Lecture 12: Adders

Based on material prepared by prof. Visvesh S. Sathe

ELECTRICAL & COMPUTER ENGINEERING

Acknowledgements

All class materials (lectures, assignments, etc.) based on material prepared by Prof. Visvesh S. Sathe, and reproduced with his permission

Visvesh S. Sathe
Associate Professor
Georgia Institute of Technology
https://psylab.ece.gatech.edu

UW (2013-2022)
GaTech (2022-present)

A note on number systems

- Digital systems use a binary (base-2) representation
  - 1001 → $1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 9$
  - Works naturally with the 2-state nature of digital logic
- Two dominant representations used in digital systems
  - Integer
    - Sign-Magnitude, 1's Compliment
    - **2's Compliment** (Overwhelmingly popular for integer computation)
    - $M = b_n\ b_{n-1}\ b_{n-2}\ b_{n-3}\ \ldots\ b_2\ b_1\ b_0 \rightarrow M = -b_n 2^n + \sum_i^{n-1} b_i 2^i$
    - Range = $[-2^{n-1}, 2^{n-1}-1]$
  - Floating Point (IEEE 754 Standard)
    - 32-bit representation as Sign-Bit_Exponent_Mantissa (1,8,23)
    - $F = -1^{sign}\left(1 + \sum_{i=1}^{23} b_{23-i} 2^{i-1}\right) \times 2^{exp-127}$
    - Range {0, [$1.4 \times 10^{-45*}$, 3.4×10^{38}], NaN}
  - So, how do I get a much wider range with the same number of bits?

A note on number systems

- Digital systems use a binary (base-2) representation
  - 1001 → $1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 9$
  - Works naturally with the 2-state nature of digital logic
- Two dominant representations used in digital systems
  - Integer
    - Sign-Magnitude, 1's Compliment
    - **2's Compliment** (Overwhelmingly popular for integer computation)
    - $M = b_n\ b_{n-1}\ b_{n-2}\ b_{n-3}\ \ldots\ b_2\ b_1\ b_0 \rightarrow M = -b_n 2^n + \sum_i^{n-1} b_i 2^i$
    - Range = $[-2^{n-1}, 2^{n-1}-1]$
  - Floating Point (IEEE 754 Standard)
    - 32-bit representation as Sign-Bit_Exponent_Mantissa (1,8,23)
    - $F = -1^{sign}\left(1 + \sum_{i=1}^{23} b_{23-i} 2^{i-1}\right) \times 2^{exp-127}$
    - Range {0, [1.4 X 10^{-45*}, 3.4X10^{38}], NaN}
  - So, how do I get a much wider range with the same number of bits?
- This course: 2's complement integer representation

Number Systems: 2's Complement

- MSB weighted negatively, added to remaining positive n-1 bit number
  - 1xxxxx → Negative number
  - 0xxxxxx → Positive number
  - Covers range from -2^{n-1} to $2^{n-1}-1$
- Key motivations
  - Represent positive and negative numbers efficiently
  - Enable +, - operations to be readily performed with an adder*
- But bit-wise, logical representation allows for easy conversion
  - 15 as an 8-bit number is trivial, how about -15*
  - Will have to solve for b_n, b_{n-1}, b_{n-2}, b_{n-3} … b_2, b_1, b_0 ?

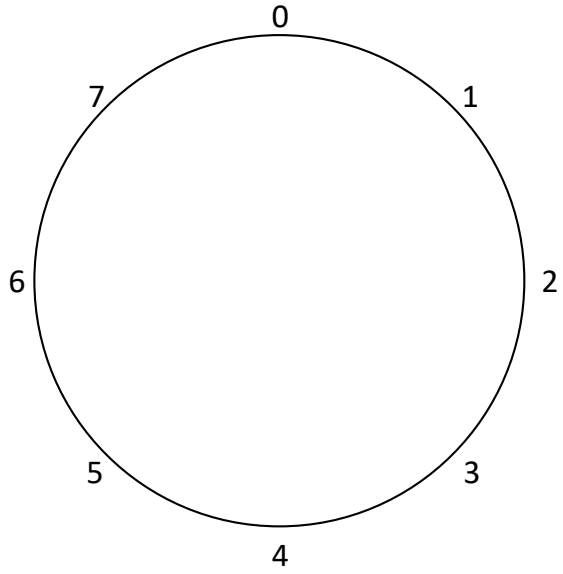
$$M = b_n \; b_{n-1} \; b_{n-2} \; b_{n-3} \; \ldots \; b_2 \; b_1 \; b_0$$

$$M = -b_n 2^n + \sum_i^{n-1} b_i 2^i$$

2's Complement

- Let $\tilde{X}$ be a binary representation for –X
  - X + $\tilde{X}$ = 0
- n-bit 2's complement
  - Motivation: 8-bit addition: 255 + 1 = ?
  - Divide number space into 2 halfs (almost)
  - Let X = [0,2^{n-1} -1] represent positive numbers
  - Let $\tilde{X}$= 2^n – X
  - X + $\tilde{X}$ = 2^n = 0 (n-bit representation)
- Binary representation for $\tilde{X}$
  - 2^n – X = (2^n-1) - X + 1 = ~X + 1
- Useful algebraic representation of a 2's complement number $M = b_n\ b_{n-1}\ \ldots\ b_2\ b_1 b_0$

