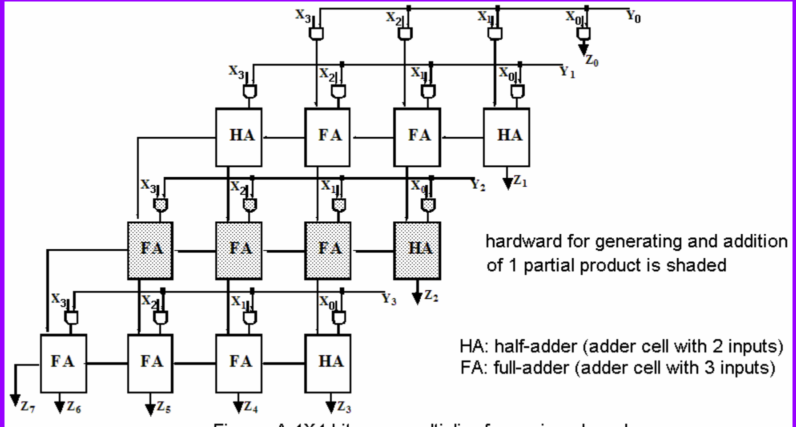
❑ Array multiplier:

✓ Generating N partial products needs N M-bit AND gates

✓ Requires (N-1) M-bit adders to add N partial results

✓ Shifting of partial results: simple routing (not active logic)

✓ Can be compacted into a rectangle →efficient layout



X3 X2 X1 X0 Y0 Z0
X3 X2 X1 X0 Y1
HA FA FA HA
X3 X2 X1 X0 Y2 Z1
FA FA FA HA
X3 X2 X1 X0 Y3 Z2
FA FA FA HA
Z7 Z6 Z5 Z4 Z3
hardward for generating and addition of 1 partial product is shaded
HA: half-adder (adder cell with 2 inputs)
FA: full-adder (adder cell with 3 inputs)

Figure: A 4X4 bit-array multiplier for unsigned numbers

## The M×N Array Multiplier — Critical Path

❑ Propagation delay of M×N array multiplier

✓ Partial sum adders: ripple-carry structures

✓ There are many almost identical-length paths

✓ Propagation delay $t_{mult} \approx [(M-1)+(N-2)]t_{carry}+(N-1)t_{sum}+t_{and}$

$t_{carry}$: delay between $C_{in}$ and $C_{out}$

$t_{sum}$: delay between $C_{in}$ and Sum of full adder

$t_{and}$: delay of AND gate

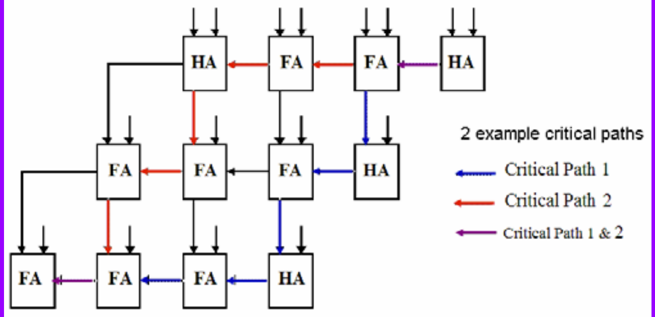


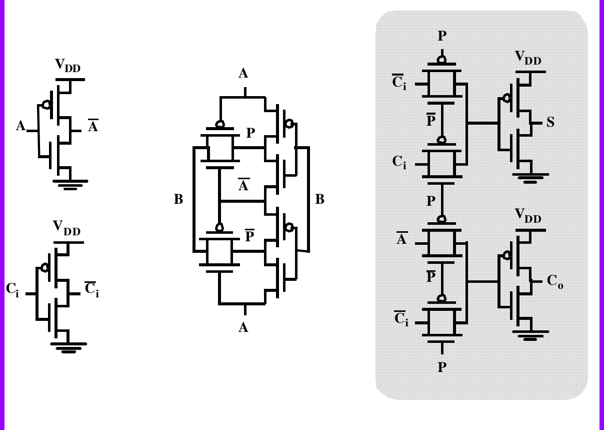
Figure: simplified ripple-carry based 4X4 multiplier

