CHAPTER 7

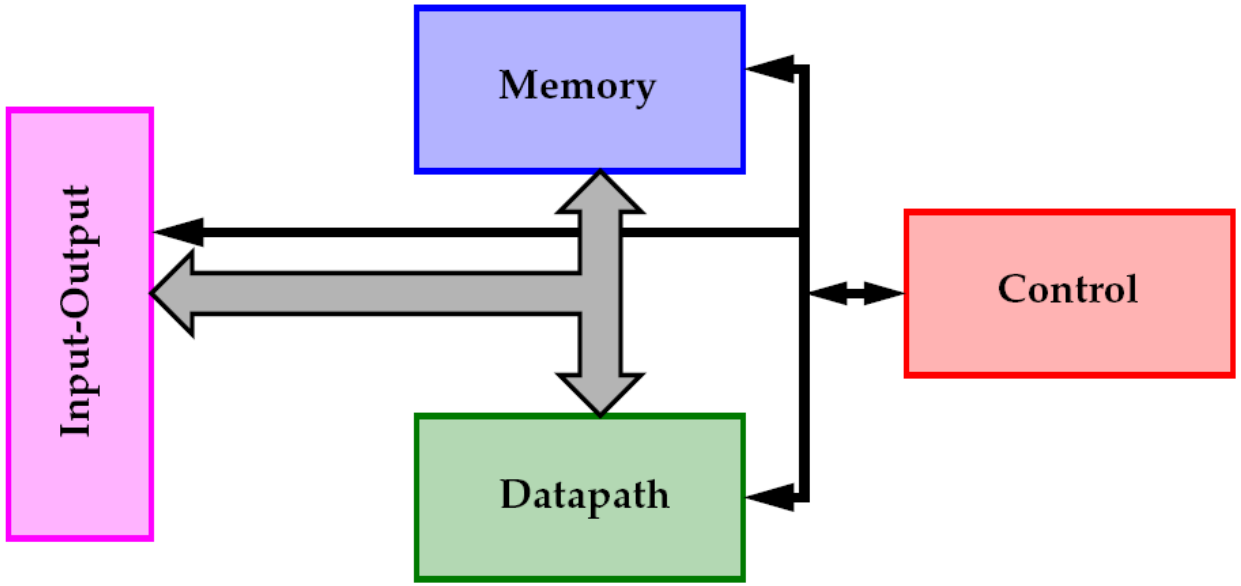
Datapath Subsystems

Outline

1. **Addition/Subtraction**
2. One/Zero Detectors
3. Comparators
4. Counters
5. Shifters
6. Multiplication

Digital Device Components

- A simple processor illustrates many of the basic components used in any digital system:



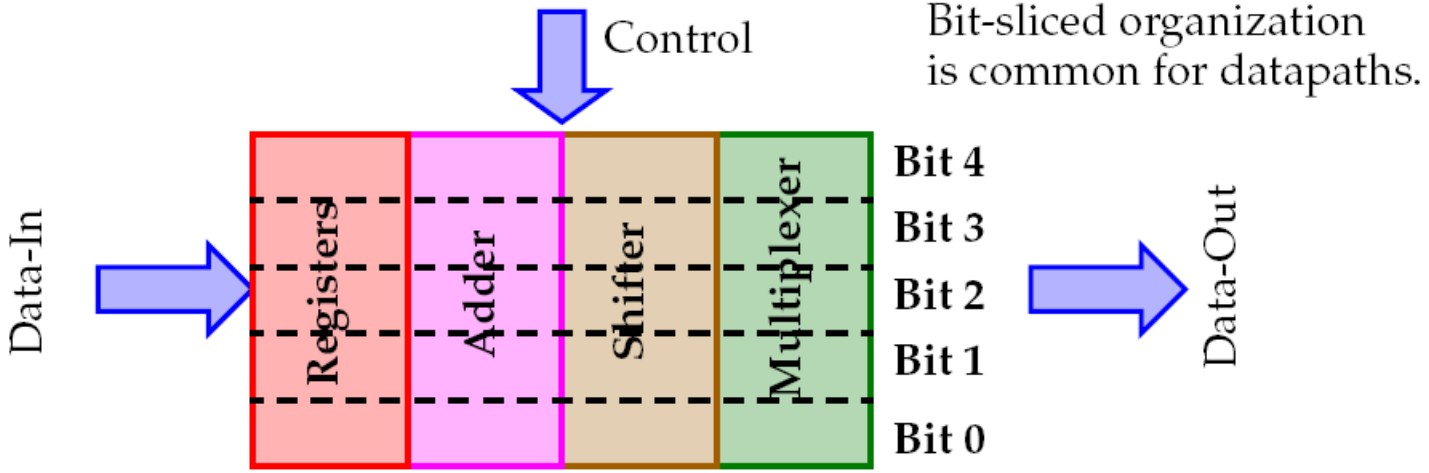
- Datapath: The core -- all other components are support units that store either the results of the datapath or determine what happens in the next cycle.

Digital Device Components

- Memory:
  - A broad range of classes exist determined by the way data is accessed:
    - Read-Only vs. Read-Write
    - Sequential vs. Random access
    - Single-ported vs. Multi-ported access
  - Or by their data retention characteristics:
    - Dynamic vs. Static
  - Stay tuned for a more extensive treatment of memories.
- Control:
  - A FSM (sequential circuit) implemented using random logic, PLAs or memories.
- Interconnect and Input-Output:
  - Parasitic resistance, capacitance and inductance affects performance of wires both on and off the chip.
  - Growing die size increases the length of the on-chip interconnect, increasing the value of the parasitic.

Digital Device Components

- Datapath elements include adder, multiplier, shifter, etc.
  - The speed of these elements often dominates the overall system performance so optimization techniques are important.
  - However, as we will see, the task is non-trivial since there are multiple equivalent logic and circuit topologies to choose from, each with adv./disadv. in terms of speed, power and area.
  - Also, optimizations focused at one design level, e.g., sizing transistors, leads to inferior designs.

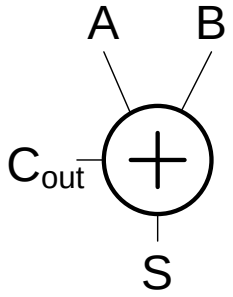


Single-Bit Addition

Half Adder

$$S = A \oplus B$$

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



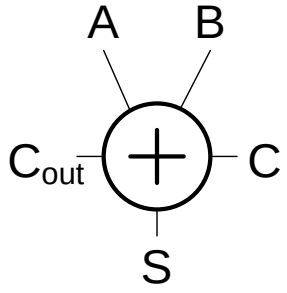
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder

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

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

$$= AB + BC + AC$$



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 • B
  - Propagate: C_{out} = C
    - P = A ⊕ B
  - Kill: C_{out} = 0 independent of C
    - K = ~A • ~B

A	B	C	G	P	K	C_{out}	S
0	0	0	0	0	1	0	0
		1				0	1
0	1	0	0	1	0	0	1
		1				1	0
1	0	0	0	1	0	0	1
		1				1	0
1	1	0	1	0	0	1	0
		1				1	1

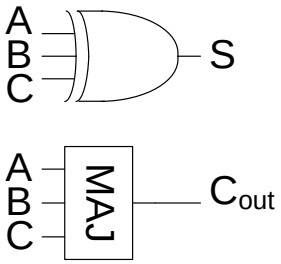
Full Adder Design I

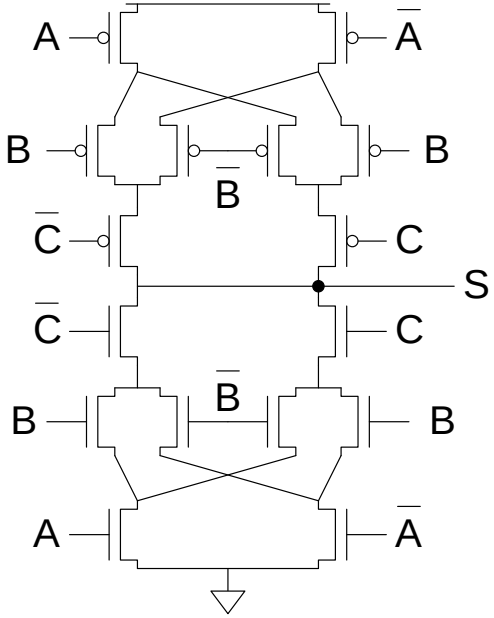
- Brute force implementation from eqns

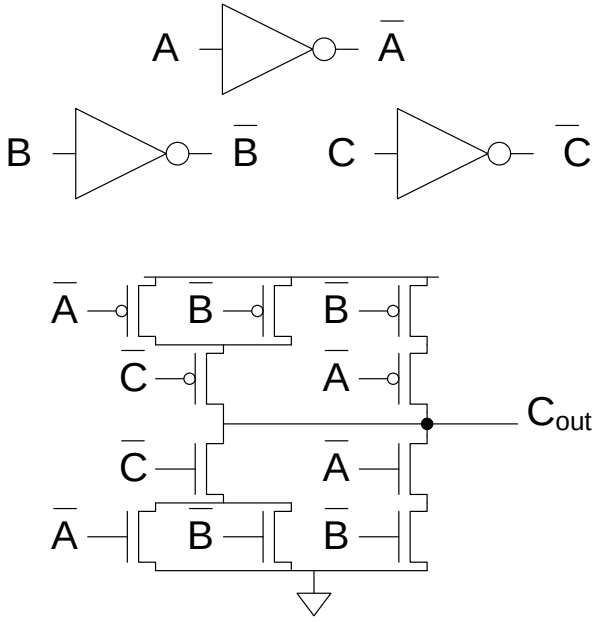
$$S = A \oplus B \oplus C \qquad \text{(16 transisotrs)}$$

$$C_{\text{out}} = MAJ(A,B,C) \qquad \text{(10 transisors)}$$

INV (6 transistors)

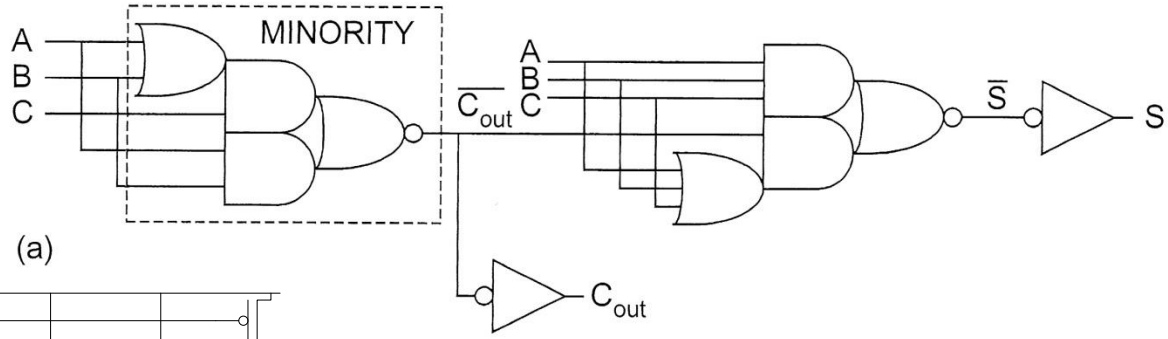




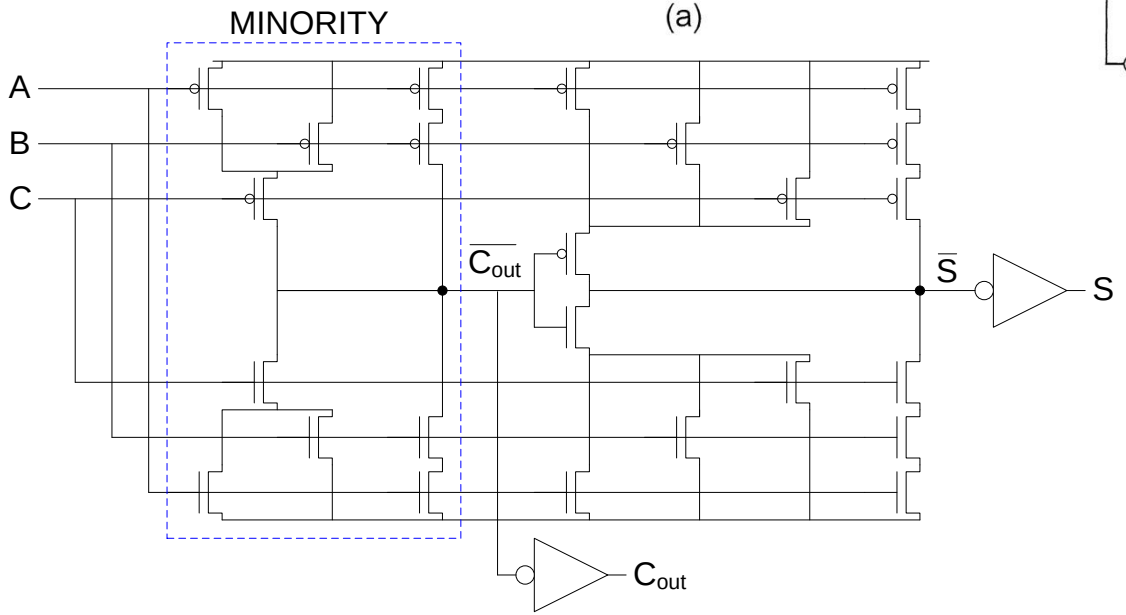


Full Adder Design II

- Factor S in terms of C_{out} : S = ABC + (A + B + C)(~C_{out})
- Critical path is usually C to C_{out} in ripple adder
  - 28 transistors

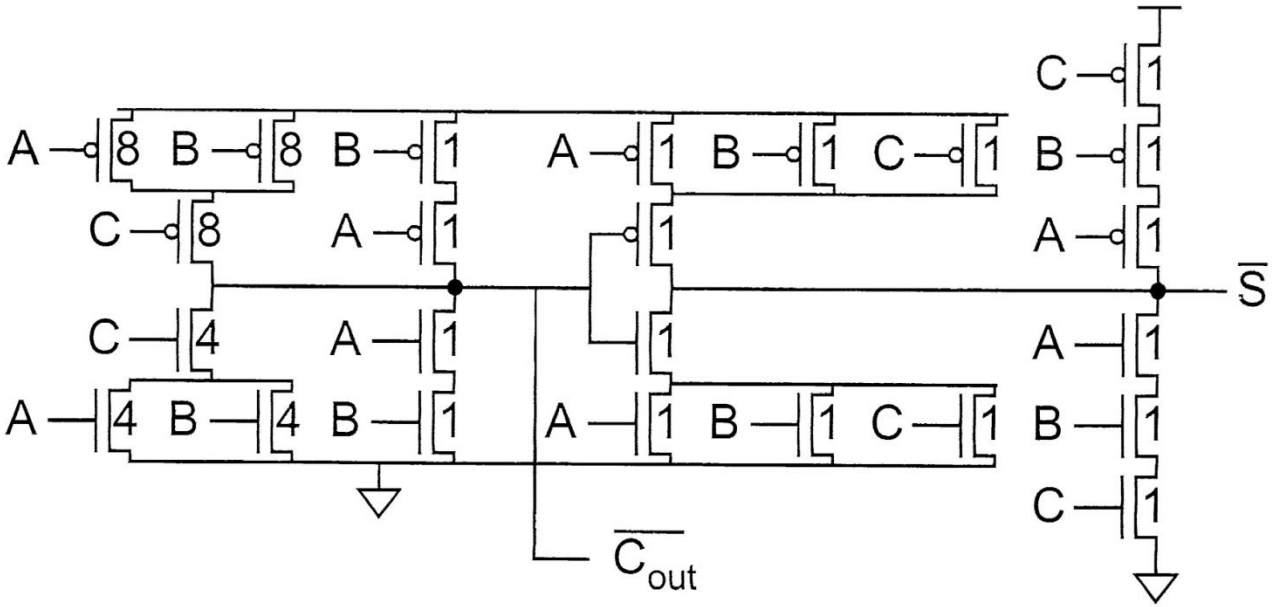


(a)



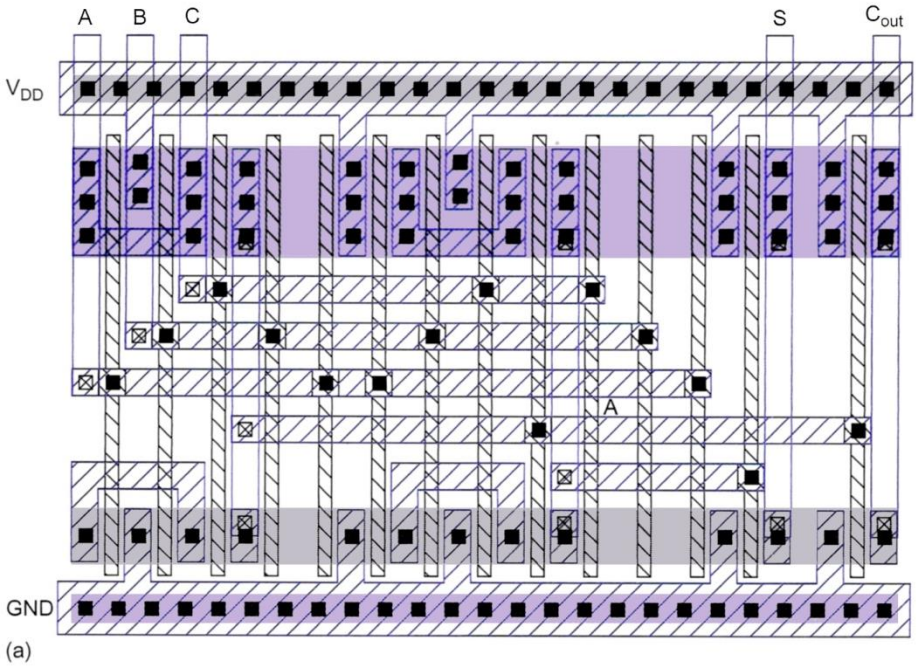
Full Adder Design II

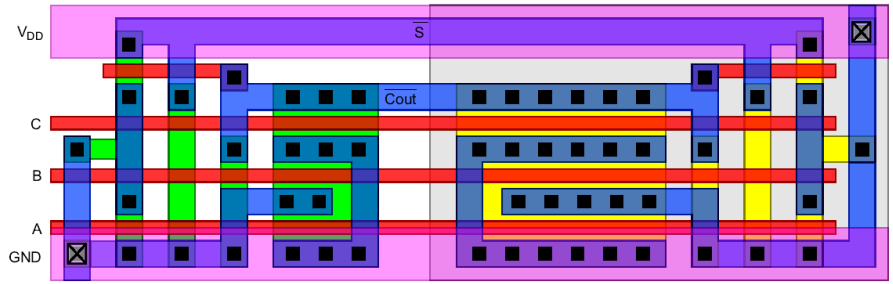
- Feed carry-in (C) to the inner inputs.
- Use minimum size at summing (S) logic for minimizing branching effort on the critical path.
- Asymmetric gate to reduce the logical effort from C ->~C_{out}
- Eliminate output inverter: 24 transistors

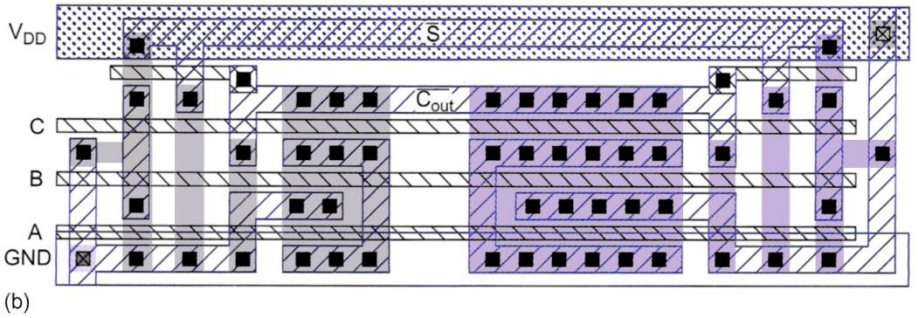


Layout

- Standard cell layout
  - Fixed pMOS (nMOS) location & pitch
- Datapath layout
  - Smaller bit-pitch (height) with wider MOS size

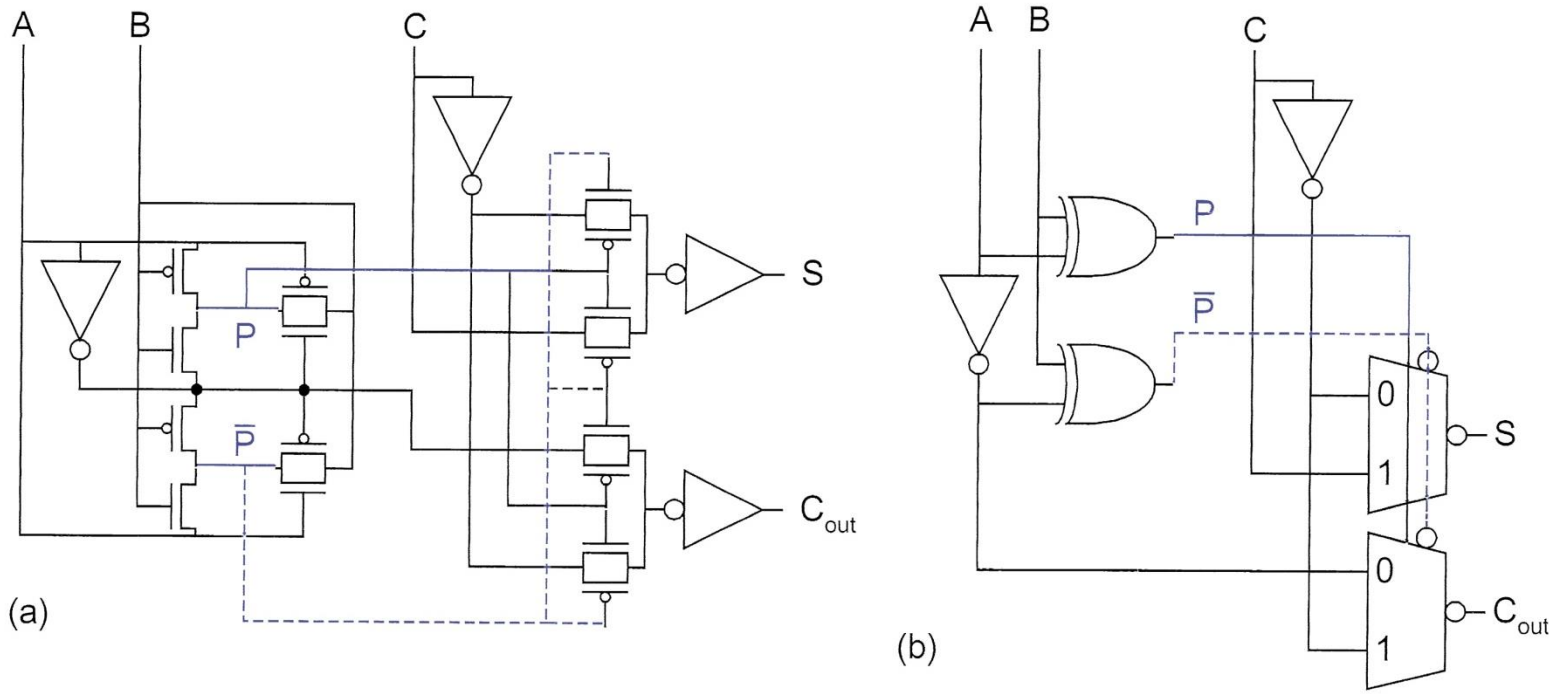






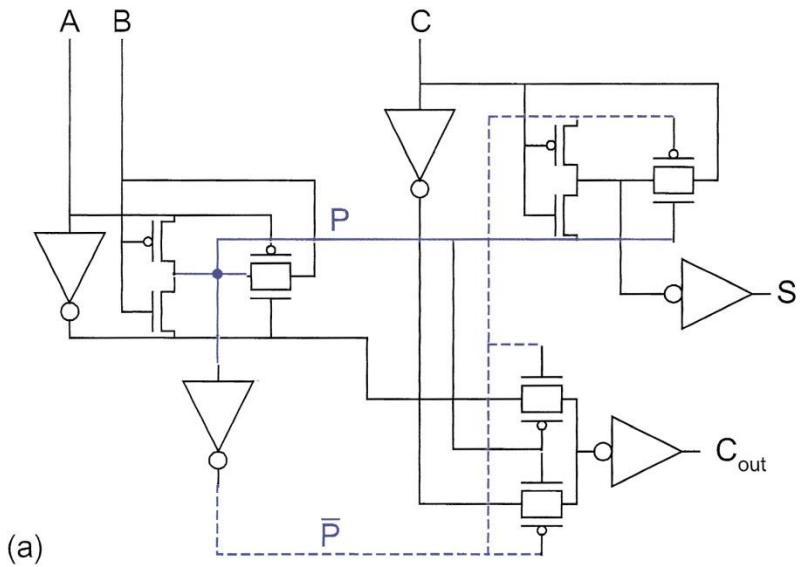
Full Adder Design III

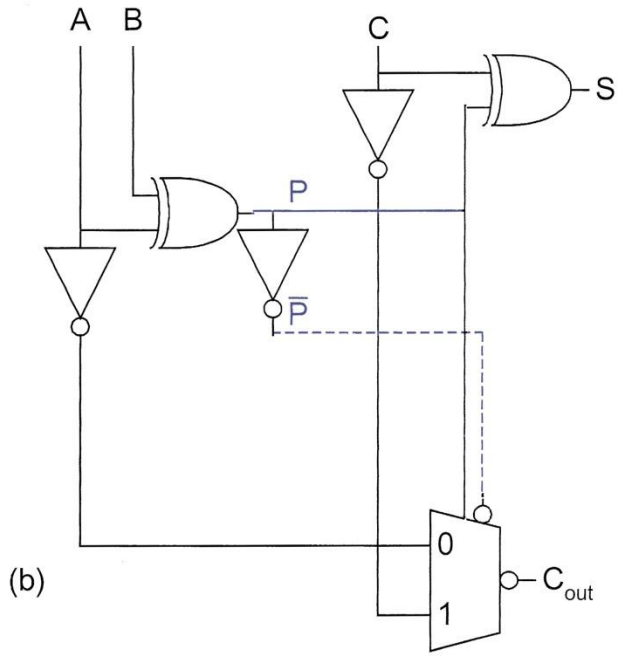
- Transmission gate full adder
  - 24 Transistors, providing buffered outputs of proper polarity with equal delay.



Full Adder Design IV

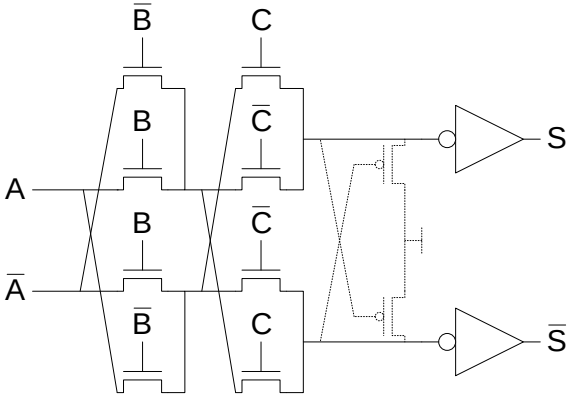
- Zhuang full adder
  - 22 Transistors, extra inverter delay by computing ~P from P.

