University of Southern California

Viterbi School of Engineering

EE577A

VLSI System Design

Data Path Design

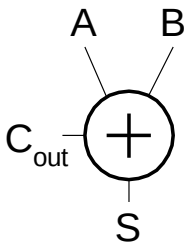
References: Professor Massoud Pedram's lecture slides, books listed in the syllabus, and online resources

Shahin Nazarian

Spring 2013

Single-Bit Addition

Half Adder

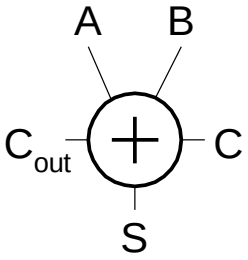


$$S = A \oplus B$$

$$C_{out} = A \cdot B$$

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder



$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A,B,C) = AB + (A+B)C$$
$$= AB + (A \oplus B)C$$

A	B	C	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

PGK

- For a full adder, define what happens to carries
  - Generate: C_{out} = 1 independent of C

$$G = A \cdot B$$

  - Propagate: C_{out} = C

$$P = A \oplus B$$

  - Kill: C_{out} = 0 independent of C

$$K = \overline{A} \cdot \overline{B}$$

  - Note that

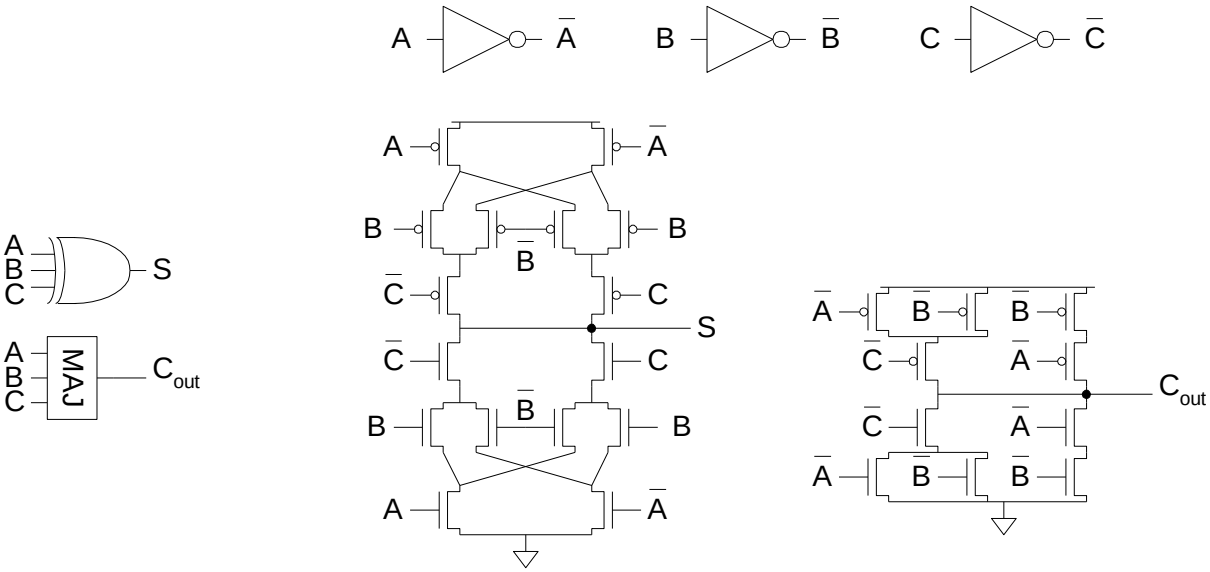
$$\overline{P} = A.B + \overline{A}.\overline{B} = G + K$$

Full Adder Design I

- **Brute force implementation from equations**

$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$

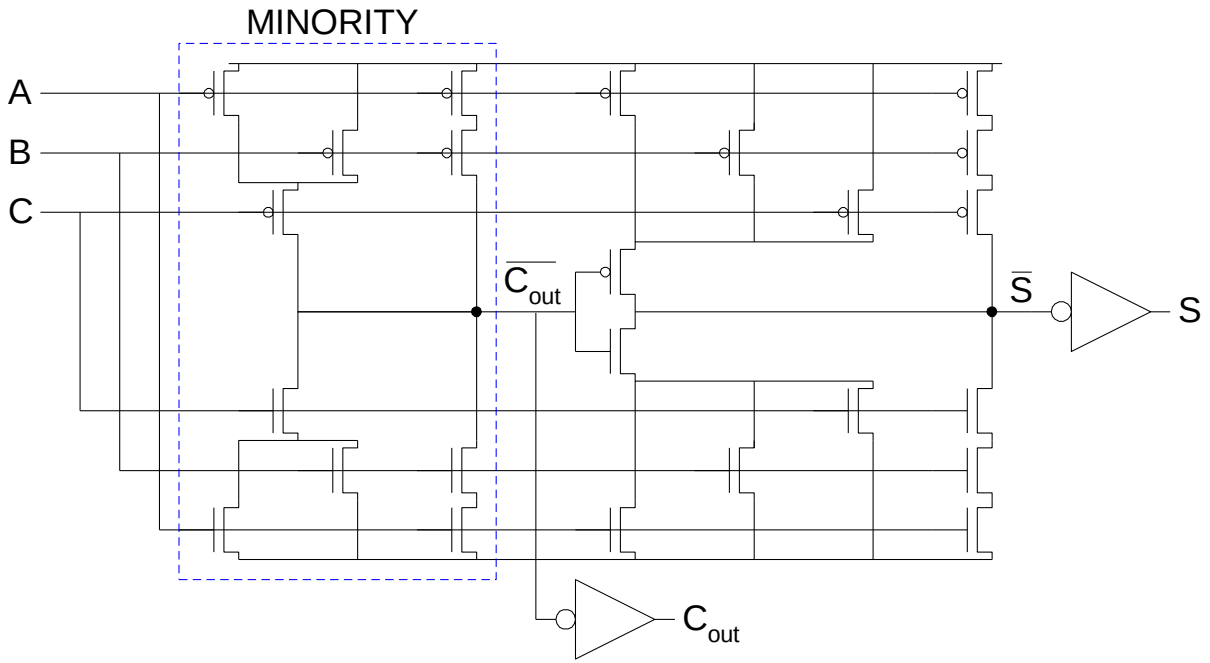


Full Adder Design II

- Factor S in terms of C_{out}

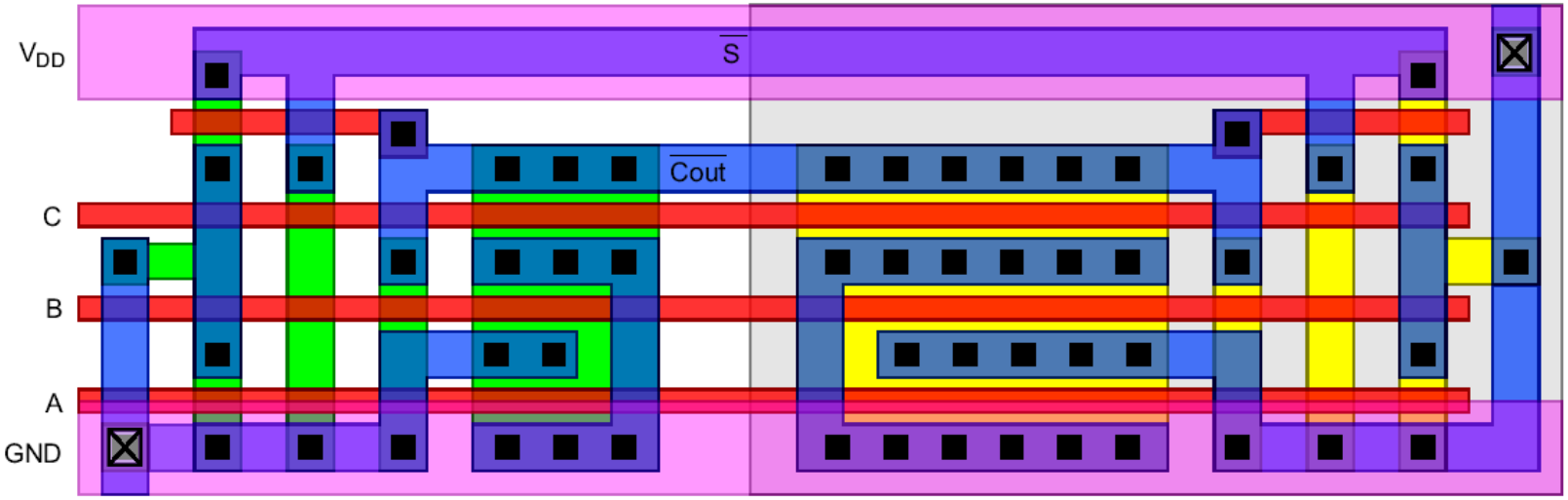
$$S = ABC + (A + B + C)\overline{C_{out}}$$

- Critical path is usually C to C_{out} in a ripple adder



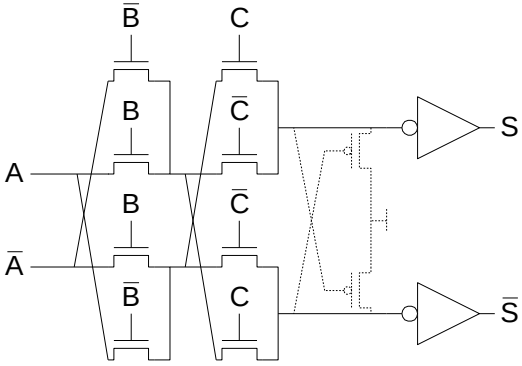
Layout

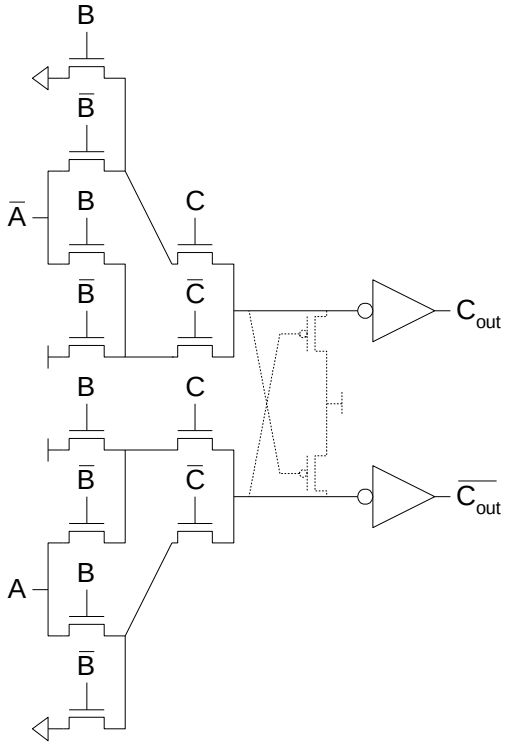
- Clever layout circumvents usual line of diffusion
  - Use wide transistors on critical path
  - Eliminate output inverters



Full Adder Design III

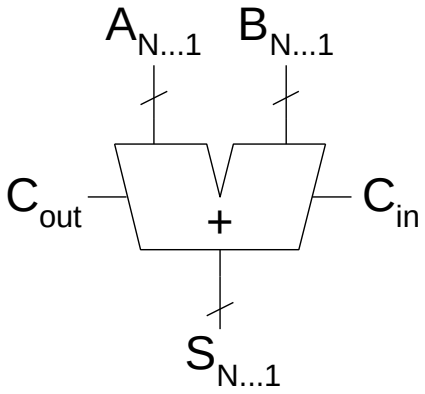
- Complementary Pass Transistor Logic (CPL)
  - Slightly faster, but more area

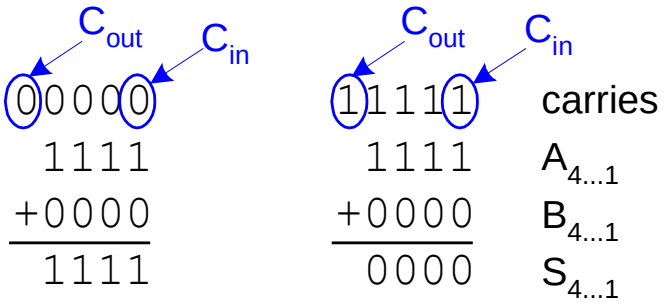




Carry Propagate Adders (CPA)

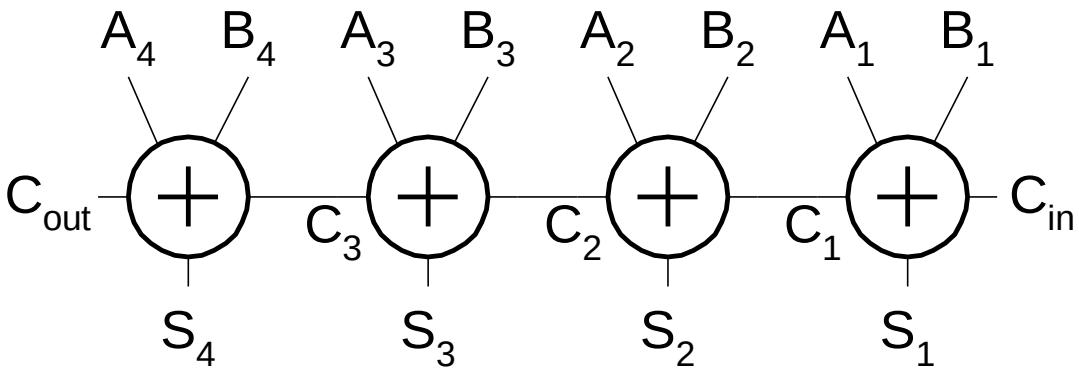
- N-bit adder called CPA
  - Each sum bit depends on all previous carries
  - How do we compute all these carries quickly?





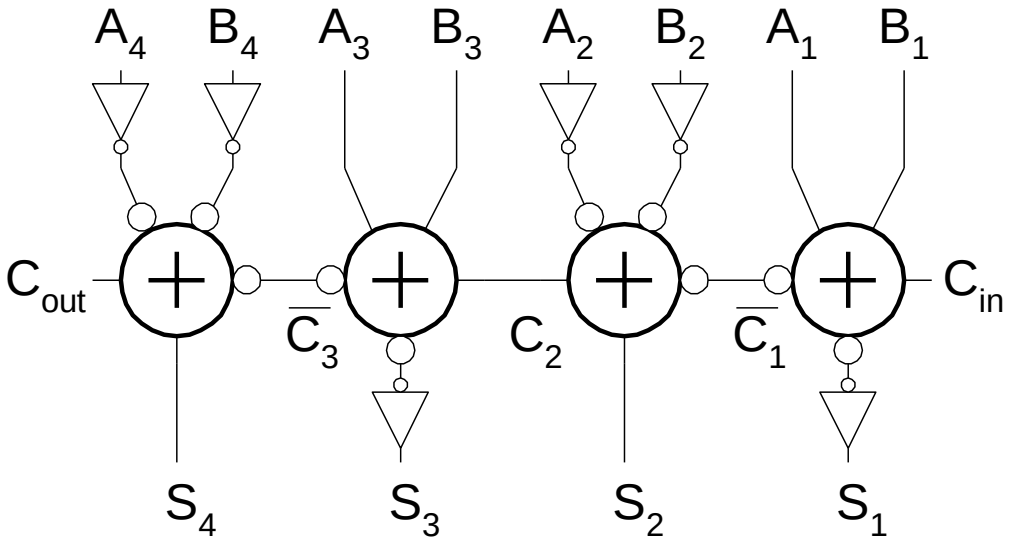
Carry-Ripple Adder (CRA)

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - Design full adder to have fast carry delay



Inversions

- Critical path passes through majority gate
  - Built from minority + inverter
  - Eliminate inverter and use inverting full adder



Generate / Propagate

- Equations often factored into G and P
- Generate and propagate for groups spanning $i{:}j$ ($j \le i$)

$$\forall k = i, i-1, \ldots, j+1:$$

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

$$G_{i:j} \cdot P_{i:j} = 0$$
$$G_i \cdot P_i = 0$$

- Base case

$$G_{i:i} \equiv G_i = A_i B_i$$
$$P_{i:i} \equiv P_i = A_i \oplus B_i$$

$$G_{0:0} \equiv G_0 = C_{in}$$
$$P_{0:0} \equiv P_0 = 0$$

- Because the carry into bit i is the carry-out of bit $i-1$, which is: $C_{i-1} = G_{i-1:0}$, the sum is: $S_i = P_i \oplus G_{i-1:0}$