$$t_{mult} \approx [(M-1)+(N-2)]t_{carry}+(N-1)t_{sum}+(N-1)t_{and}$$

## Adder Cells in Array Multiplier

❑ Minimizing delay of array multiplier requires minimization of both $t_{carry}$ and $t_{sum}$ → It helps if $t_{carry}=t_{sum}$

❑ A full adder with comparable $t_{sum}$ and $t_{carry}$ delays using TG EXOR (24 transistors)



Identical Delays for Carry and Sum

## Carry-Save Multiplier

❑ Carry-save multiplier: more efficient

✓ Fact: multiplication result does not change when output carry bits are passed diagonally downwards instead of to the right

✓ An extra M-bit full-adder is added (vector-merging adder) to generate final result

✓carry bits are not immediately added, but are rather “saved” for next adder stage

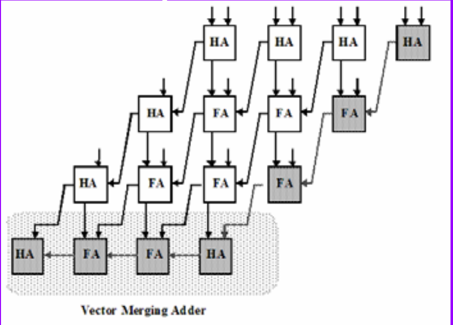
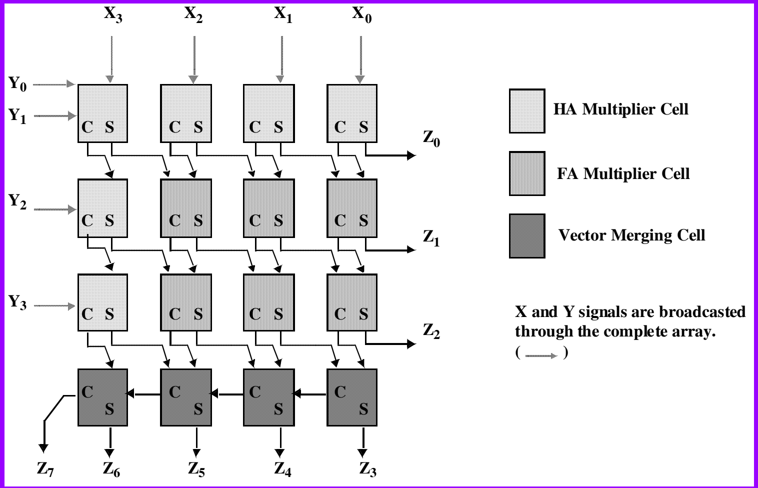


Figure. A 4X4 carry-save multiplier, the critical path is highlightedin gray

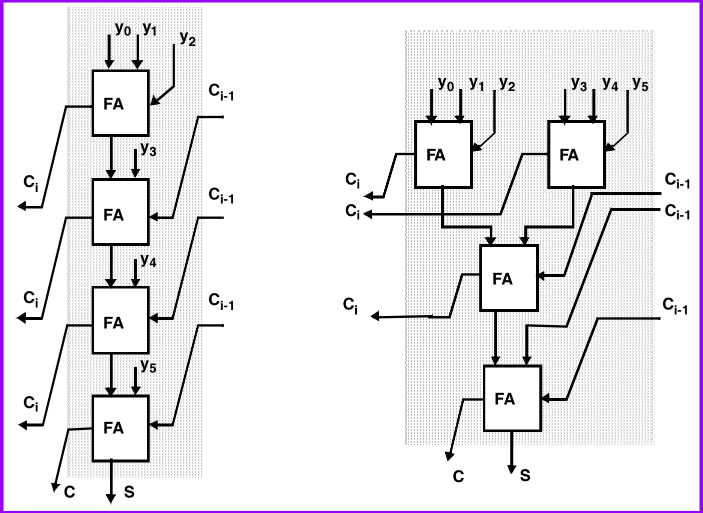
$$t_{mult} = (N-1)t_{carry} + (N-1)t_{and} + t_{merge}$$