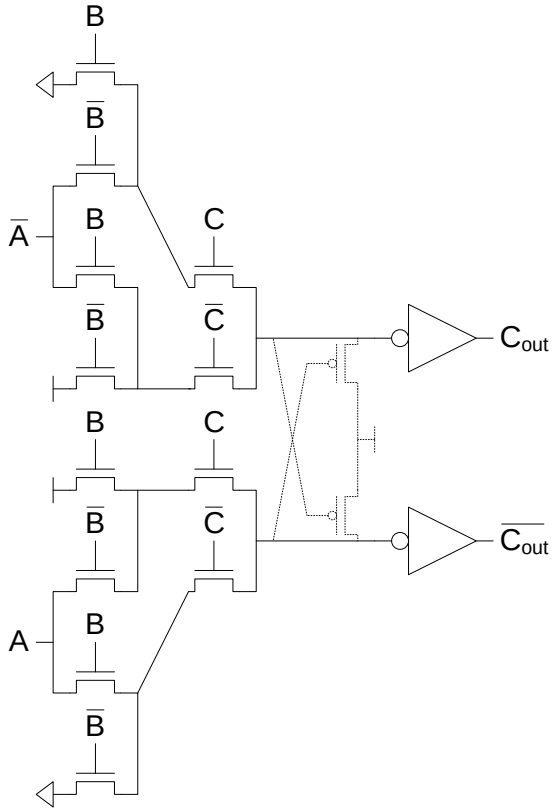




Full Adder Design V

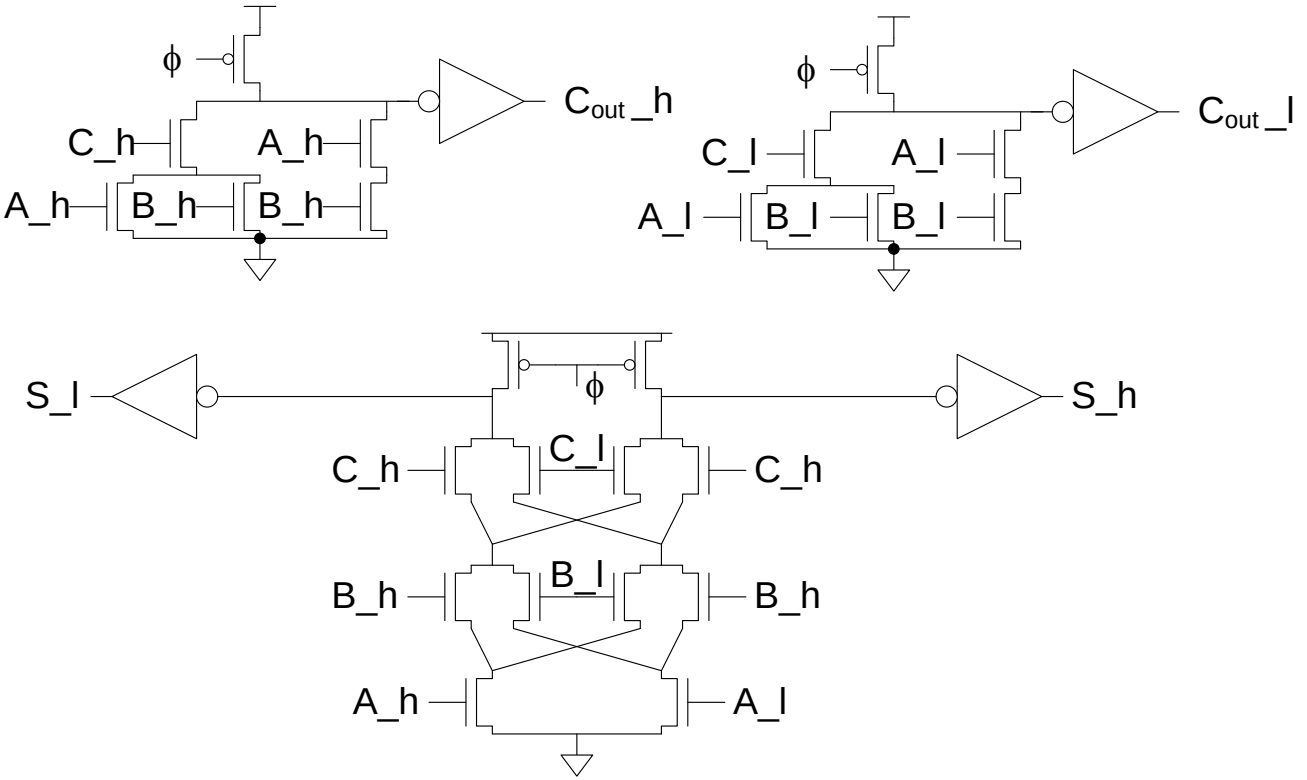
- Complementary Pass Transistor Logic (CPL)
  - Compared to I : x2 faster, 30% lower power.
  - Compared to II: Slightly faster, but more area





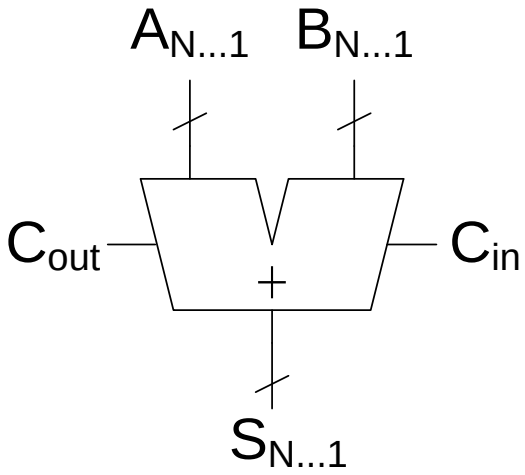
Full Adder Design VI

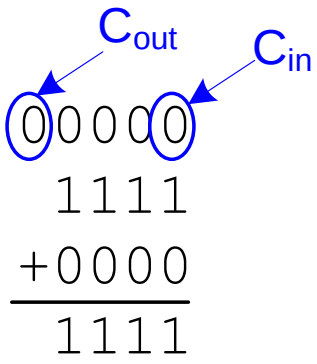
- Footless dual-rail domino
- Very fast, but large and power hungry
- Used in very fast multipliers

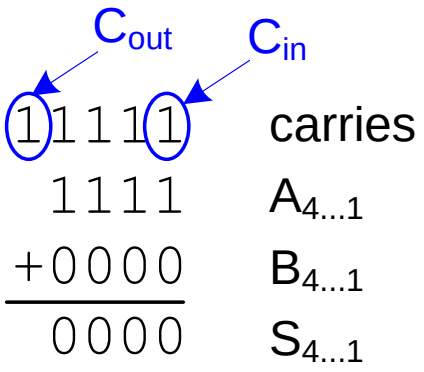


Carry Propagate Adders

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

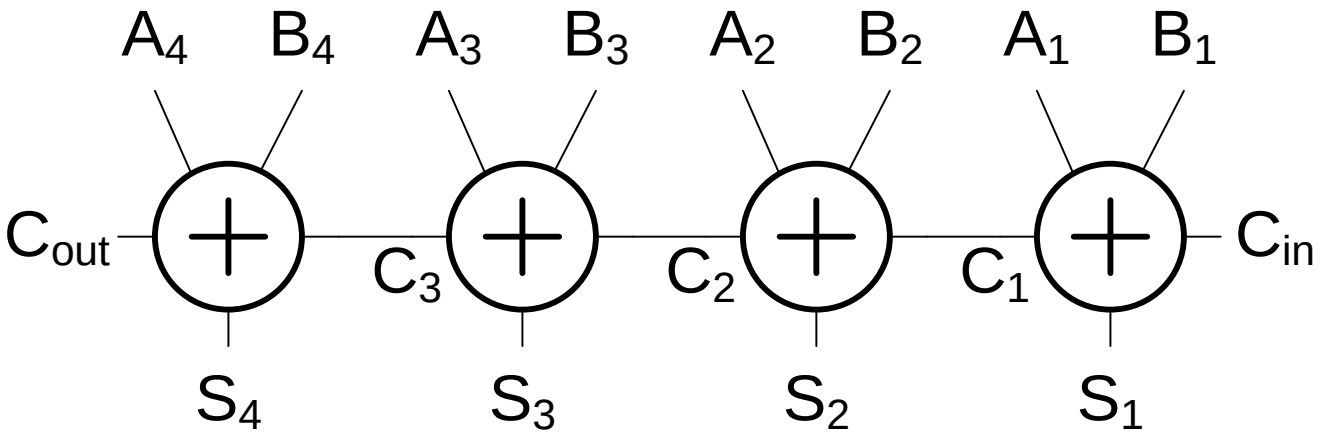






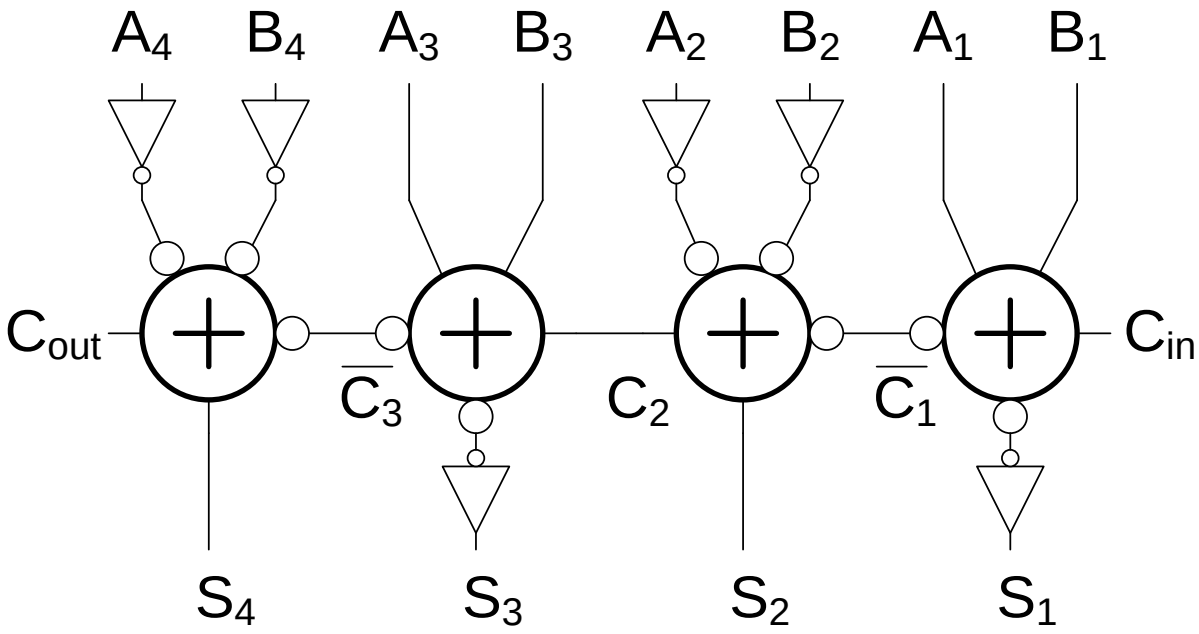
Carry-Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out} (minimize $t_{C \rightarrow Cout}$)
  - 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

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

$$P_{i:j} = P_{i:k} \bullet P_{k-1:j}$$

Valency-2 group PG Logic

- Base case

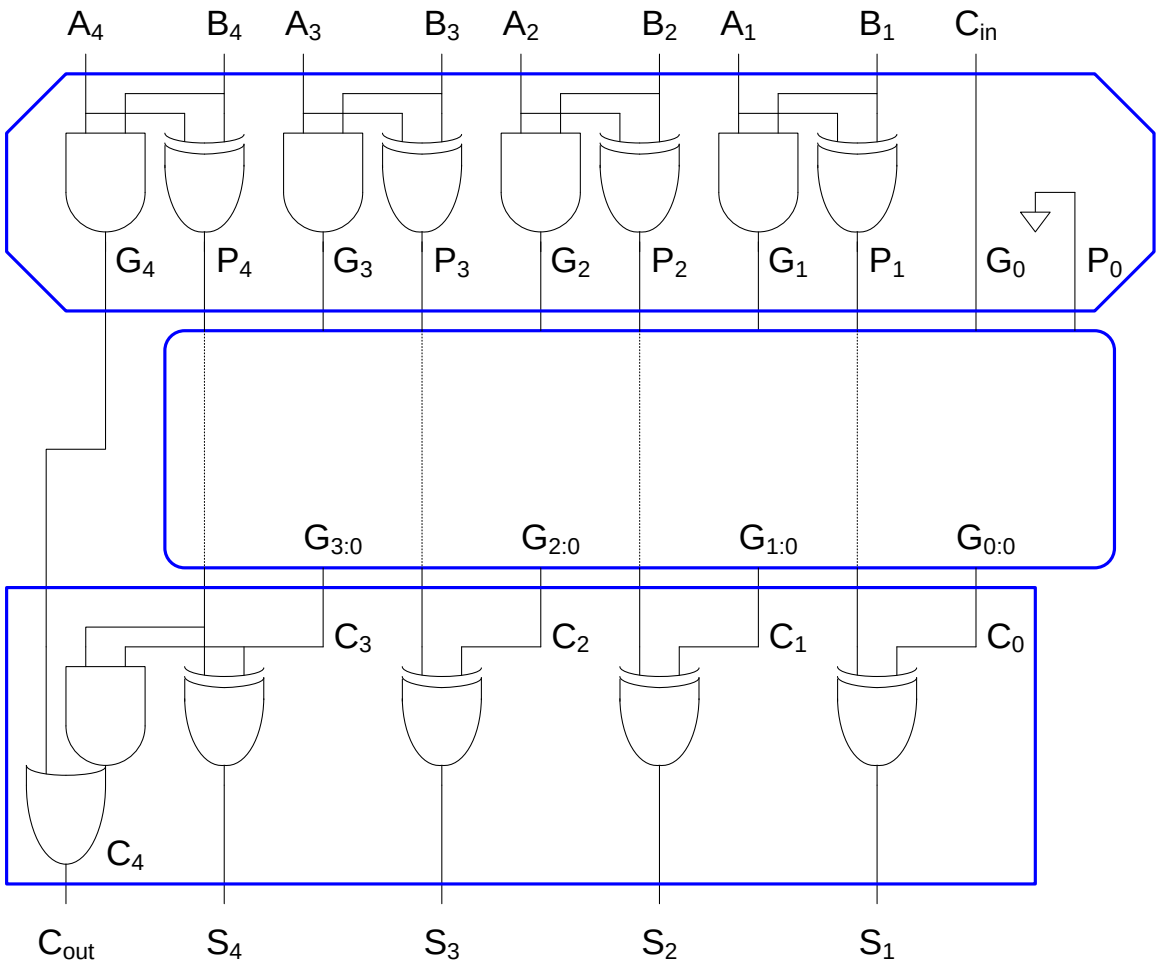
$$G_{i:i} \equiv G_i = A_i \bullet B_i \qquad G_{0:0} \equiv G_0 = C_{in}$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} \equiv P_0 = 0$$

- Sum:

$$C_{i-1} = G_{i-1:0}, \quad S = P \oplus C, \quad S_i = P_i \oplus G_{i-1:0}$$

PG Logic



1: Bitwise PG logic

2: Group PG logic

3: Sum logic

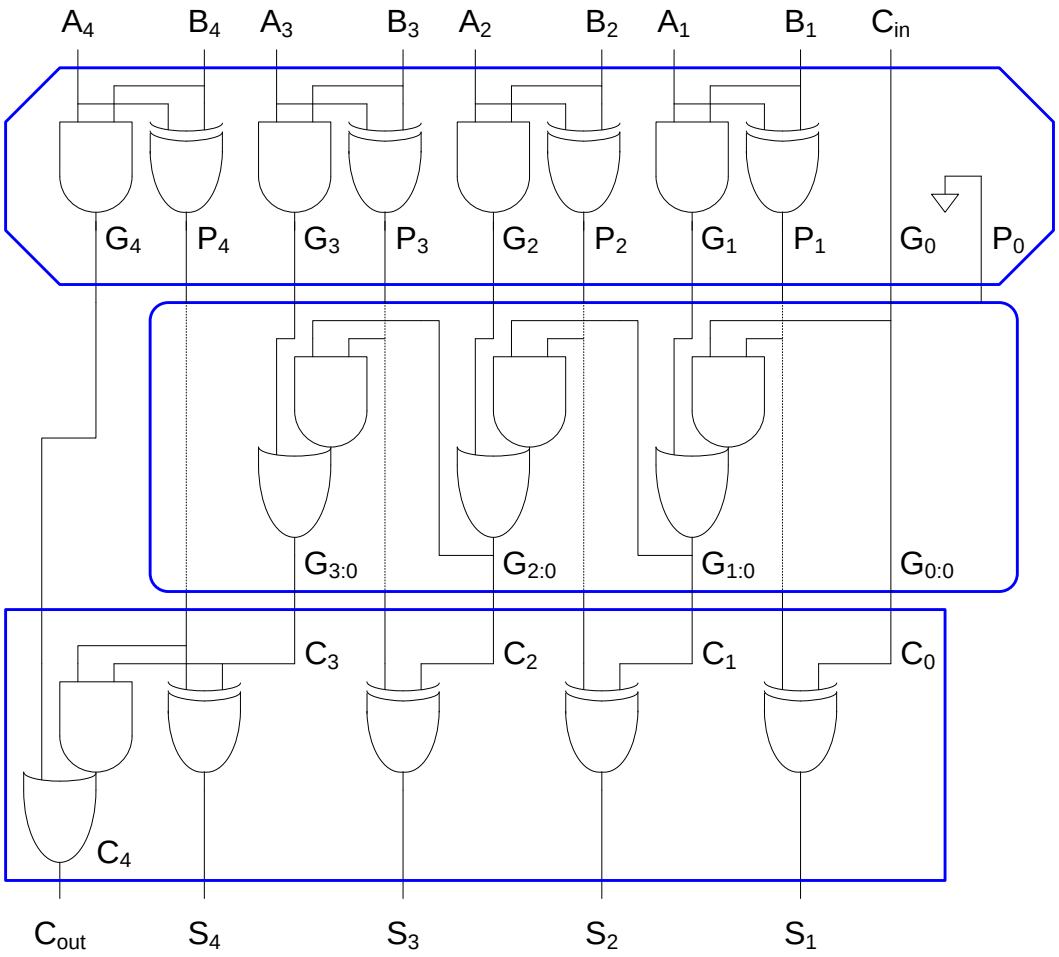
Carry-Ripple Revisited

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

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

$$C_{out} = G_{i:0}$$

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



Carry-Ripple PG Diagram

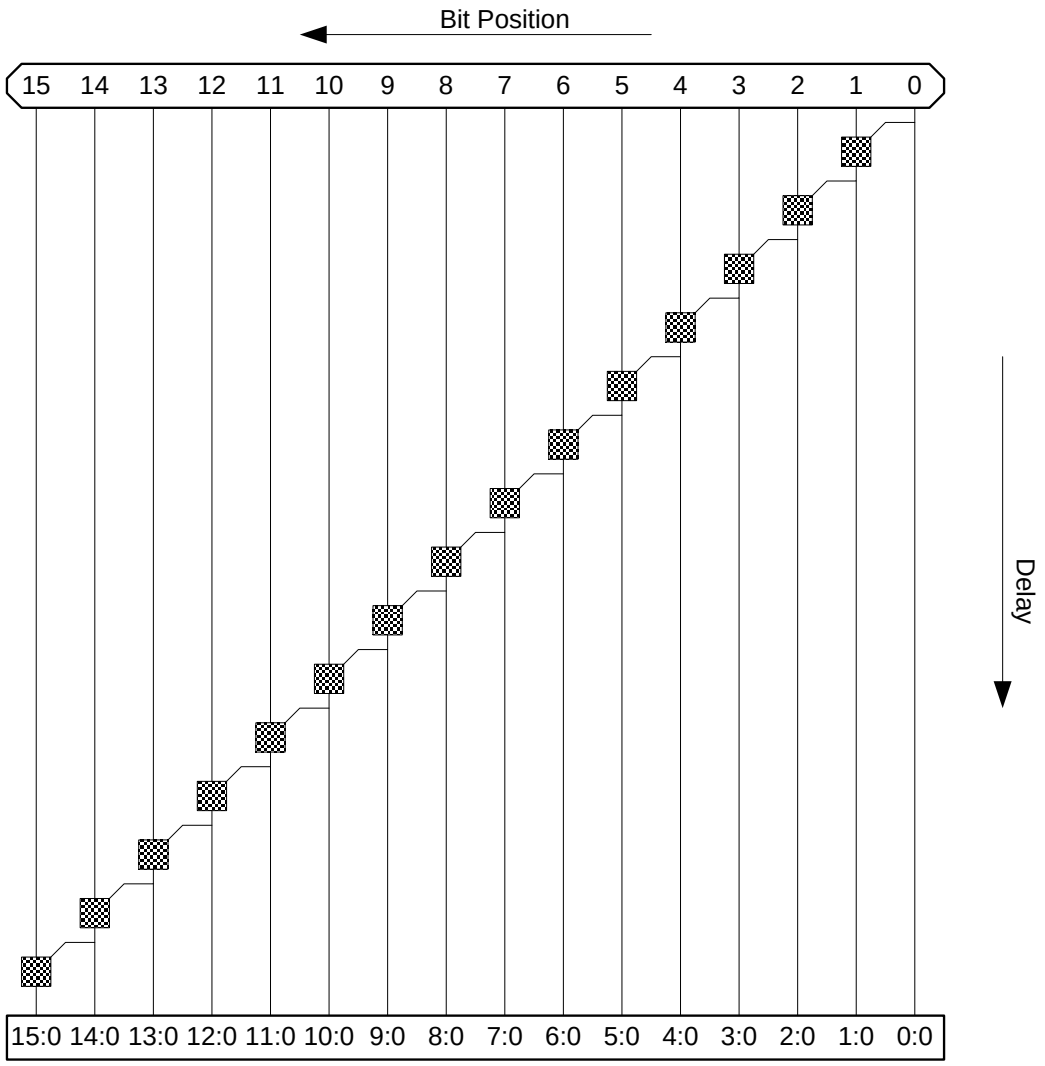
For N-bit adder

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

t_{pg} : delay of 1-bit propagate /generate gate

t_{AO} : delay of AND-OR gate in grey cell

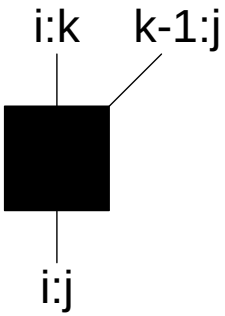
t_{xor} : delay of final sum XOR

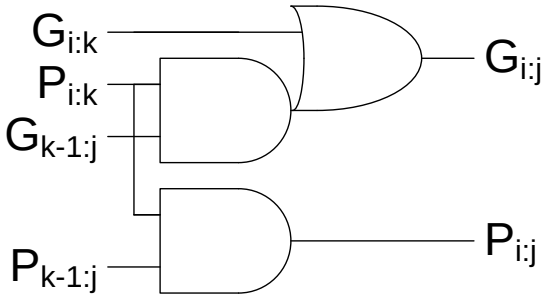


PG Diagram Notation

Group generate & propagate (AND-OR & AND)

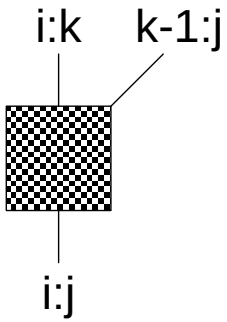
Black cell

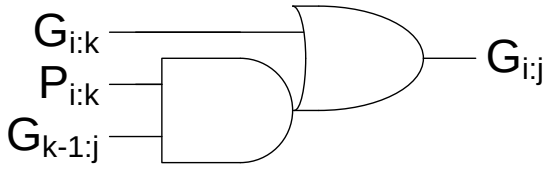




Group generate only (AND-OR)

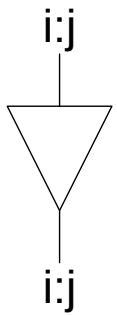
Gray cell

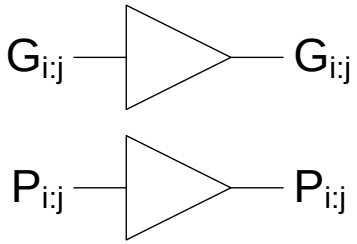




Minimize the load on critical path

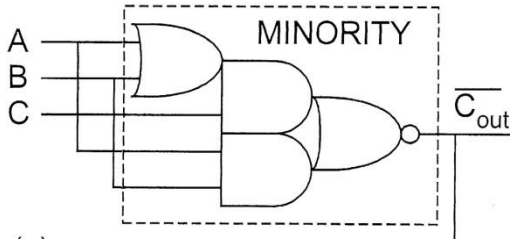
Buffer

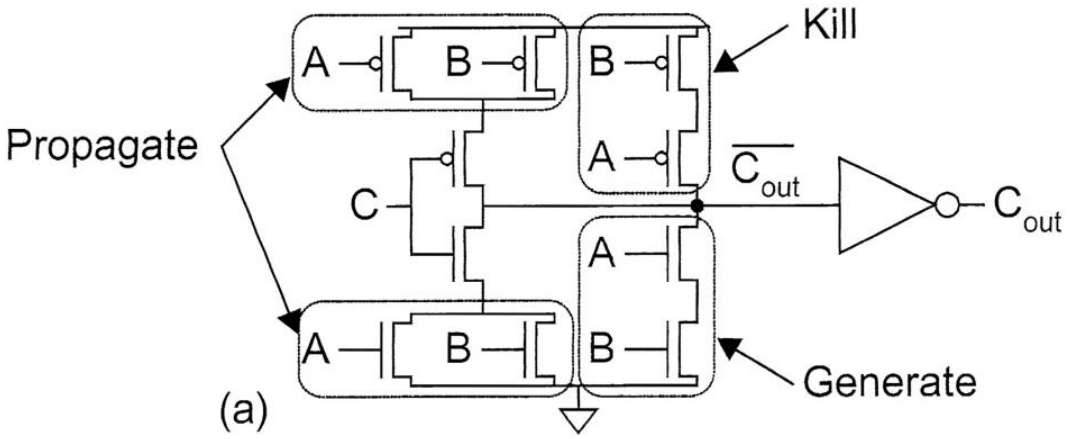




Manchester Carry Chain Adder

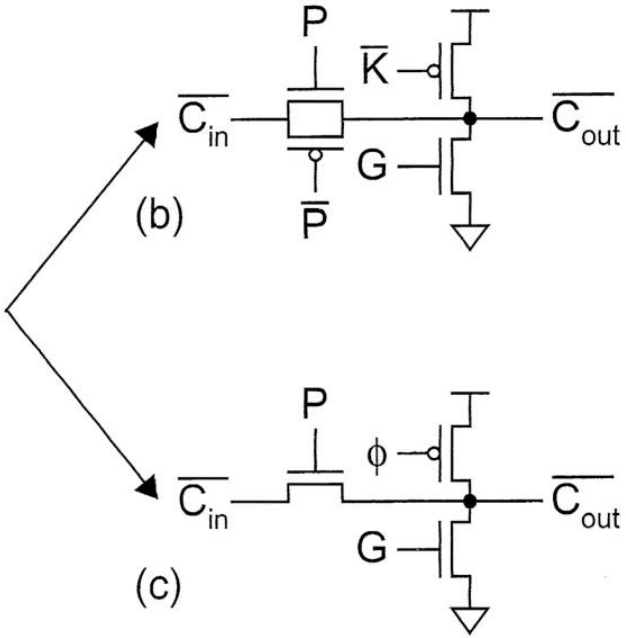
Majority gate





Pass Transistor Logic

Static

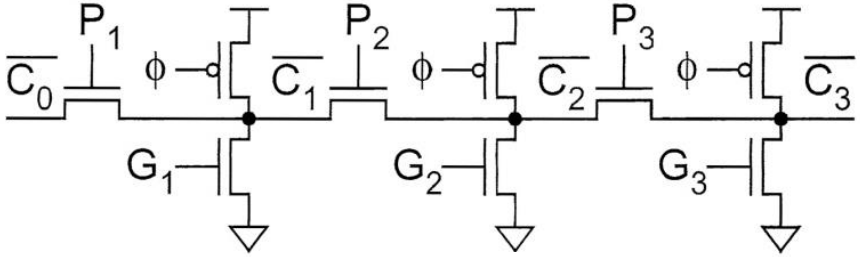


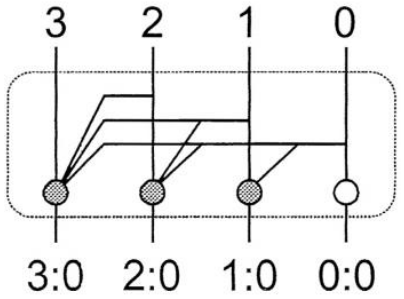
Dynamic

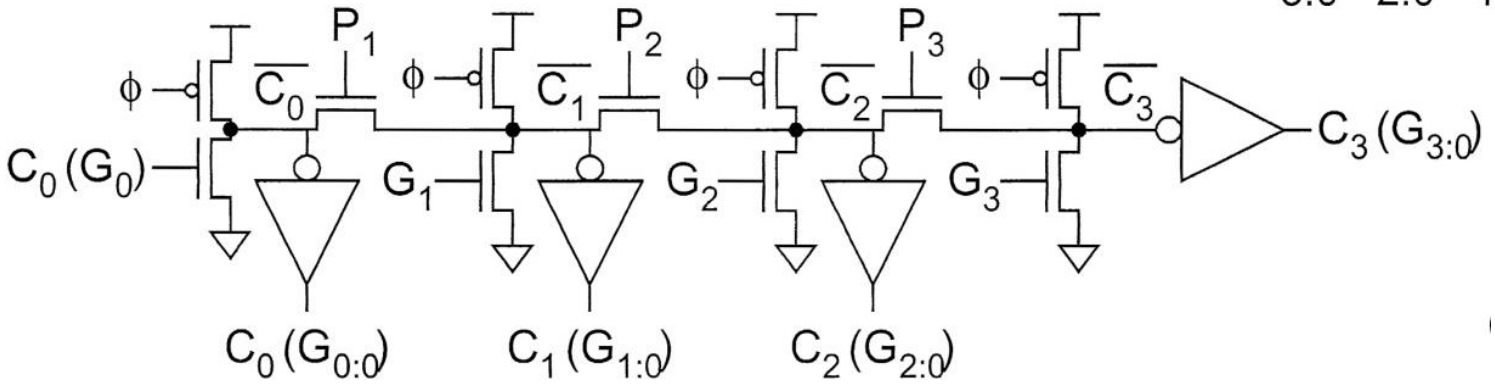
Manchester Carry Chain

- Valency-4 group generate

$$C_0 = G_{0:0} = C_0$$
$$C_1 = G_{1:0} = G_1 + P_1 C_0$$
$$C_2 = G_{2:0} = G_2 + P_2 \left(G_1 + P_1 C_0 \right)$$
$$C_3 = G_{3:0} = G_3 + P_3 \left(G_2 + P_2 \left(G_1 + P_1 C_0 \right) \right)$$