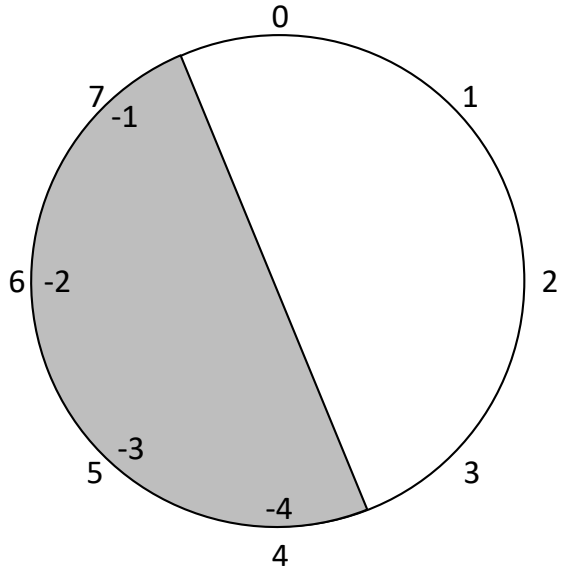
$$M = -\left(b_n 2^{n+1} - \sum_i^n b_i 2^i\right) = -b_n 2^n + \sum_i^{n-1} b_i 2^i$$



2's Complement

- Let $\tilde{X}$ be a binary representation for –X
  - X + $\tilde{X}$ = 0
- n-bit 2's complement
  - Motivation: 8-bit addition: 255 + 1 = ?
  - Divide number space into 2 halfs (almost)
  - Let X = [0,2^{n-1} -1] represent positive numbers
  - Let $\tilde{X}$= 2^n – X
  - X + $\tilde{X}$ = 2^n = 0 (n-bit representation)
- Binary representation for $\tilde{X}$
  - 2^n – X = (2^n-1) - X + 1 = ~X + 1
- Useful algebraic representation of a 2's complement number $M = b_n\ b_{n-1}\ \ldots\ b_2\ b_1\ b_0$

$$M = -\left(b_n 2^{n+1} - \sum_i^n b_i 2^i\right) = -b_n 2^n + \sum_i^{n-1} b_i 2^i$$



One Last Note on 2-s Complement

$$M = -b_n 2^n + \sum_i^{n-1} b_i 2^i$$

$$-\mathrm{M} = b_n 2^n + \sum_i^{n-1} -b_i 2^i = -(1-b_n)2^n + 2^n + \sum_i^{n-1}(1-b_i)2^i - 2^i$$

$$= -(1-b_n)2^n + 2^n + \sum_i^{n-1}(1-b_i)2^i - 2^i$$

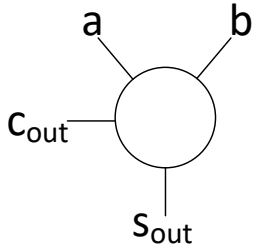
$$= -(1-b_n)2^n + \sum_i^{n-1}(1-b_i)2^i + 2^n - \sum_i^{n-1} 2^i$$

$$= \overline{-b_n}\ \ 2^n + \sum_i^{n-1} \bar{b}_i 2^i + 2^n - (2^n - 1)$$

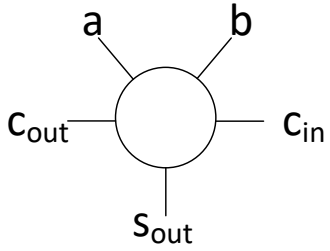
$$= -\overline{b_n}\ \ 2^n + \sum_i^{n-1} \bar{b}_i 2^i + 1 = \sim M + 1$$

- Given a number M, use algebraic representation to show that –M is derived based on ~M+1

1-Bit Adder



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

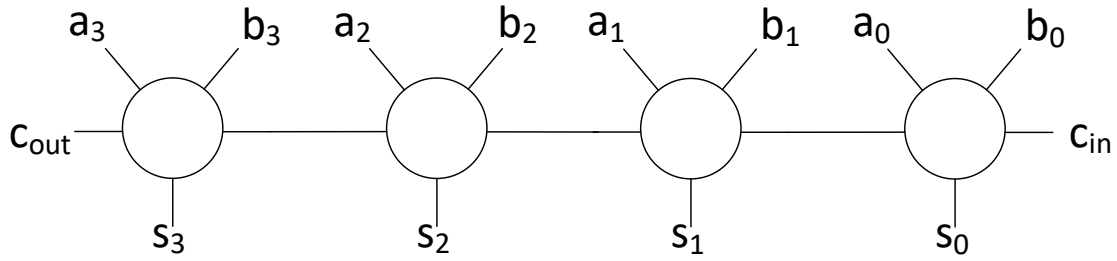


A	B	C_{in}	S_{out}	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Exercise: Implement a full-adder from 2-half adders and any other gates

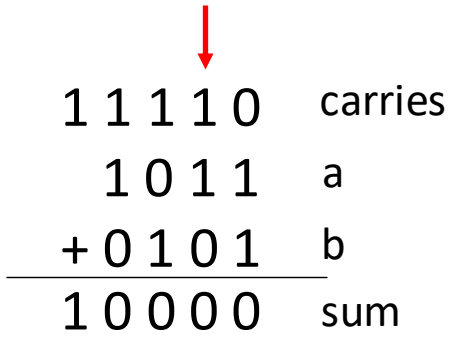
N-bit addition

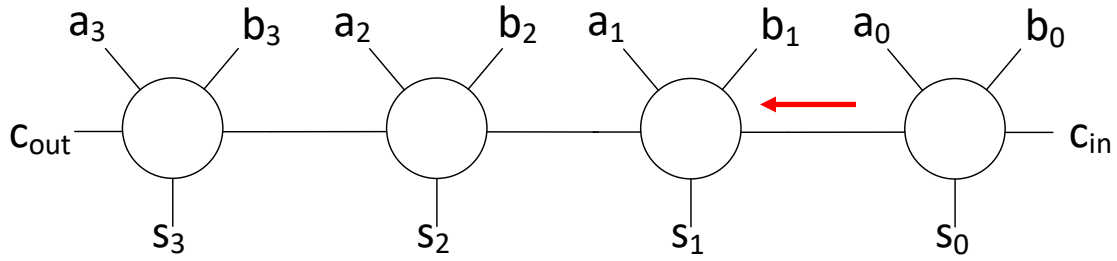
1 1 1 1 0	carries
1 0 1 1	a
+ 0 1 0 1	b
1 0 0 0 0	sum