## Multiplier Floorplan

❑ Rectangle floorplan of carry-save multiplier


$X_3$ $X_2$ $X_1$ $X_0$
$Y_0$ $Y_1$ $Y_2$ $Y_3$
C S
$Z_0$ $Z_1$ $Z_2$ $Z_3$ $Z_4$ $Z_5$ $Z_6$ $Z_7$
HA Multiplier Cell
FA Multiplier Cell
Vector Merging Cell
X and Y signals are broadcasted through the complete array. ( ⟶ )

## Wallace-Tree Multiplier



## Multipliers —Summary

❑ Optimization goals different from binary adder

❑ Once again: identify critical path

❑ Other possible techniques

✓ Logarithmic versus linear (Wallace tree multiplier)

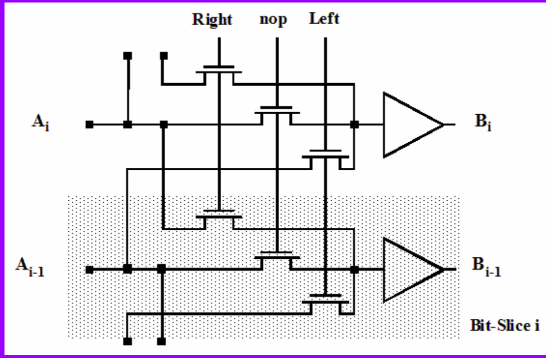
✓ Data encoding (Booth)

✓ Pipelining

❑ First Glimpse at System Level Optimization

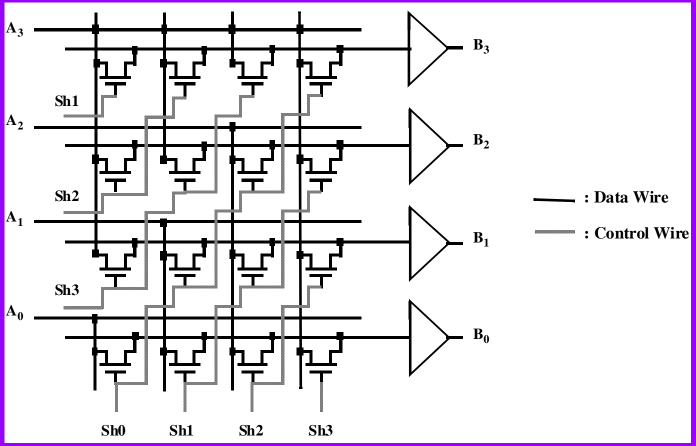
## The Binary Shifter

- ❑ Shifter: used in floating-point units, scalers, multiplications by constant numbers.
- ❑ It can be implemented by appropriate signal wiring
- ❑ A 1-bit left-right shifter: depending on control signals, input word is either shifted left or right or remain unchanged.
- ❑ N-bit shifters can be built by cascading 1-bit shifters, but very slow for large N