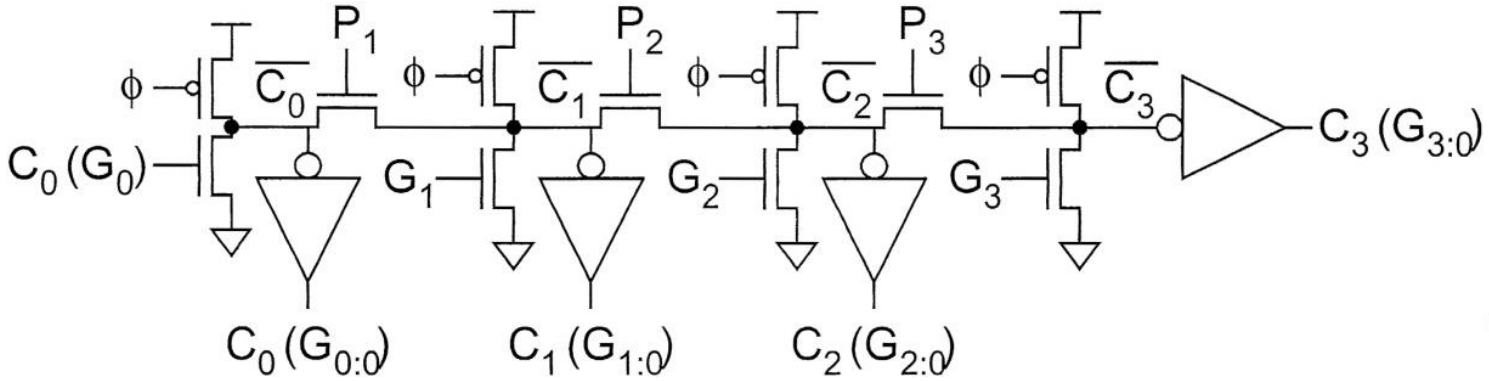


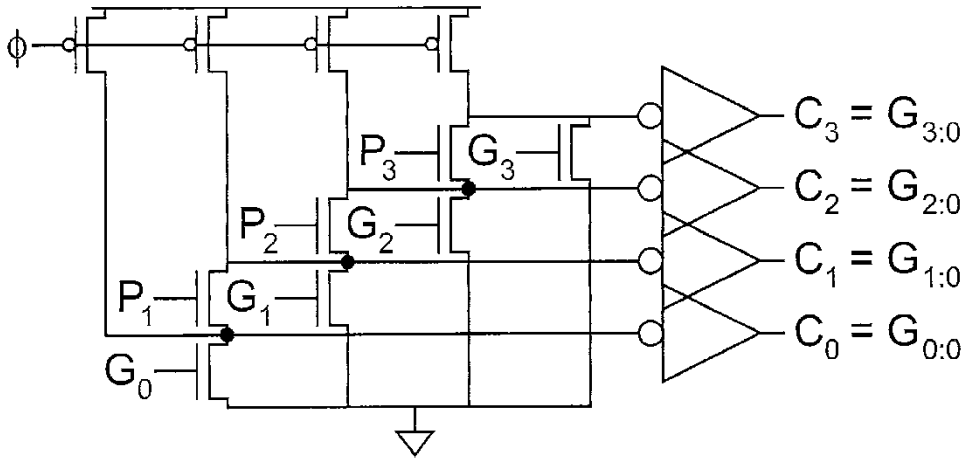




Manchester Carry Chain - Domino

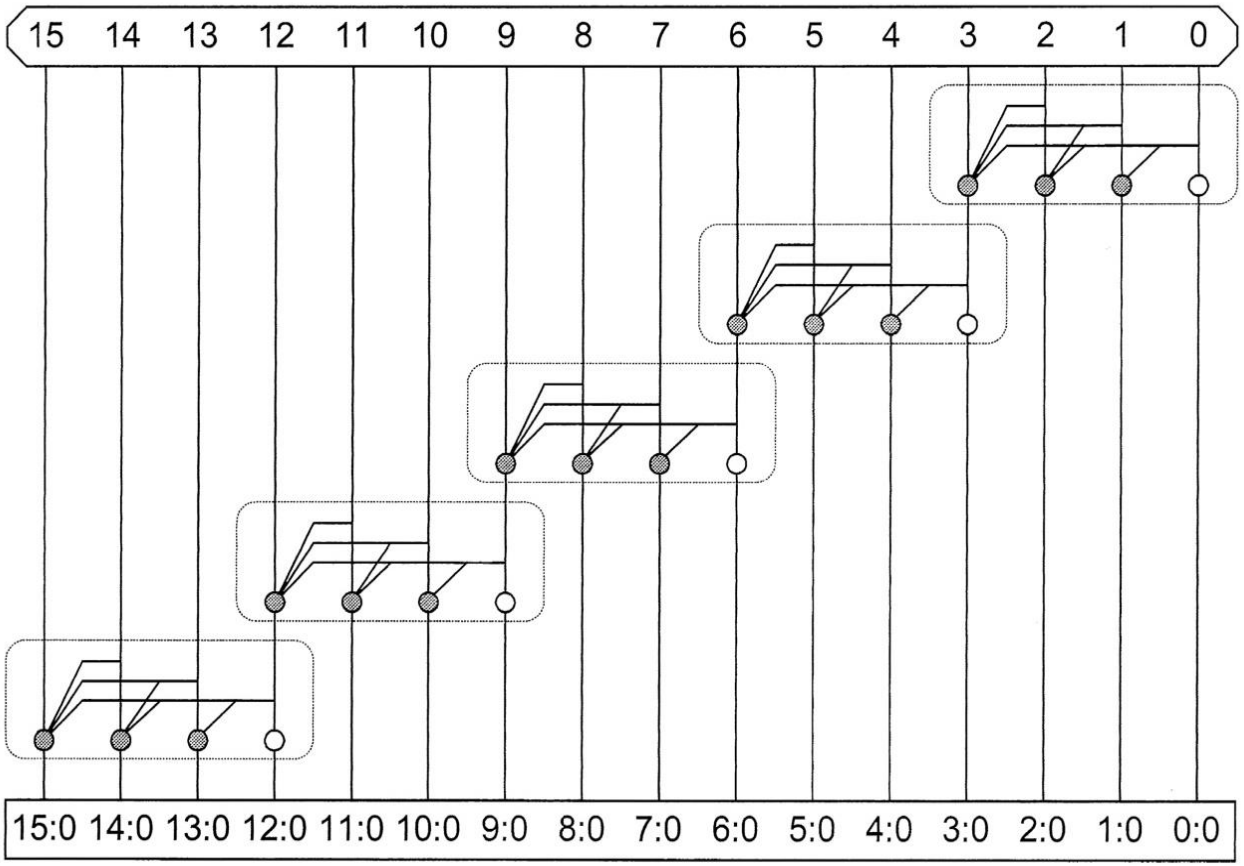
- Equivalence of Manchester carry chain and multiple-output domino gate





Manchester Carry Chain Adder

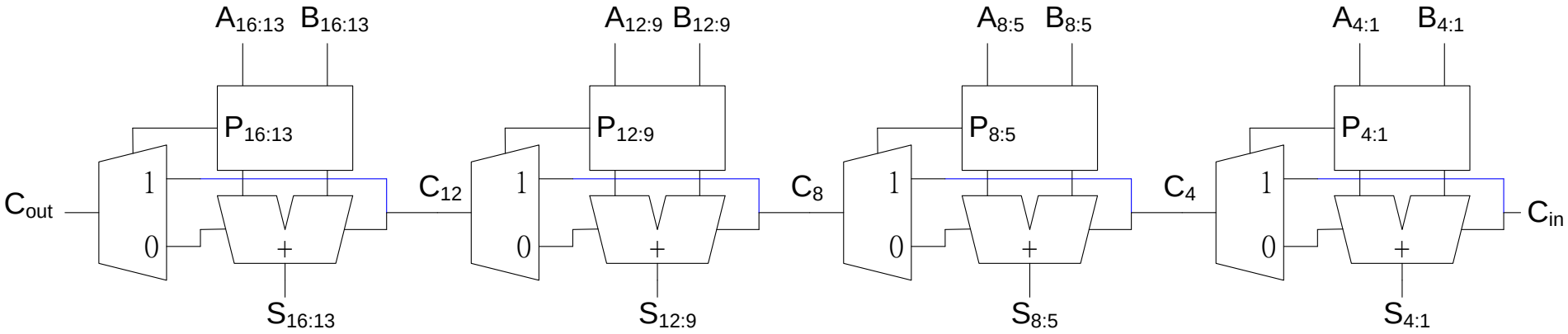
- Valency-4 stages. Similar to carry-ripple adder with N/3 stages



Carry-Skip Adder

- Carry-ripple is slow through all N stages
- Carry-skip allows carry to skip over groups of n bits
  - Decision based on n-bit propagate signal

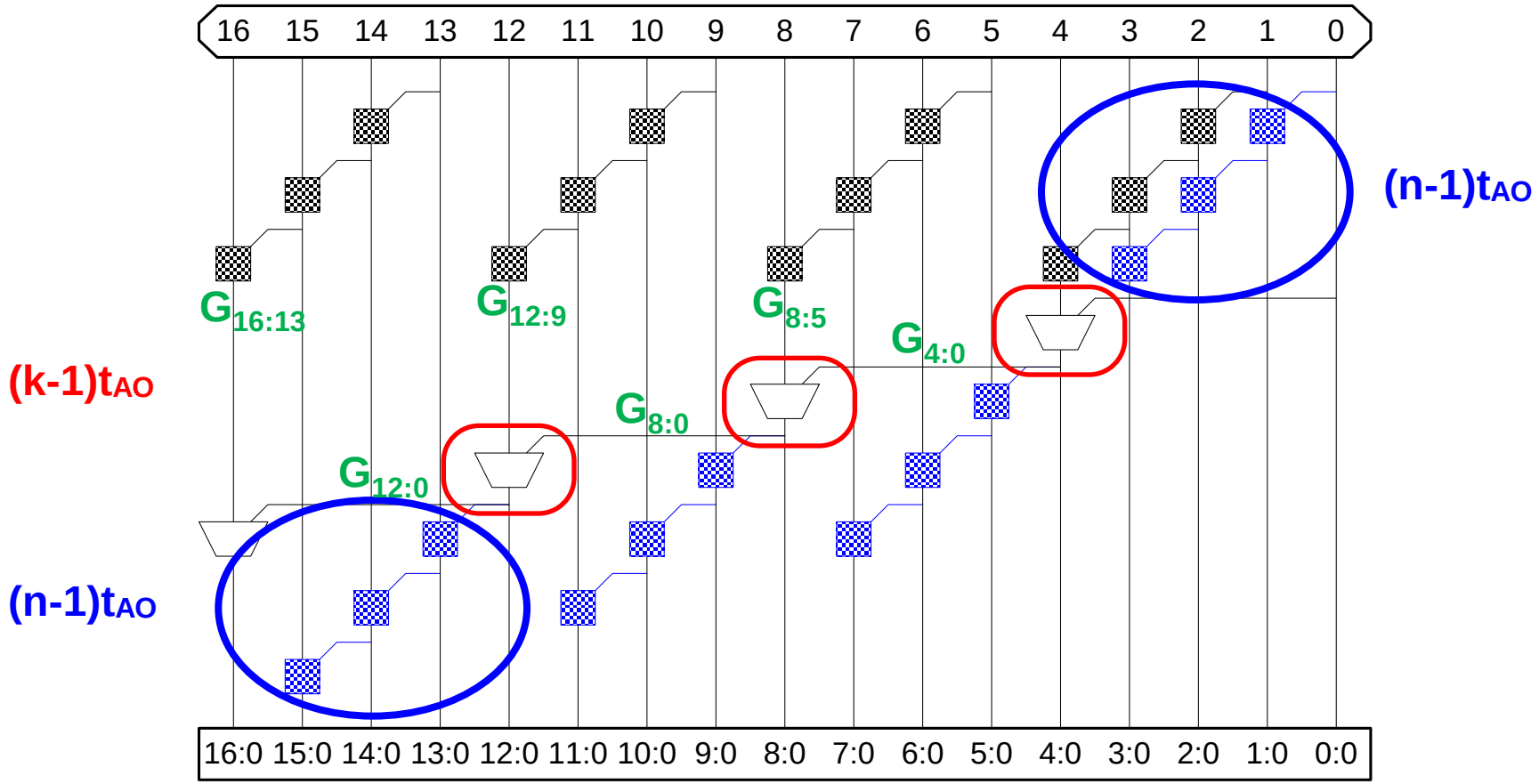
$$G_{16:0} = G_{16:13} + P_{16:13} \bullet G_{12:0} \qquad G_{8:0} = G_{8:5} + P_{8:5} \bullet G_{4:0}$$



$$G_{12:0} = G_{12:9} + P_{12:9} \bullet G_{8:0} \qquad G_{4:0} = G_{4:1} + P_{4:1} \bullet G_{0:0}$$

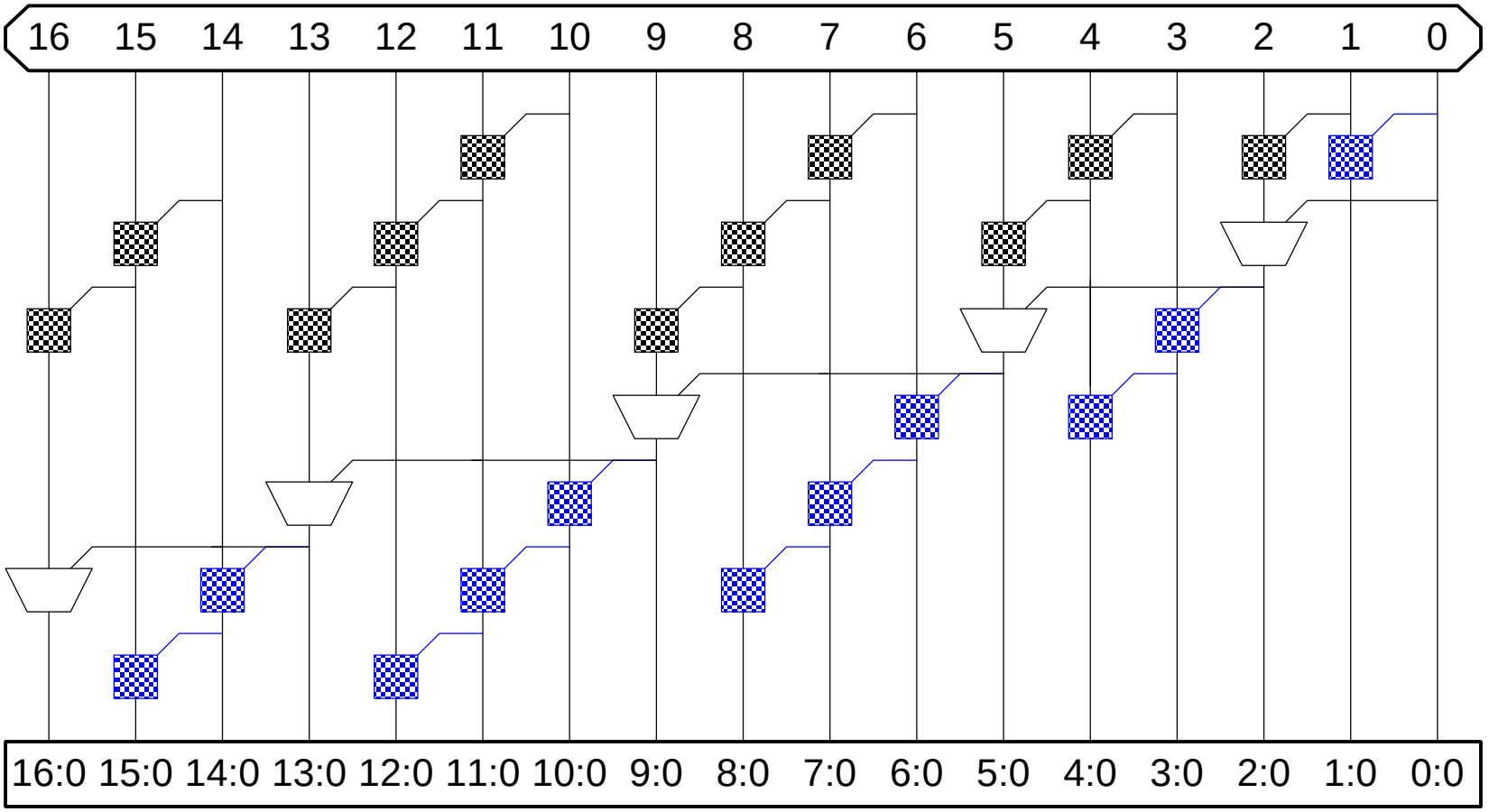
Carry-Skip PG Diagram

For k n-bit groups (N = nk) $t_{\text{skip}} = t_{pg} + \left[2(n-1)+(k-1)\right]t_{AO} + t_{\text{xor}}$

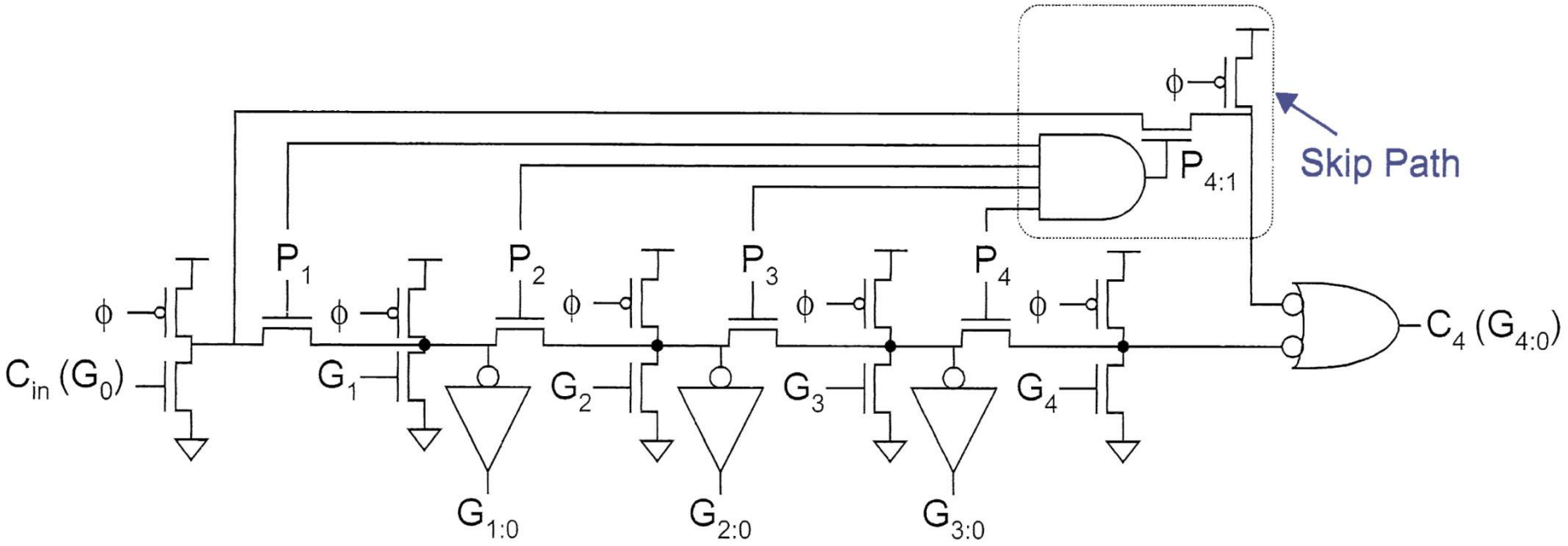


Variable Group Size

Delay grows as O(sqrt(N)) : [2,3,4,4,3] saves 2 levels of logic compared to [4,4,4,4], shorter group at B/E, longer group at middle.



Carry-Skip Adder Manchester Stage

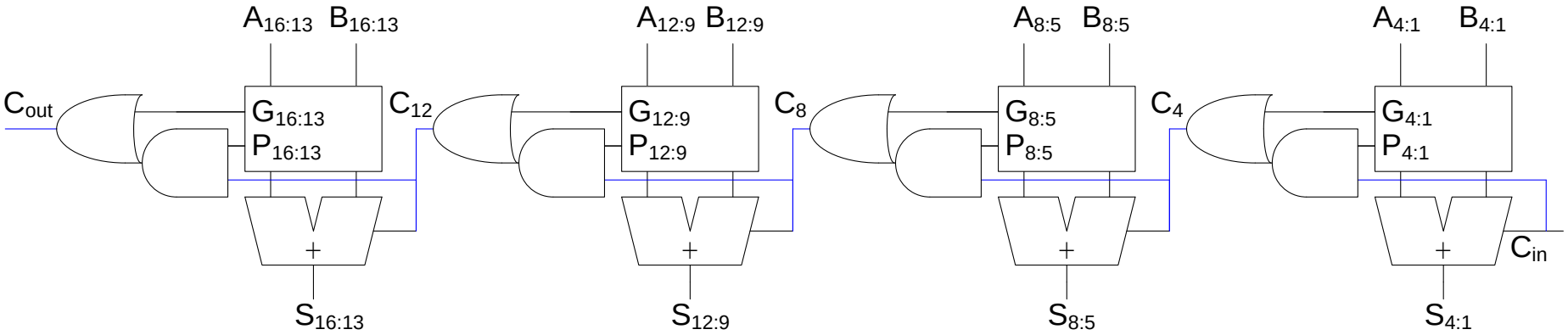


Carry-Lookahead Adder

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

$$G_{16:0} = G_{16:13} + P_{16:13} \bullet G_{12:0}$$

$$G_{8:0} = G_{8:5} + P_{8:5} \bullet G_{4:0}$$

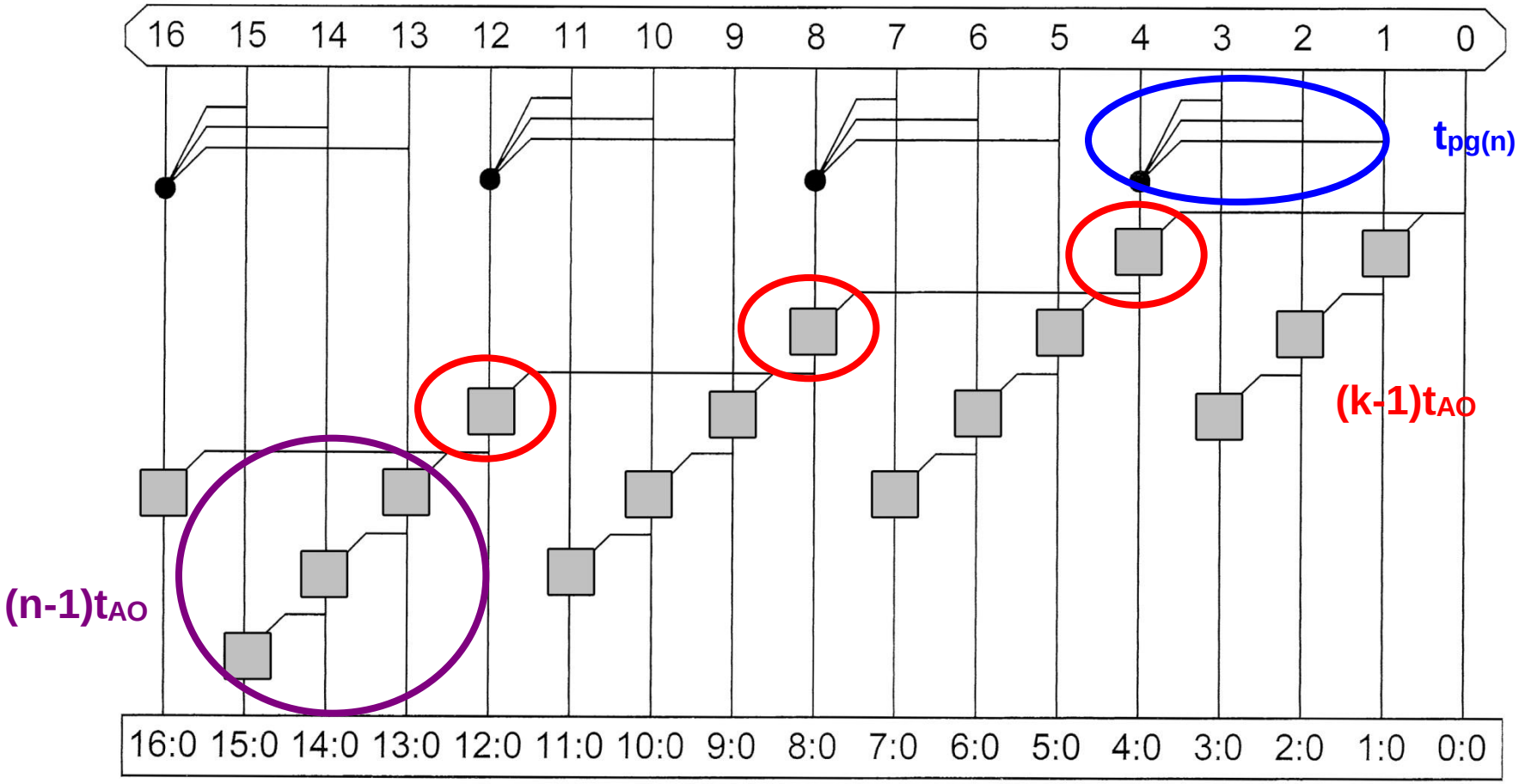


$$G_{12:0} = G_{12:9} + P_{12:9} \bullet G_{8:0}$$

$$G_{4:0} = G_{4:1} + P_{4:1} \bullet G_{0:0}$$

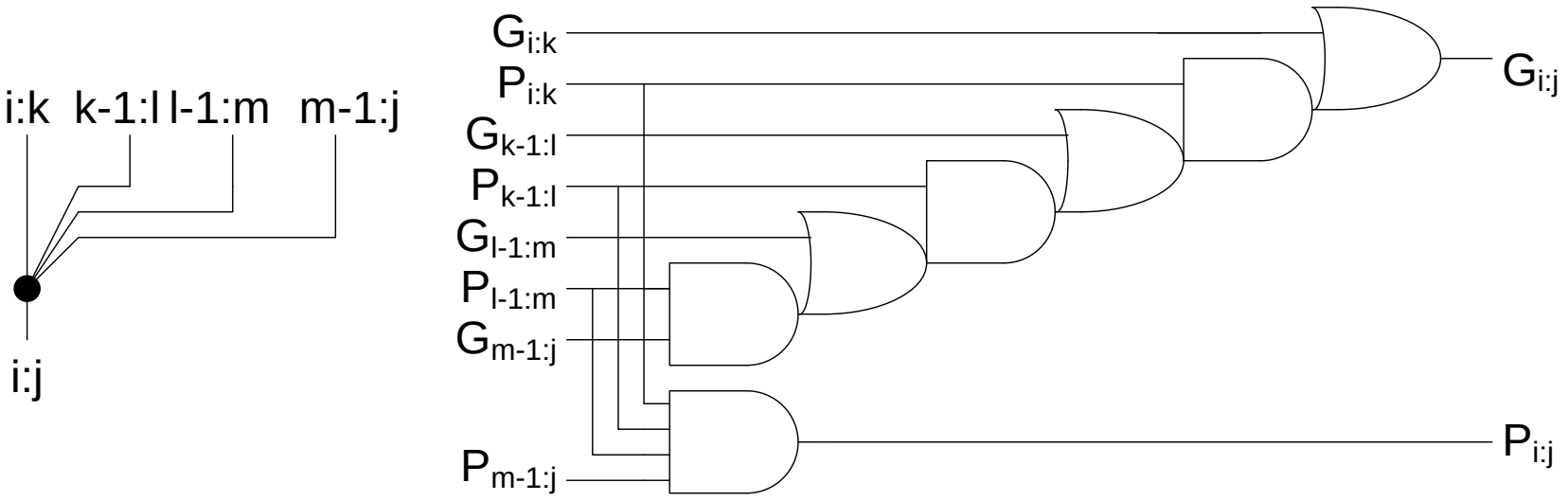
CLA PG Diagram

For k n-bit groups (N = nk) $t_{cla} = t_{pg} + t_{pg(n)} + [(n-1)+(k-1)]t_{AO} + t_{\text{xor}}$