- Sequential traversal of each adder ➔ t_{delay} = N*D
  - Also known as the Ripple-carry adder
- **Carry propagation sets addition latency**
  - Faster adder topologies must speed-up carry generation
  - Last bit to get C_{in} sets overall adder delay

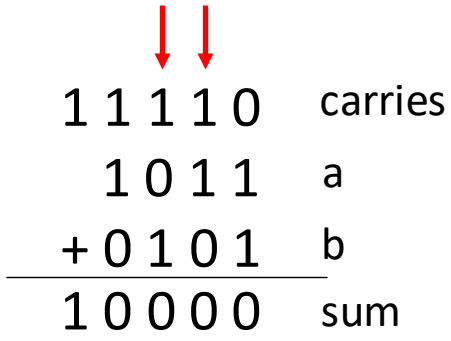
N-bit addition

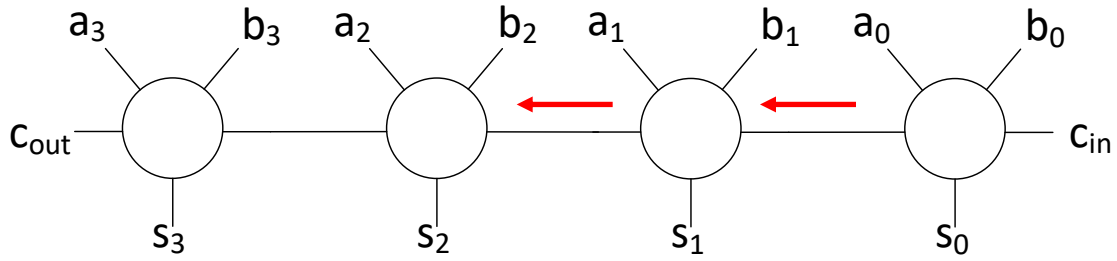




- Sequential traversal of each adder ➔ t_{delay} = N*D
  - Also known as the Ripple-carry adder
- **Carry propagation sets addition latency**
  - Faster adder topologies must speed-up carry generation
  - Last bit to get C_{in} sets overall adder delay

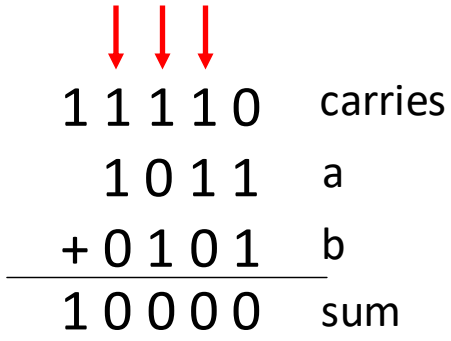
N-bit addition

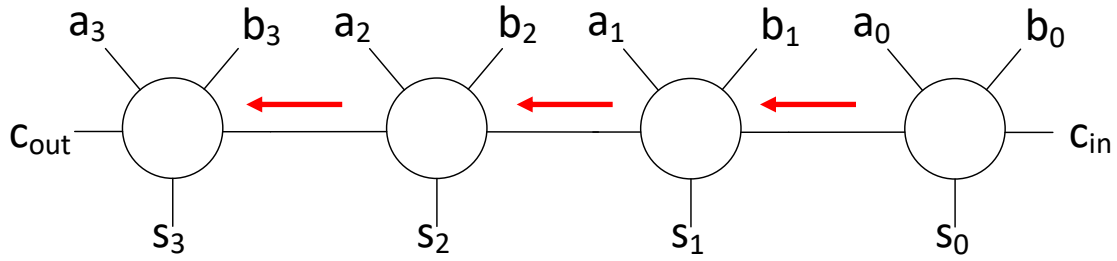




- Sequential traversal of each adder ➔ t_{delay} = N*D
  - Also known as the Ripple-carry adder
- **Carry propagation sets addition latency**
  - Faster adder topologies must speed-up carry generation
  - Last bit to get C_{in} sets overall adder delay