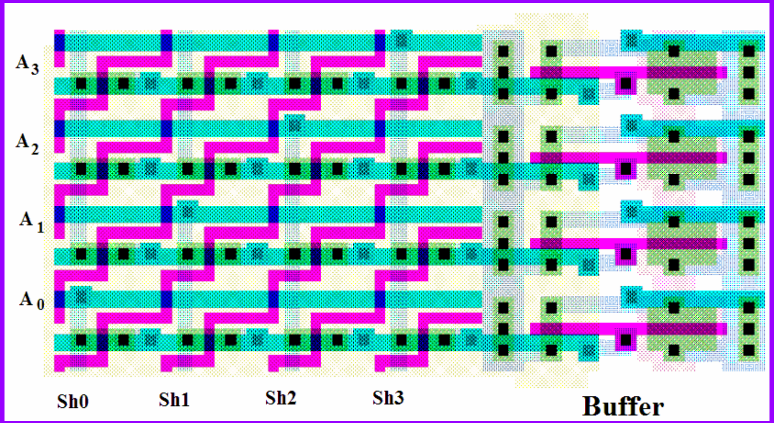
Right=1 ⇒ $B_{i-1} = A_i$
Left=1⇒ $B_i = A_{i-1}$
nop=1 ⇒ $B_i = A_i$.

## The Barrel Shifter



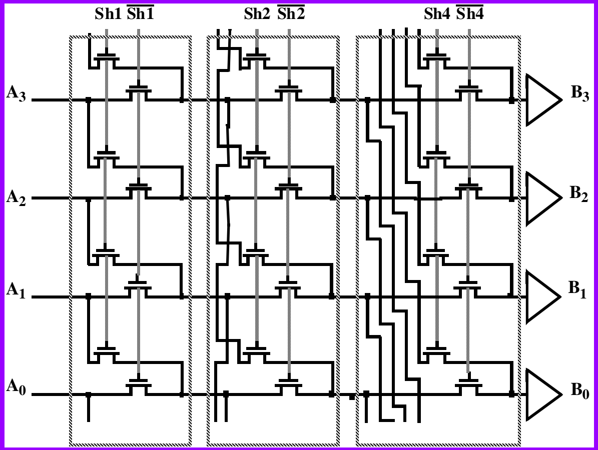
**Area Dominated by Wiring**

## 4x4 barrel shifter



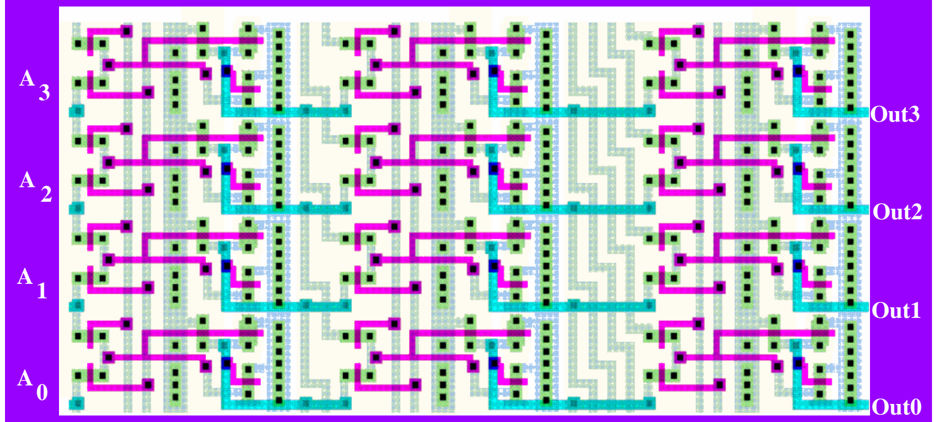
$\text{Width}_{\text{barrel}} \sim 2\, p_m\, M$