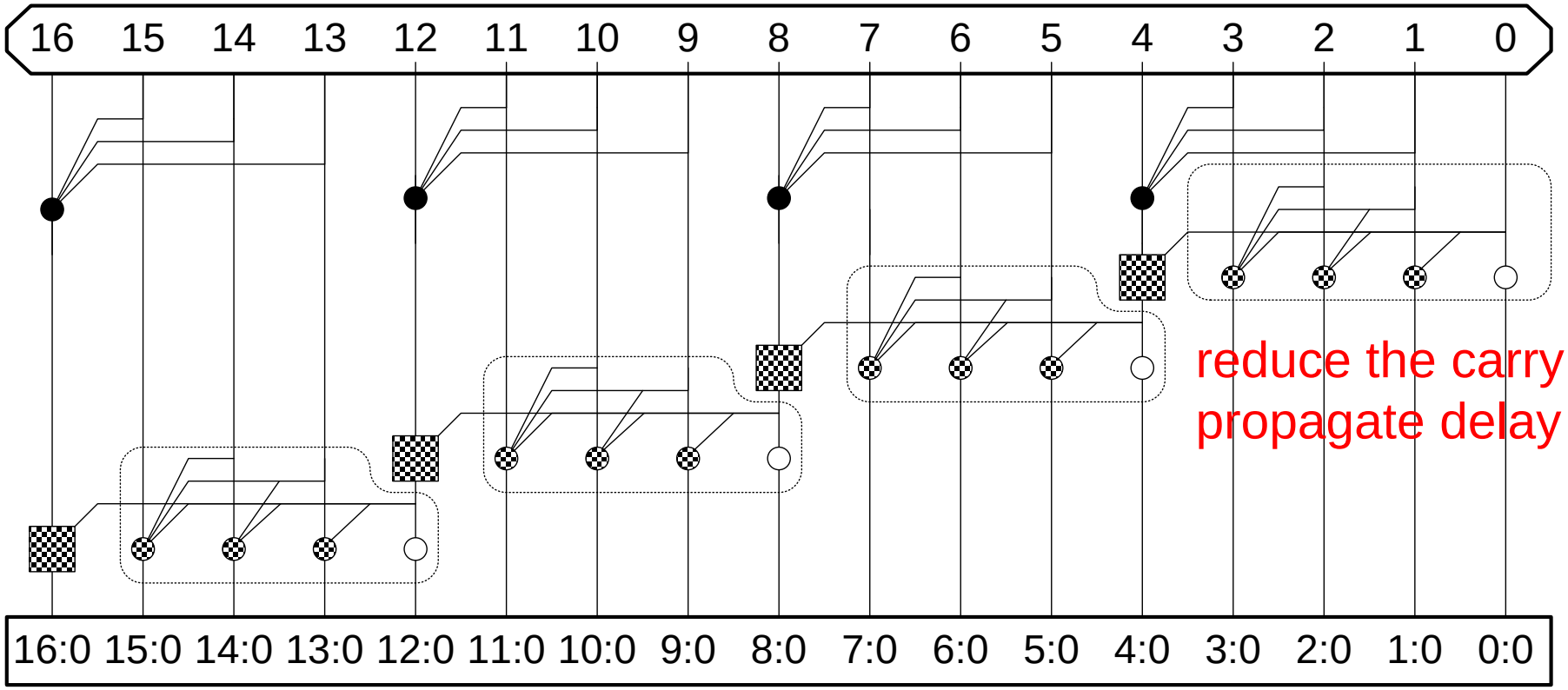


Higher-Valency Cells

$$Delay : t_{pg(n)}$$



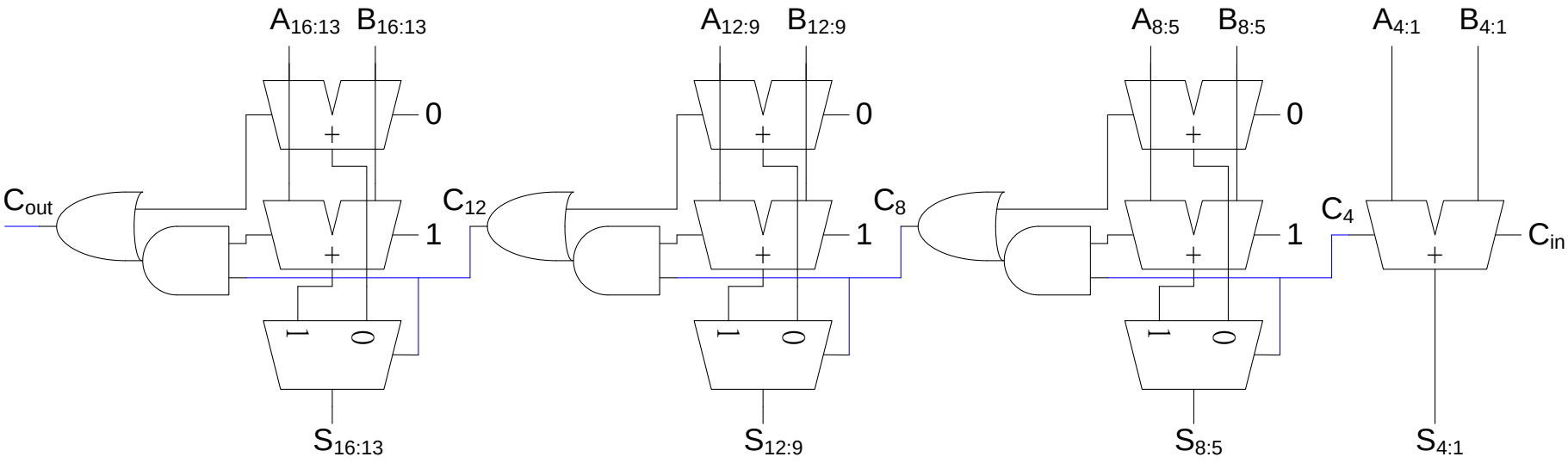
CLA PG Diagram: Manchester



Carry-Select Adder

- Trick for critical paths 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 sum

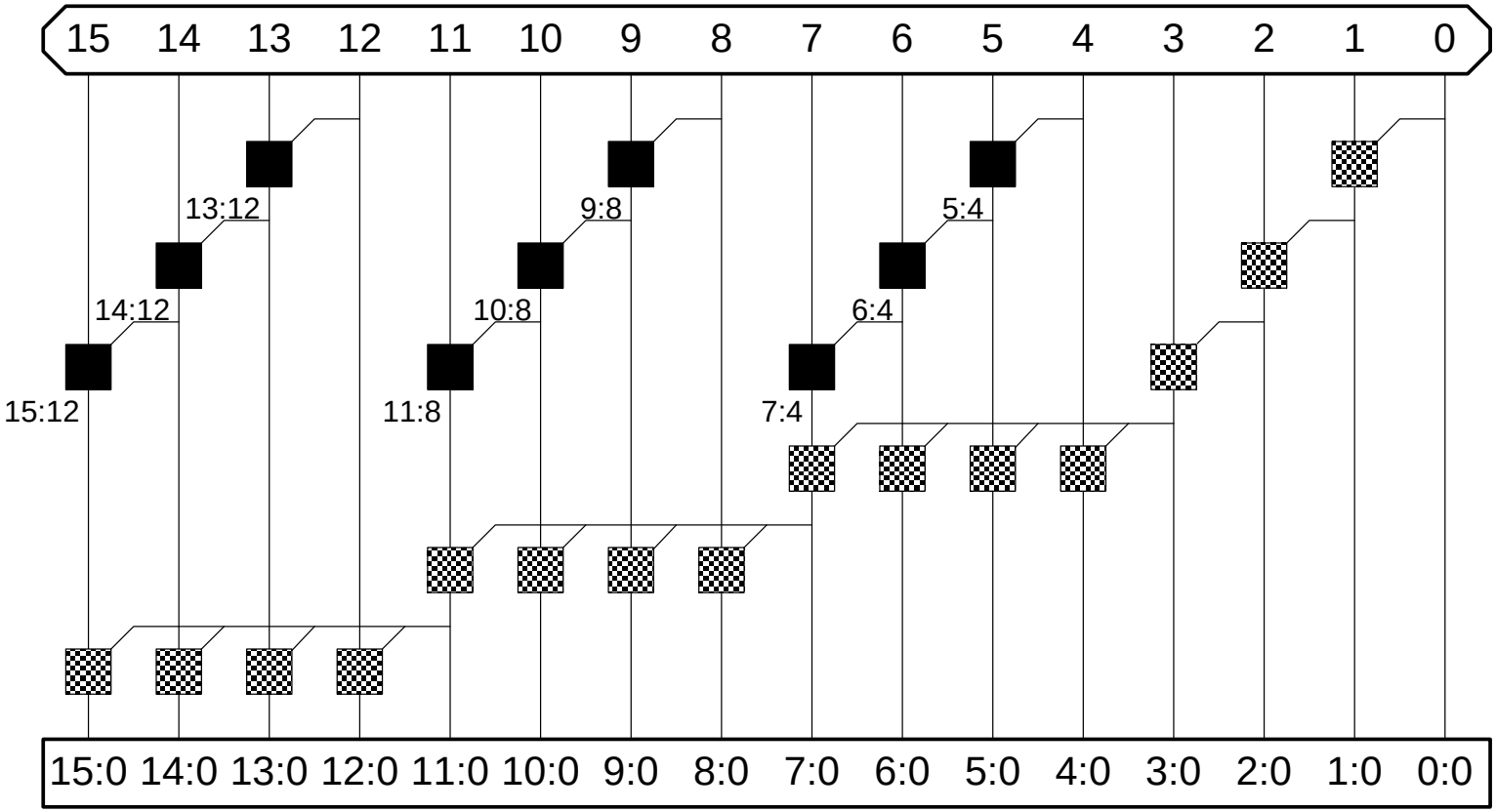
$$t_{\text{select}} = t_{pg} + \left[n + (k-2)\right] t_{AO} + t_{\text{mux}}$$



Carry-Increment Adder

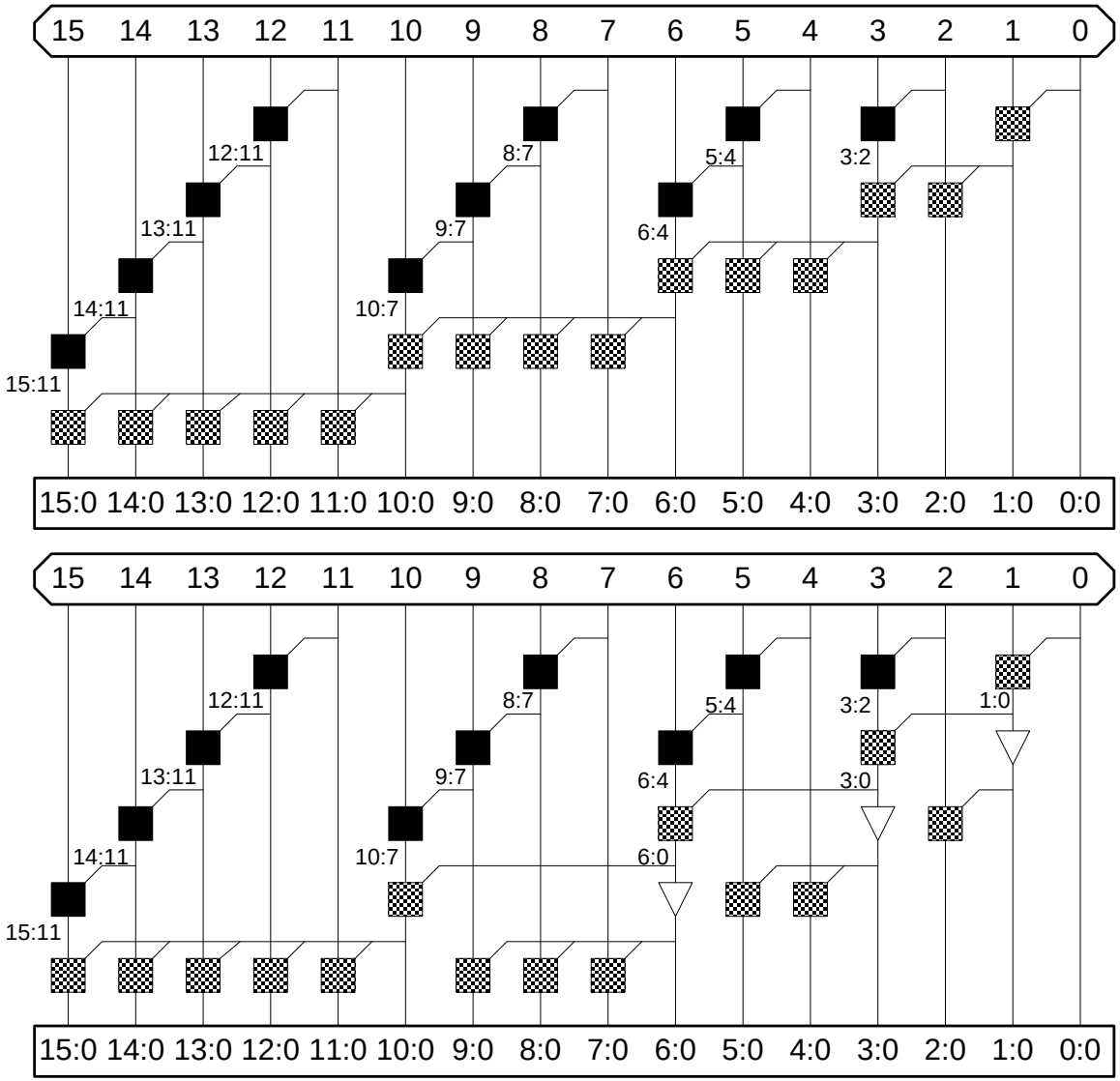
- Factor initial PG and final XOR out of carry-select

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



Variable Group Size

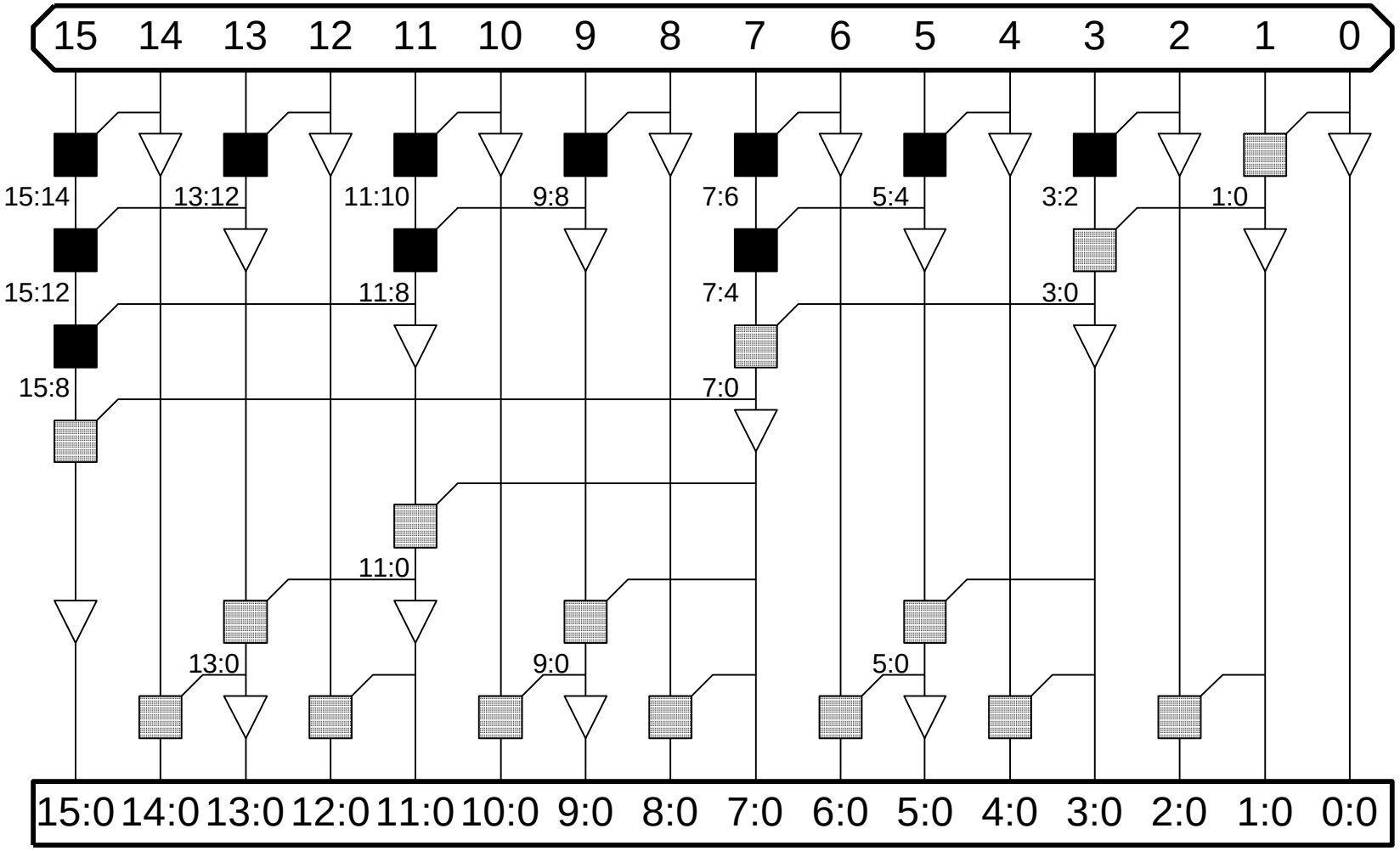
- Also buffer noncritical signals
- Reduce branch effort



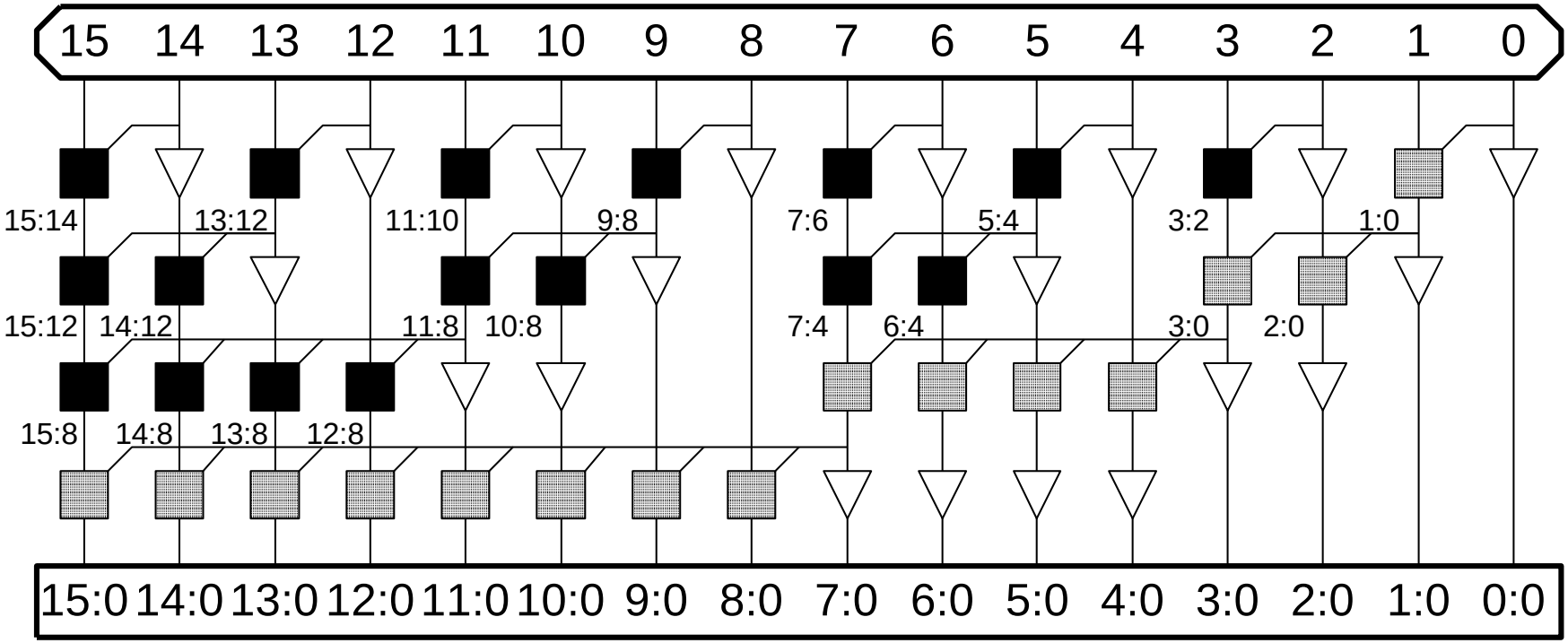
Tree Adder

- If lookahead is good, lookahead across lookahead!
  - Recursive lookahead gives O(log N) delay
- Many variations on tree adders

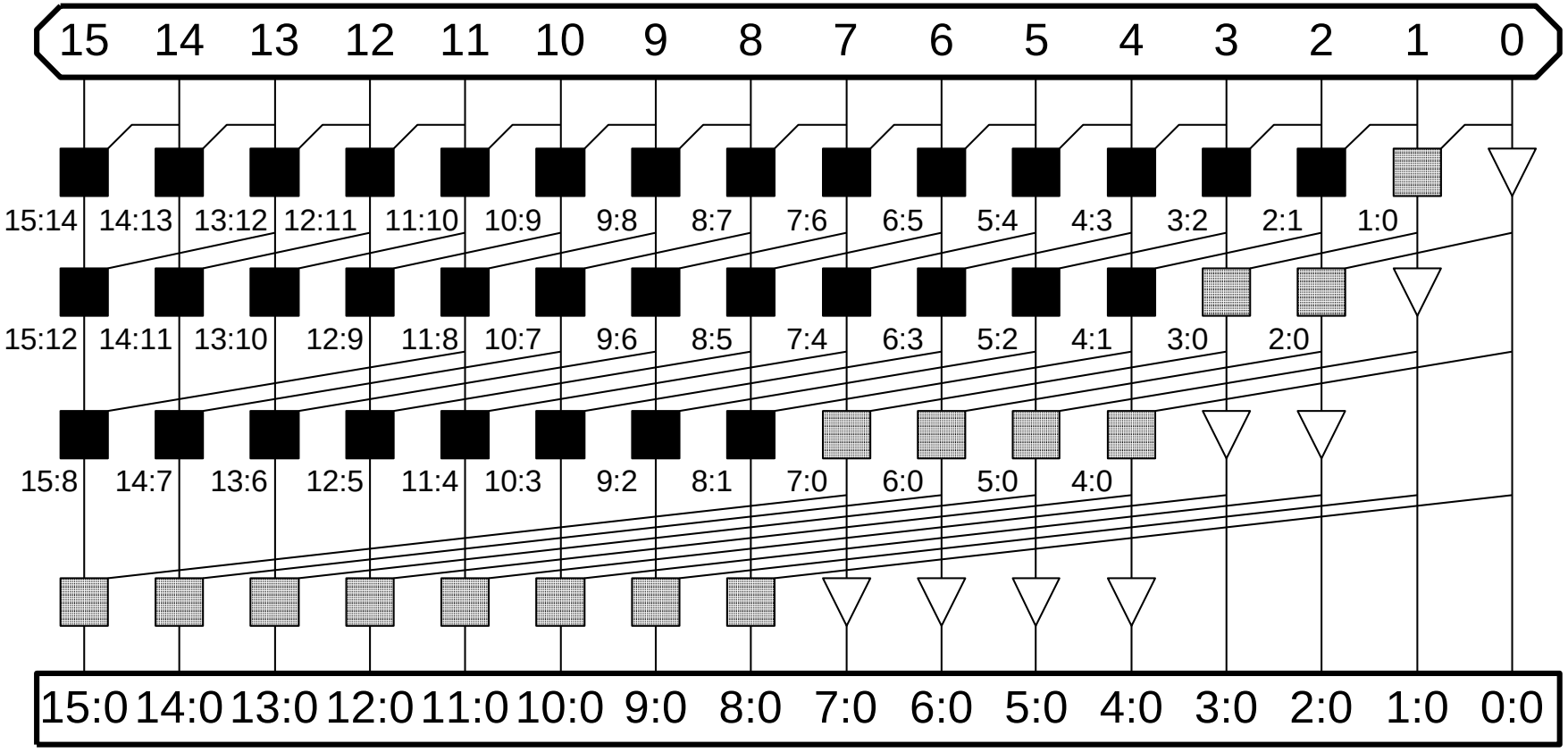
Brent-Kung


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

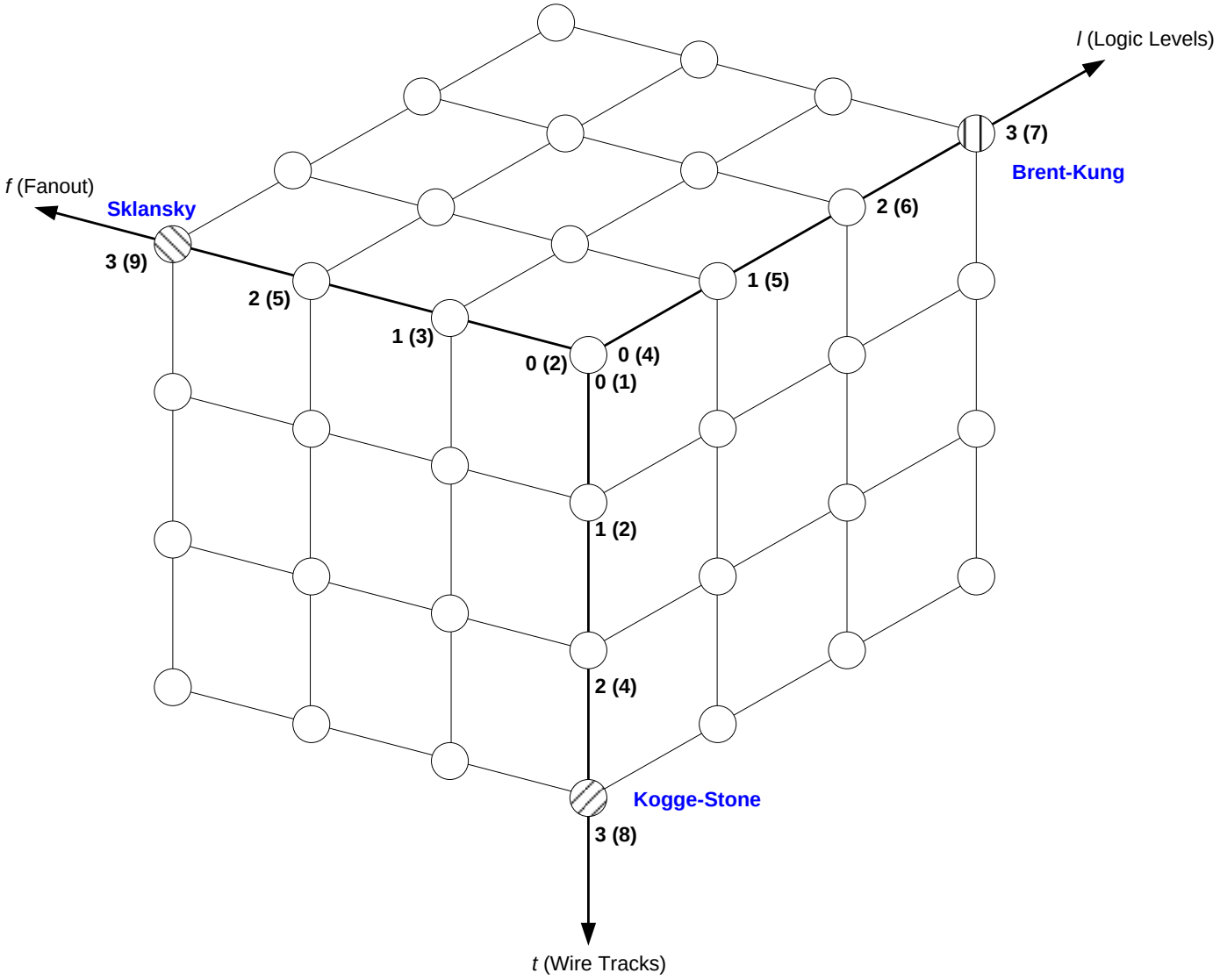

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15:14 14:13 13:12 12:11 11:10 10:9 9:8 8:7 7:6 6:5 5:4 4:3 3:2 2:1 1:0
15:12 14:11 13:10 12:9 11:8 10:7 9:6 8:5 7:4 6:3 5:2 4:1 3:0 2:0
15:8 14:7 13:6 12:5 11:4 10:3 9:2 8:1 7:0 6:0 5:0 4:0
15:0 14:0 13:0 12:0 11:0 10:0 9:0 8:0 7:0 6:0 5:0 4:0 3:0 2:0 1:0 0:0