Comment on Calculating the Generate/Propagate Signals

- In some textbooks, you will find the following:

$$G_{i:i} \equiv G_i = A_i B_i$$

$$P_{i:i} \equiv P_i = A_i + B_i$$

$$G_i \cdot P_i \neq 0 \text{ and } G_{i:j} \cdot P_{i:j} \neq 0$$

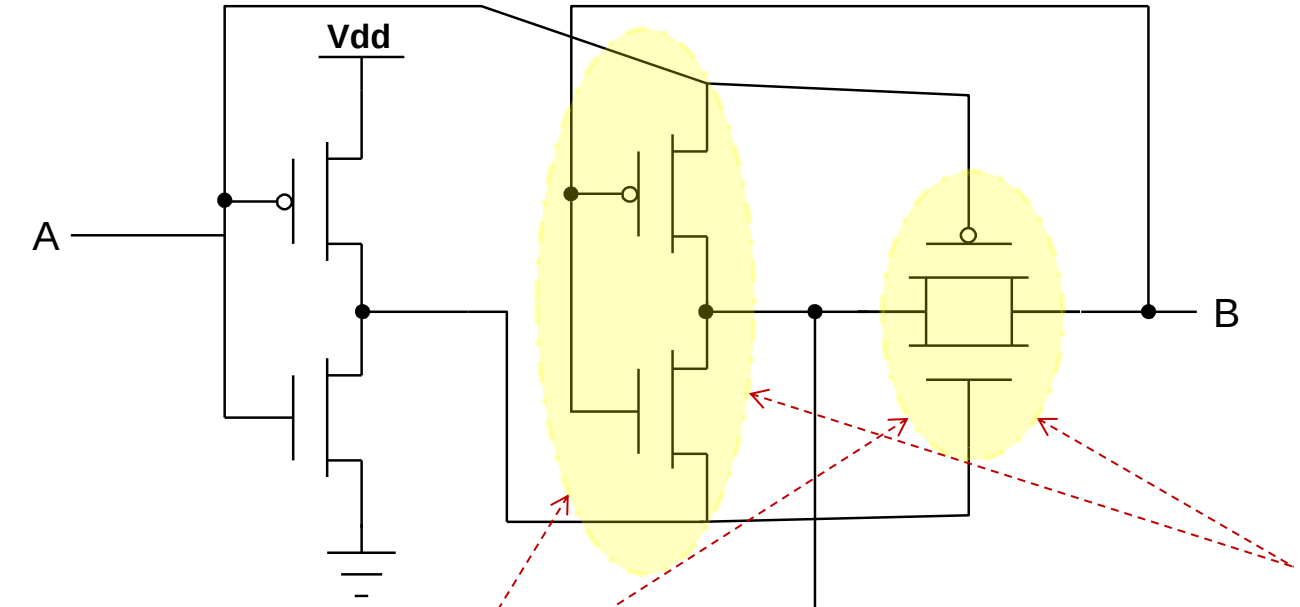
- If you use these equations, then therefore, we must calculate the sum bit as follows:

$$S_i = P_i \oplus G_i \oplus G_{i-1:0}$$

- Expressions for $G_{i:j}$ and $P_{i:j}$ remain the same, including

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0}$$

Efficient Realization of XOR Gate

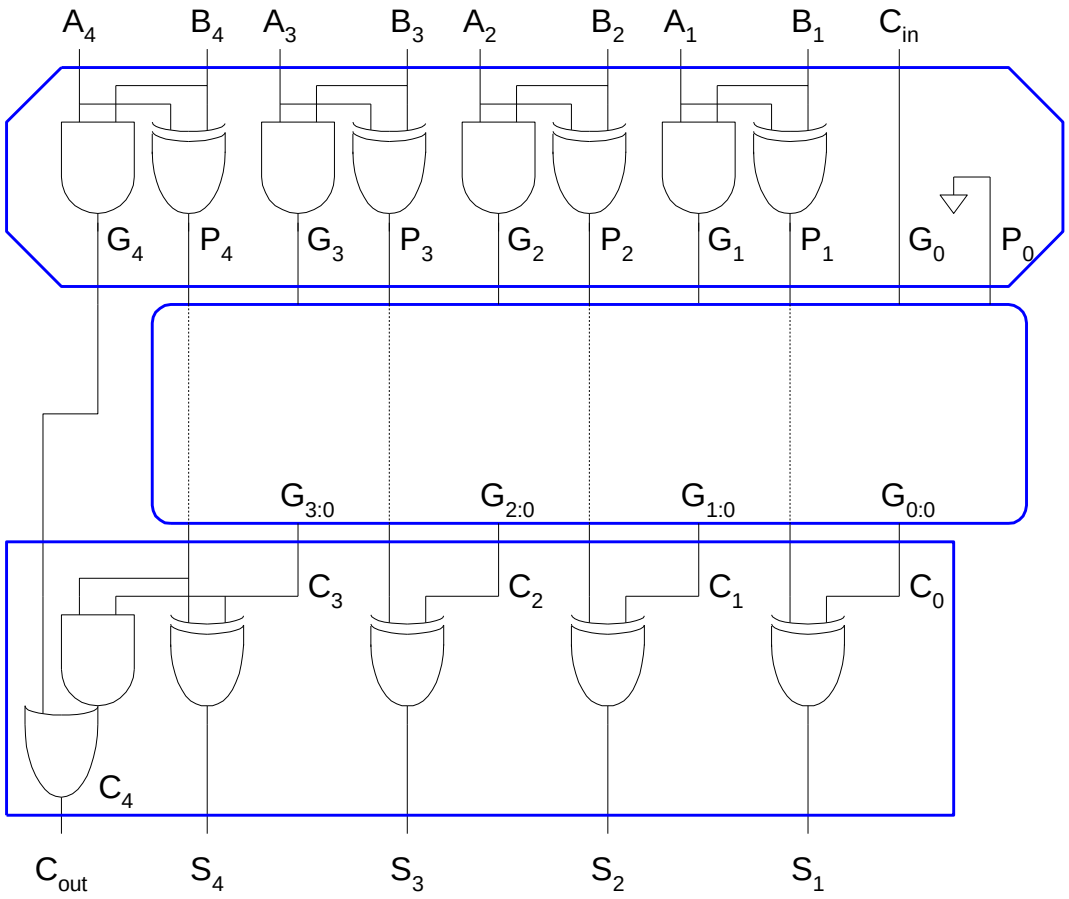


When A=1, TG is open and this structure is an inverter

When A=0, TG is conducting and this structure is a weak buffer!

XOR function:
If A=0, pass input B; Otherwise, invert input B

PG Logic



1: Bitwise PG logic

2: Group PG logic

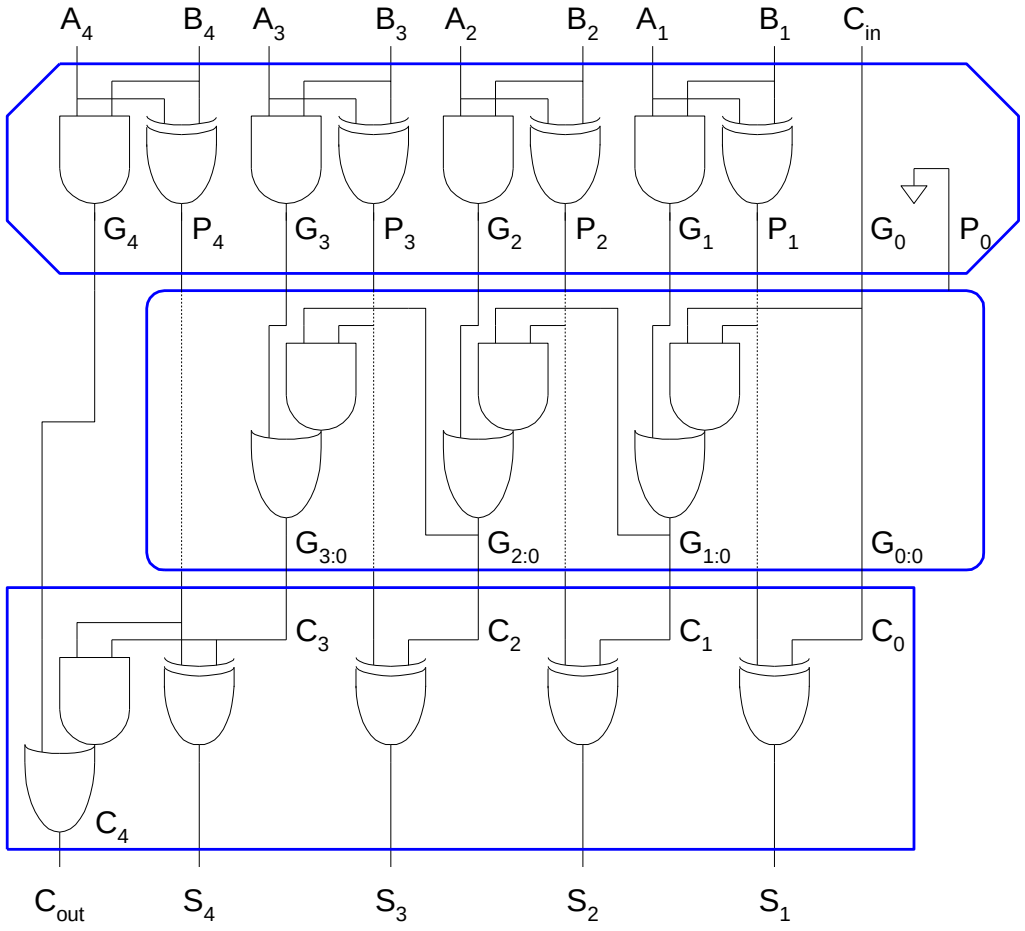
Key step

3: Sum logic

CRA Revisited

$$C_i = A_i B_i + (A_i + B_i) C_{i-1}$$
$$= A_i B_i + (A_i \oplus B_i) C_{i-1}$$
$$= G_i + P_i C_{i-1}$$

$$G_{i:0} = G_i + P_i \cdot G_{i-1:0}$$



CRA PG Diagram

- Adding two operands: A[1:16] and B[1:16]
- The Bitwise PG Logic and Sum Logic blocks are not shown
- Gray cells contain only the group generate logic
- Final sum is computed as:

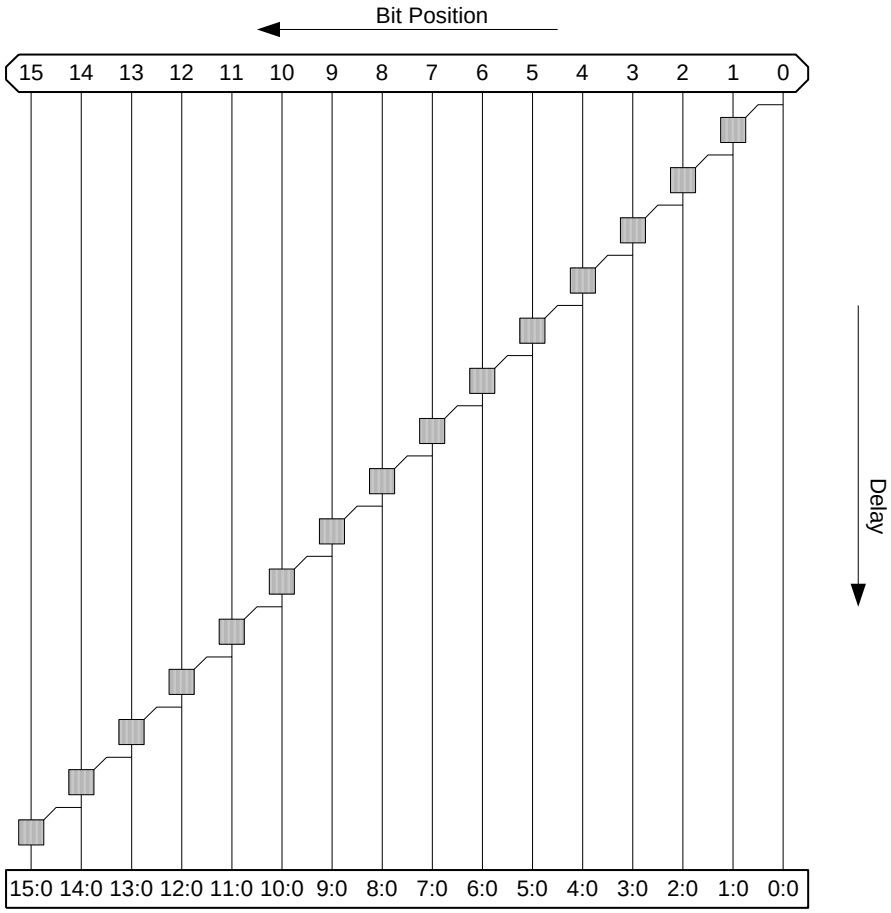
$$S_{16} = P_{16} \oplus G_{15:0} = P_{16} \oplus C_{15}$$

- Final carry-out is computed as:

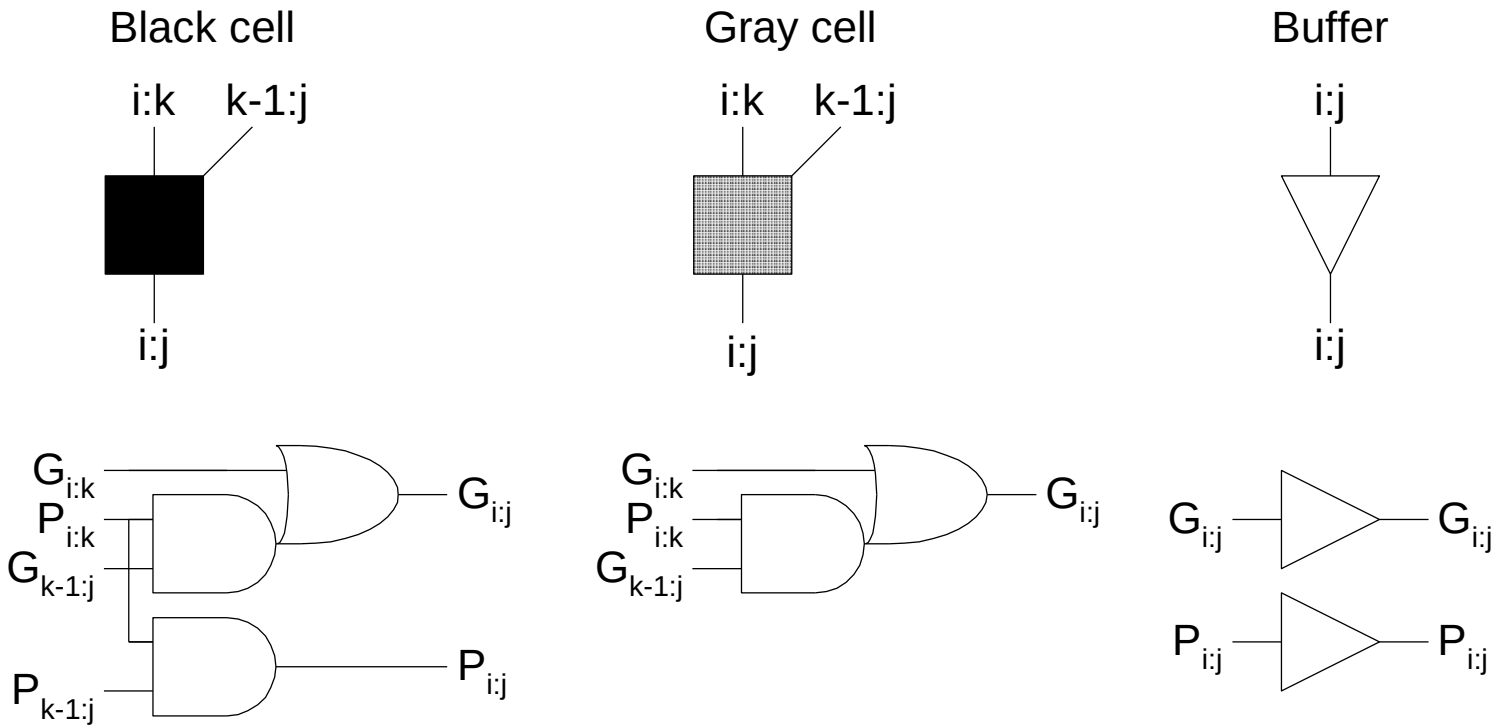
$$C_{16} = G_{16:0} = G_{16} + P_{16}C_{15}$$