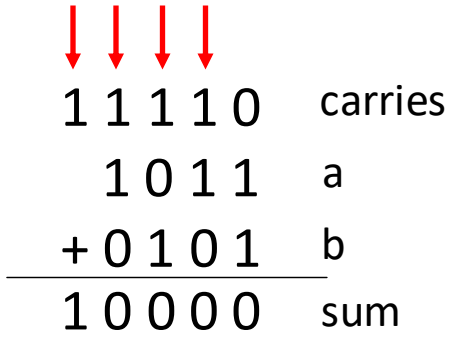
N-bit addition

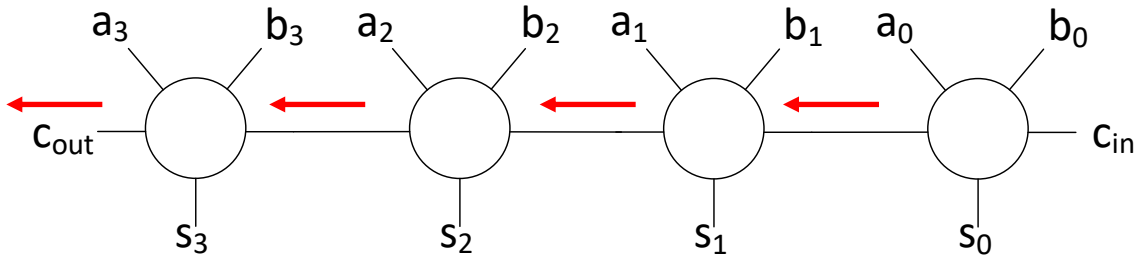




- Sequential traversal of each adder ➔ t_{delay} = N*D
  - Also known as the Ripple-carry adder
- **Carry propagation sets addition latency**
  - Faster adder topologies must speed-up carry generation
  - Last bit to get C_{in} sets overall adder delay

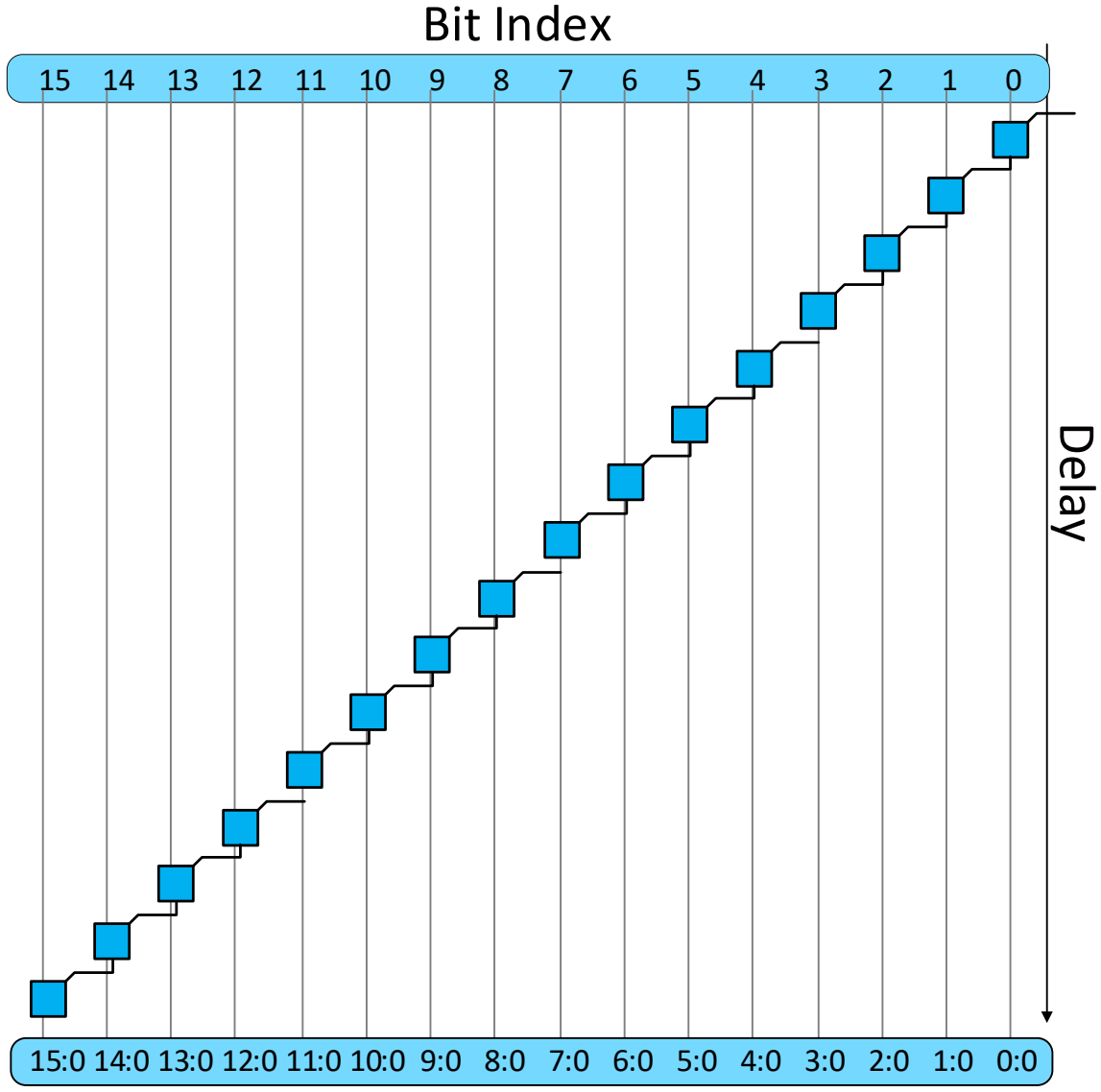
N-bit addition



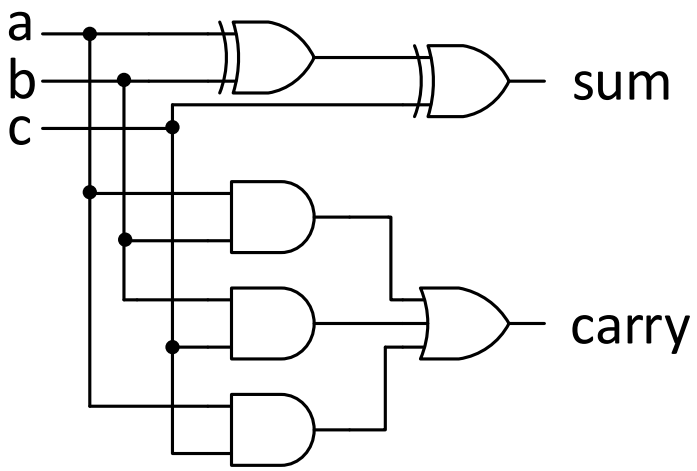


- Sequential traversal of each adder ➔ t_{delay} = N*D
  - Also known as the Ripple-carry adder
- **Carry propagation sets addition latency**
  - Faster adder topologies must speed-up carry generation
  - Last bit to get C_{in} sets overall adder delay