Tree Adder Taxonomy

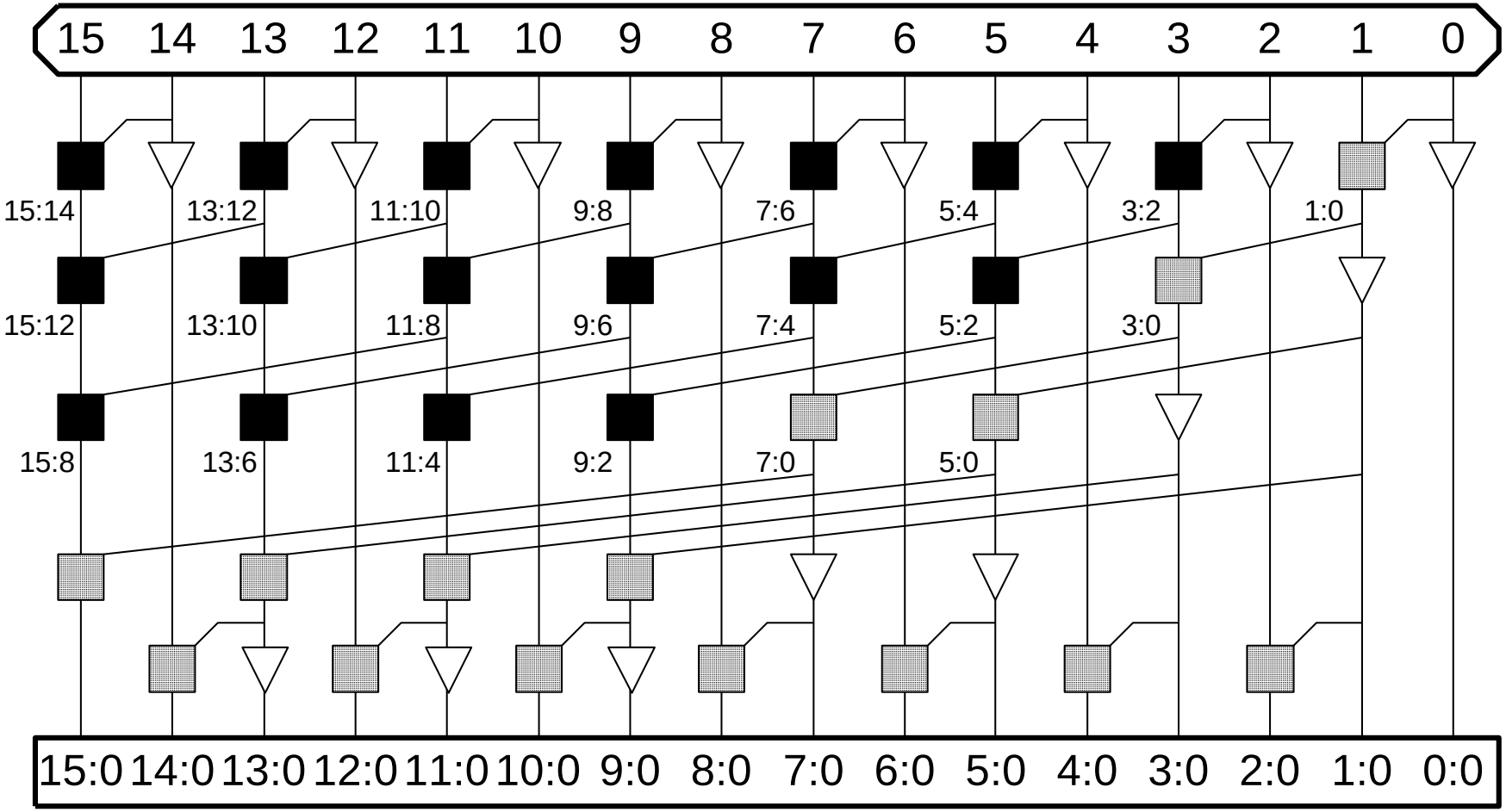
- Ideal N-bit tree adder would have
  - $L = \log_2 N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (l, f, t)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- Known tree adders sit on plane defined by

$$l + f + t = L\text{-}1$$

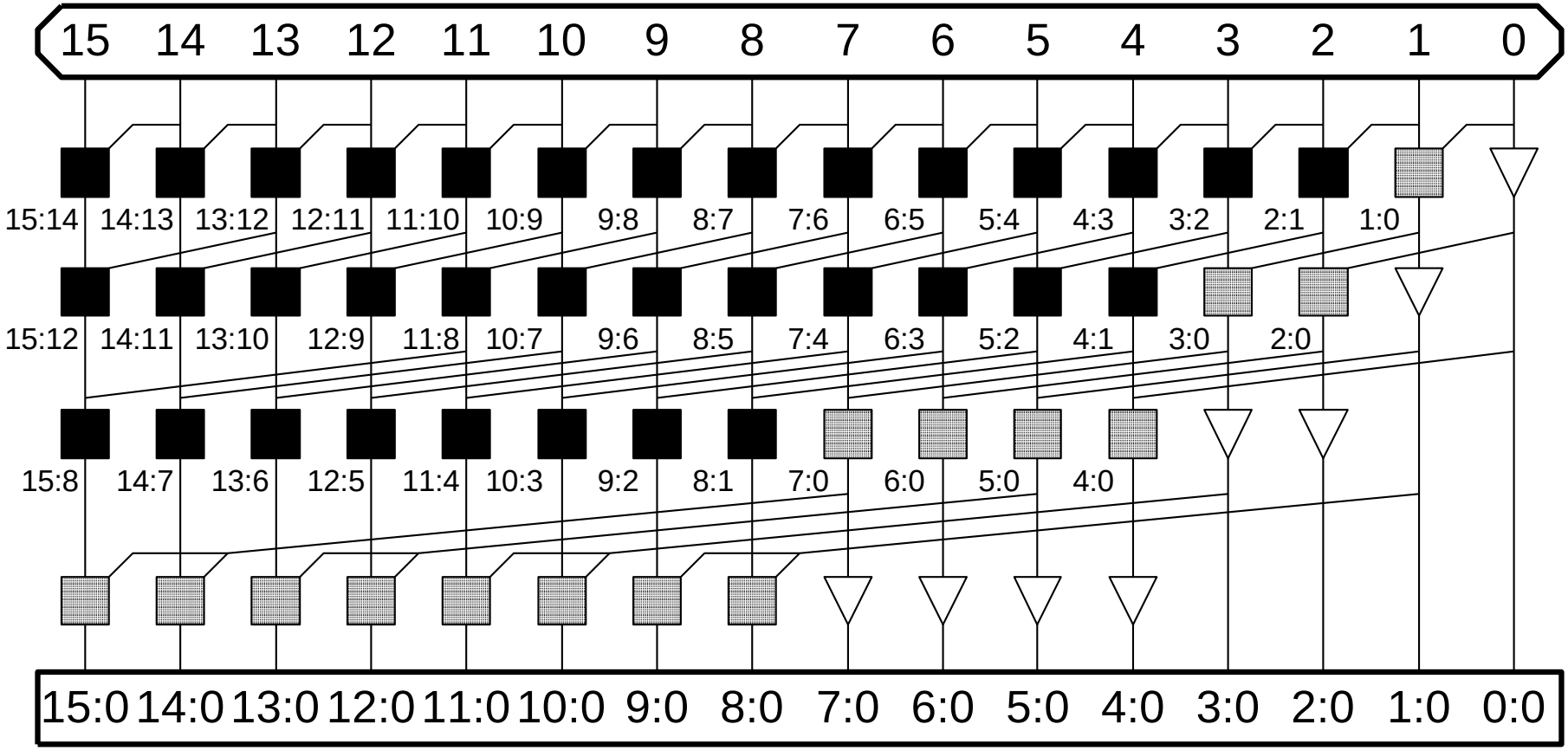
Tree Adder Taxonomy



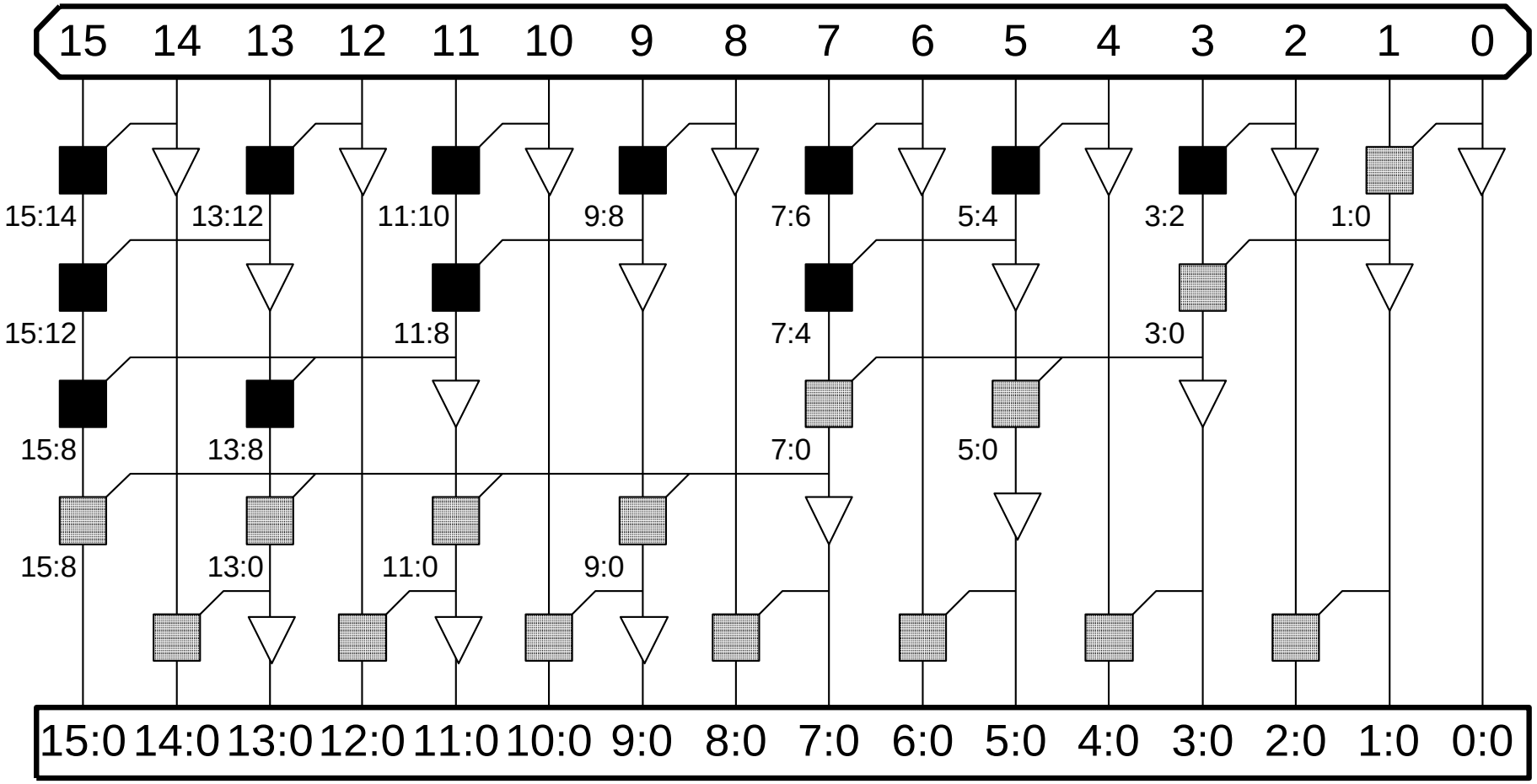
Han-Carlson


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

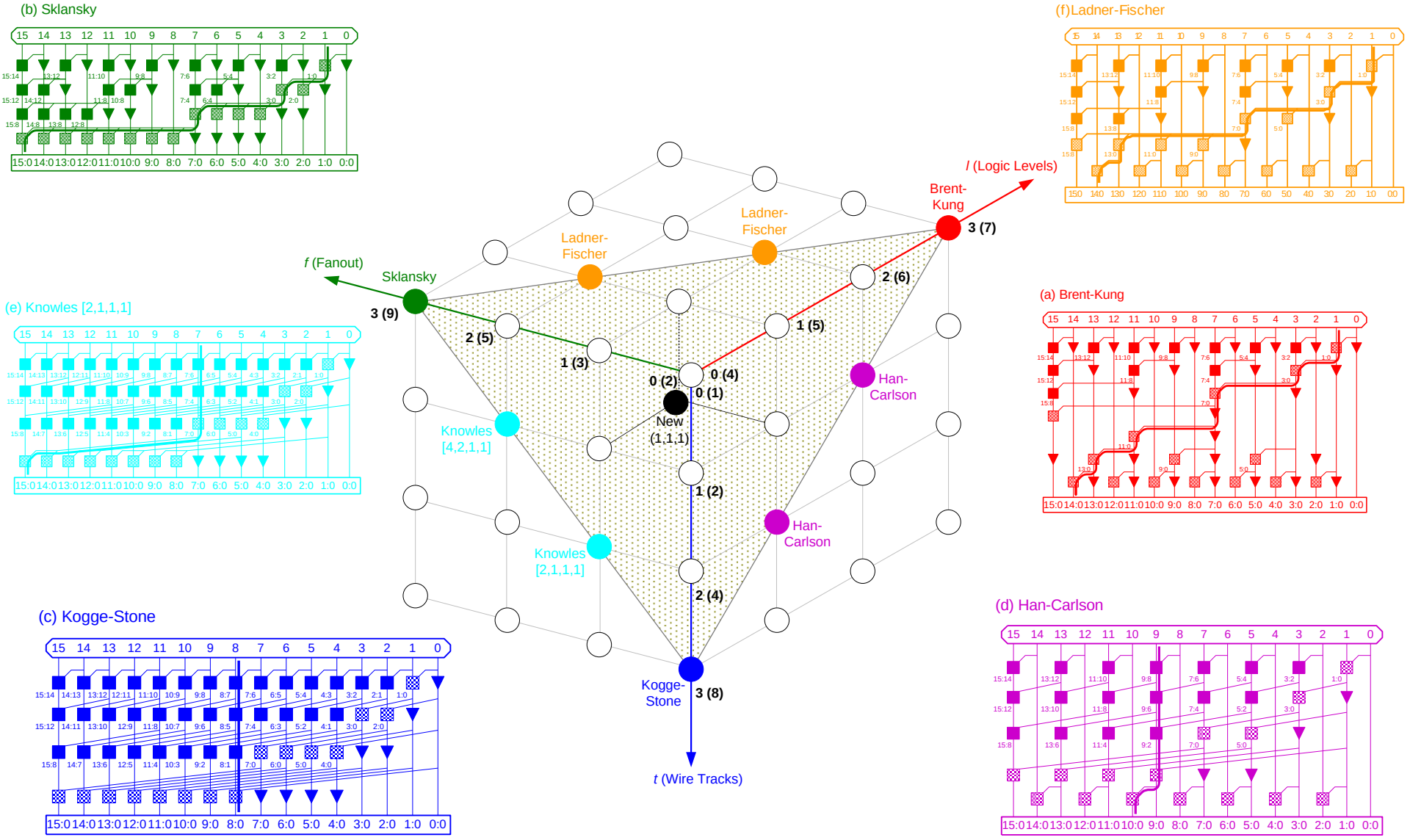
Knowles [2, 1, 1, 1]



Ladner-Fischer



Taxonomy Revisited



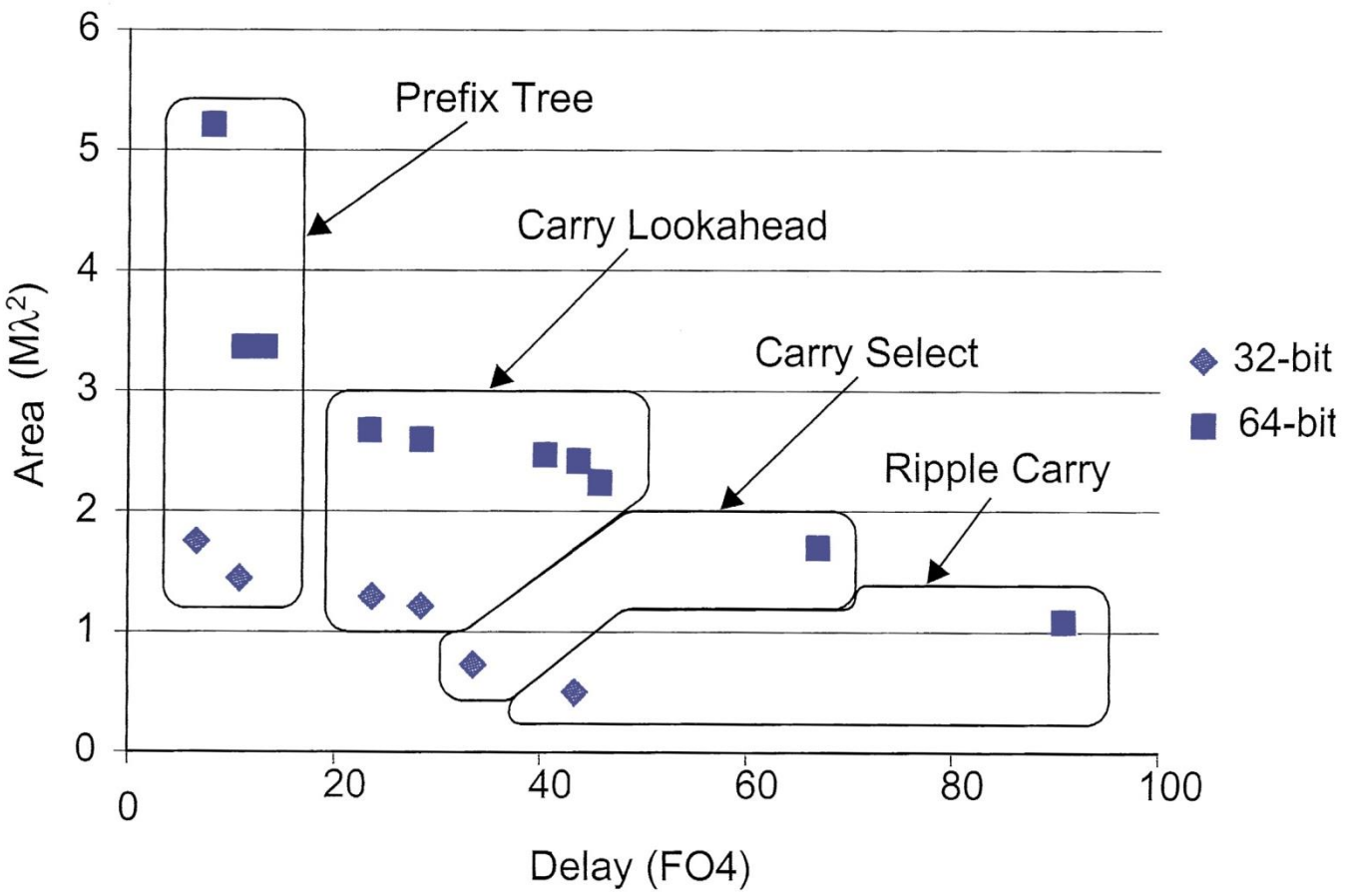
Summary

Adder architectures offer area / power / delay tradeoffs.

Choose the best one for your application.

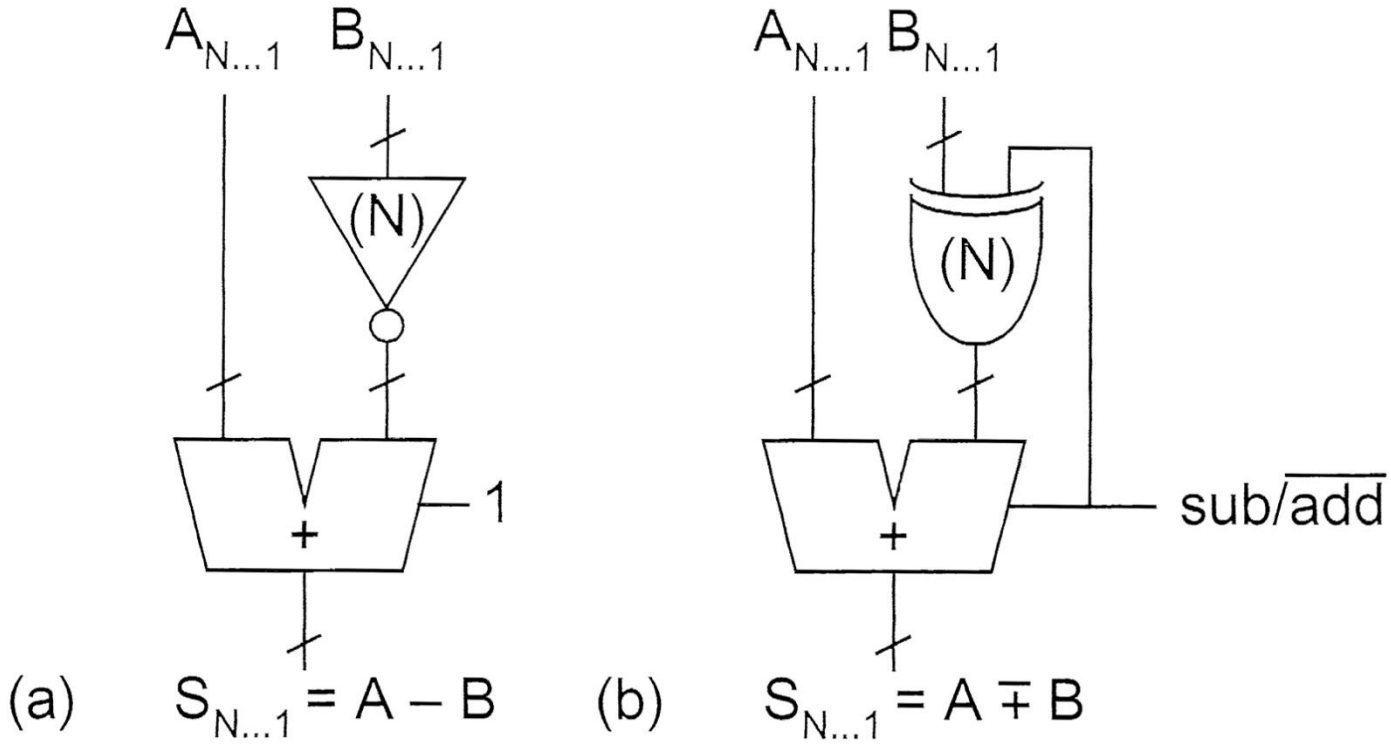
Architecture	**Classification**	**Logic Levels**	**Max Fanout**	**Tracks**	**Cells**
Carry-Ripple		N-1	1	1	N
Carry-Skip n=4		N/4 + 5	2	1	1.25N
Carry-Inc. n=4		N/4 + 2	4	1	2N
Brent-Kung	(L-1, 0, 0)	$2\log_2 N - 1$	2	1	2N
Sklansky	(0, L-1, 0)	$\log_2 N$	N/2 + 1	1	$0.5\ N\log_2 N$
Kogge-Stone	(0, 0, L-1)	$\log_2 N$	2	N/2	$N\log_2 N$

Summary


Prefix Tree
Carry Lookahead
Carry Select
Ripple Carry
32-bit
64-bit
Area (Mλ²)
Delay (FO4)
0
1
2
3
4
5
6
0
20
40
60
80
100

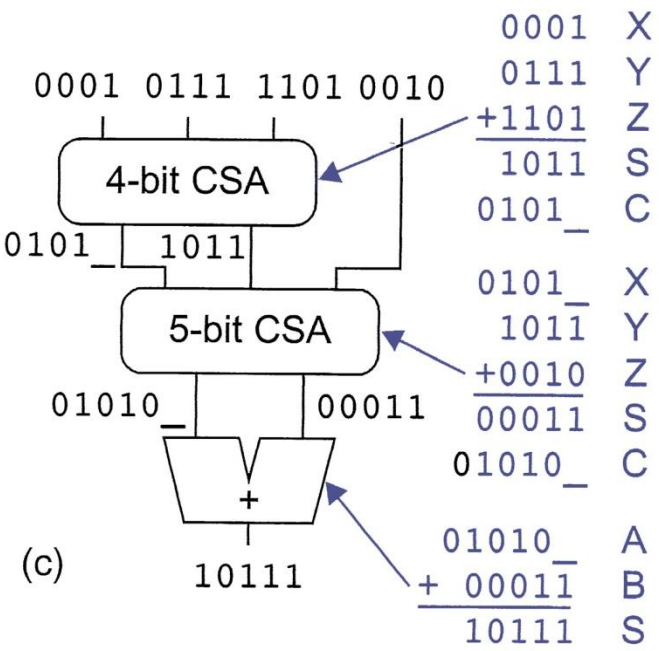
Subtraction

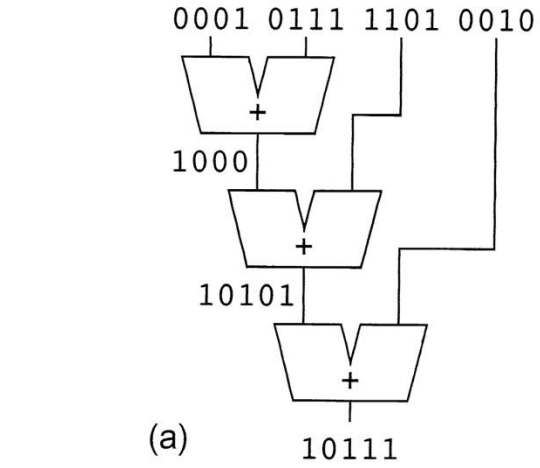
$$A - B = A + \overline{B} + 1$$

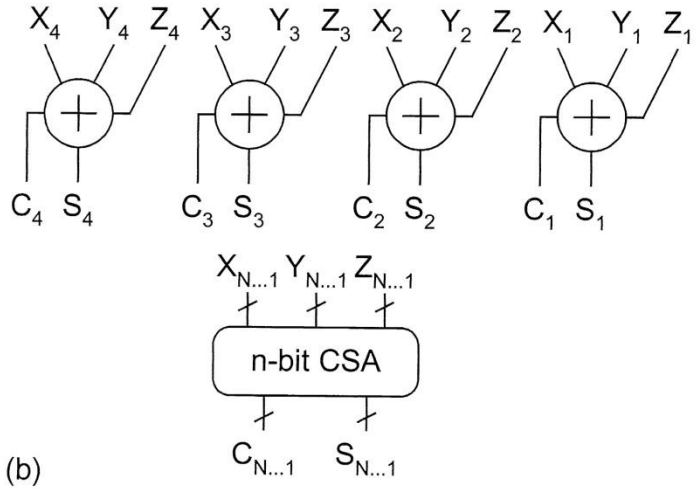


Multiple-Input Addition

- Carry-Save Adder (CSA)
  - *k* N-bit words can be summed with *k*-2 CSA and 1 CPA.





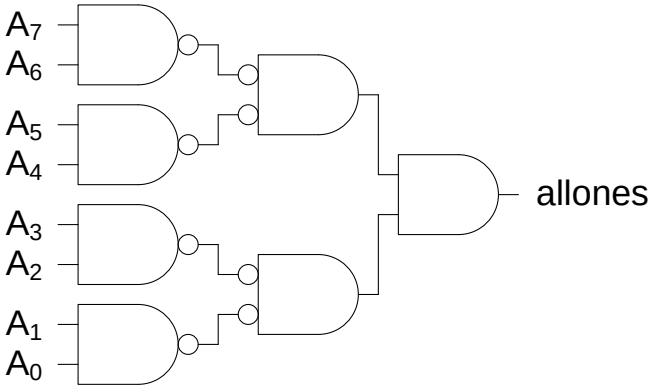


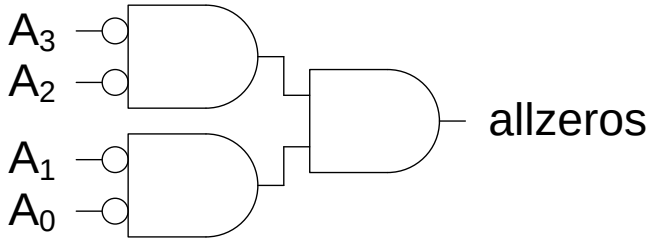
Outline

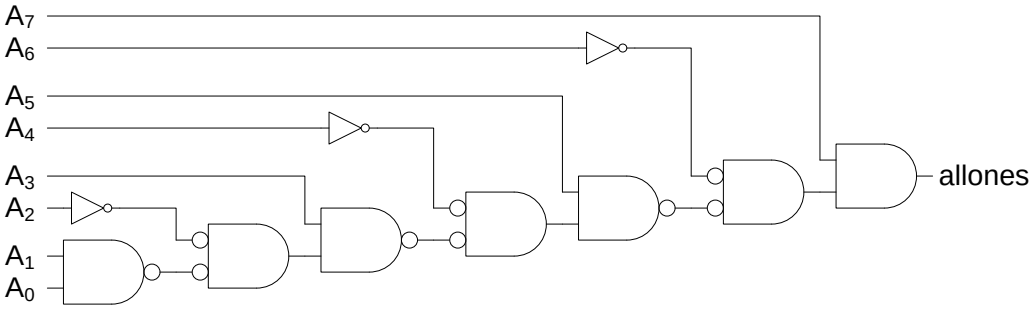
1. Addition/Subtraction
2. **One/Zero Detectors**
3. Comparators
4. Counters
5. Shifters
6. Multiplication

1's & 0's Detectors

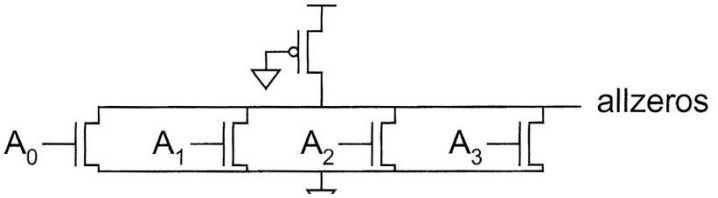
- 1's detector: N-input AND gate
- 0's detector: NOTs + 1's detector (N-input NOR)







Mimic the adder delay



"wired-OR"

Outline

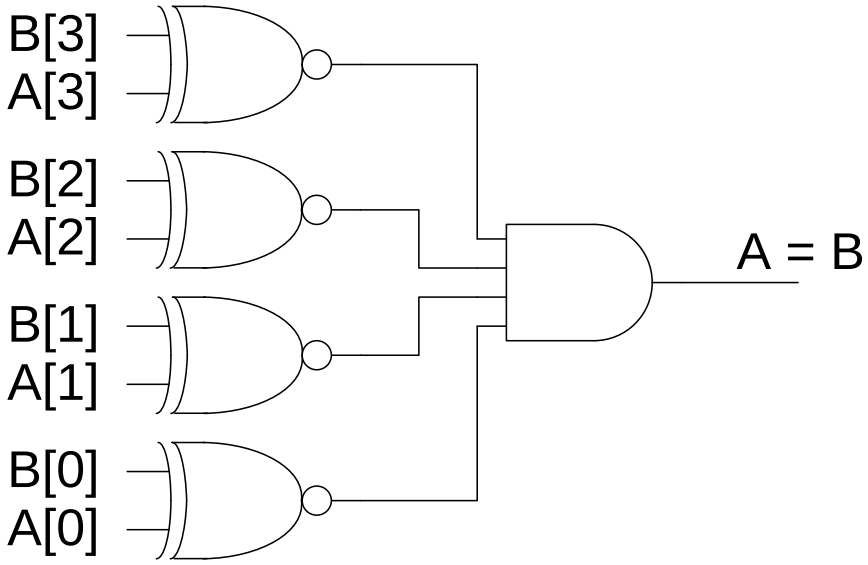
1. Addition/Subtraction
2. One/Zero Detectors
3. **Comparators**
4. Counters
5. Shifters
6. Multiplication