- The critical path delay from carry-in to sum bit is:

$$t_{\text{ripple}} = t_{pg} + (N-1)t_{AO} + t_{\text{xor}}$$



PG Diagram Notation

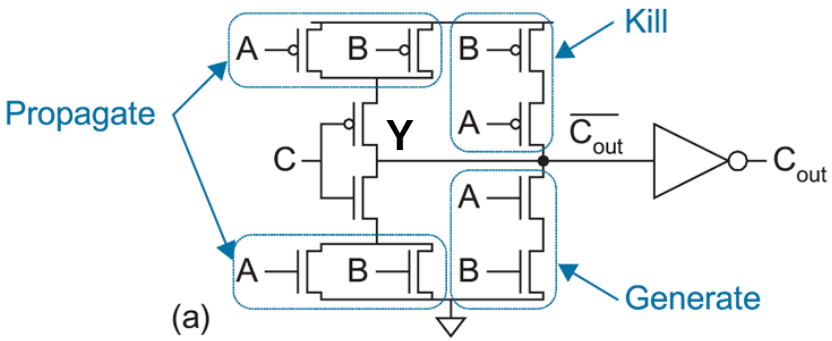


Notice that, for the Gray cell, vertical input line carries both $G_{i:k}$ and $P_{i:k}$ signals whereas the diagonal input line only provides the $G_{k-1:j}$ signal

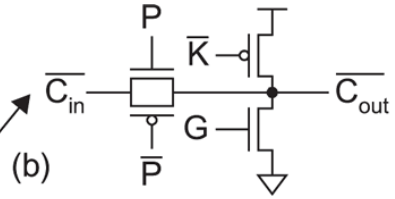
Carry Chain Designs (Optional)

Recall that:

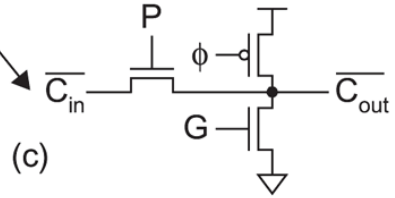
$$P = A \oplus B, \quad G = A \cdot B, \quad K = \overline{A} \cdot \overline{B}$$



Static Version



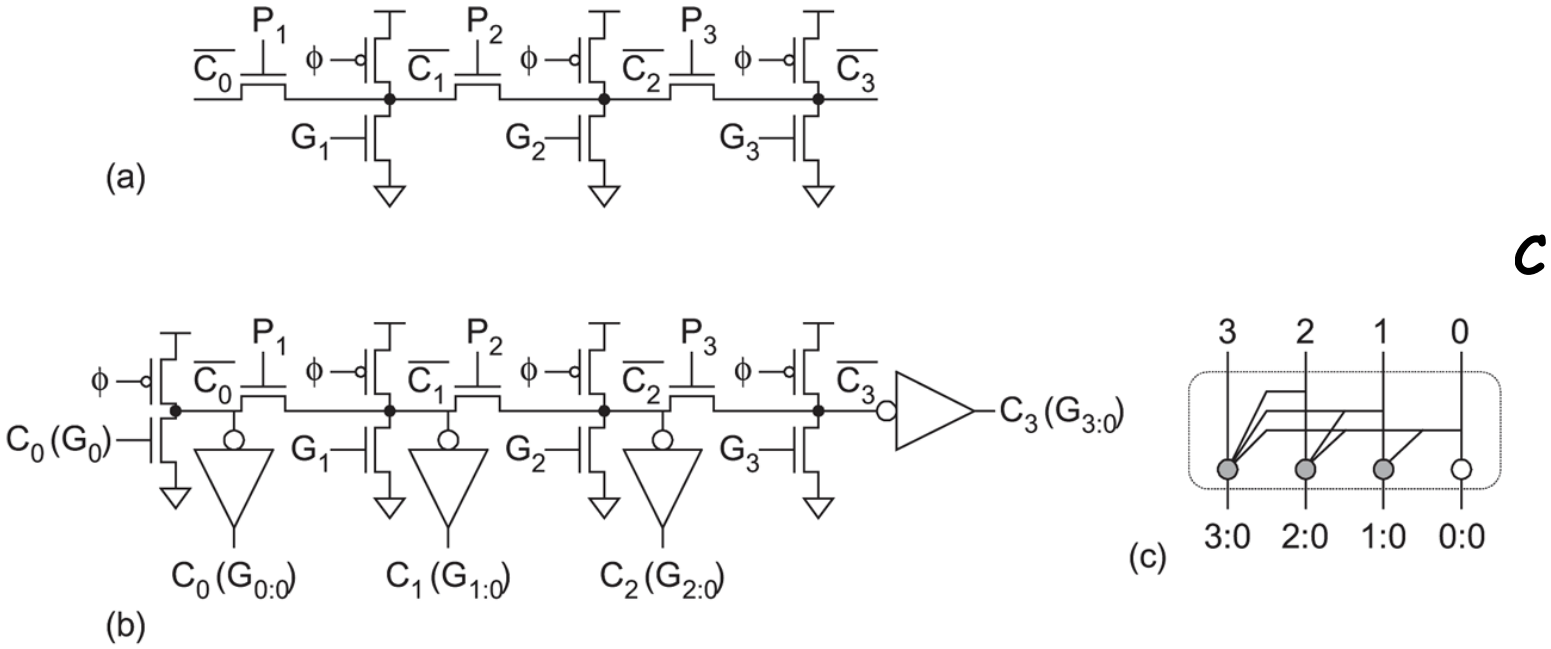
Dynamic Version



C	A, B	Y	G	K	C_{out}
1	$A+B$	0	-	0	1
1	$\overline{A} \cdot \overline{B}$	float	0	**1**	0
0	$\overline{A}+\overline{B}$	1	0	-	0
0	$A \cdot B$	float	**1**	0	1

$$C_{out} = C_{in} \cdot (A+B) + G = MAJ(A, B, C_{in})$$
$$= C_{in} \cdot P + G$$
$$\overline{C}_{out} = \overline{C}_{in} \cdot (\overline{A}+\overline{B}) + K = MIN(A, B, C_{in})$$
$$= \overline{C}_{in} \cdot P + K$$

Manchester Carry Chain (Optional)

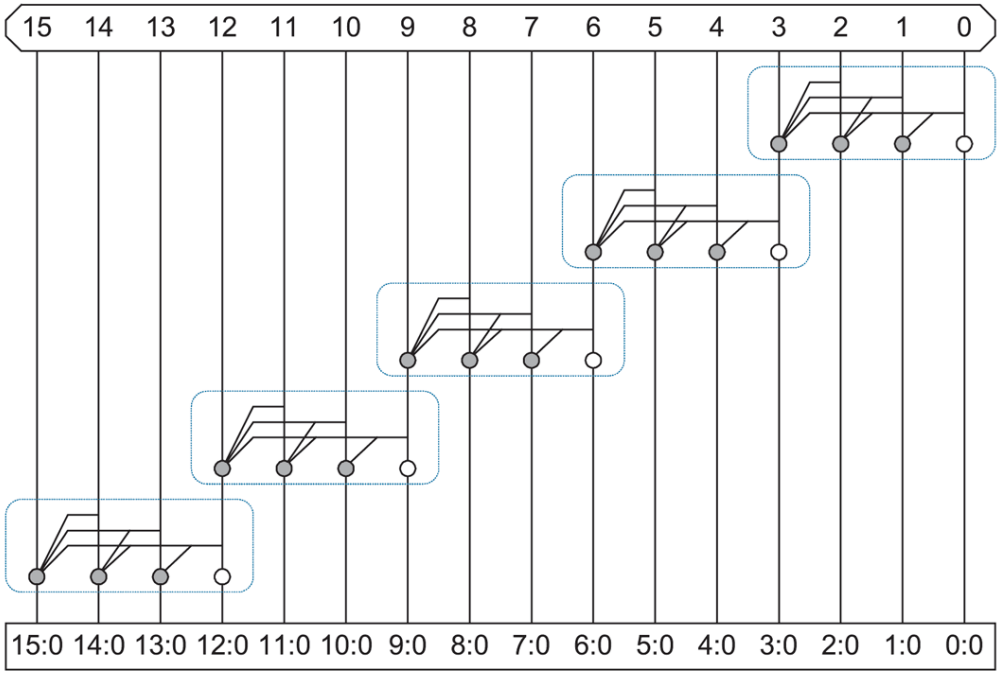


Manchester carry chains

C

One may also use the static version of Manchester Carry Chain (MCC)

MCC Adder PG Network (Optional)



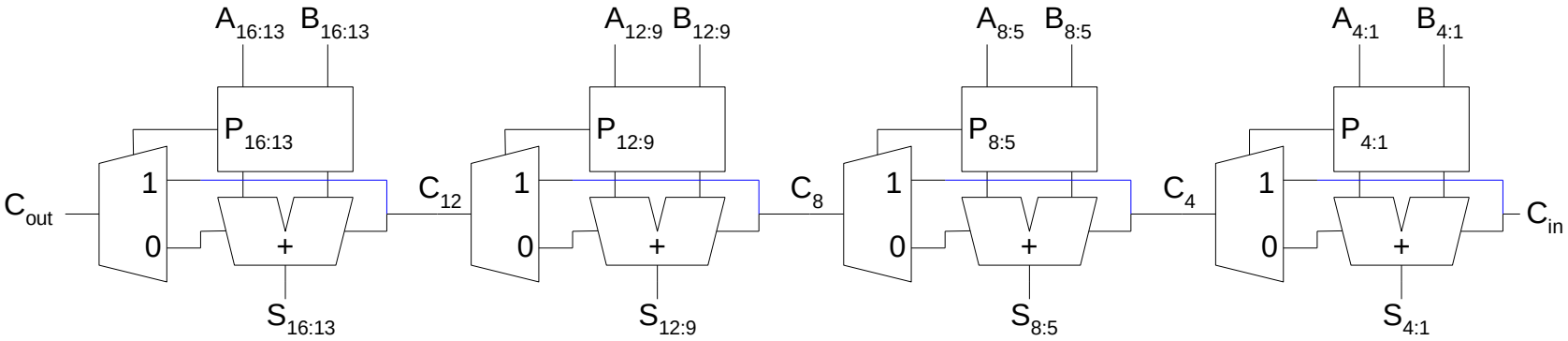
Manchester carry chain adder group PG network

- With valency-n MCC stages i.e., *N=(n –1).k* :

$$t_{\text{mcca}} = t_{pg} + kt_{mcc(n)} + t_{\text{xor}}$$

Carry-Bypass Adder (CBA)

- Carry-bypass allows carry to skip over groups of *n* bits
  - Decision is based on the *n*-bit propagate signal
  - The bypass speeds up the addition because either the carry-in propagates through the bypass path, or the carry-in is (first killed and subsequently re-) generated in the chain. In both cases, the delay is smaller than the normal ripple carry adder



Example: *n*=4

$$C_{12} = \overline{P}_{12:9} \cdot G_{12:9} + P_{12:9} \cdot C_8$$

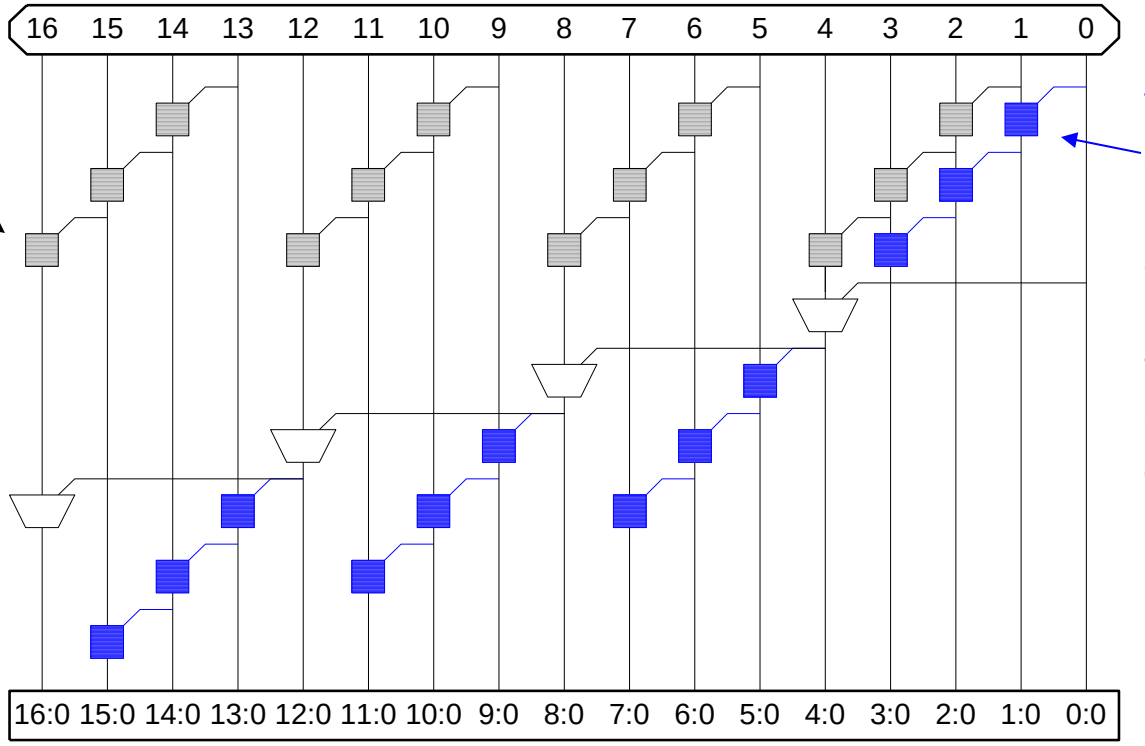
Since $G_{12:9} \subseteq \overline{P}_{12:9}$,

$$C_{12} = G_{12:9} + P_{12:9} \cdot C_8$$

This means that the MUX gate may be replaced with a simple AO gate

CBA PG Diagram

4-bit CRA determines if group generates a carry

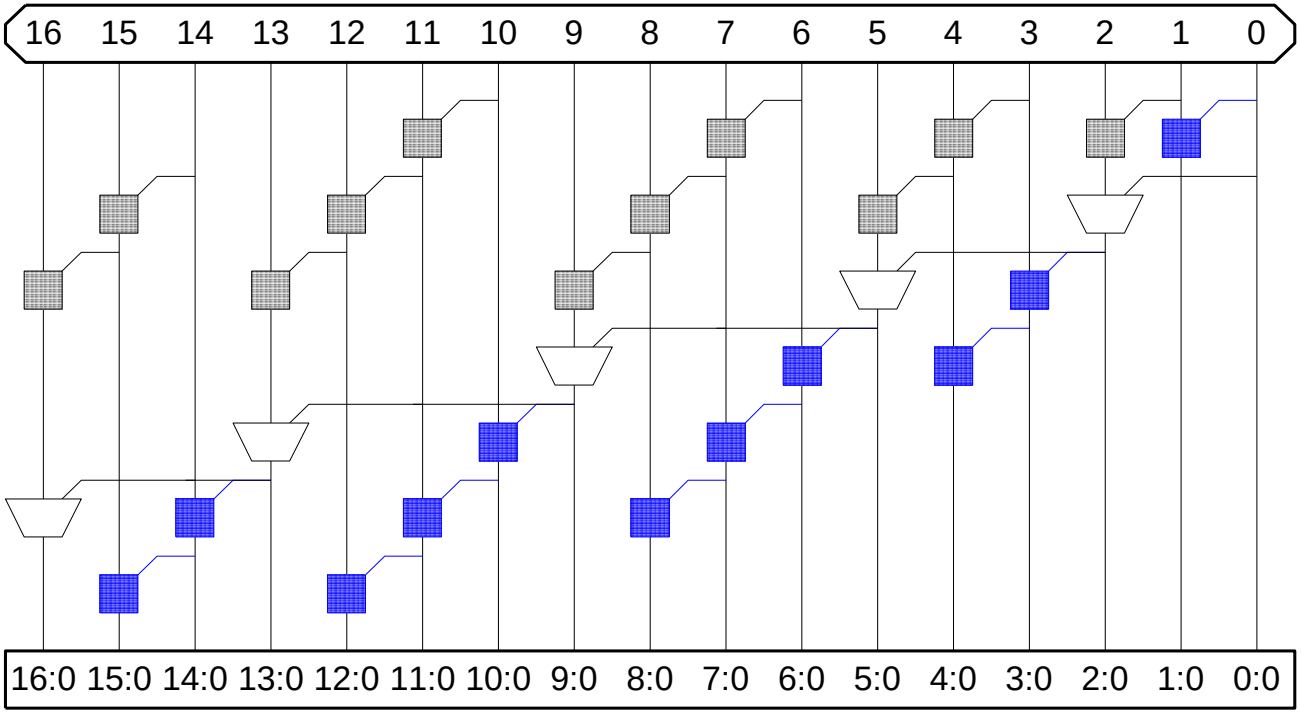


The *same* 4-bit CRA's that can produce a carry-out when a carry-in is bypassed to them: Results used for sum bit calculations only