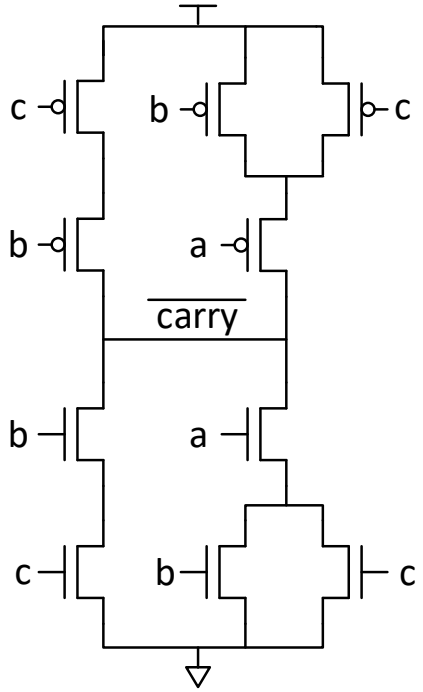
Delay-Diagram for N-bit Ripple Carry Adder

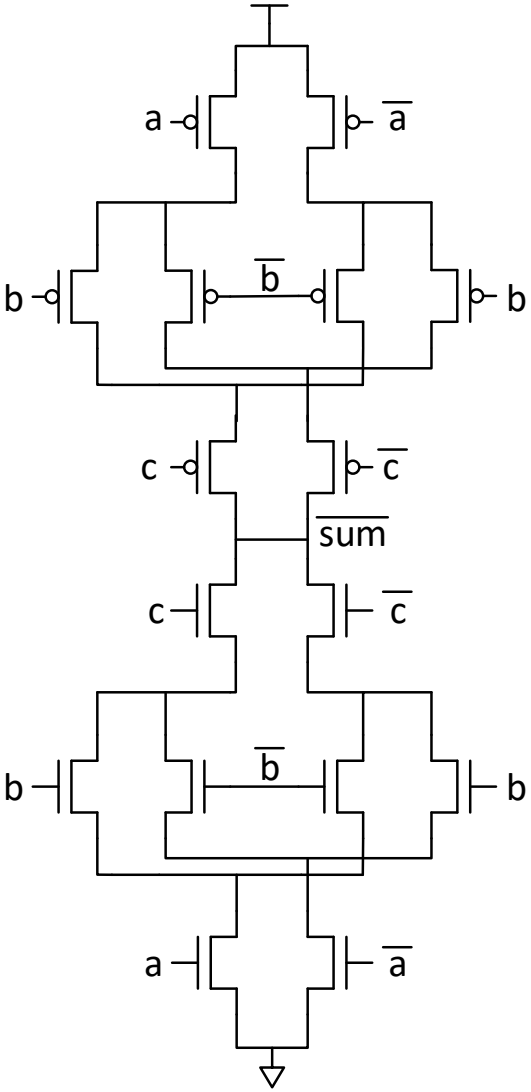


Full-Adder Implementation



Naïve Implementation

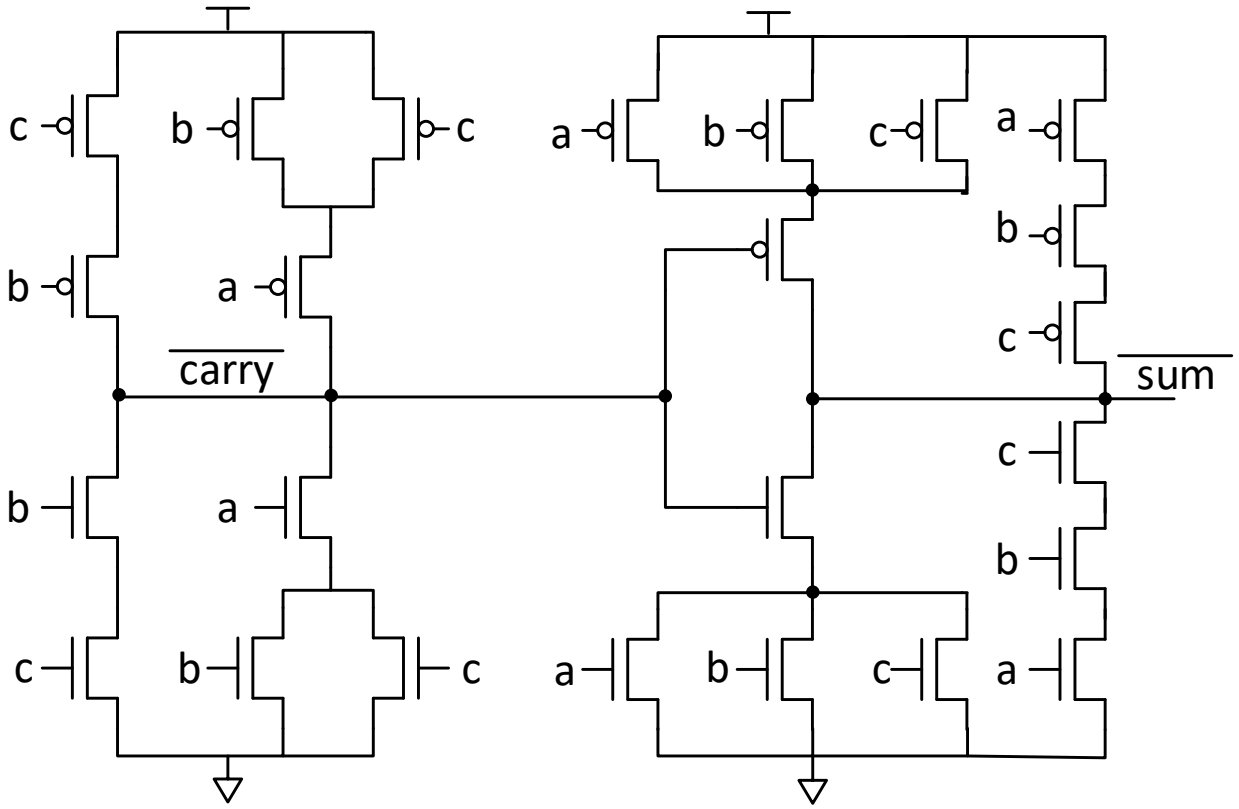




CMOS complex gate Implementation

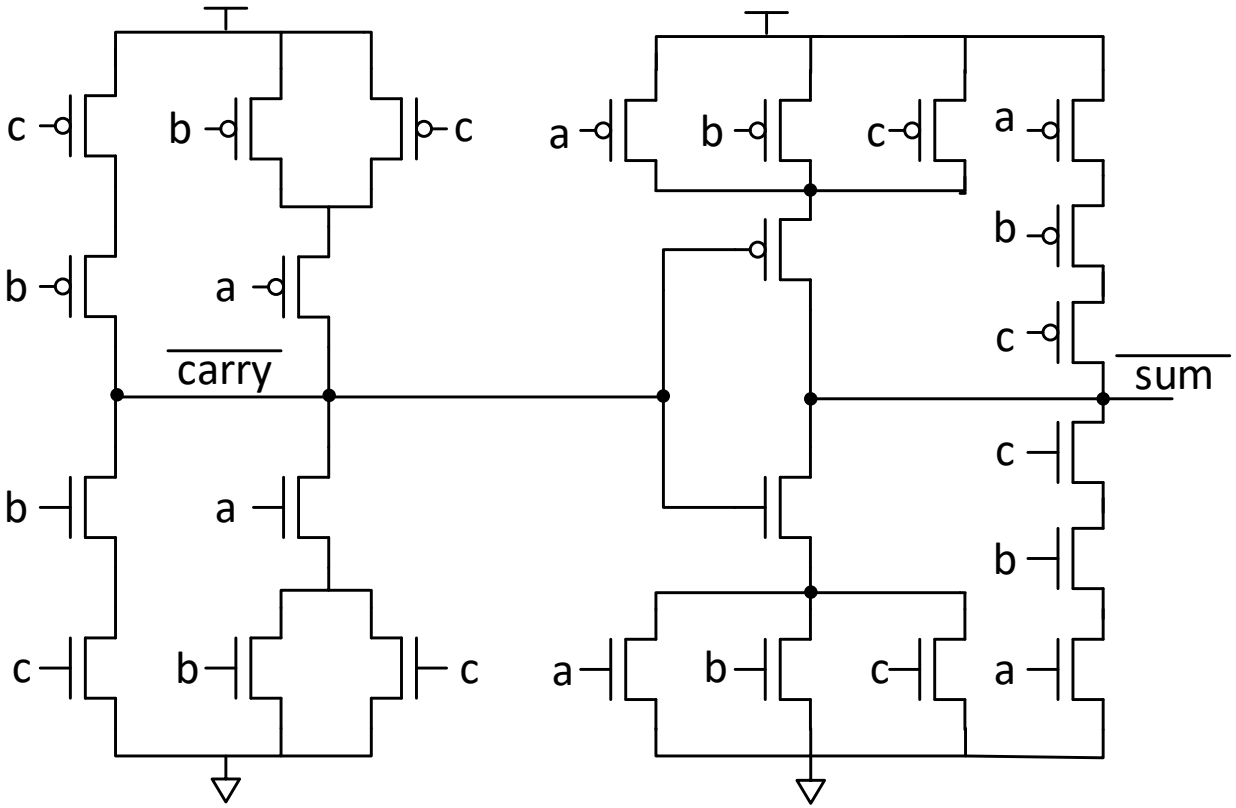
- Naive implementation is not area-efficient
- CMOS complex gate implementation
  - Sum: 3-input XOR
  - Carry: 3-input majority function

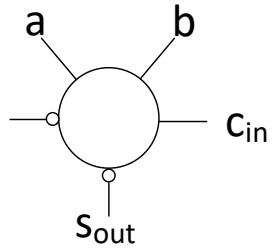
Optimizing the Full-adder



- Full-adder cell
  - Carry generation is critical - generate the Cout from A, B and C_{in}
  - S_{out} can be efficiently be generated using A,B,C_{in} and C_{out}
  - Available in most std-cell libraries

Optimizing the Full-adder





- Full-adder cell
  - Carry generation is critical - generate the Cout from A, B and C_{in}
  - S_{out} can be efficiently be generated using A,B,C_{in} and C_{out}
  - Available in most std-cell libraries