Comparators

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

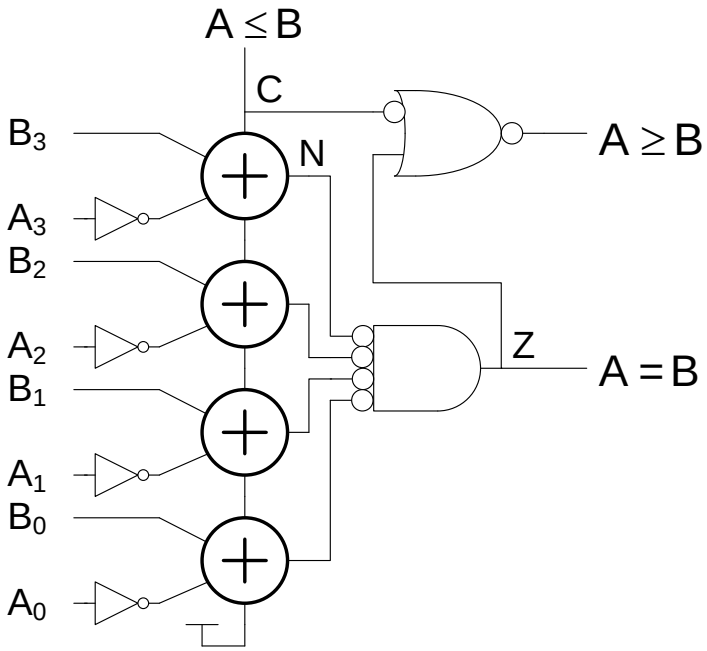
Equality Comparator

- Check if each bit is equal (XNOR, aka 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

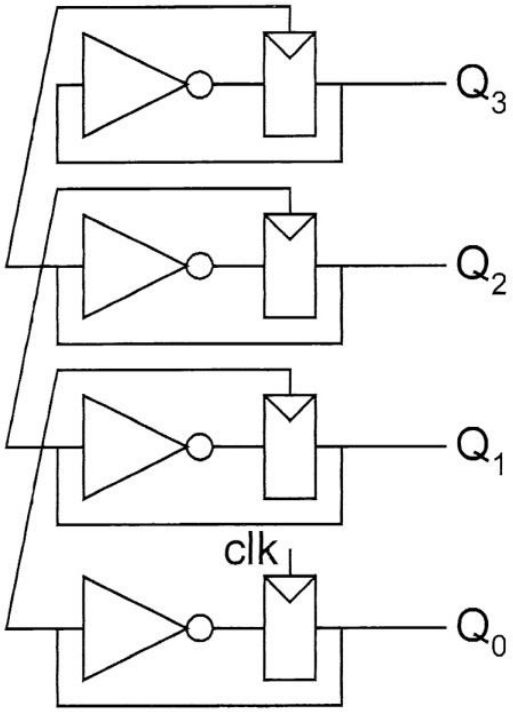
Relation	Unsigned Comparison	Signed Comparison
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C + Z}$	$\overline{(N \oplus V) + Z}$
$A > B$	$\overline{C}$	$(N \oplus V)$
$A \leq B$	C	$(\overline{N \oplus V})$
$A \geq B$	$\overline{C} + Z$	$(N \oplus V) + Z$

Outline

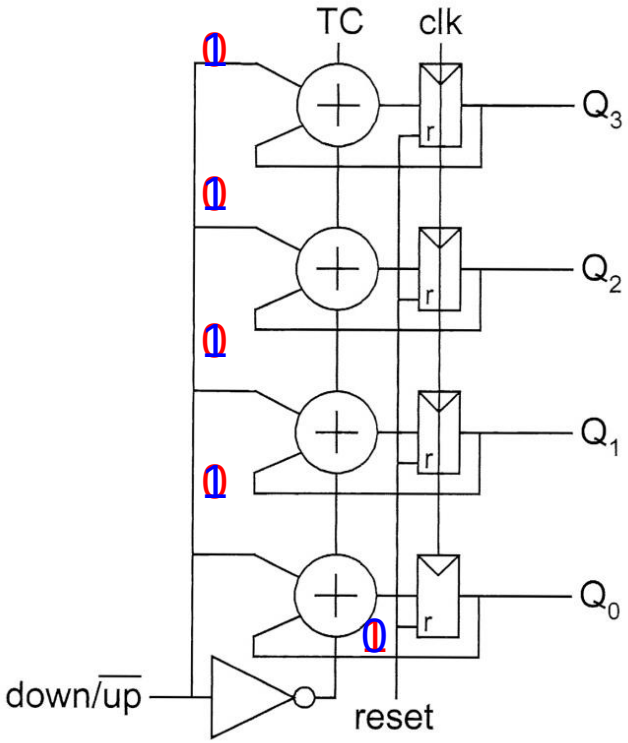
1. Addition/Subtraction
2. One/Zero Detectors
3. Comparators
4. **Counters**
5. Shifters
6. Multiplication

Binary Counters

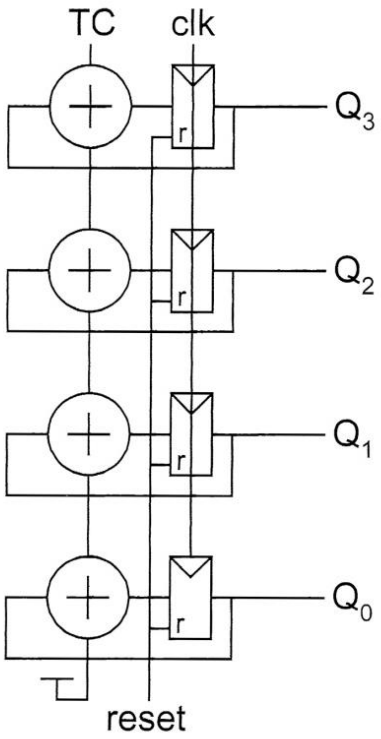
Asynchronous counter



Synchronous up/down counter



Synchronous incrementer



Half adder only

Linear Feedback Shift Register

- The input to the LFSR register comes from the XOR of particular bits of the register : pseudo-random-bit-sequence (PRBS) generator

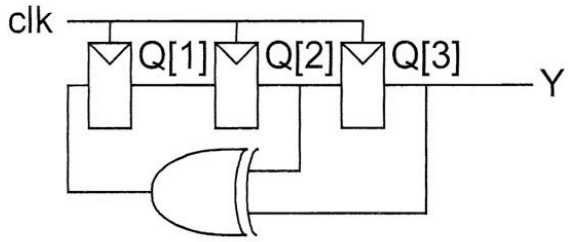


Table 10.6 LFSR sequence

Cycle	Q [1]	Q [2]	Q [3] / Y
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1
repeats forever			

Table 10.7 Characteristic polynomials

N	Polynomial
3	$1 + x^2 + x^3$
4	$1 + x^3 + x^4$
5	$1 + x^3 + x^5$
6	$1 + x^5 + x^6$
7	$1 + x^6 + x^7$
8	$1 + x^1 + x^6 + x^7 + x^8$
9	$1 + x^5 + x^9$
15	$1 + x^{14} + x^{15}$
16	$1 + x^4 + x^{13} + x^{15} + x^{16}$
23	$1 + x^{18} + x^{23}$
24	$1 + x^{17} + x^{22} + x^{23} + x^{24}$
31	$1 + x^{28} + x^{31}$
32	$1 + x^{10} + x^{30} + x^{31} + x^{32}$

Linear Feedback Shift Register

Example

Sketch an 8-bit linear-feedback shift register. How long is the pseudo-random bit sequence that it produces?

Solution: Figure 10.57 shows an 8-bit LFSR using the four taps after the first, sixth, seventh, and eighth bits, as given in Table 10.7. It produces a sequence of $2^8-1 = 255$ bits before repeating.

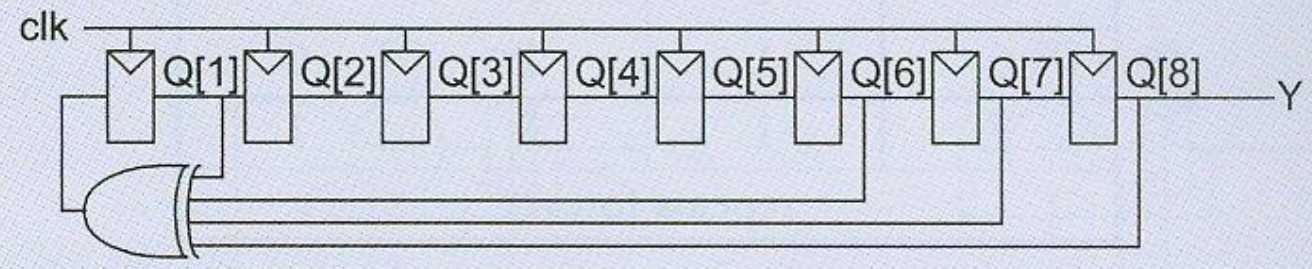
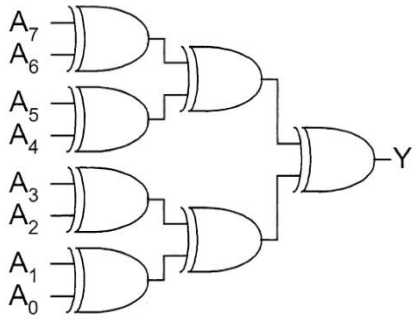


FIG 10.57 8-bit LFSR

Coding

- Error-detecting and error-correcting codes (ECC) by adding extra bits to the data
  - Parity: check number of 1's is even
    single-bit error detecting

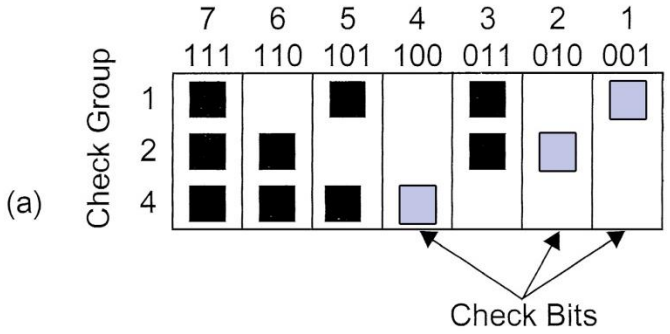
$$A_n = PARITY = A_0 \oplus A_1 \oplus A_2 \oplus \ldots \oplus A_{n-1}$$

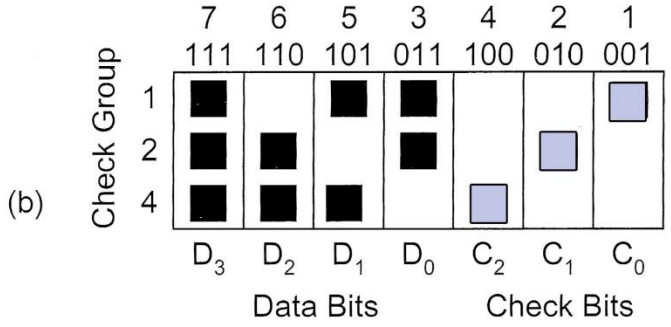


  - ECC : single error correcting & double error detecting (SECDED)

    4 data bits $D_{0\sim3}$ and 3 check bits $C_{0\sim2}$

$$C_0 = D_3 \oplus D_1 \oplus D_0$$
$$C_1 = D_3 \oplus D_2 \oplus D_0$$
$$C_2 = D_3 \oplus D_2 \oplus D_1$$





Gray Code

- Consecutive numbers differ in only one bit
  - No glitch when applying to decoder
  - Power saving by reducing transition

Binary → Gray

$$G_{N-1} = B_{N-1}$$

$$G_i = B_{i+1} \oplus B_i$$

Gray → Binary

$$B_{N-1} = G_{N-1}$$

$$B_i = B_{i+1} \oplus G_i \quad N-1 > i \geq 0$$

Table 10.9 3-bit Gray code

Number	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Outline

1. Addition/Subtraction
2. One/Zero Detectors
3. Comparators
4. Counters
5. **Shifters**
6. Multiplication

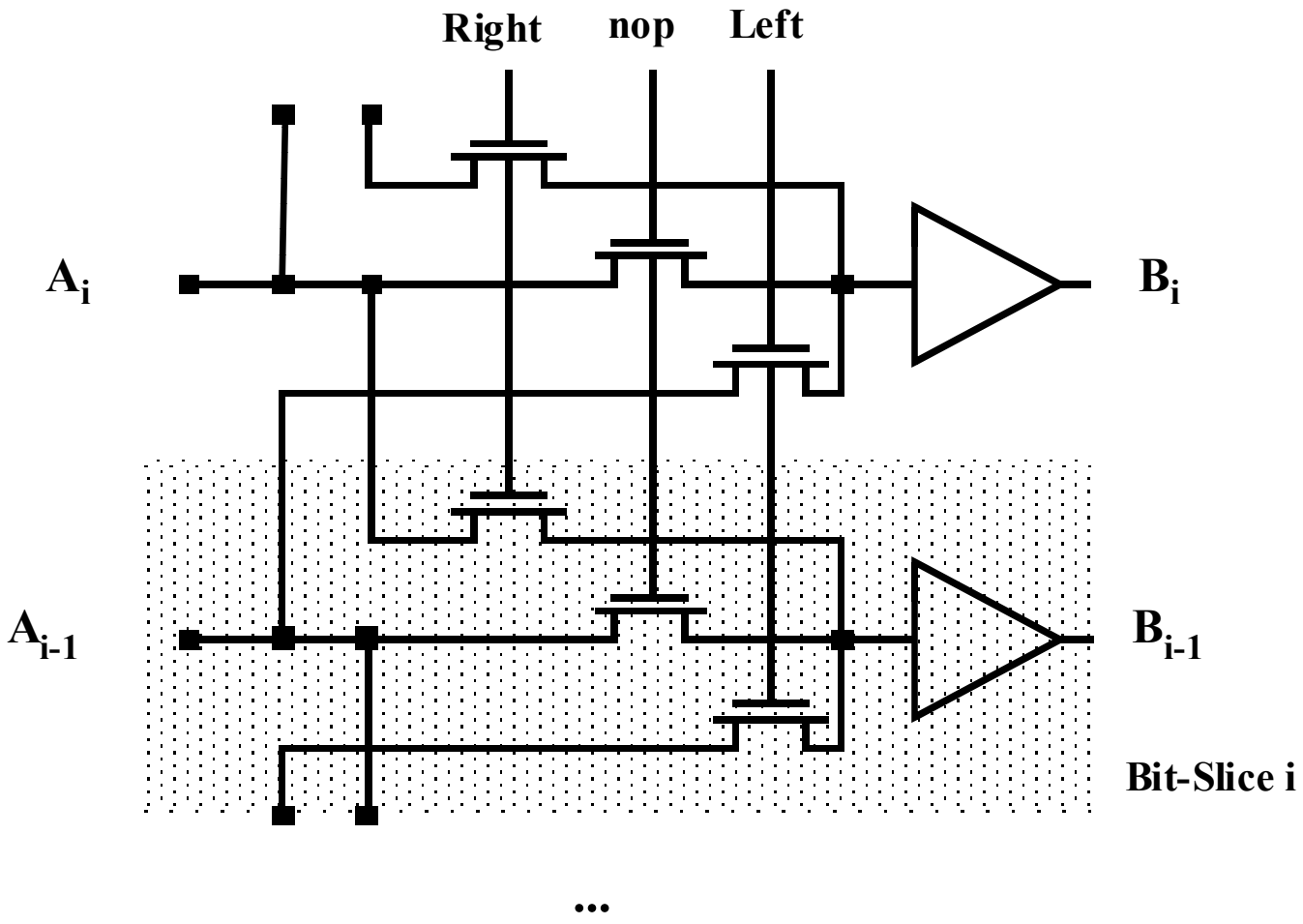
Shifters

- Logical Shifter:
  - Shifts number left or right and fills with 0's
    - 1011 LSR 1 = 0101 1011 LSL1 = 0110
- Arithmetic Shifter:
  - Shifts number left or right. Rt shift sign extends, Lf shift fills with 0's
    - 1011 ASR1 = 1101 1011 ASL1 = 0110
- Barrel Shifter (Rotator):
  - Shifts number left or right and fills with lost bits
    - 1011 ROR1 = 1101 1011 ROL1 = 0111

Combinational shifters

- Useful for arithmetic operations, bit field extraction, etc.
- Latch-based shift register can shift only one bit per clock cycle.
- A multiple-shift shifter requires additional connectivity.

The Binary Shifter

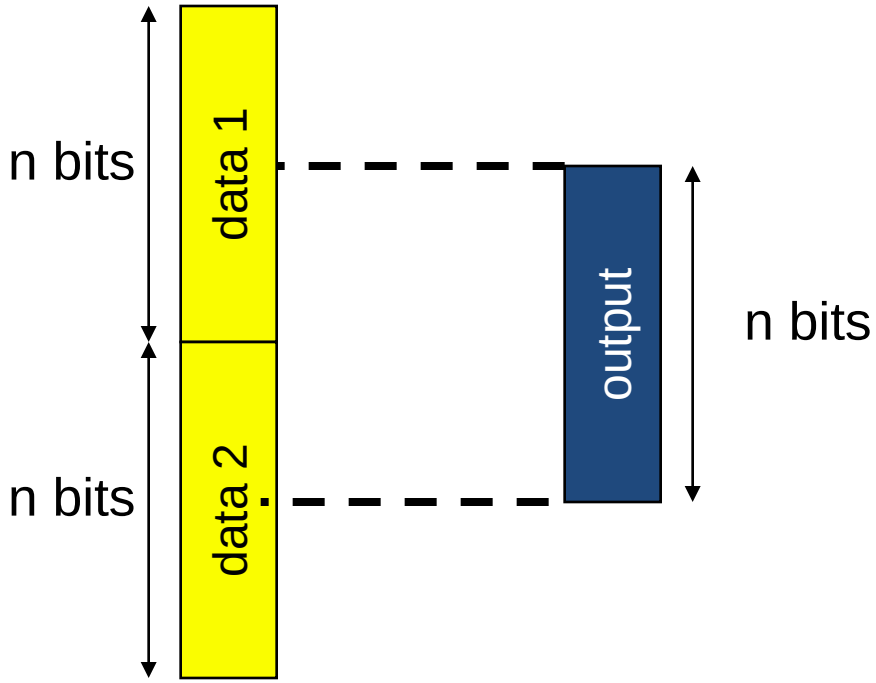


Barrel shifter

- Can perform n-bit shifts in a single cycle.
- Efficient layout.
- Does require transmission gates and long wires.

Barrel shifter structure

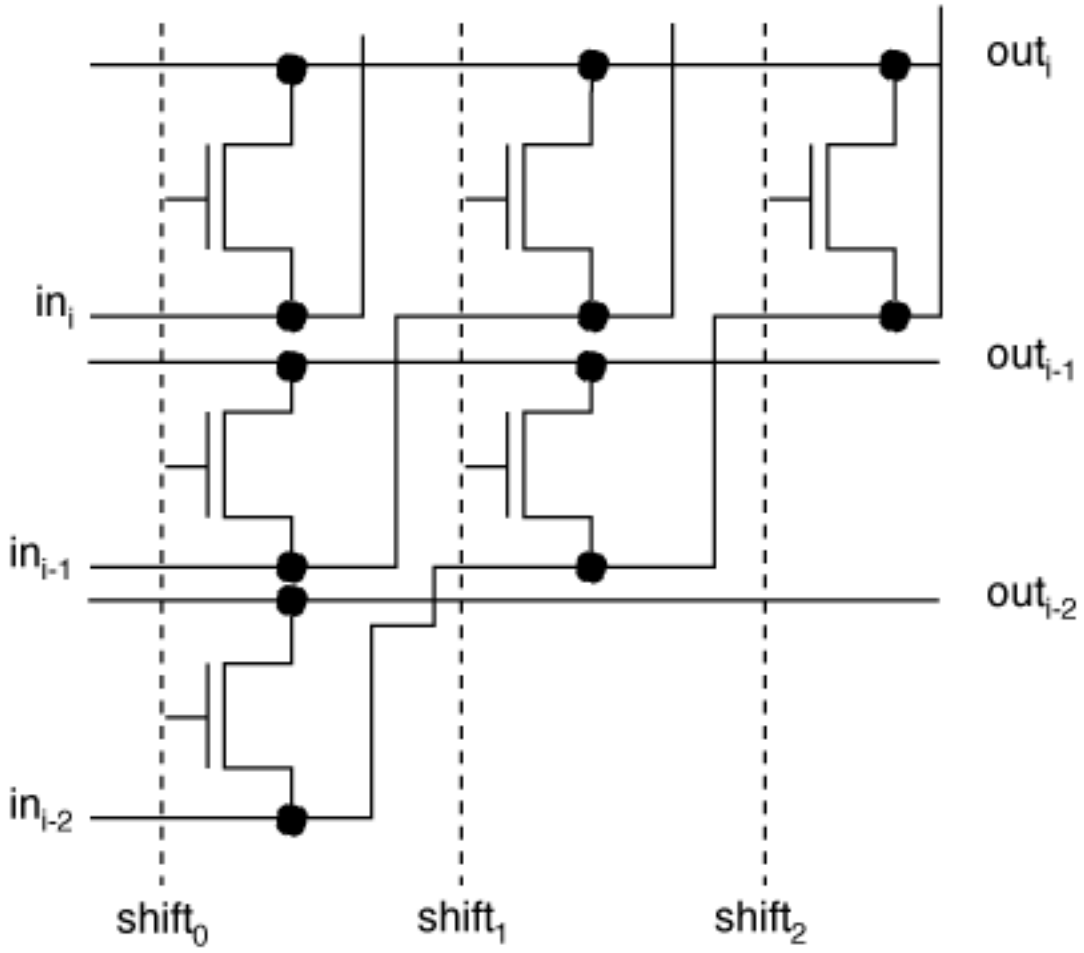
Accepts *2n* data inputs and *n* control signals, producing *n* data outputs.



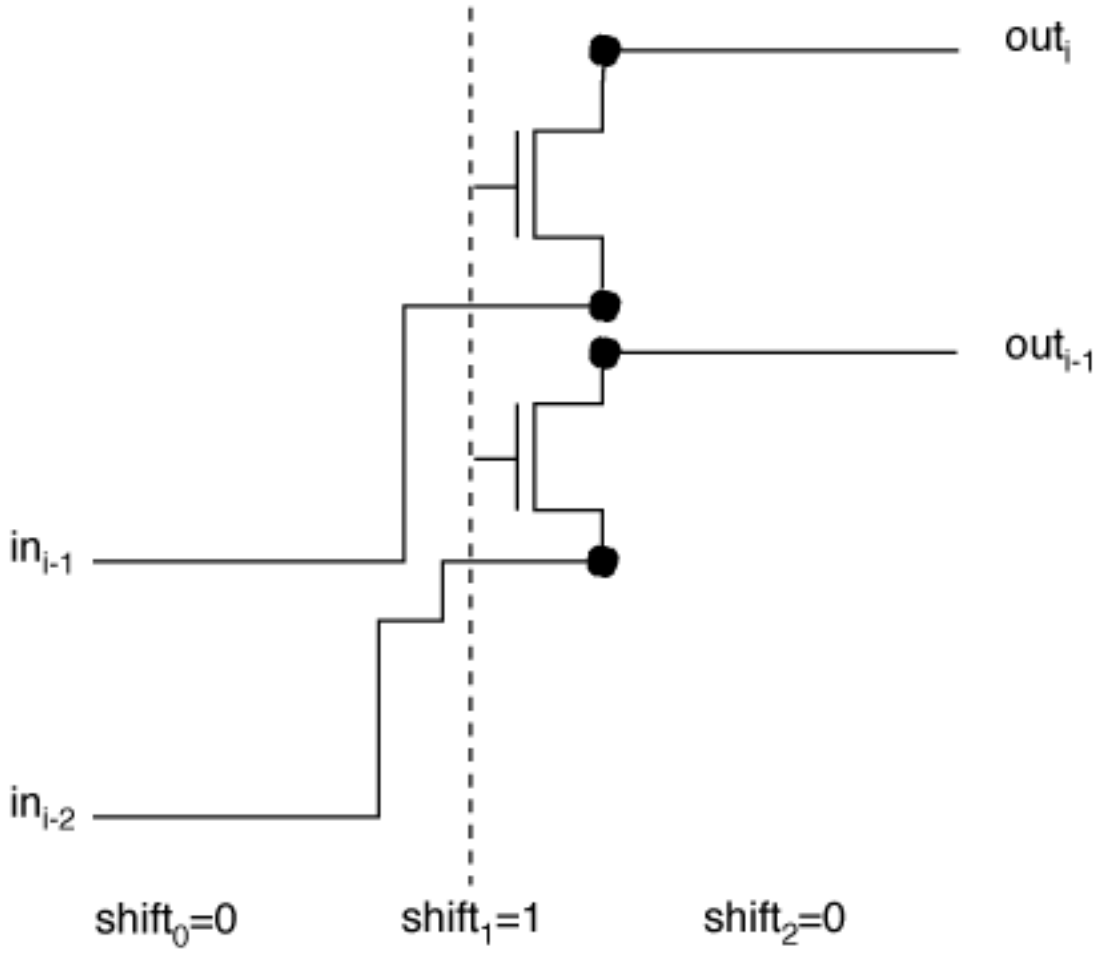
Barrel shifter operation

- Selects arbitrary contiguous *n* bits out of *2n* input bits.
- Examples:
  - right shift: data into top, 0 into bottom;
  - left shift: 0 into top, data into bottom;
  - rotate: data into top and bottom.

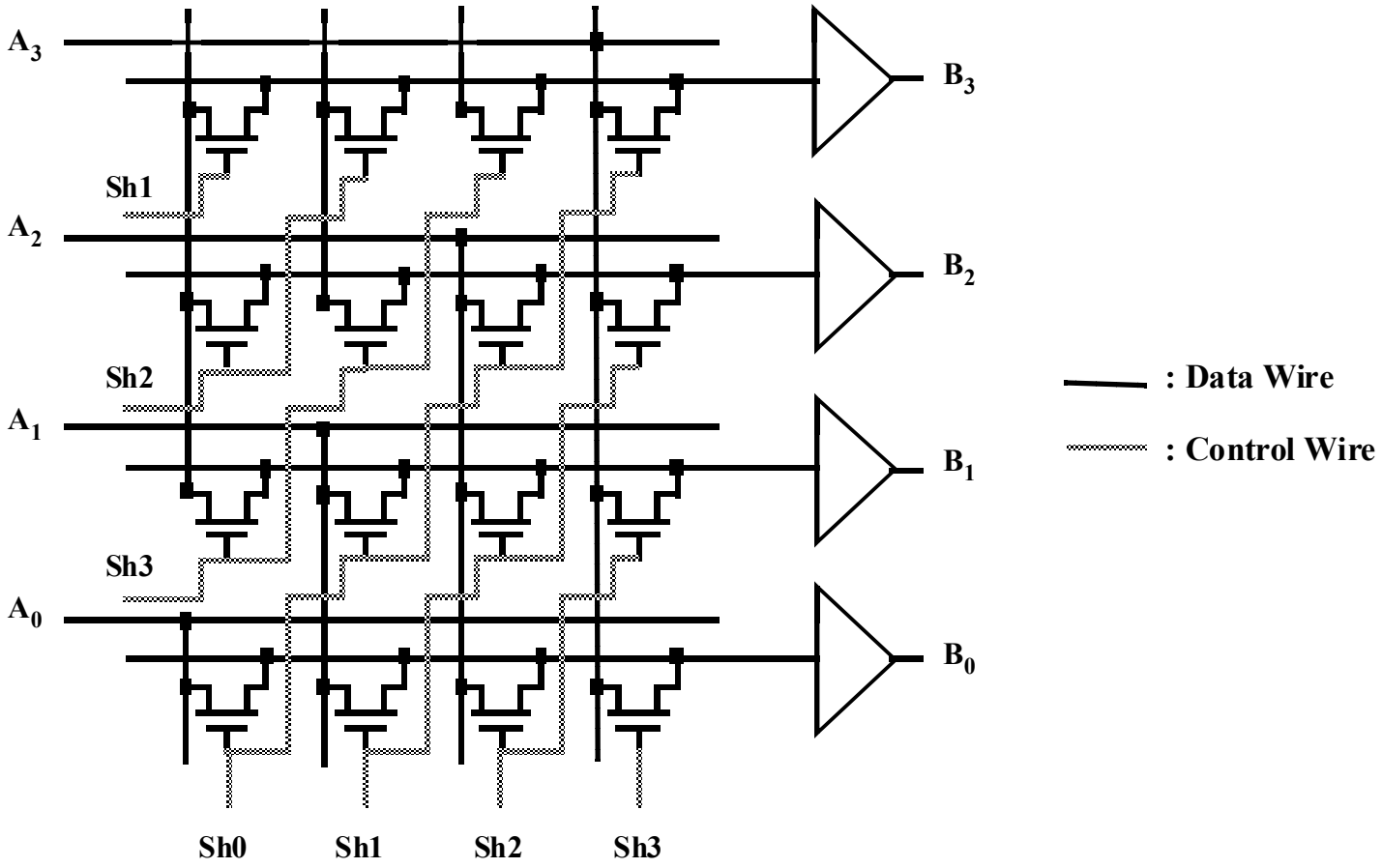
Barrel shifter cell



Barrel shifter in action



The Barrel Shifter

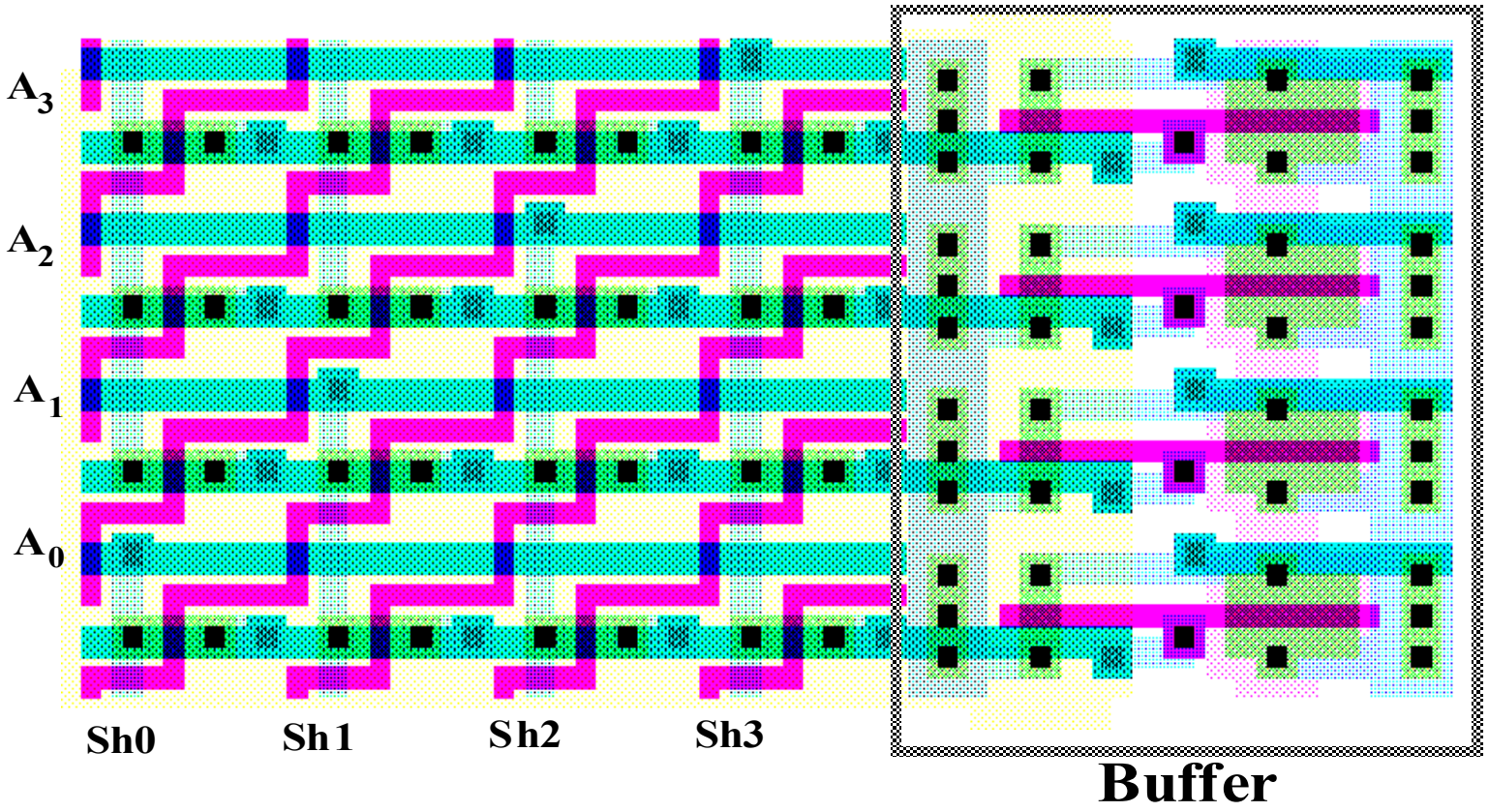


Area Dominated by Wiring

Barrel shifter layout

- Two-dimensional array of *2n* vertical X *n* horizontal cells.
- Input data travels diagonally upward. Output wires travel horizontally.
- Control signals run vertically. Exactly one control signal is set to 1, turning on all transmission gates in that column.