For *k* groups of *n* bits each (*N* = *k.n*) and replacing MUX's w/ AO gates:

$$t_{\text{skip}} = t_{pg} + 2(n-1)t_{AO} + (k-1)t_{mux} + t_{\text{xor}} = t_{pg} + (2n+k-3)t_{AO} + t_{\text{xor}}$$

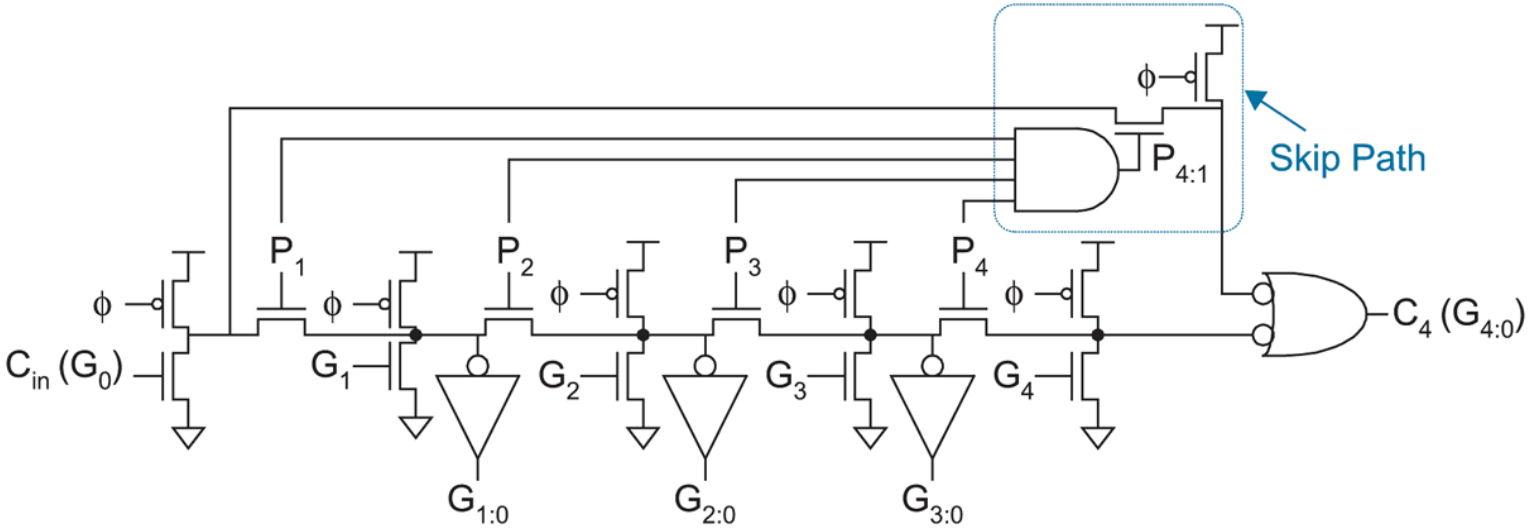
CBA w/ Variable Group Size



Group sizes: (2,3,4,4,3)

Delay grows as $O(\sqrt{N})$

CBA Manchester Stage (Optional)

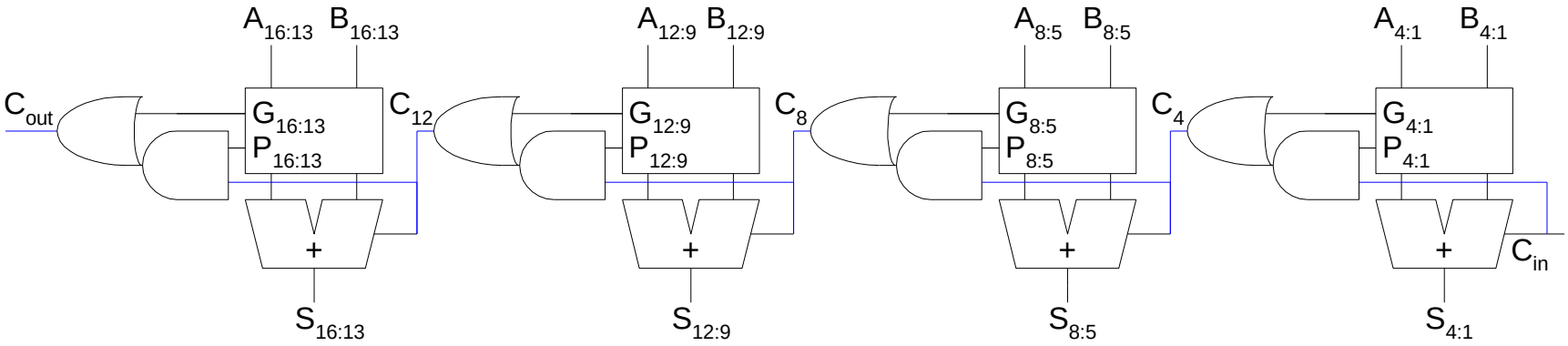


Carry-skip adder Manchester stage

Skips across carry groups of 4 bits at a time

Carry-Lookahead Adder (CLA)

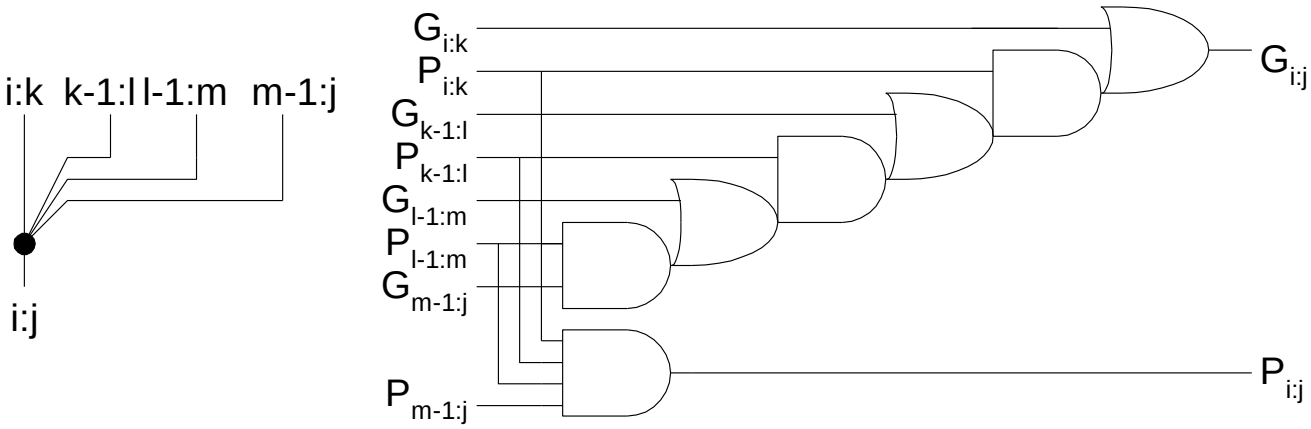
- Carry-lookahead adder computes $G_{i:0}$ for many bits in parallel
- Uses higher-valency cells with more than two inputs



Higher-Valency Group PG Cells

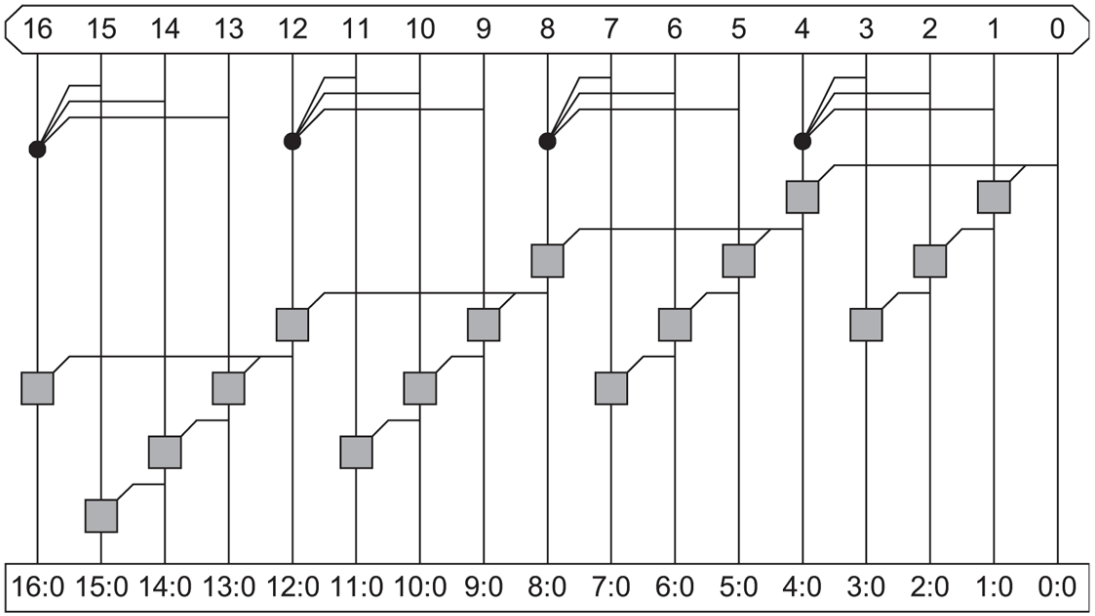
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot (G_{k-1:l} + P_{k-1:l} \cdot (G_{l-1:m} + P_{l-1:m} \cdot G_{m-1:j}))$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:l} \cdot P_{l-1:m} \cdot P_{m-1:j}$$



Valency-4 black cell

CLA PG Diagram

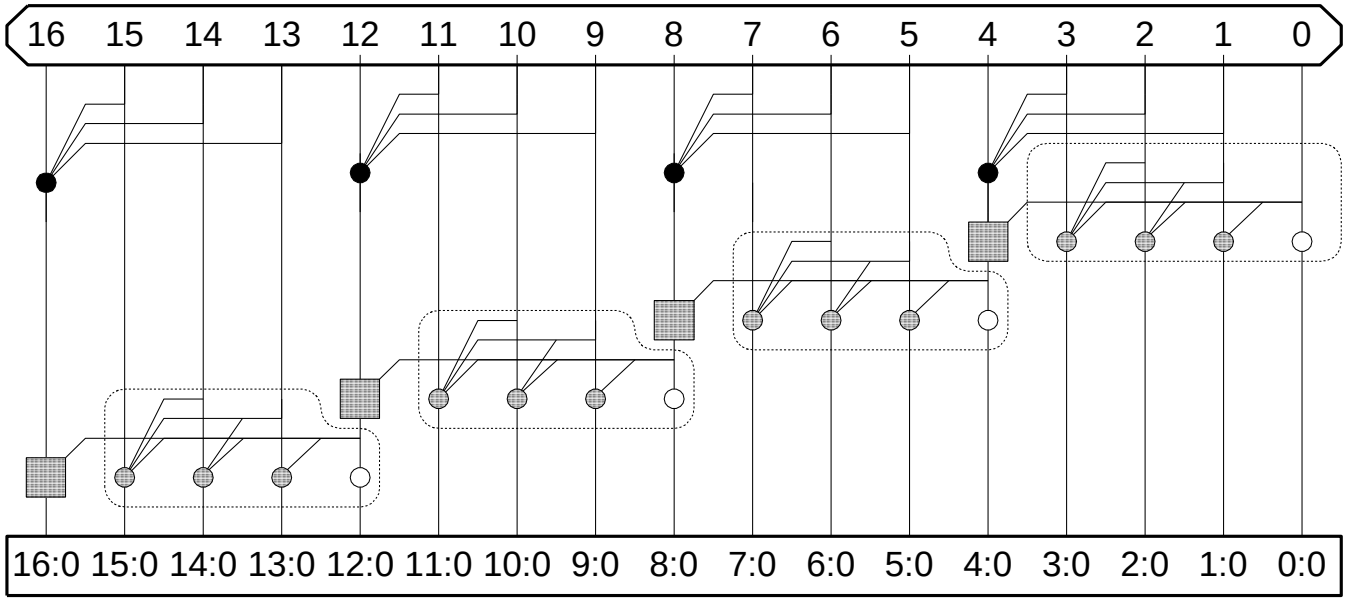


Carry-lookahead adder group PG network

- For *k* groups of *n*-bits each :

$$t_{\text{CLA}} = t_{pg} + t_{pg(n)} + \left[(n-1)+(k-1)\right] t_{AO} + t_{\text{xor}}$$

Improved CLA with MCCs (Optional)

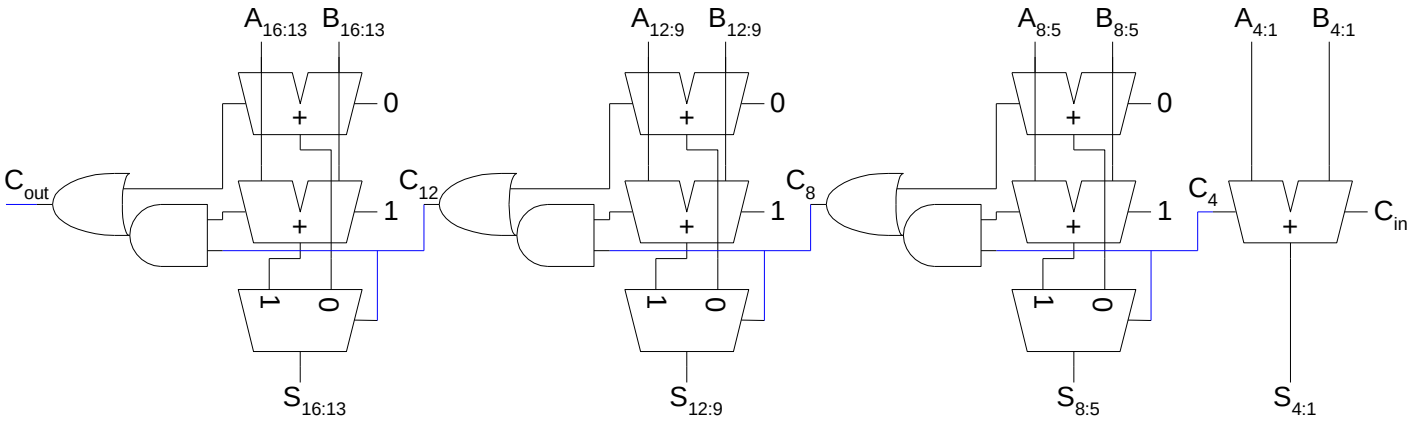


- For *k* groups of *n*-bits each and valency-n MCC blocks:

$$t_{\text{CLA+MCC}} = t_{pg} + t_{pg(n)} + (k-1)t_{AO} + t_{mcc(n)} + t_{\text{xor}}$$

Carry-Select Adder (CSA)

- For critical paths which are dependent on late input X,
  - Precompute two possible outputs for X = 0, 1
  - Select proper output when X arrives
- Carry-select adder precomputes *n*-bit sums for both possible carries into the *n*-bit group



$$t_{\text{select}} = t_{pg} + [n + (k-2)]t_{AO} + t_{\text{mux}}$$

Example Delay Optimization

- Consider designing an N-bit carry select adder (CSA) comprising of *k* groups of *n*-bits each ($N=k\times n$). Write the expression for the worst-case delay from any input to any output in this adder. What are the optimum value for *n* and *k* which minimize the delay of the N-bit adder? What is the minimum delay of this adder?