Interesting…
Pulldown=Pullup

Optimizing the Full-adder

S

c \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

C

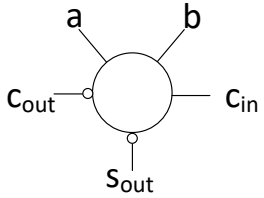
c \ ab	00	01	11	10
0	0	0	1	1
1	0	0	1	1

S

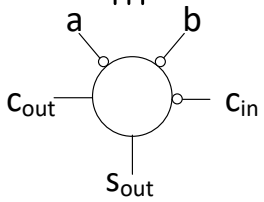
$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0				
1				

C

$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0				
1				



|||



- Notice that the pulldown and pullup of Sum and Carry are **identical**
  - Complete the Karnaugh map for S and Cout in terms of $\bar{A}, \bar{B}, \bar{C}$.
  - Inverting the inputs inverts the outputs
  - Exploit this in the ripple carry adder to avoid inversion (bubble-propagation)

Optimizing the Full-adder

S

c \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

C

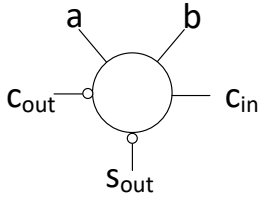
c \ ab	00	01	11	10
0	0	0	1	1
1	0	0	1	1

S

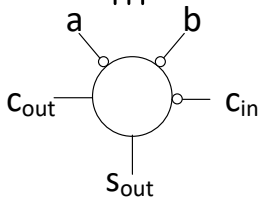
$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0	1	0	1	0
1	0	1	0	1

C

$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0	1	1	0	0
1	1	1	0	0



|||



- Notice that the pulldown and pullup of Sum and Carry are **identical**
  - Complete the Karnaugh map for S and Cout in terms of $\bar{A}, \bar{B}, \bar{C}$.
  - Inverting the inputs inverts the outputs
  - Exploit this in the ripple carry adder to avoid inversion (bubble-propagation)

Optimizing the Full-adder

c \ ab	00	01	11	10
0	0	1	0	1
1	1	0	1	0

S

c \ ab	00	01	11	10
0	0	0	1	1
1	0	0	1	1

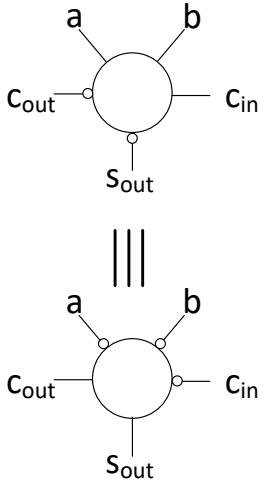
C

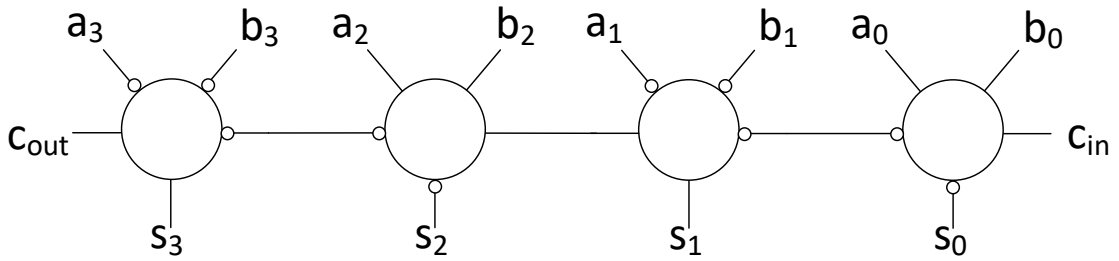
$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0	1	0	1	0
1	0	1	0	1

S

$\bar{c}$ \ $\bar{a}\bar{b}$	00	01	11	10
0	1	1	0	0
1	1	1	0	0

C



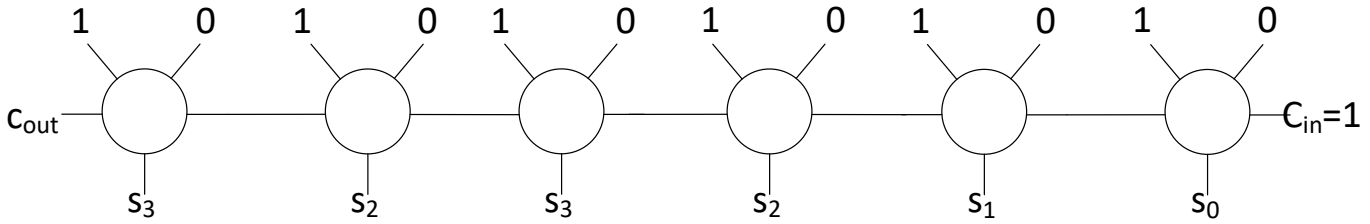


Aviod N/2 inversions!