4x4 barrel shifter



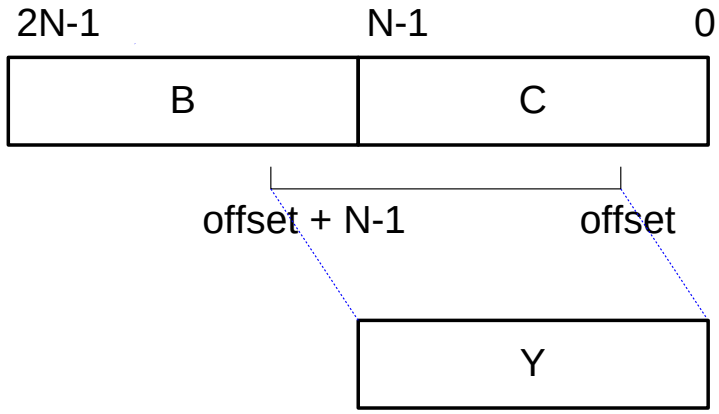
$$Width_{barrel} \sim 2\, p_m\, M$$

Analysis

- Large number of cells, but each one is small.
- Delay is large, considering long wires and transmission gates.

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

- Computing N-k requires an adder

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	$A_{N-1}…A_0$	$A_{N-1}…A_0$	$N-k$

Simplified Funnel Shifter

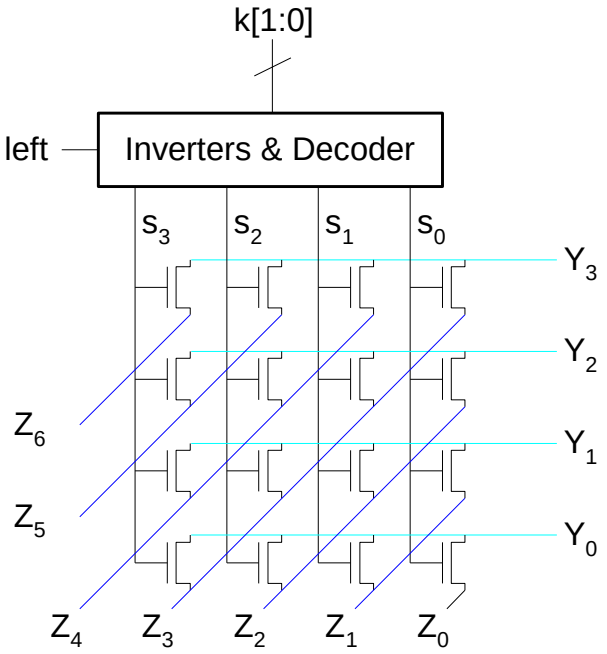
- Optimize down to 2N-1 bit input

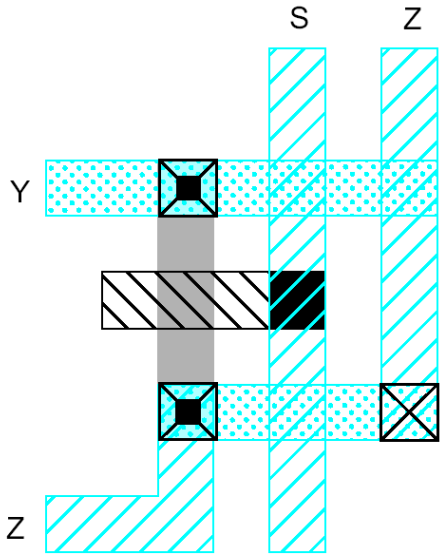
Table 10.11 Simplified funnel shifter

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

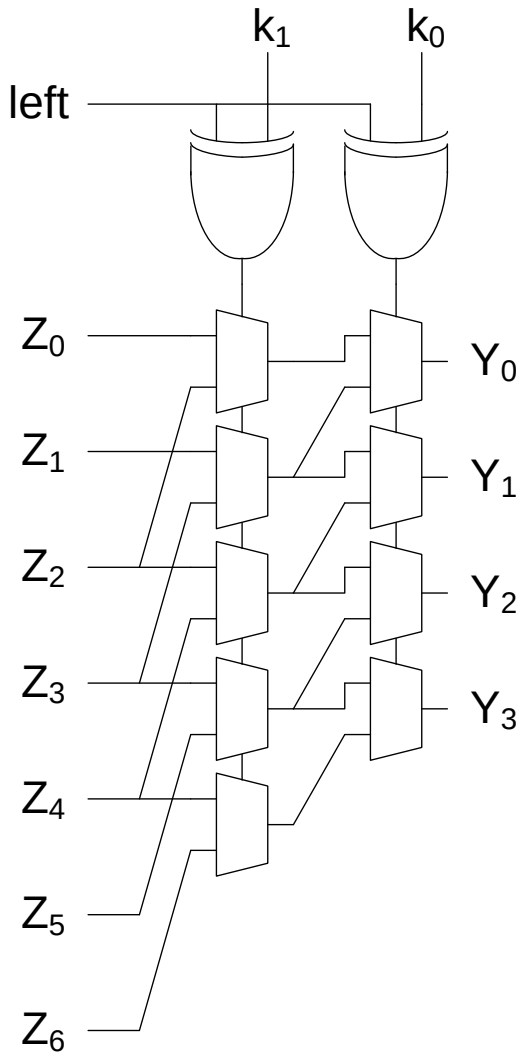
- N N-input multiplexers
  - Use 1-of-N hot select signals for shift amount
  - nMOS pass transistor design (V_t drops!)





Funnel Shifter Design 2

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



Outline

1. Addition/Subtraction
2. One/Zero Detectors
3. Comparators
4. Counters
5. Shifters
6. **Multiplication**

Multiplication

- Example:

$$\begin{array}{ll} 1100 & : 12_{10} \\ \underline{0101} & : 5_{10} \end{array}$$

Multiplication

- Example:

$$\begin{array}{ll} 1100 & : 12_{10} \\ \underline{0101} & : 5_{10} \\ 1100 & \end{array}$$

Multiplication

- Example:

$$
\begin{array}{ll}
1100 & : 12_{10} \\
\underline{0101} & : 5_{10} \\
1100 & \\
0000 &
\end{array}
$$

Multiplication

- Example:

$$
\begin{array}{rl}
1100 & : 12_{10} \\
\underline{0101} & : 5_{10} \\
1100 & \\
0000 & \\
1100 & \\
\hline
\end{array}
$$

Multiplication

- Example:

$$
\begin{array}{rl}
1100 & : 12_{10} \\
\underline{0101} & : 5_{10} \\
1100 & \\
0000 & \\
1100 & \\
0000 & \\
\hline
\end{array}
$$

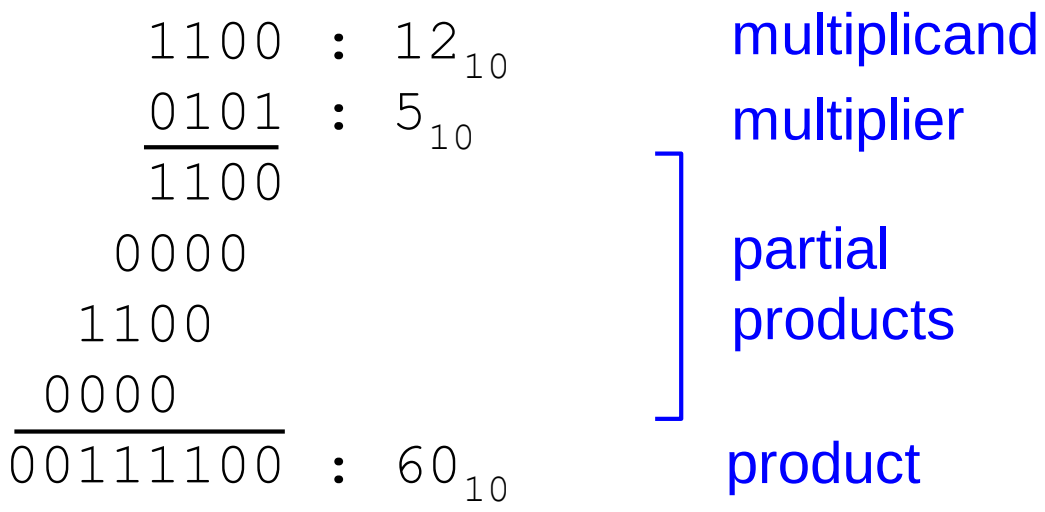
Multiplication

- Example:

$$
\begin{array}{rl}
1100 & : 12_{10} \\
0101 & : 5_{10} \\
\hline
1100 & \\
0000 & \\
1100 & \\
0000 & \\
\hline
00111100 & : 60_{10}
\end{array}
$$

Multiplication

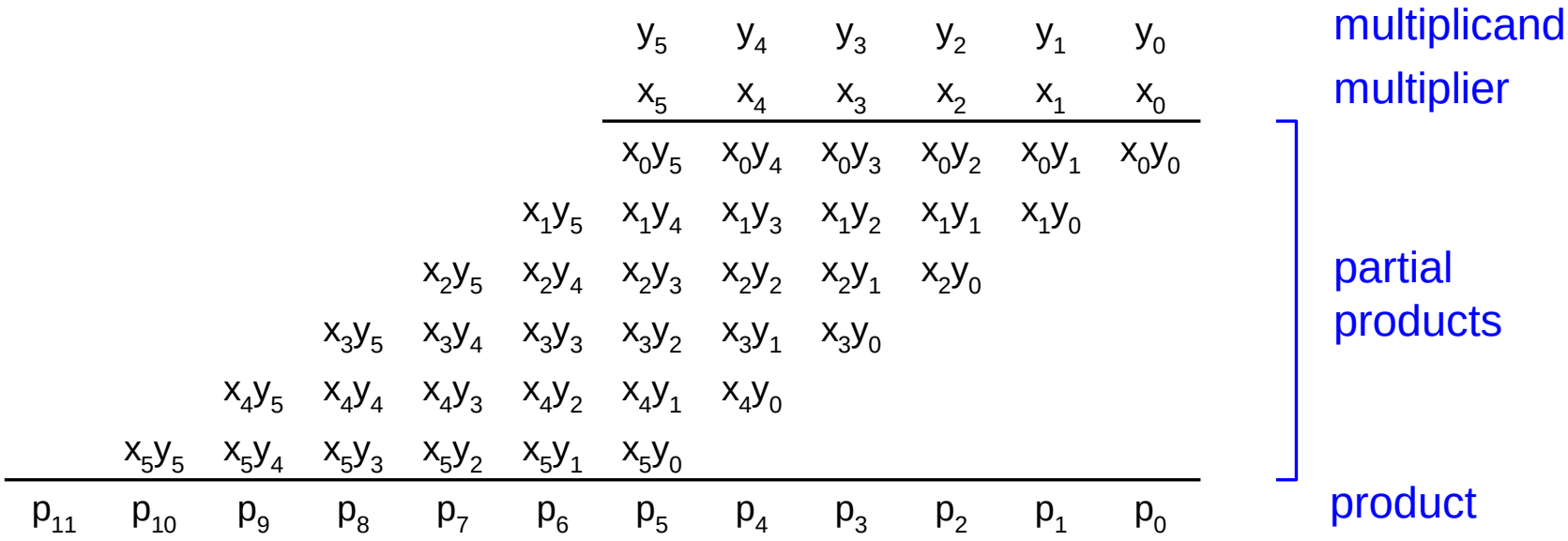
- Example:



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

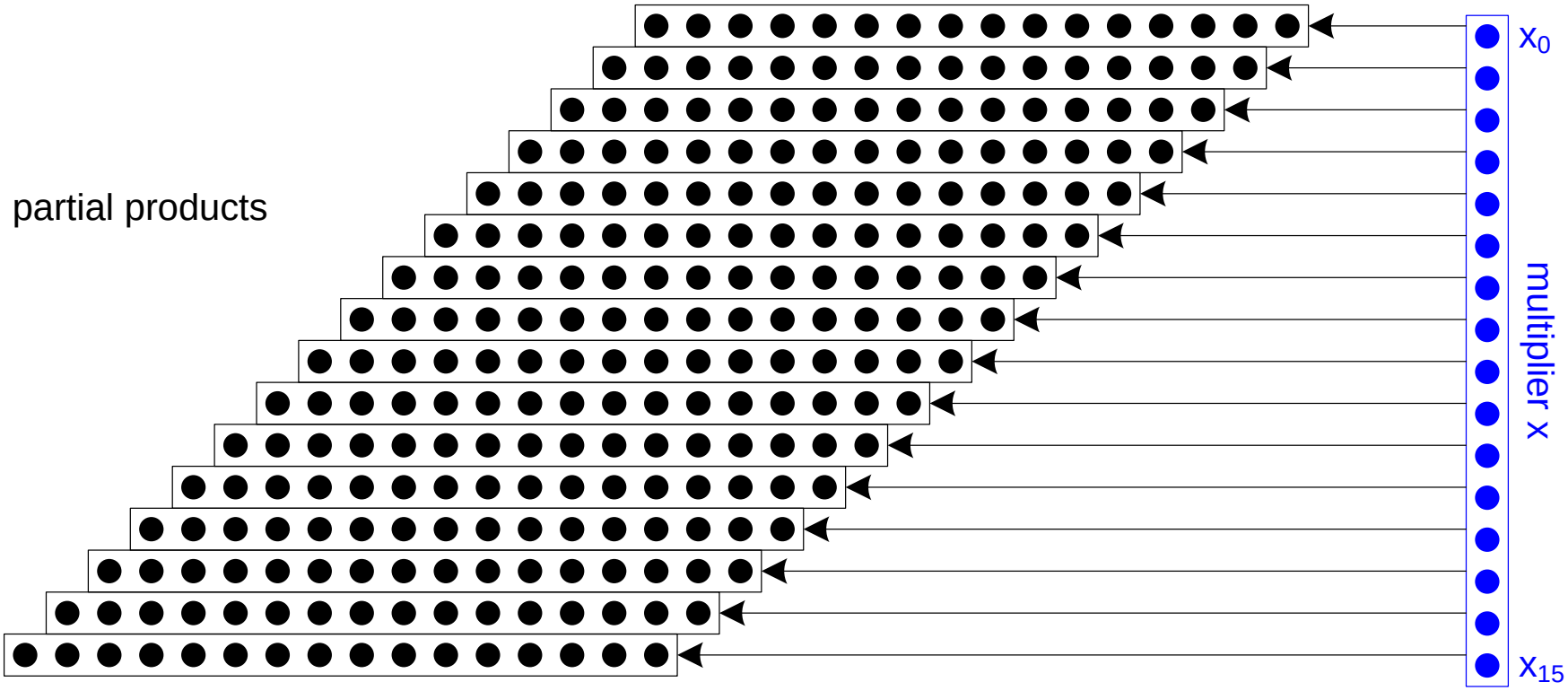
General Form

- 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

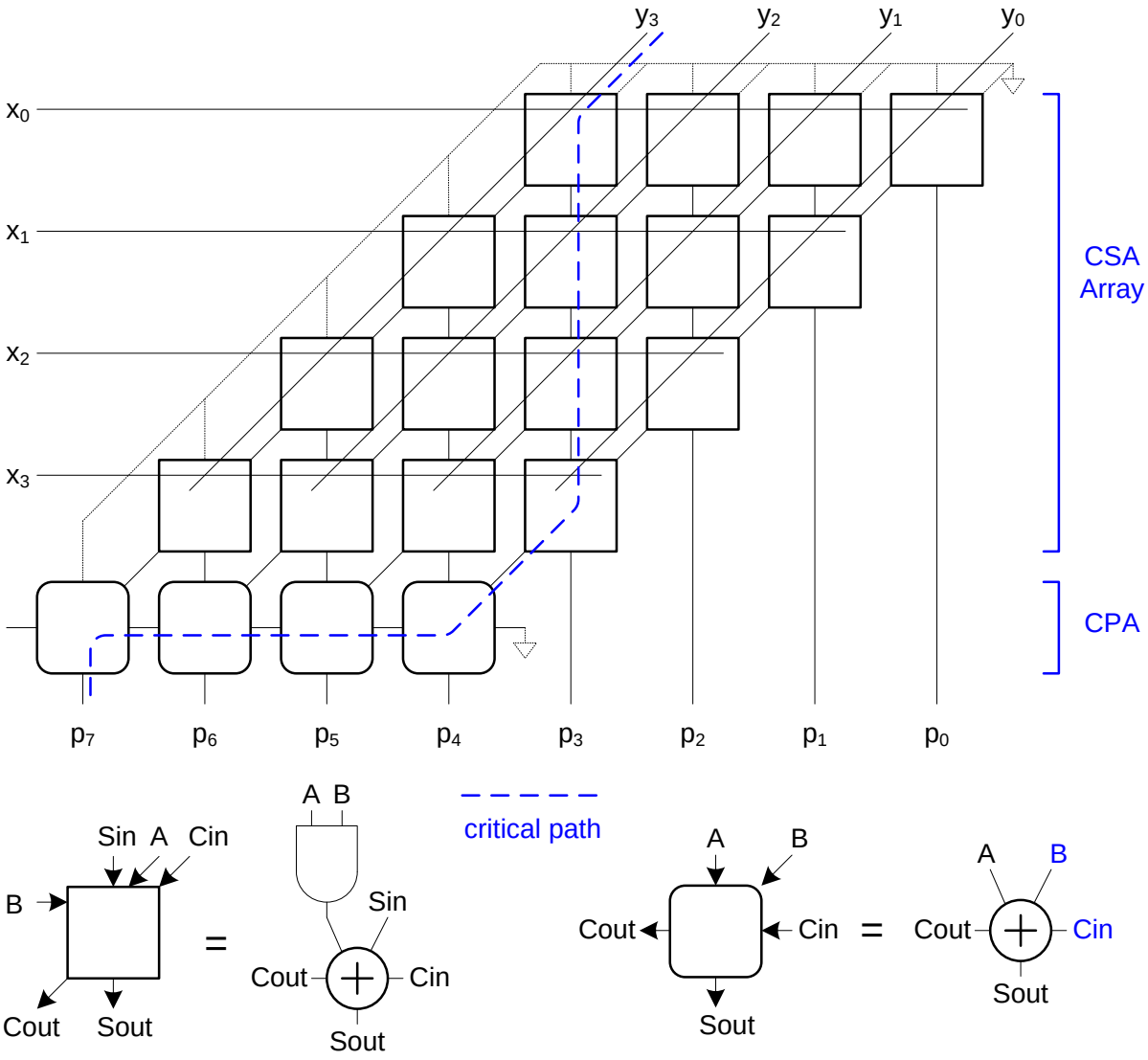
- Each dot represents a bit



Unsigned Array Multiplication

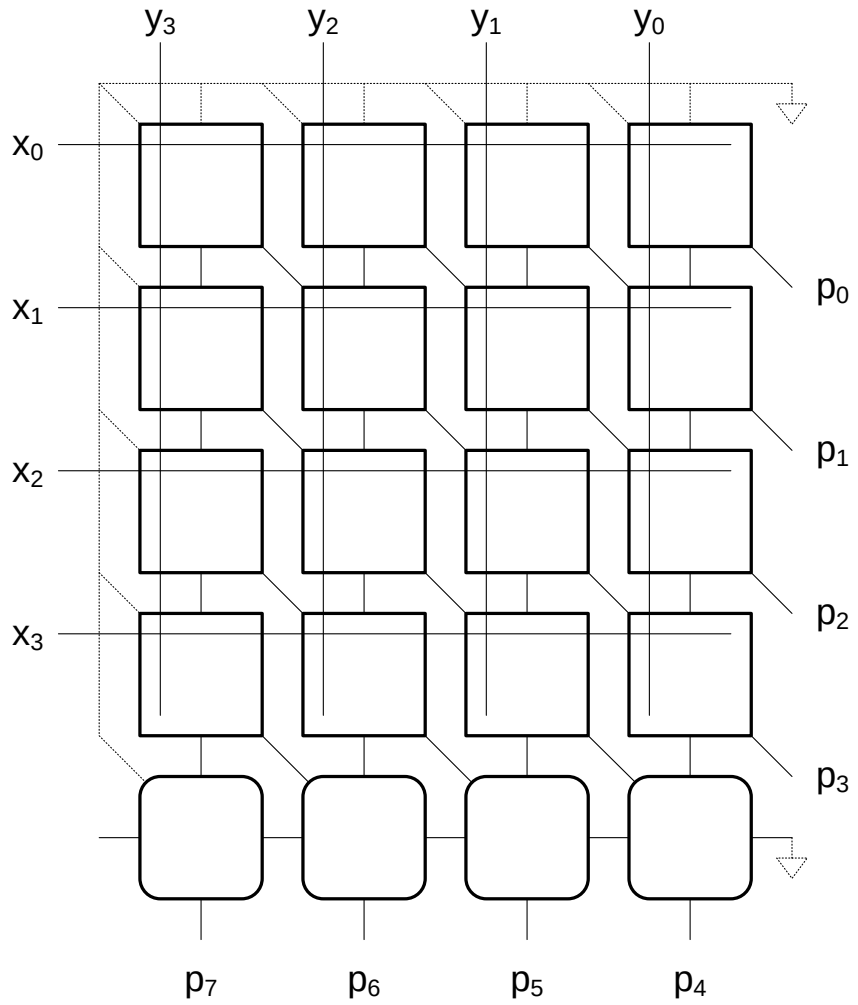
- Fast multipliers use carry-save adders (CSA) to sum the partial products.
  - A CSA typically has a delay of 1.5-2 FO4 inverters independent of the width of the partial product,
  - A carry-propagate adder (CPA) tends to have a delay of 4-15 FO4 inverters depending on the width, architecture, and circuit family.