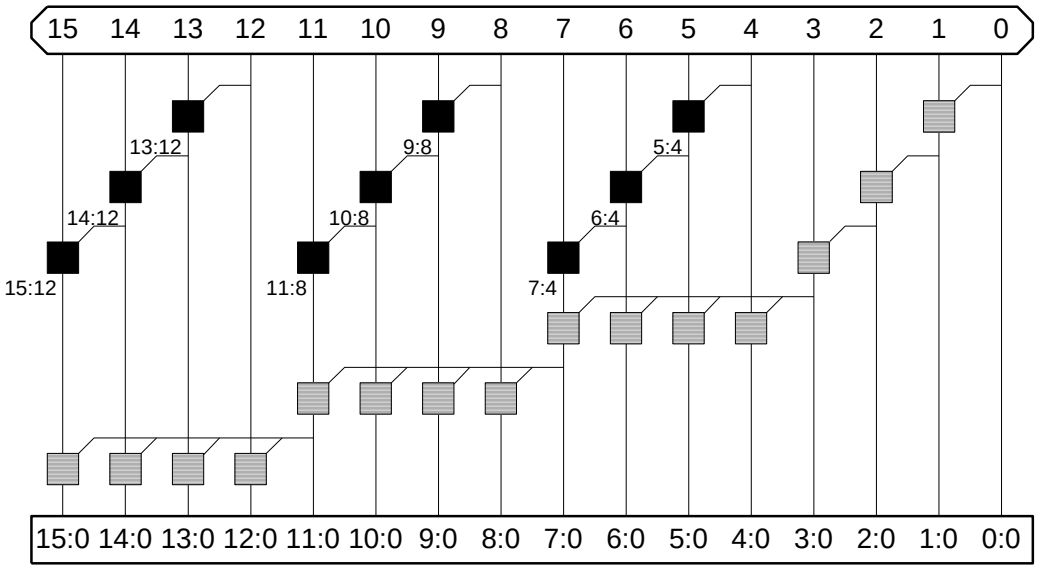
$$Delay = t_{pg} + \left(\frac{N}{k} + k - 2\right) t_{AO} + t_{mux}$$

$$\frac{\partial Delay}{\partial k} = -\frac{N}{k^2} + 1 = 0 \Rightarrow k = \sqrt{N}$$

$$Delay_{opt} = t_{pg} + 2\left(\sqrt{N} - 1\right) t_{AO} + t_{mux}$$

Carry-Increment Adder (CIA)

- It uses a short ripple chain of black cells to compute the PG signals for bits within a group
- When the carry-out from the previous group becomes available, the final gray cells in each column determine the carry-out
- Higher-valency group PG cells can be used to speed up the ripple operation to produce the first group generate signals



$$t_{\text{increment}} = t_{pg} + [(n-1)+(k-1)]t_{AO} + t_{\text{xor}}$$

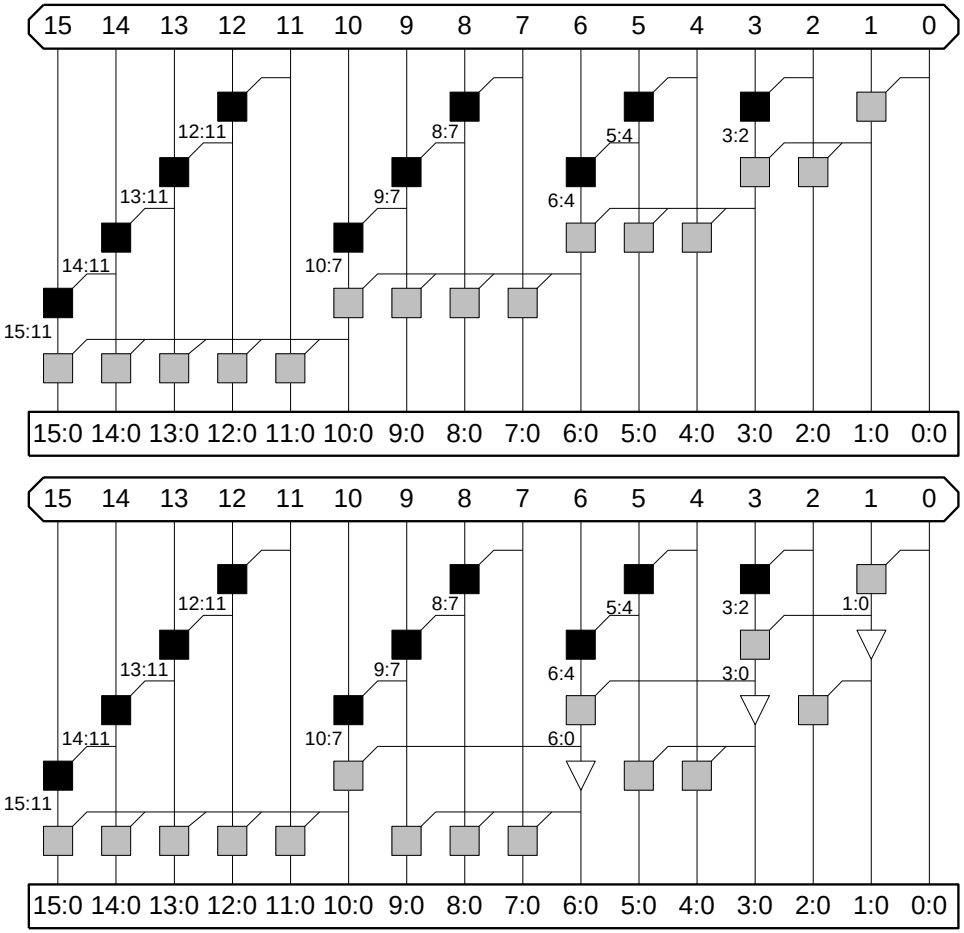
$$t_{\text{increment+CL}} = t_{pg} + t_{pg(n)} + (k-1)t_{AO} + t_{\text{xor}}$$

Variable-Length CIA

- Use variable-length groups to utilize the fact that the more significant bits complete early

$$t_{\text{increment}} = t_{pg} + \sqrt{2N}\,t_{AO} + t_{\text{xor}}$$

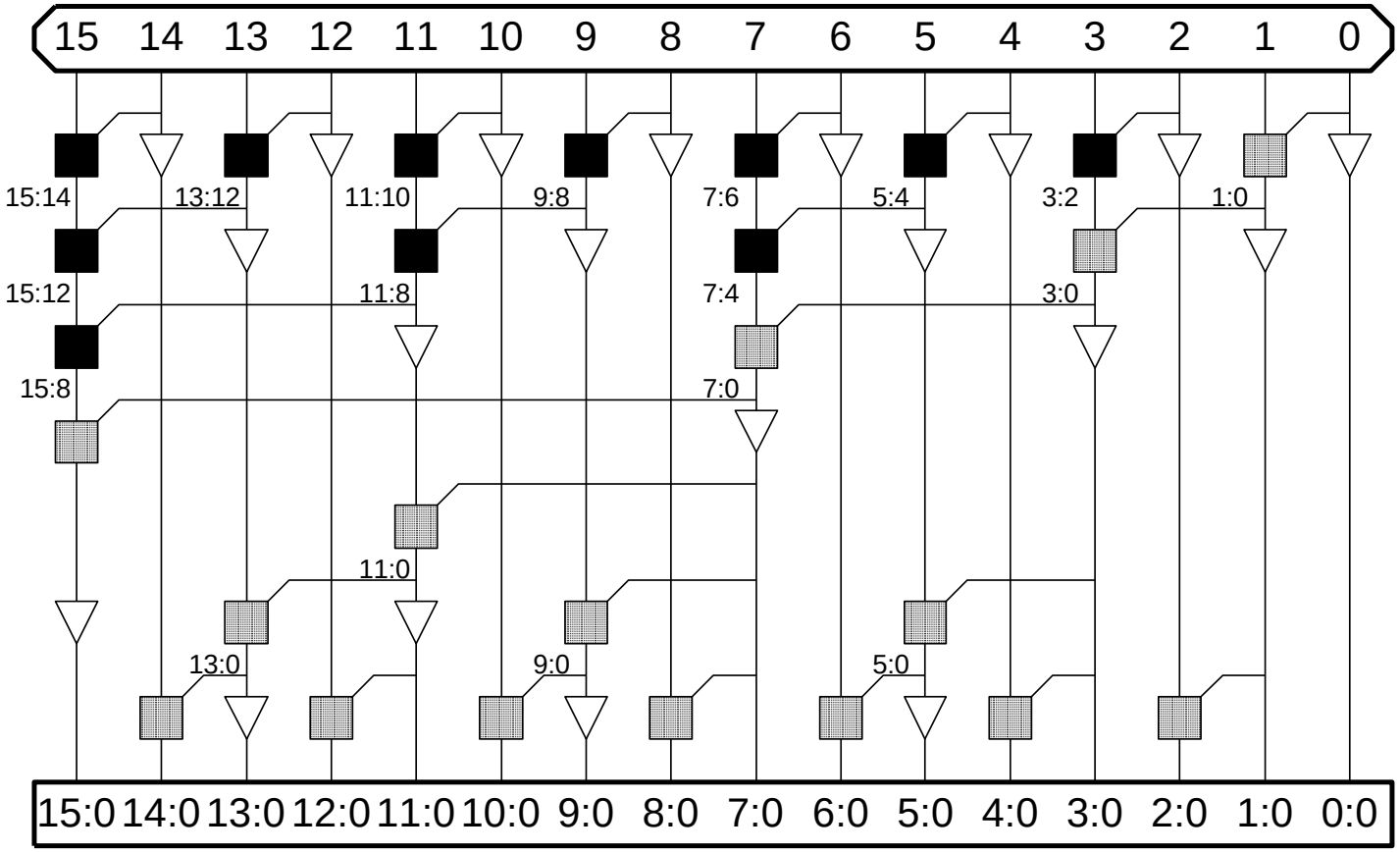
- Also do critical signal isolation by buffering along the noncritical signal paths



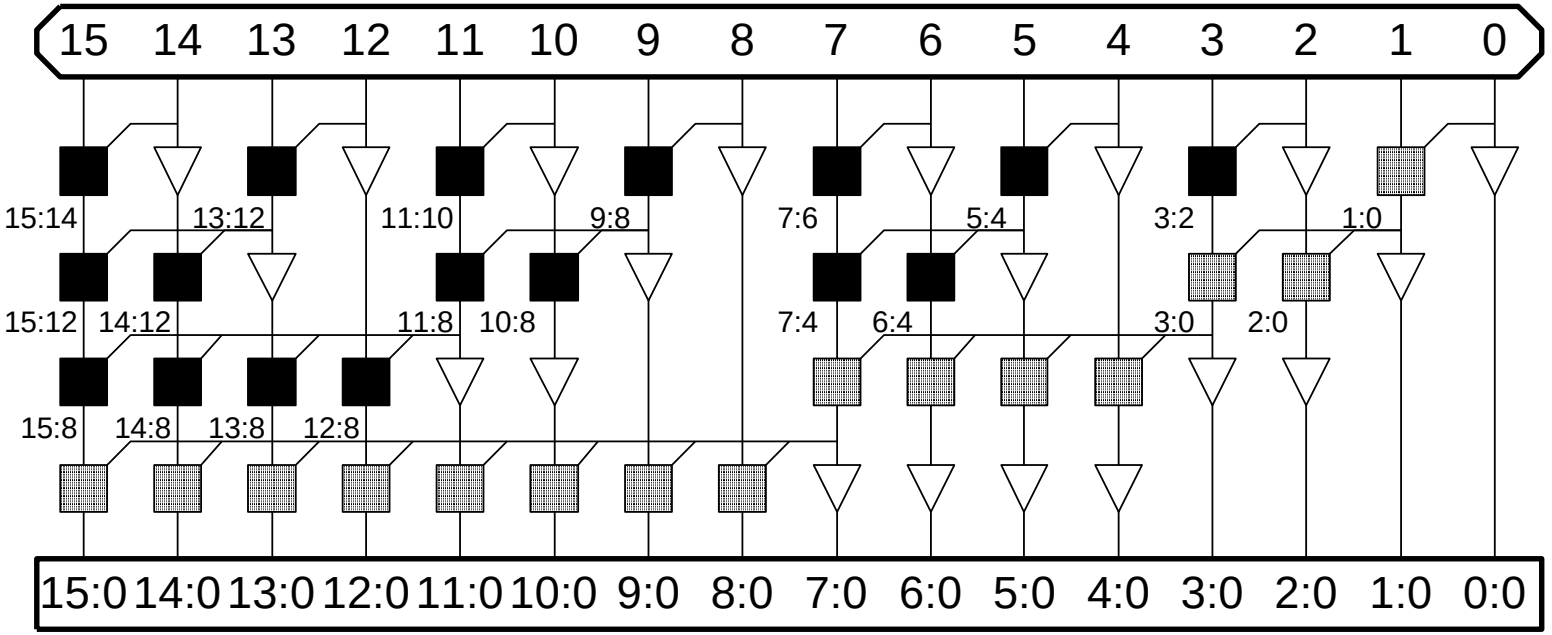
(Prefix) Tree Adders

- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives O(log N) delay
- Many variations on tree adders (also known as logarithmic adders, multilevel-lookahead adders, and parallel-prefix adders)

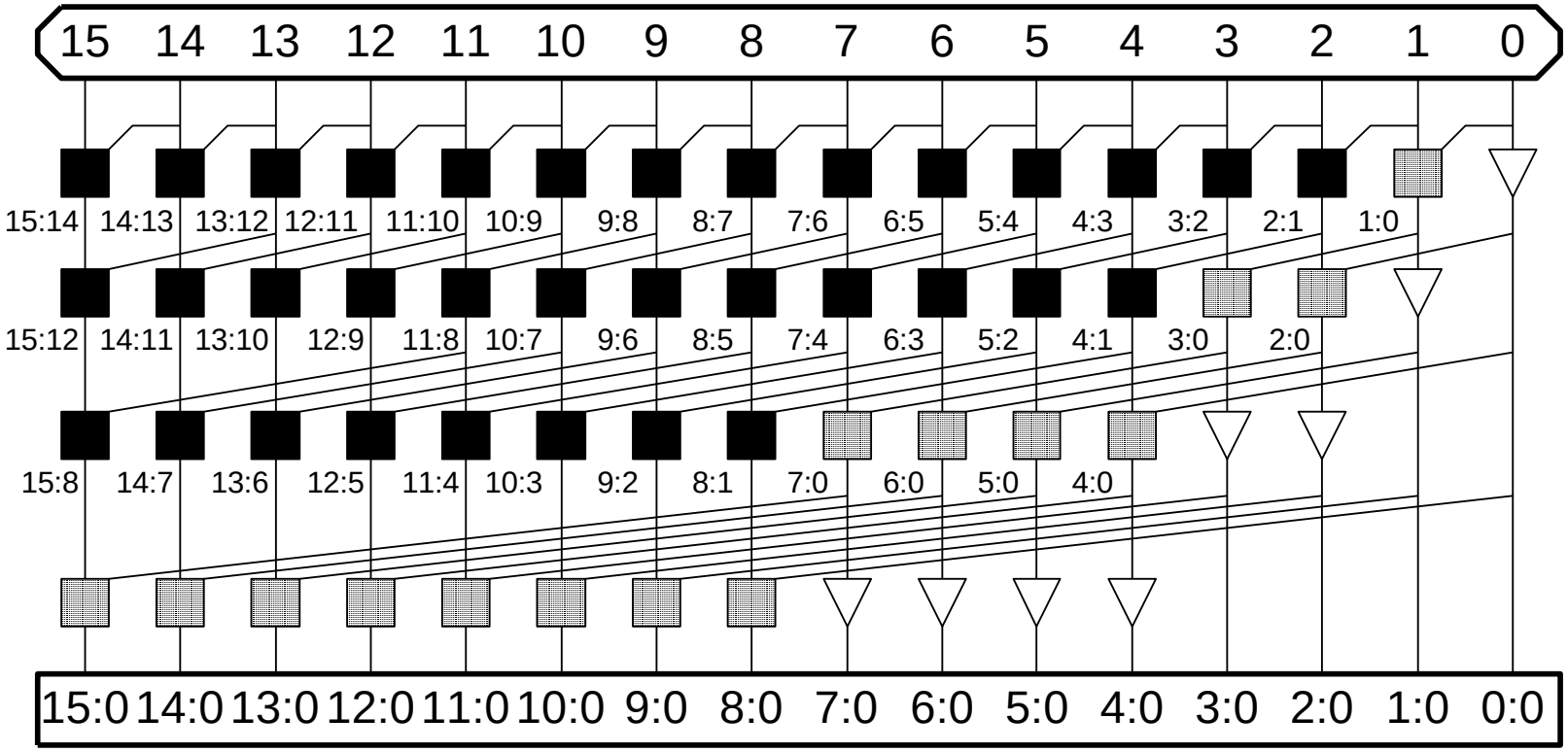
Brent-Kung


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14
13:12
11:10
9:8
7:6
5:4
3:2
1:0
15:12
11:8
7:4
3:0
15:8
7:0
11:0
13:0
9:0
5:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