## Logarithmic Shifter



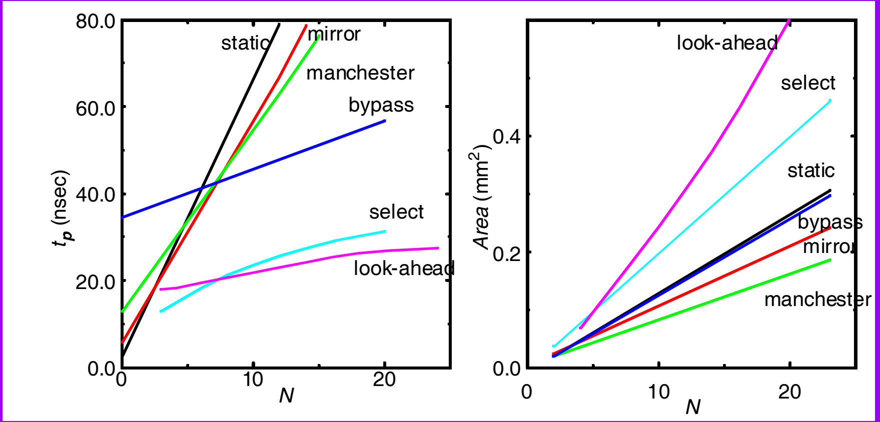
Sh1 $\overline{\text{Sh1}}$
Sh2 $\overline{\text{Sh2}}$
Sh4 $\overline{\text{Sh4}}$
$A_3$
$A_2$
$A_1$
$A_0$
$B_3$
$B_2$
$B_1$
$B_0$

## 0-7 bit Logarithmic Shifter

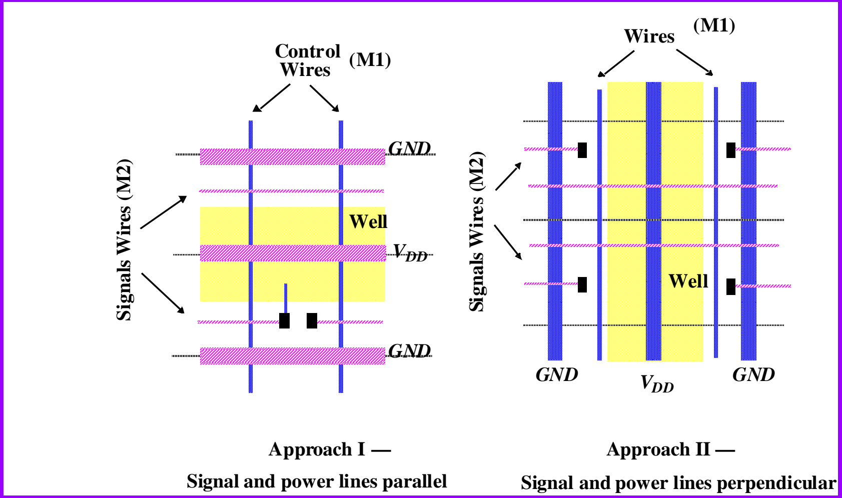


$$width_{\log} \approx p_m\left(2K + \left(1 + 2 + \ldots + 2^{K-1}\right)\right) = p_m\left(2^K + 2K - 1\right)$$

## Design as a Trade-Off



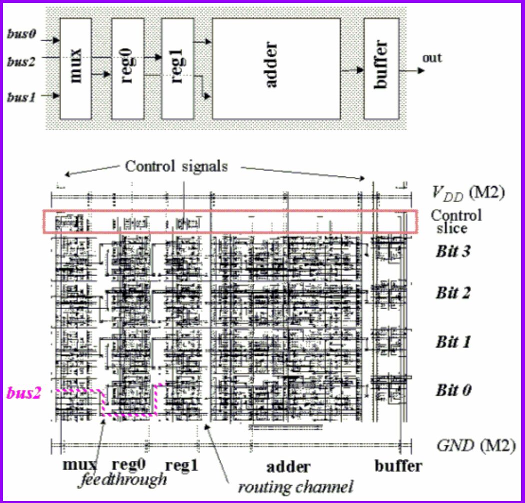
## Layout Strategies for Bit-Sliced Datapaths



Approach I —
Signal and power lines parallel

Approach II —
Signal and power lines perpendicular

## Layout of Bit-sliced Datapaths



## Layout of Bit-sliced Datapaths

(a) Datapath without feedthroughs and without pitch matching (area = 4.2 $mm^2$).

(b) Adding feedthroughs (area = 3.2 $mm^2$)

(c) Equalizing the cell height reduces the area to 2.2 $mm^2$.