Array Multiplier

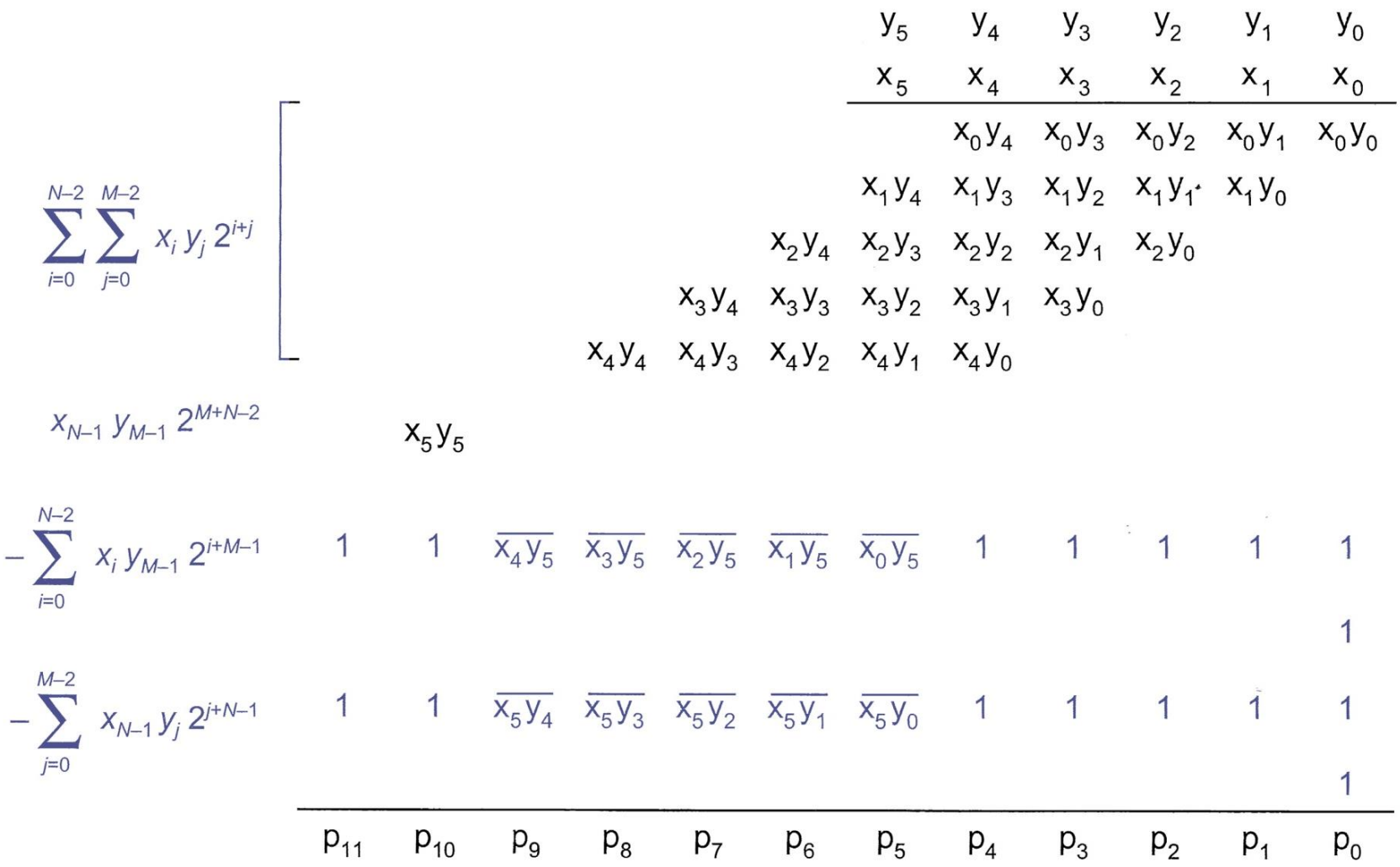


Rectangular Array

- Squash array to fit rectangular floorplan

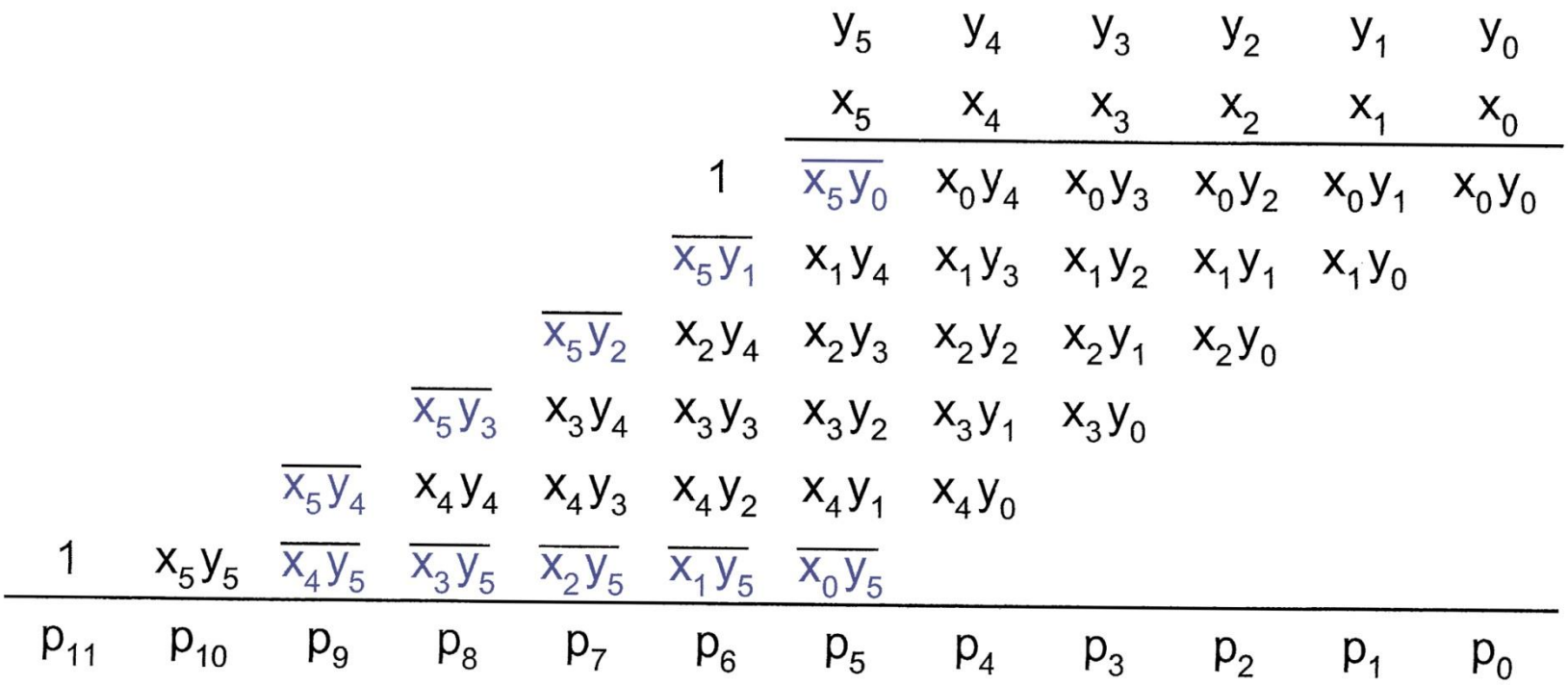


2's Complement Array Multiplication



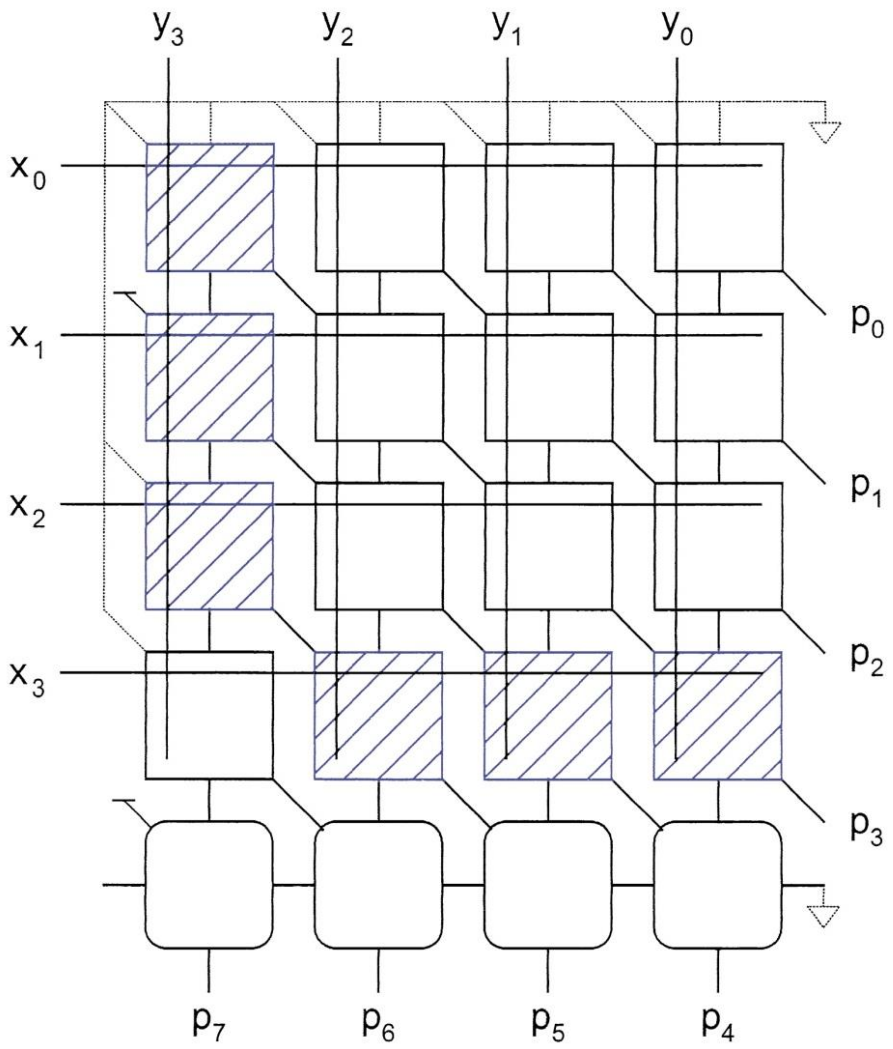
2's Complement Array Multiplication

- Modified Baugh-Wooly multiplier



Modified Baugh-Wooly multiplier

- It is almost identical to the unsigned array multiplier except
  - the AND gates are replaced by NAND gates in the hatched cells
  - 1's are added in place of 0's at a few of the unused inputs.



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

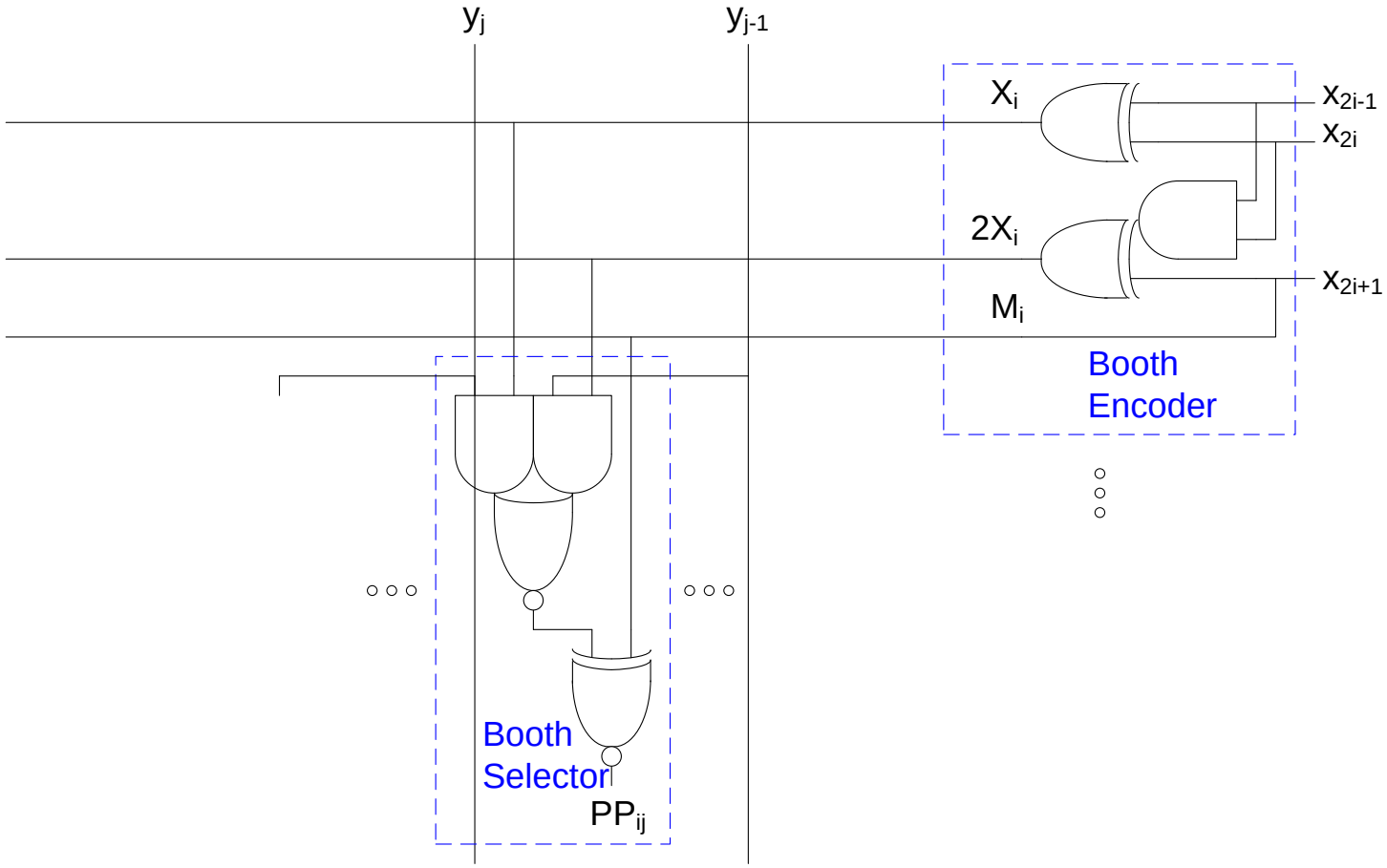
- Modified Booth encoding
  - Avoid generating 3Y by using negative partial products

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

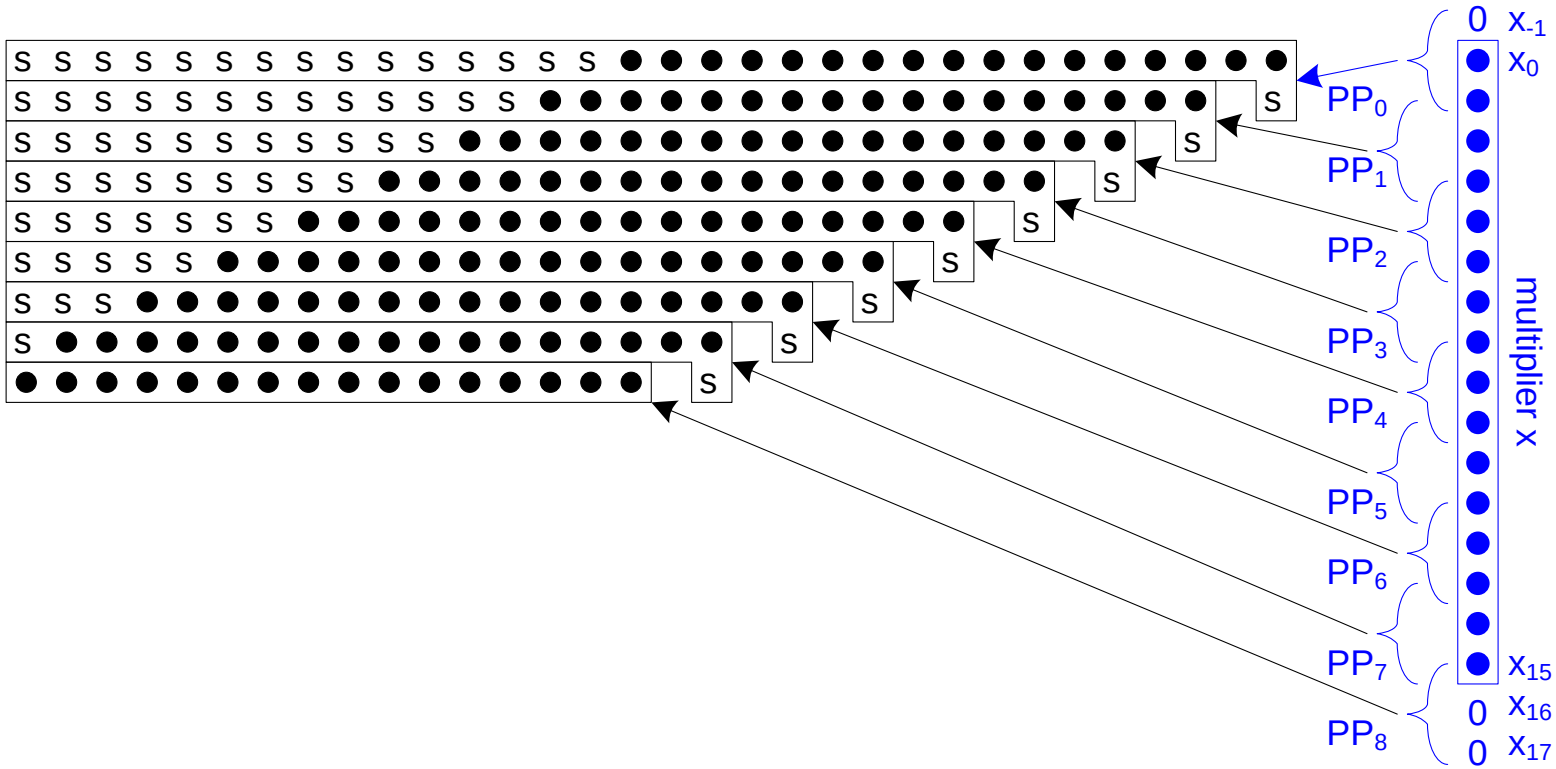
Booth Hardware

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



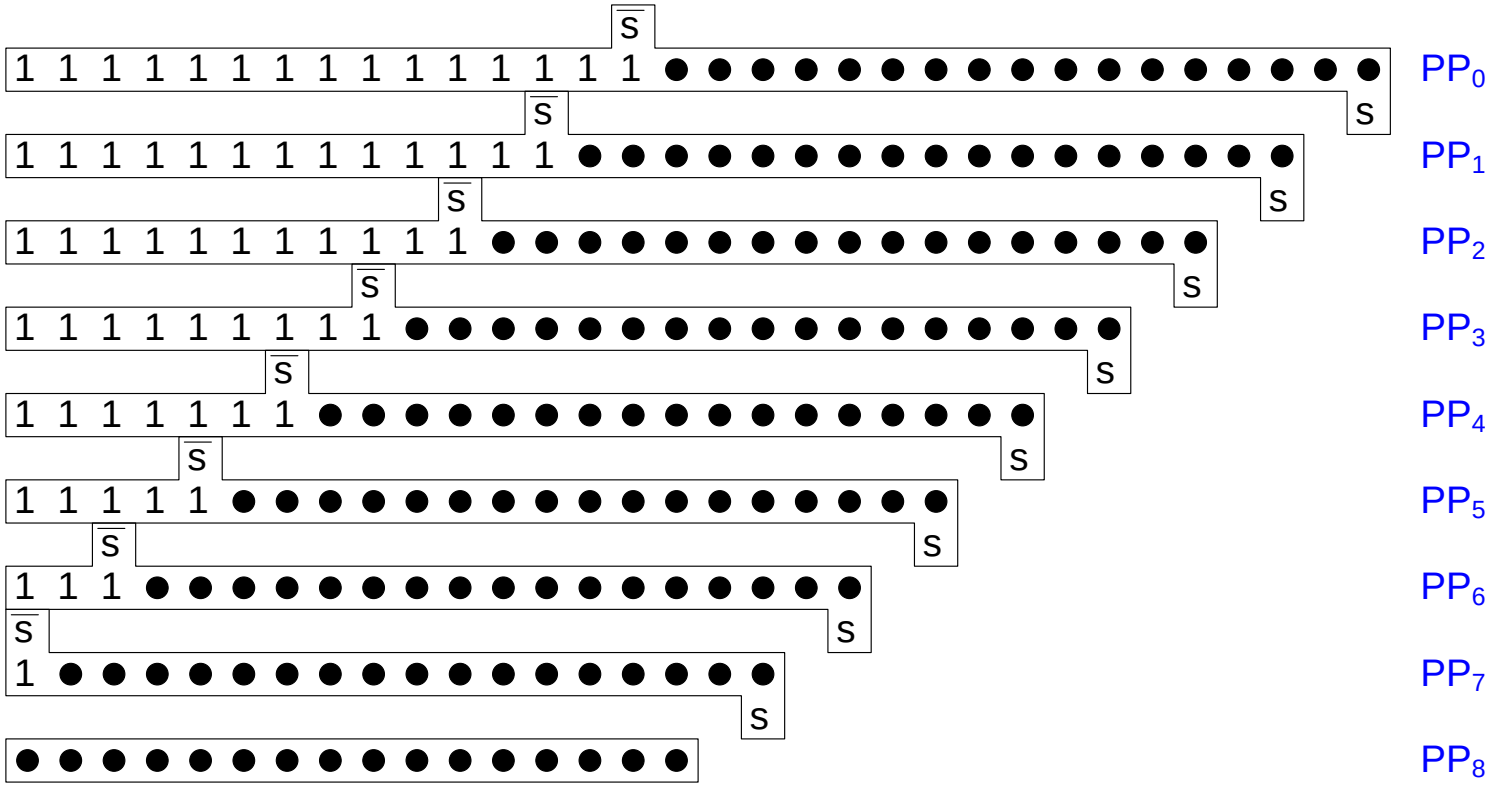
Sign Extension

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



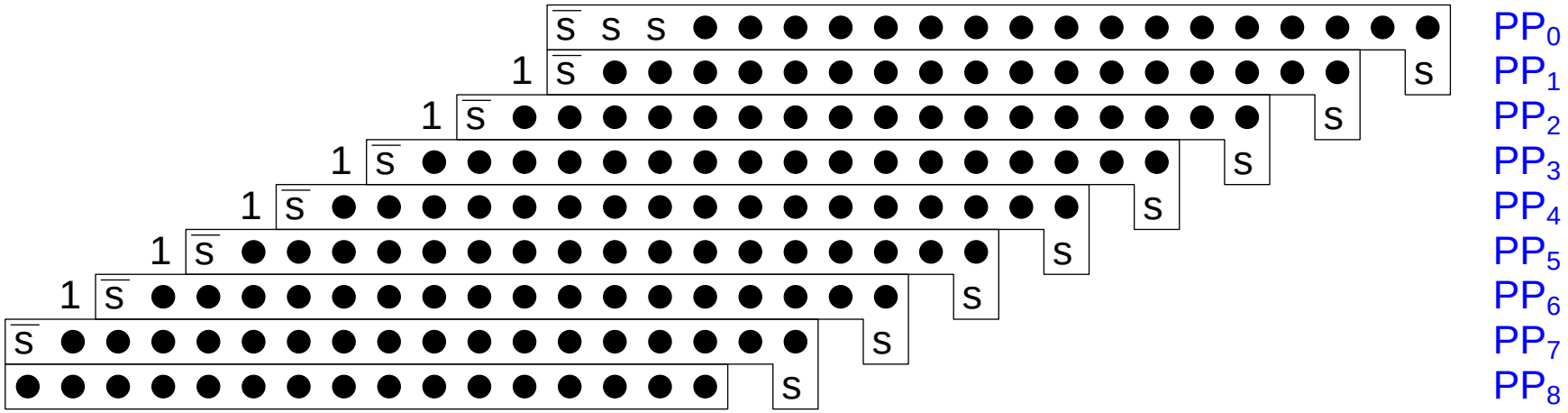
Simplified Sign Ext.

- 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



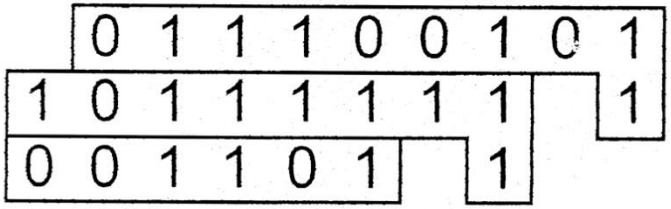
Even Simpler Sign Ext.

- No need to add all the 1's in hardware
  - Precompute the answer!

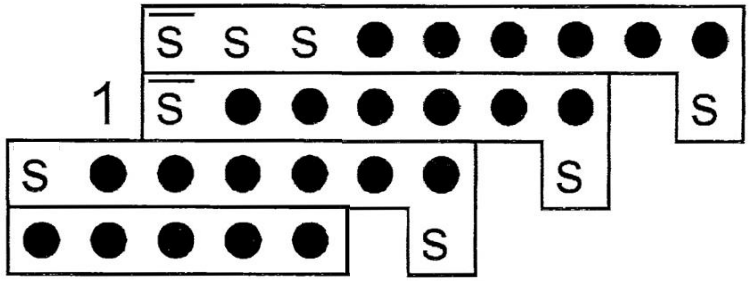


Booth Encoding

- Sketch the partial products used by a radix-4 Booth-encoded multiplier to compute $01110_2 \times 01101_2$



X_{2i+1}	X_{2i}	X_{2i-1}	P_i
1	0	n/a	$-2Y = 100101 + 1$
1	1	1	$-0Y = 111111 + 1$
n/a	0	1	$Y = 001101 + 0$



PP_0
PP_1
PP_2
PP_3

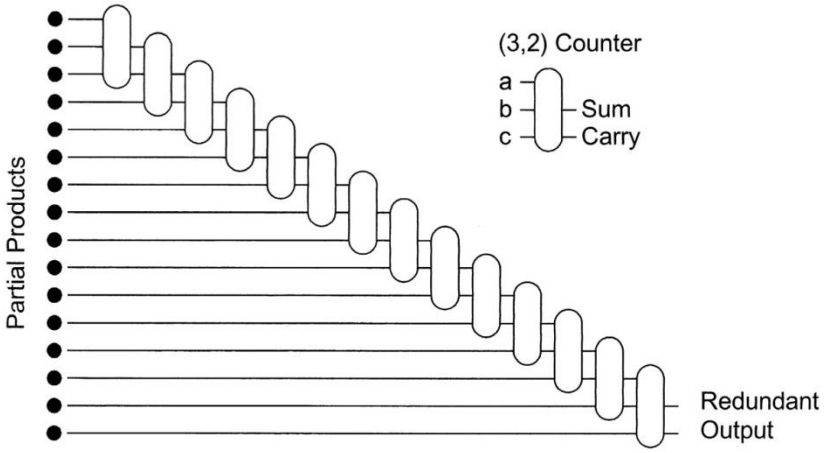
PP_3 is necessary in case PP_2 is negative

Wallace Tree Multiplication

- CSA is effective a “1’s counter”, known as (3,2) counter

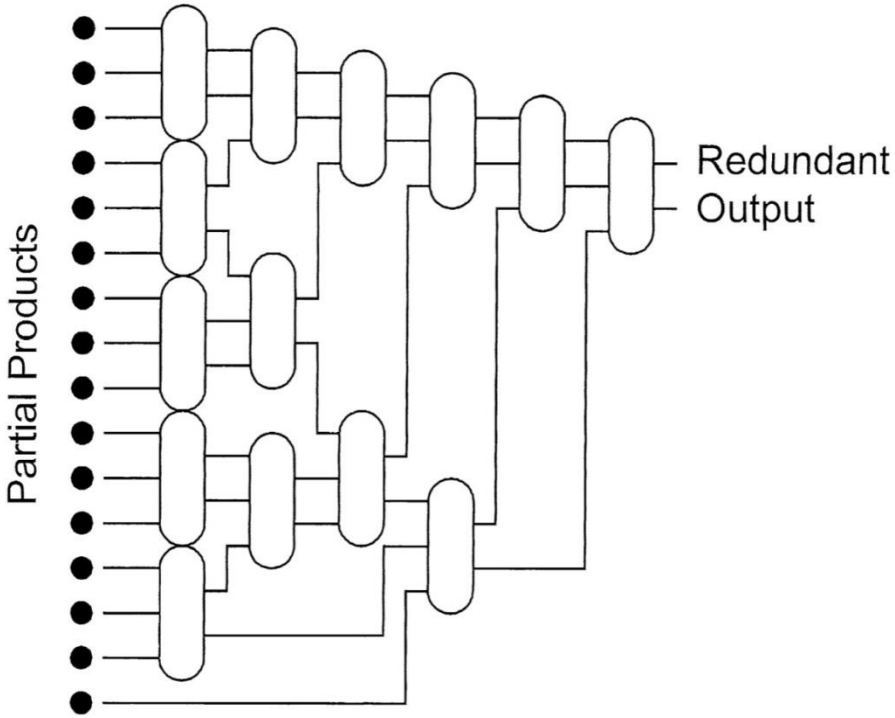
Table 10.14 An adder as a 1’s counter

A	B	C	Carry	Sum	Number of 1’s
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3



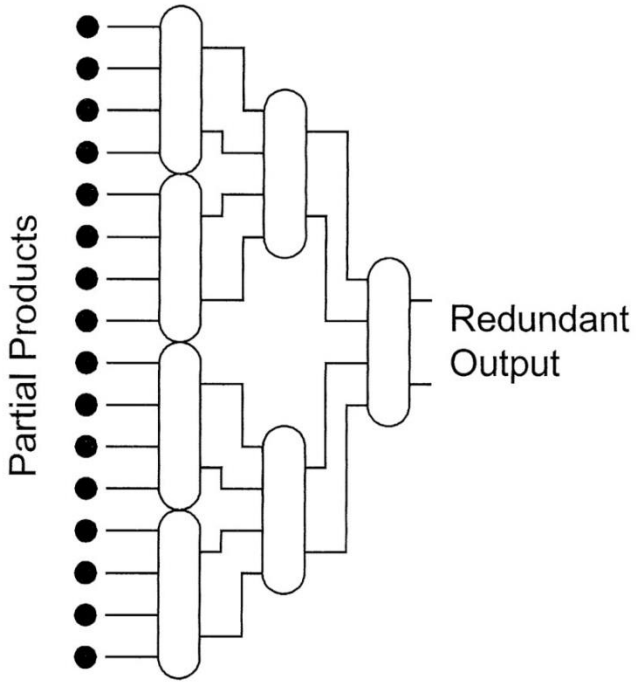
Wallace Tree Multiplication

- Wallace tree architecture can speed the column addition, requires $[\log_{3/2}(N/2)]$ levels of (3,2) counters.

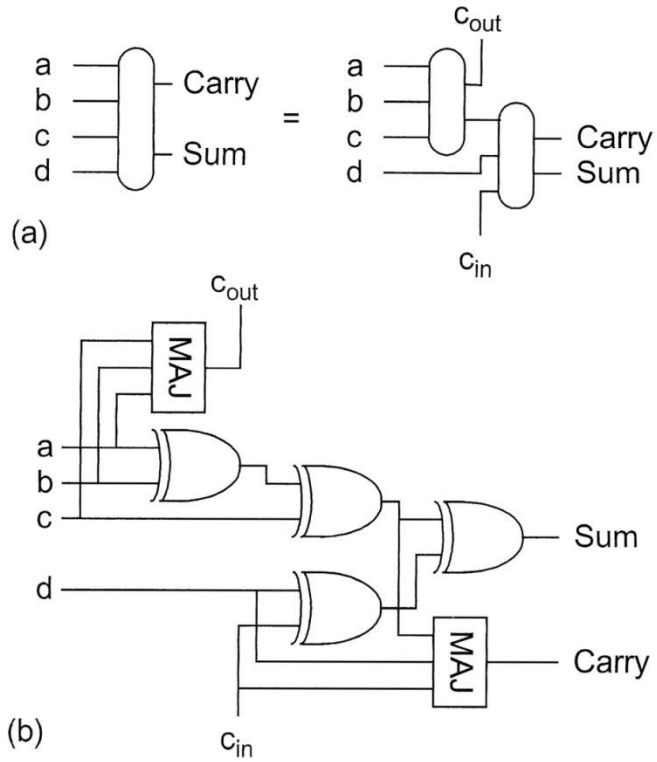


Wallace Tree Multiplication

- [4:2] compressors are used to produce a more regular layout, requires [$\log_2(N/2)$] levels of (5,3) counters.

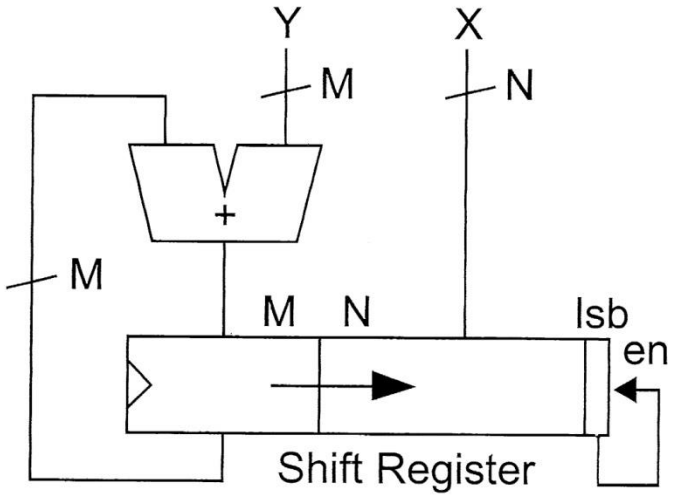


[4:2] compressor



Serial Unsigned Multiplication

- Serial multiplication need far less hardware, but requires multiple clock cycles to operate.



Step	Shift Reg	Notes
	0000 | 0101	initialize
0a	1100 | 0101	add 1*Y
0b	01100 | 010	shift right
1a	01100 | 010	add 0*Y
1b	001100 | 01	shift right
2a	111100 | 01	add 1*Y
2b	0111100 | 0	shift right
3a	0111100 | 0	add 0*Y
3b	00111100 |	shift right

Summary

- For synthesis/APR design flow: choose proper adder/multiplier with various area/speed tradeoff based on applications.
- Custom design at schematic level provide most optimization of performance.
- The wiring capacitance need to be concerned in multiplier design, compact cell and short wires can be fast as well as small and low in power.