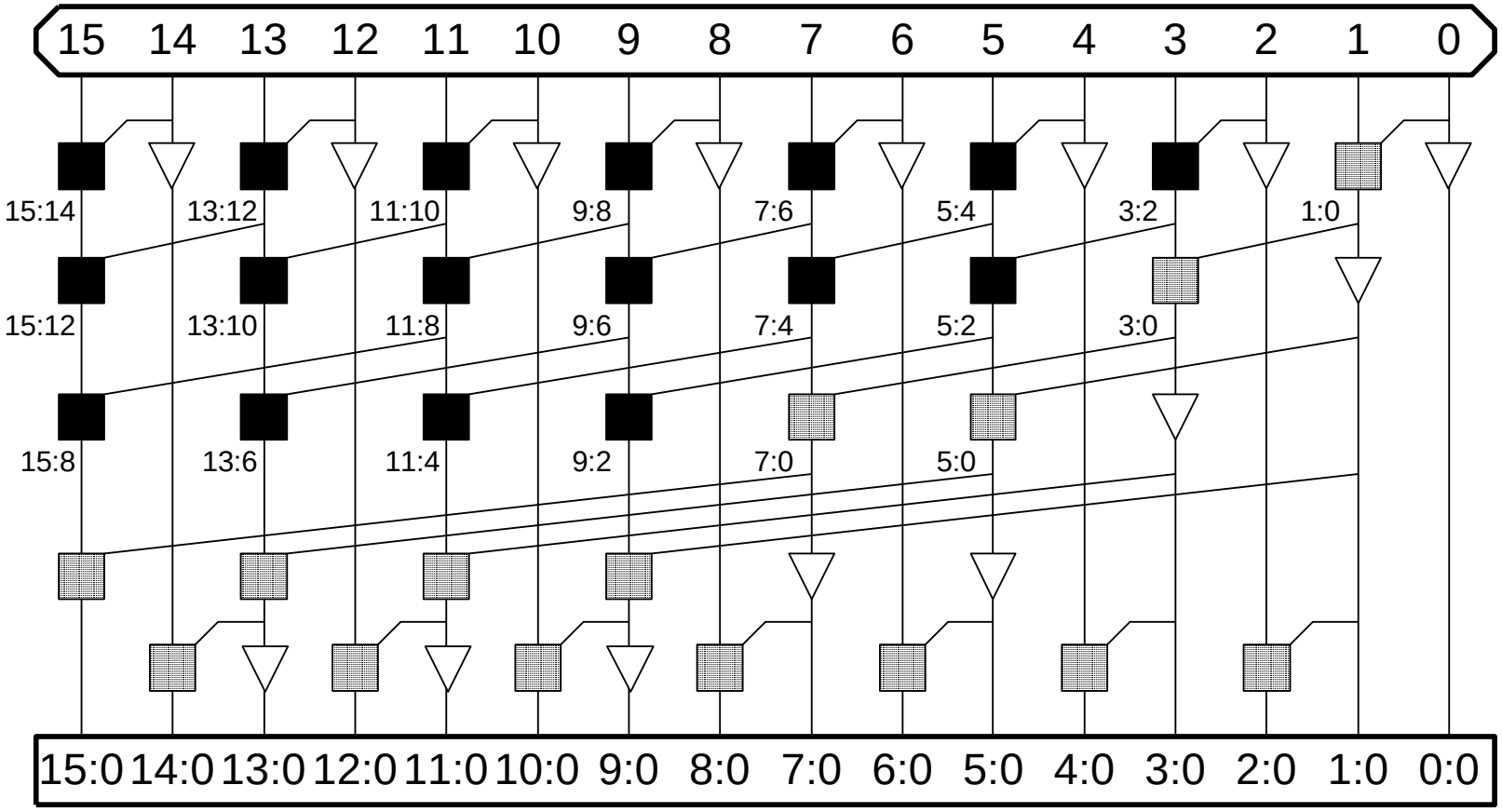
Sklansky


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 13:12 11:10 9:8 7:6 5:4 3:2 1:0
15:12 14:12 11:8 10:8 7:4 6:4 3:0 2:0
15:8 14:8 13:8 12:8
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

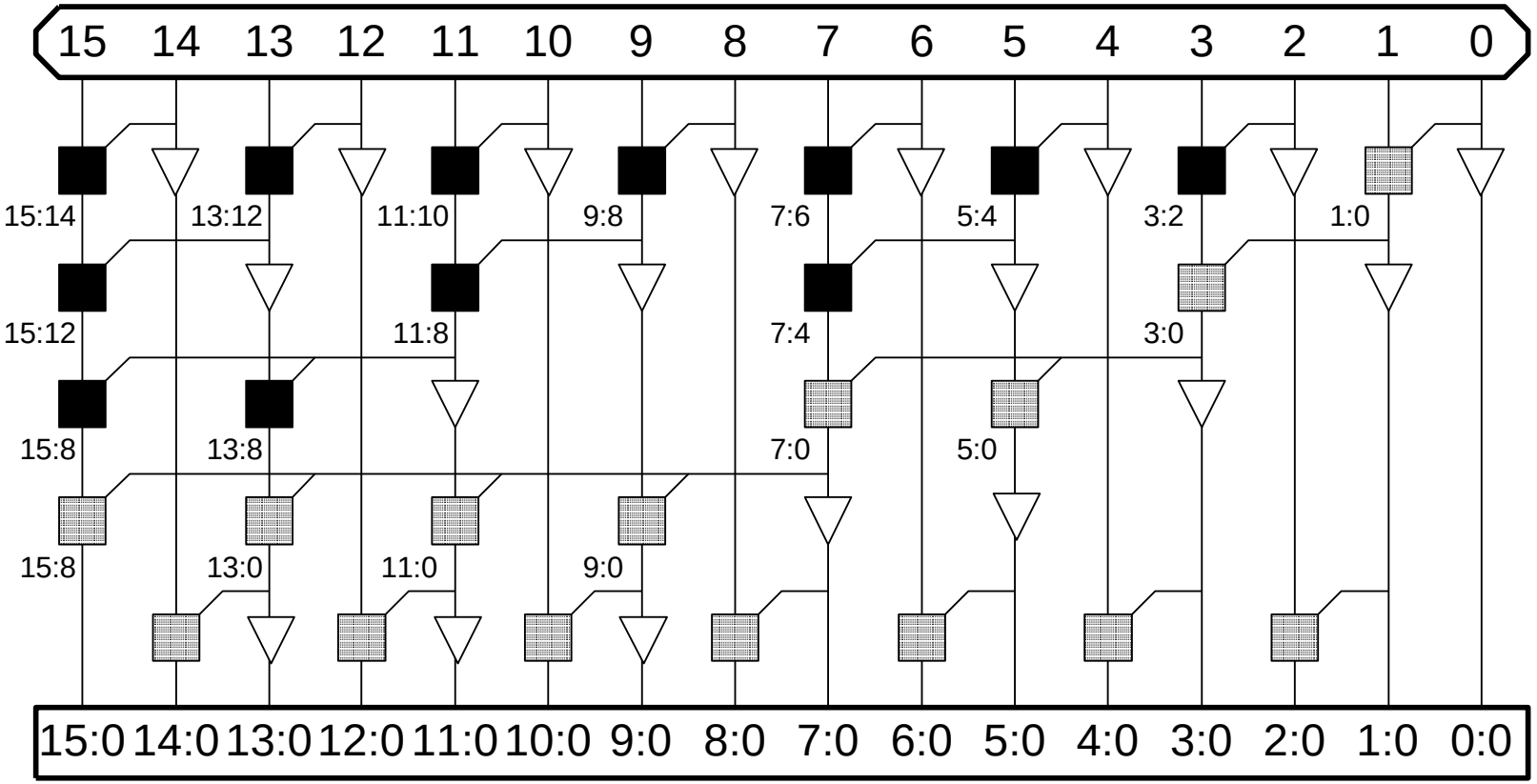
Kogge-Stone


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 14:13 13:12 12:11 11:10 10:9 9:8 8:7 7:6 6:5 5:4 4:3 3:2 2:1 1:0
15:12 14:11 13:10 12:9 11:8 10:7 9:6 8:5 7:4 6:3 5:2 4:1 3:0 2:0
15:8 14:7 13:6 12:5 11:4 10:3 9:2 8:1 7:0 6:0 5:0 4:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

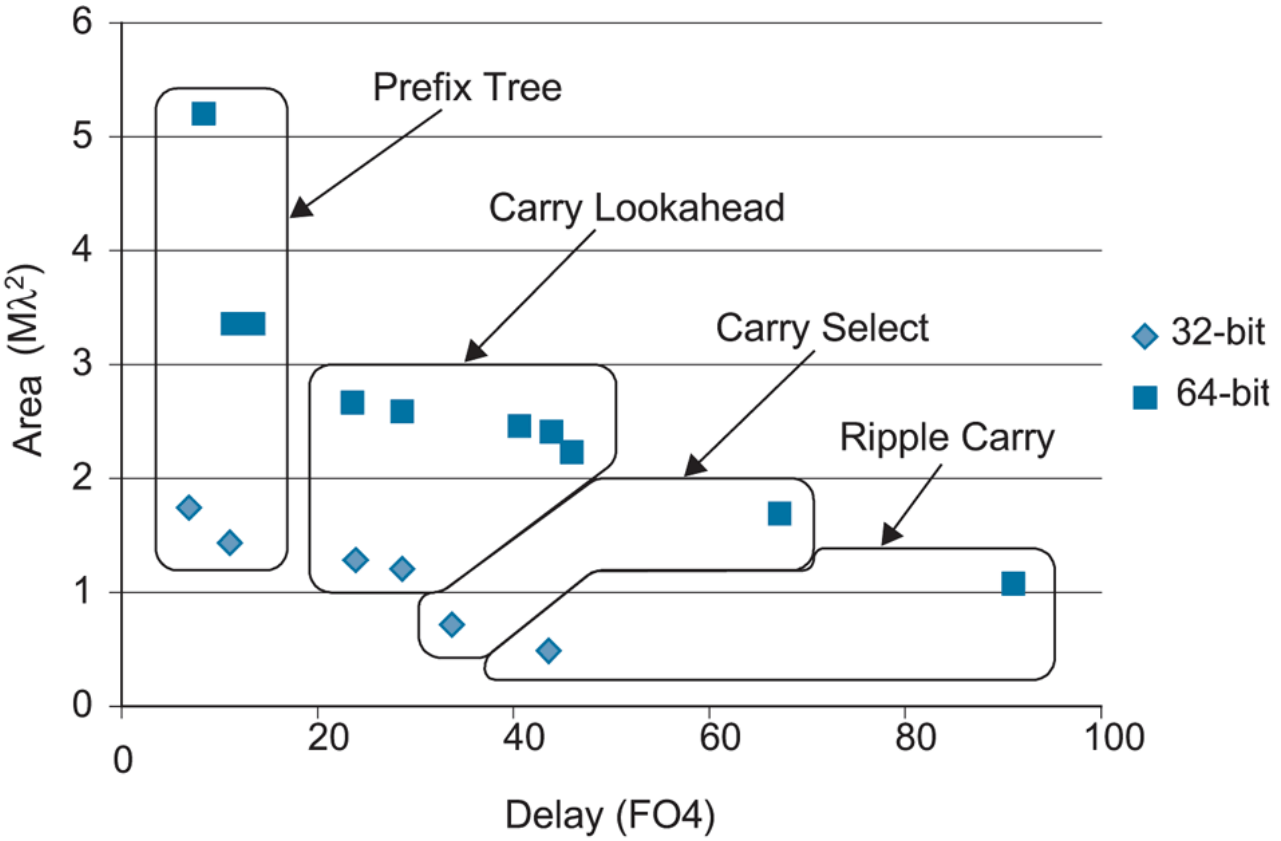
Han-Carlson


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 13:12 11:10 9:8 7:6 5:4 3:2 1:0
15:12 13:10 11:8 9:6 7:4 5:2 3:0
15:8 13:6 11:4 9:2 7:0 5:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

Ladner-Fischer



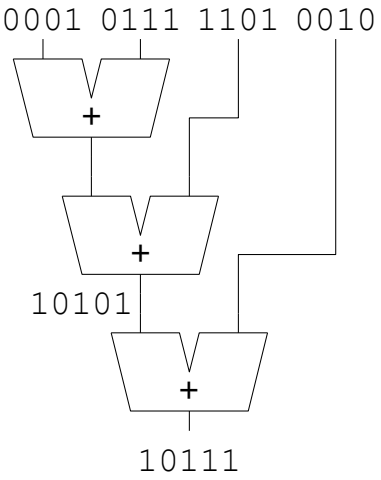
Area vs. Delay Tradeoffs



Area vs. delay of synthesized adders

Multi-input Adders

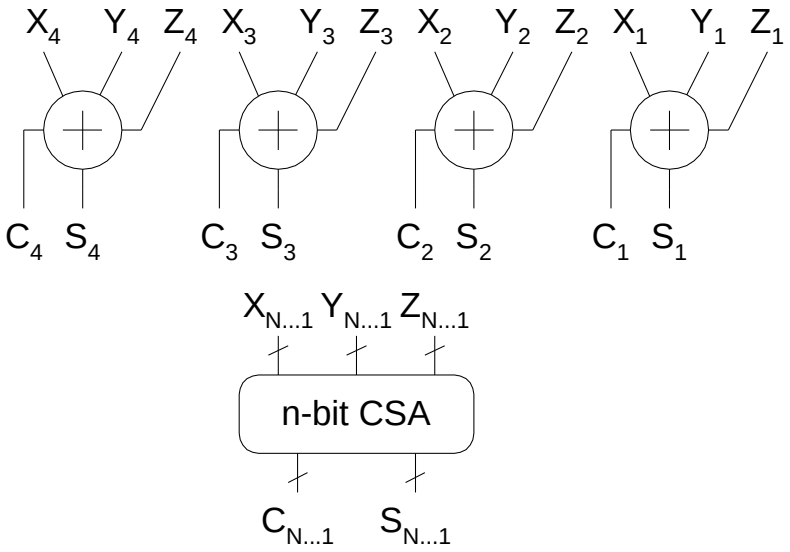
- Suppose we want to add k N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111
- Straightforward solution: k-1 N-input CPAs
  - Large and slow



Carry Save Addition

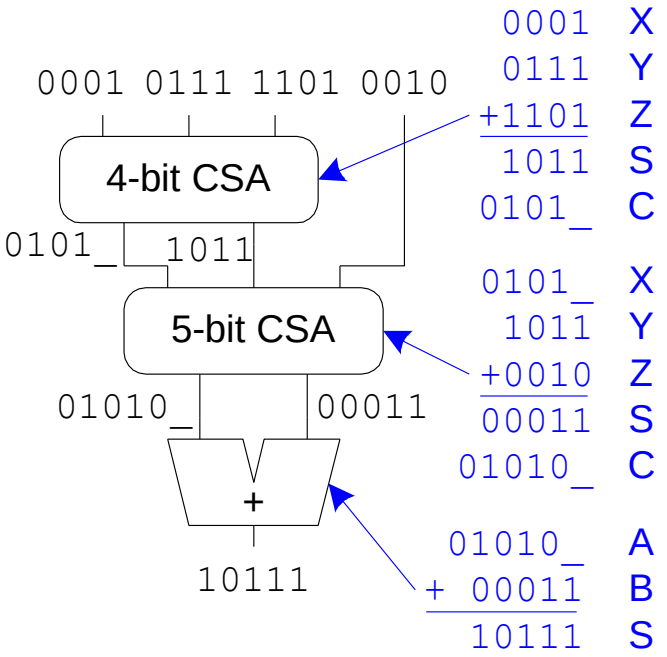
- A full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice the *weight* of sum output

- 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



Two's Complement

- A two's-complement system or two's-complement arithmetic is a system in which negative numbers are represented by the two's complement of the absolute value
- The two's complement of a binary number is the value obtained by subtracting the number from a large power of two (i.e., from 2^N for an N-bit two's complement number)
- To negate a two's complement number, invert all the bits (which yields the one's complement), then add 1 to the result. Bit overflow is ignored, which is the normal case with zero

sign bit									
0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

8-bit
two's complement integers

$$11111011_2 = -128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = (-2^7 + 2^6 + \ldots) = -5$$

Sign Extension

- When turning a two's complement number with a certain number of bits into one with more bits (e.g., when copying from a 1 byte variable to a two byte variable), the sign bit must be repeated in all the extra bits

Decimal	4-bit two's complement	8-bit two's complement
5	0101	0000 0101
-3	1101	1111 1101

sign-bit repetition in 4 and 8-bit integers

- Similarly, when a two's complement number is shifted to the right, the sign bit must be maintained. However when shifted to the left, a 0 is shifted in. These rules preserve the common semantics that left shifts multiply the number by two and right shifts divide the number by two

Addition

- Adding two's complement numbers requires no special processing if the operands have opposite signs, that is, the sign of the result is determined automatically

```
 11111 111 (carry)          0111 (carry)
  0000 1111 (15)             0111 (7)
+ 1111 1011 (−5)           + 0011 (3)
 ================          ==========
  0000 1010 (10)             1010 (−6) invalid!
```

- This process depends upon restricting to N bits of precision; a carry to the (nonexistent) N+1st most significant bit is ignored, resulting in the arithmetically correct result
- If the last two carry bits (the ones on far left of the top row) are both 1's or both 0's, the result is valid; if the last two carry bits are "1 0" or "0 1", a sign overflow has occurred

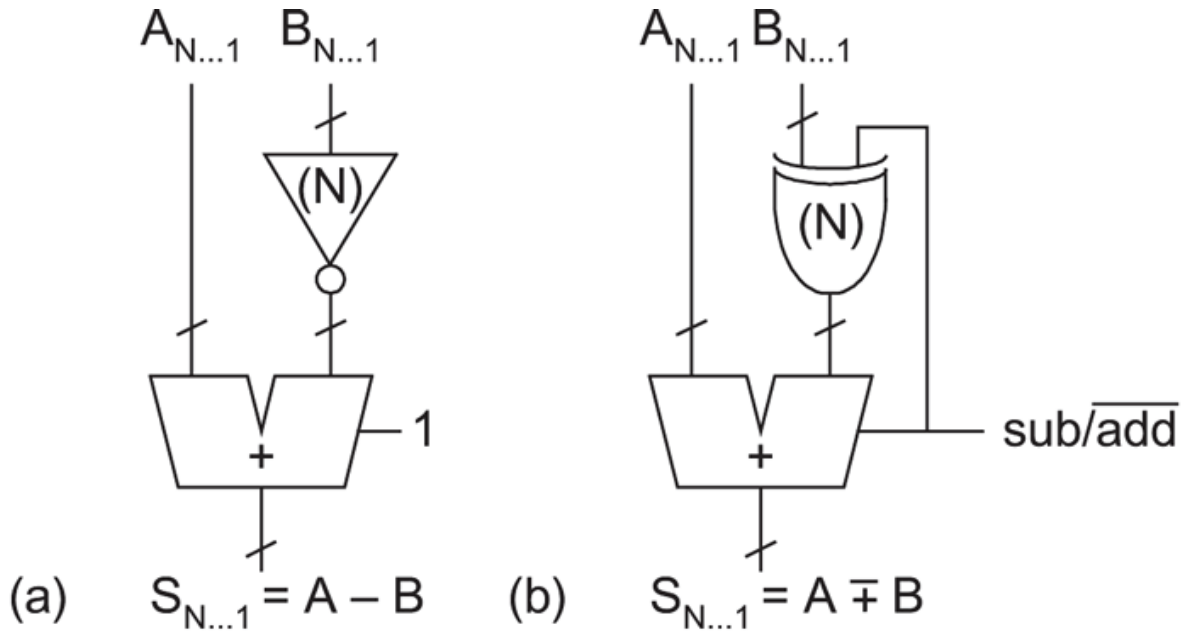
Subtraction

- Like addition, the advantage of using two's complement is the elimination of the need to examine signs of the operands to determine if addition or subtraction is needed

```
 11110 000 (borrow)        11100 000 (borrow)
  0000 1111 (15)            0000 1111 (15)
− 1111 1011 (−5)          − 0010 0011 (35)
 ===========               ===========
  0001 0100 (20)            1110 1100 (−20)
```

- Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrows; overflow has occurred when these two bits are different

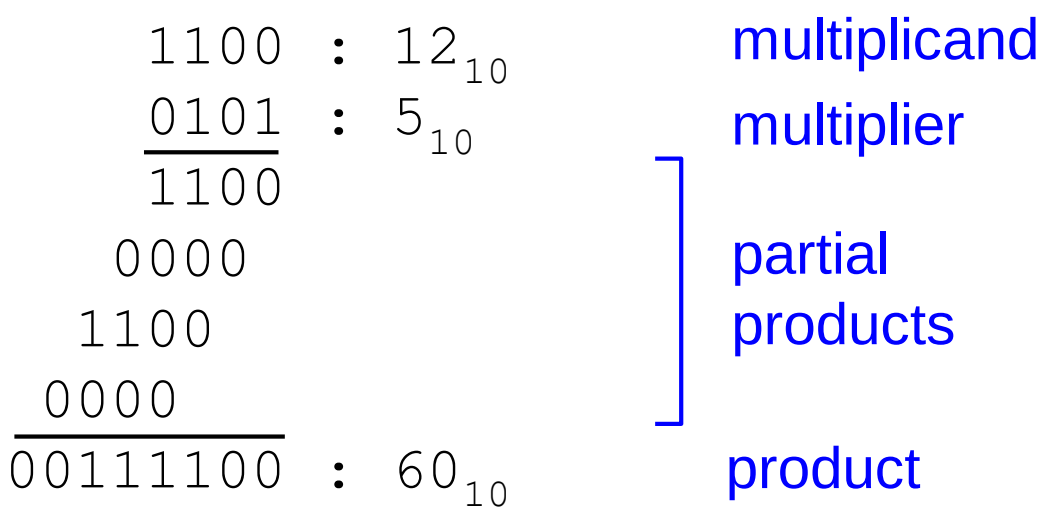
Subtraction Circuit



Subtracters

Multiplication

- **Example:**

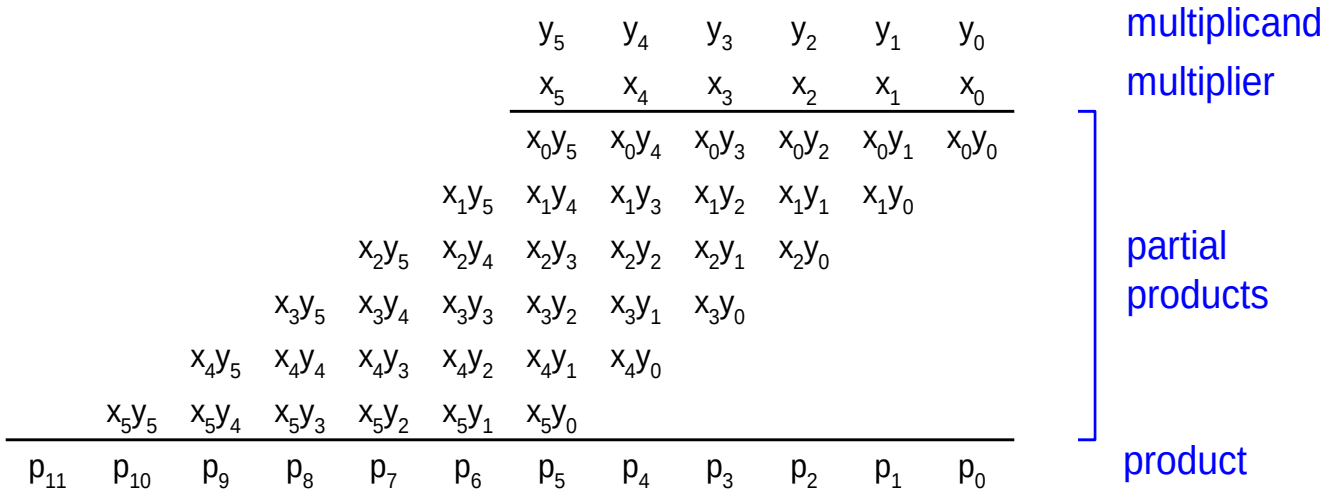


- **M x N-bit multiplication**
  - **Produce N M-bit partial products**
  - **Sum these to produce M+N-bit product**

General Form

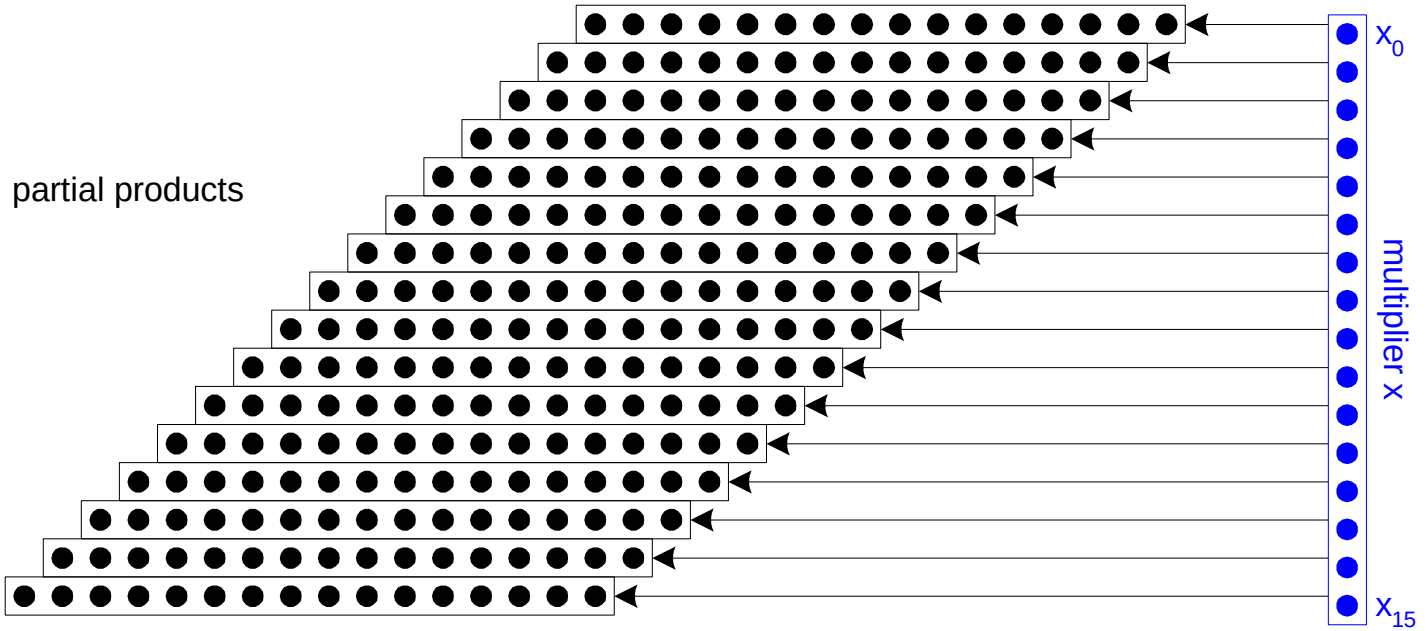
- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product:

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

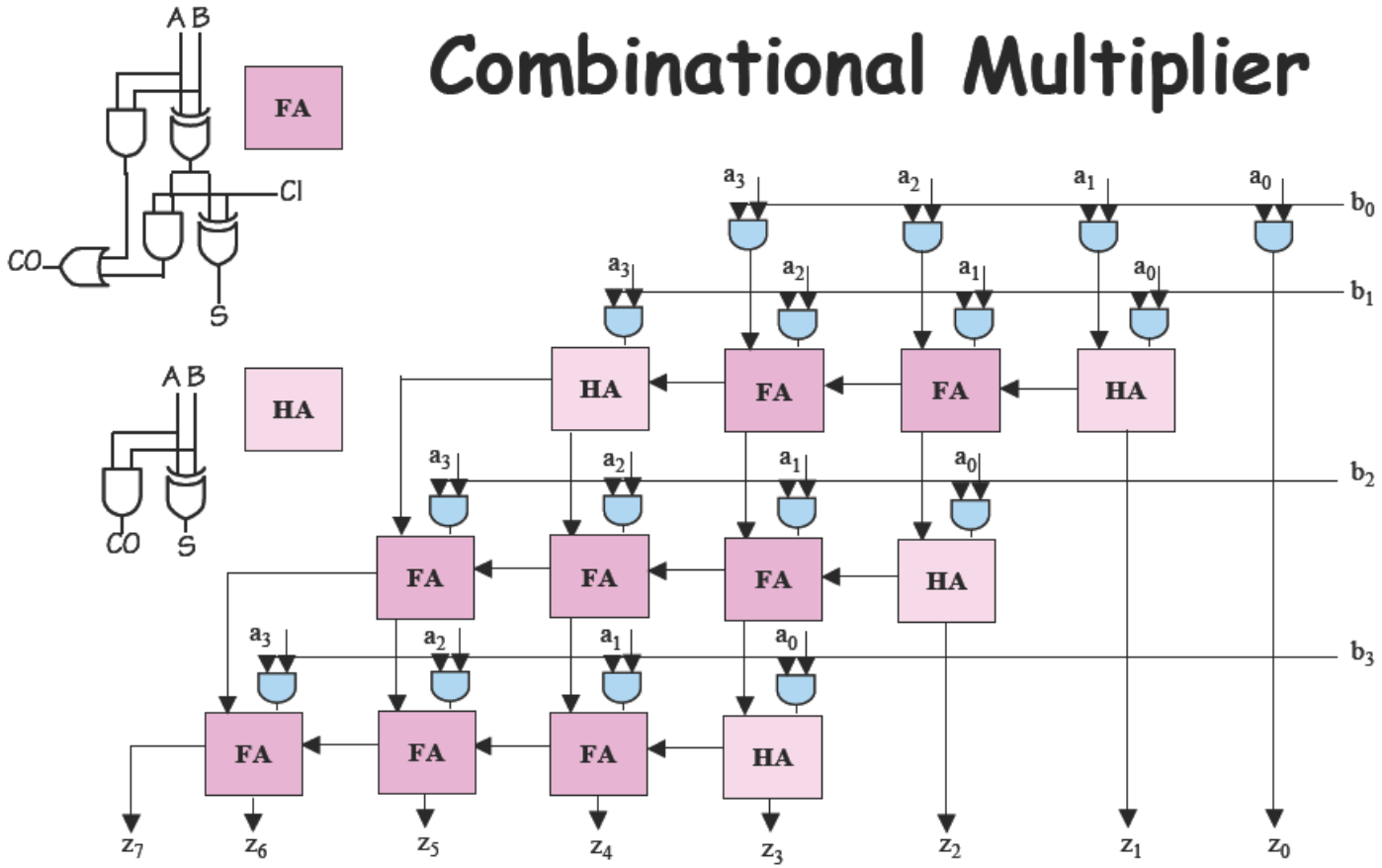


Dot Diagram

- Each dot represents a bit



Four-Bit Multiplier



Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Hardware = $\Theta(N^2)$

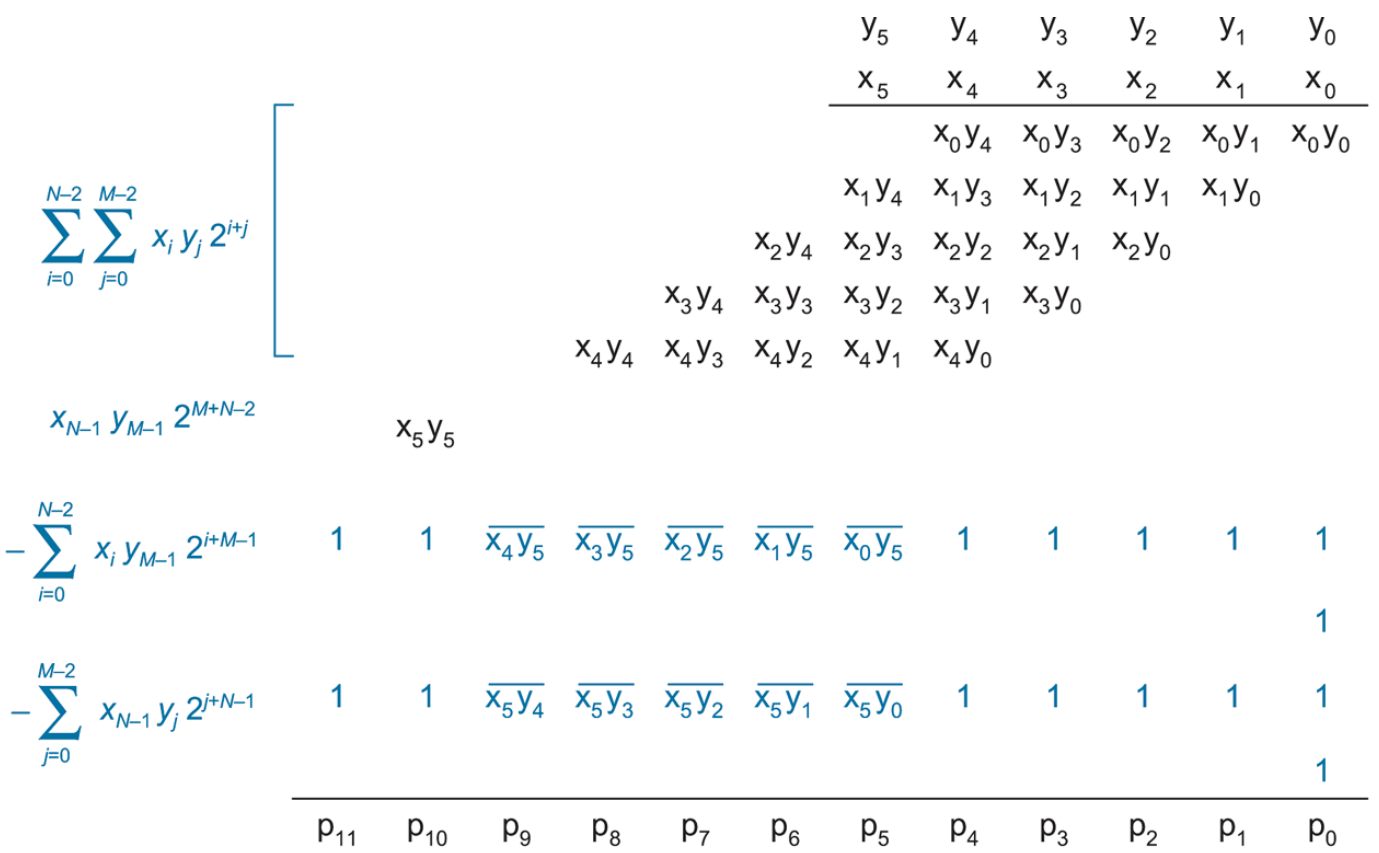
2's Complement Array MUL

- Recall that the MSB of a 2's complement number has a negative weight

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

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

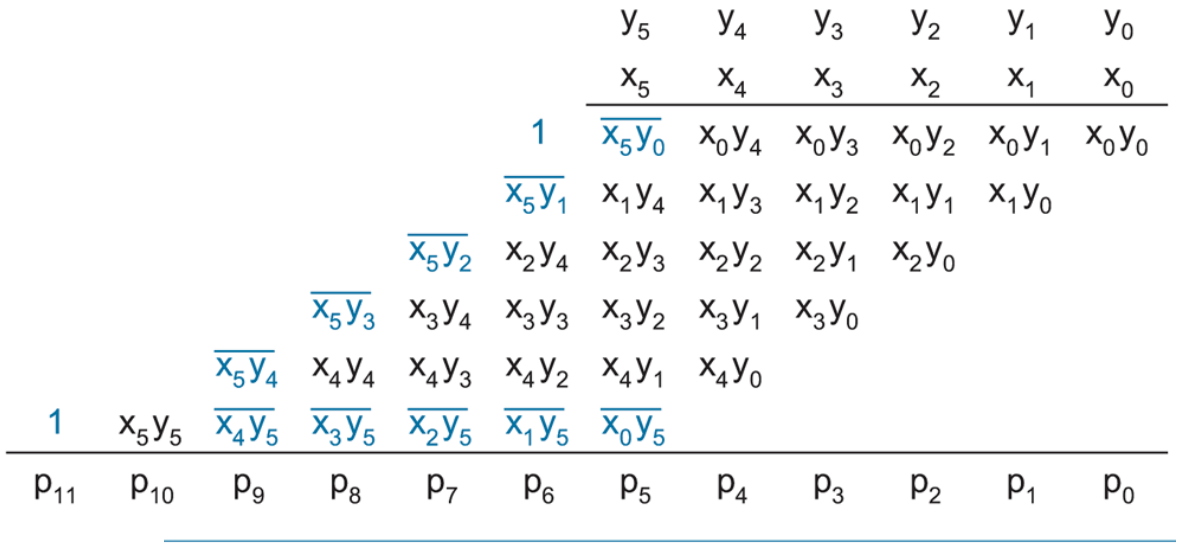
Baugh-Wooley MUL



Partial products for 2's complement multiplier

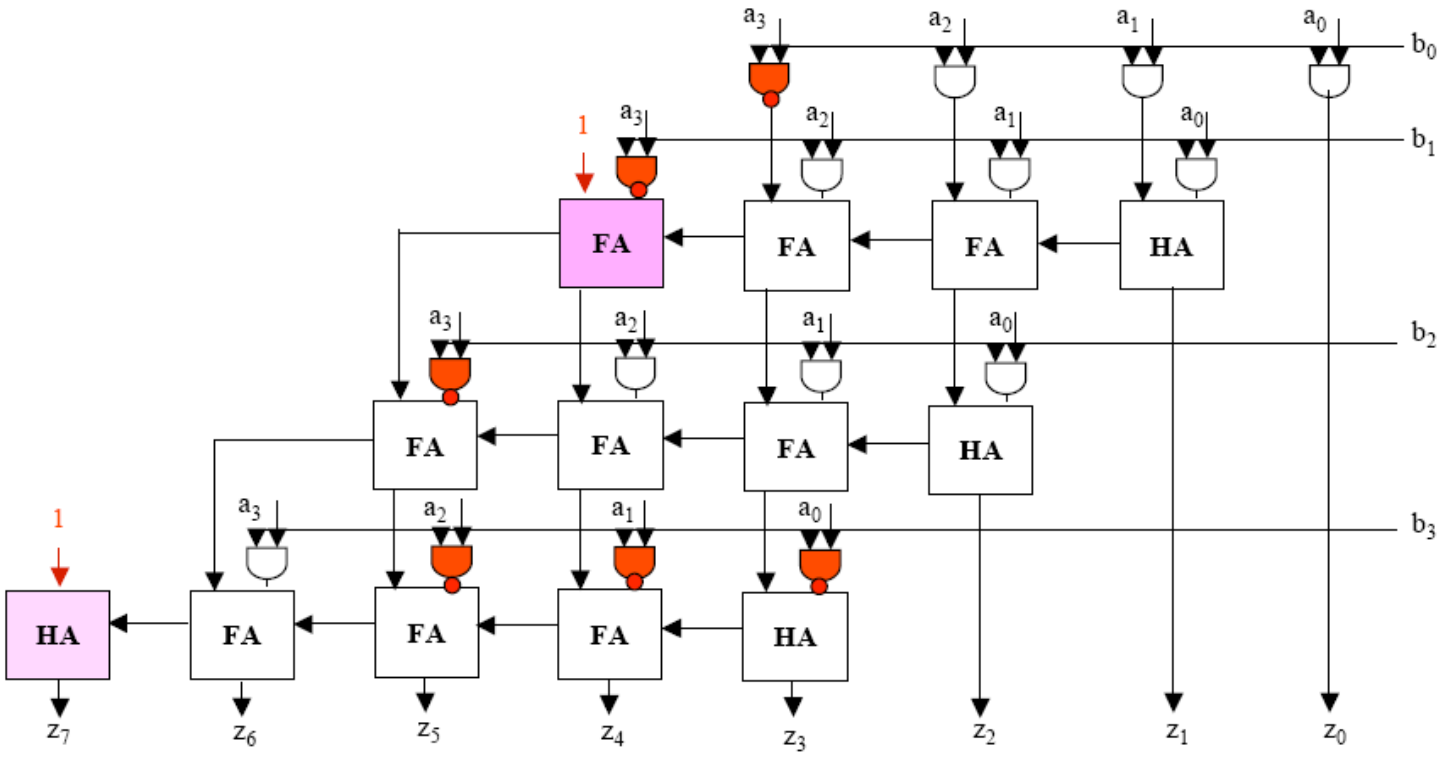
Simplified Partial Products

- Precompute the sums of the constant 1's and push some of the terms upward into extra columns

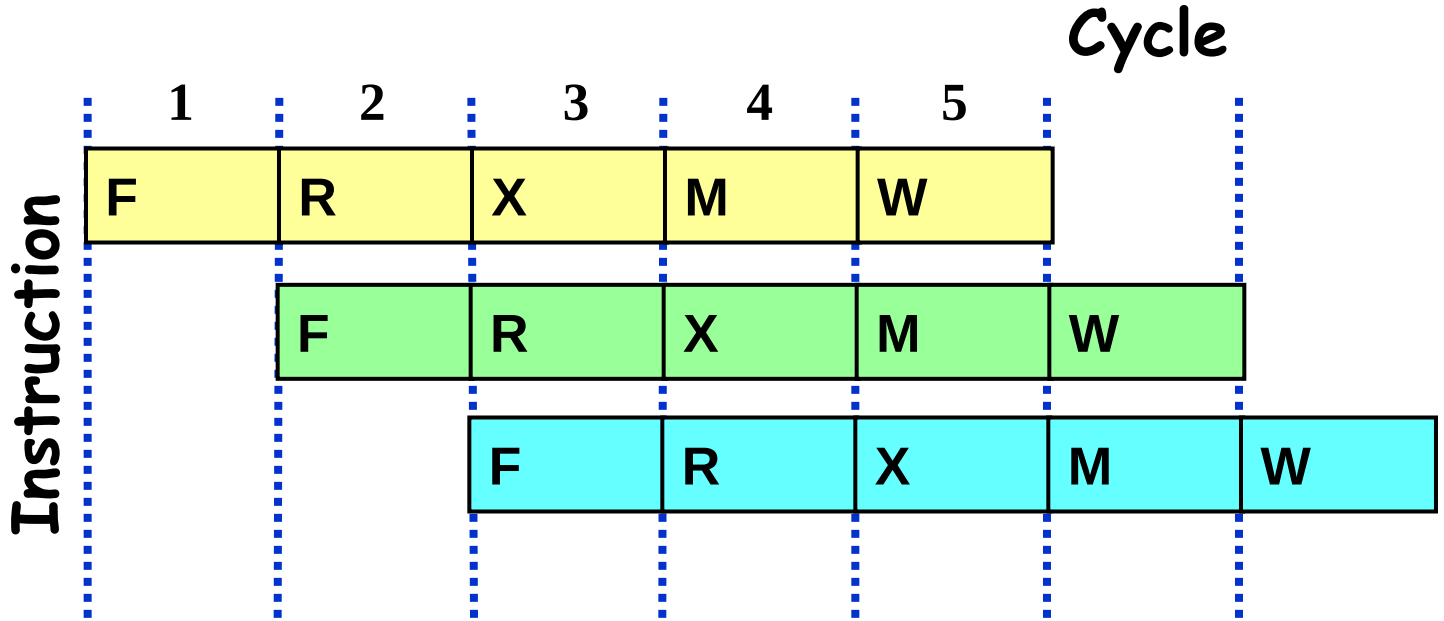


Simplified partial products for 2's complement multiplier

Four-Bit 2's Complement Multiplier

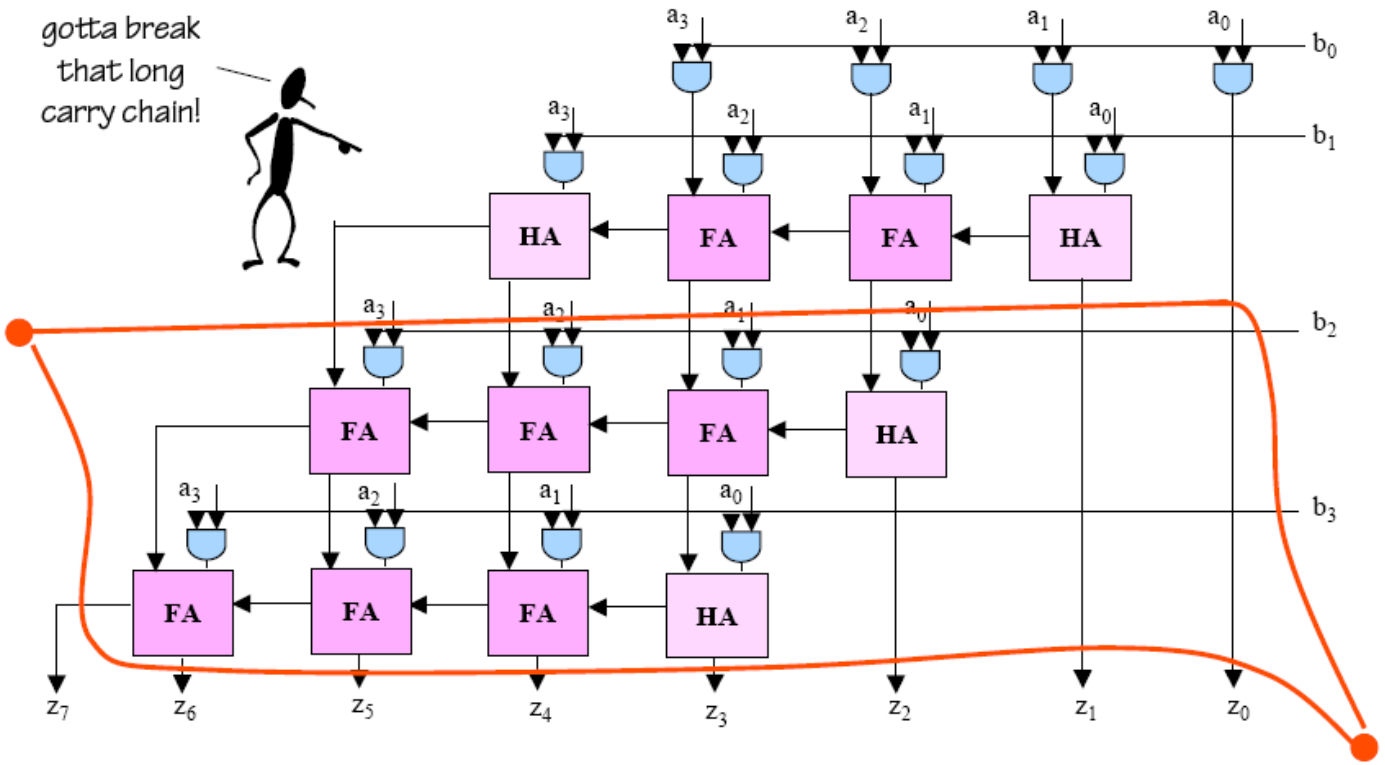


Pipelining



- Maintains latency, but increases the throughput
  - Before pipelining: $t_{clk} = t_F + t_R + t_X + t_M + t_W$
  - After pipelining: $t_{clk} = max(t_F, t_R, t_X, t_M, t_W) + t_{ck2q} + t_{setup}$

Increase Throughput with Pipelining



Before pipelining: Throughput = ~1/(2N) = Θ(1/N)

After pipelining: Throughput = ~1/N = Θ(1/N)