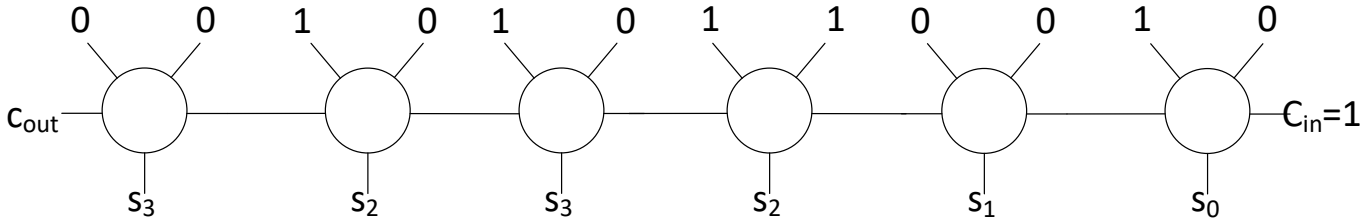
- Notice that the pulldown and pullup of Sum and Carry are **identical**
  - Complete the Karnaugh map for S and Cout in terms of $\bar{A}, \bar{B}, \bar{C}$.
  - Inverting the inputs inverts the outputs
  - Exploit this in the ripple carry adder to avoid inversion (bubble-propagation)

Faster Addition Algorithms

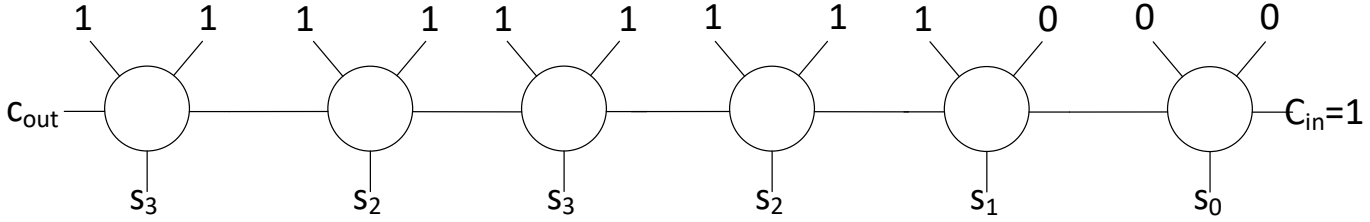
A



B

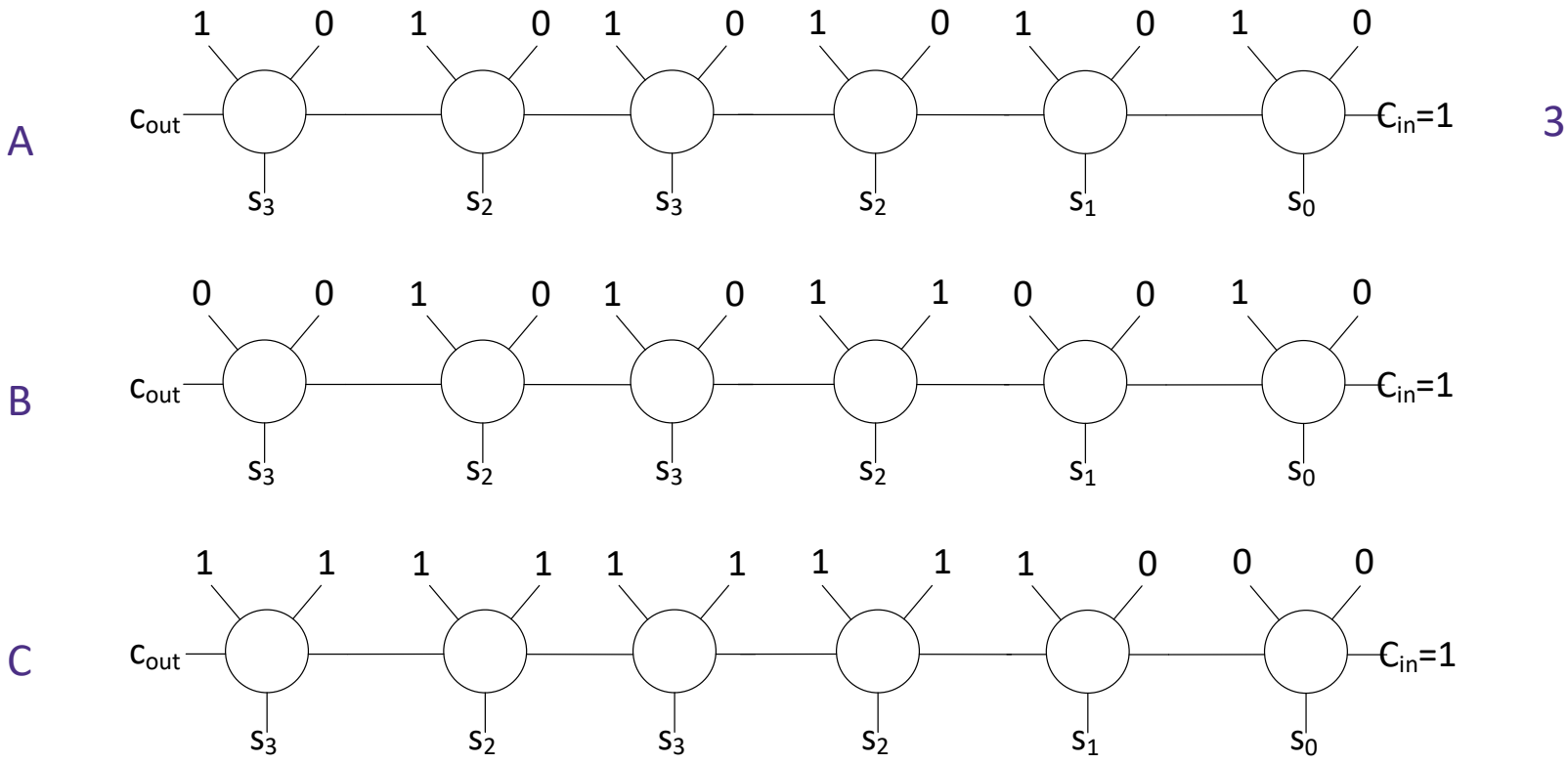


C



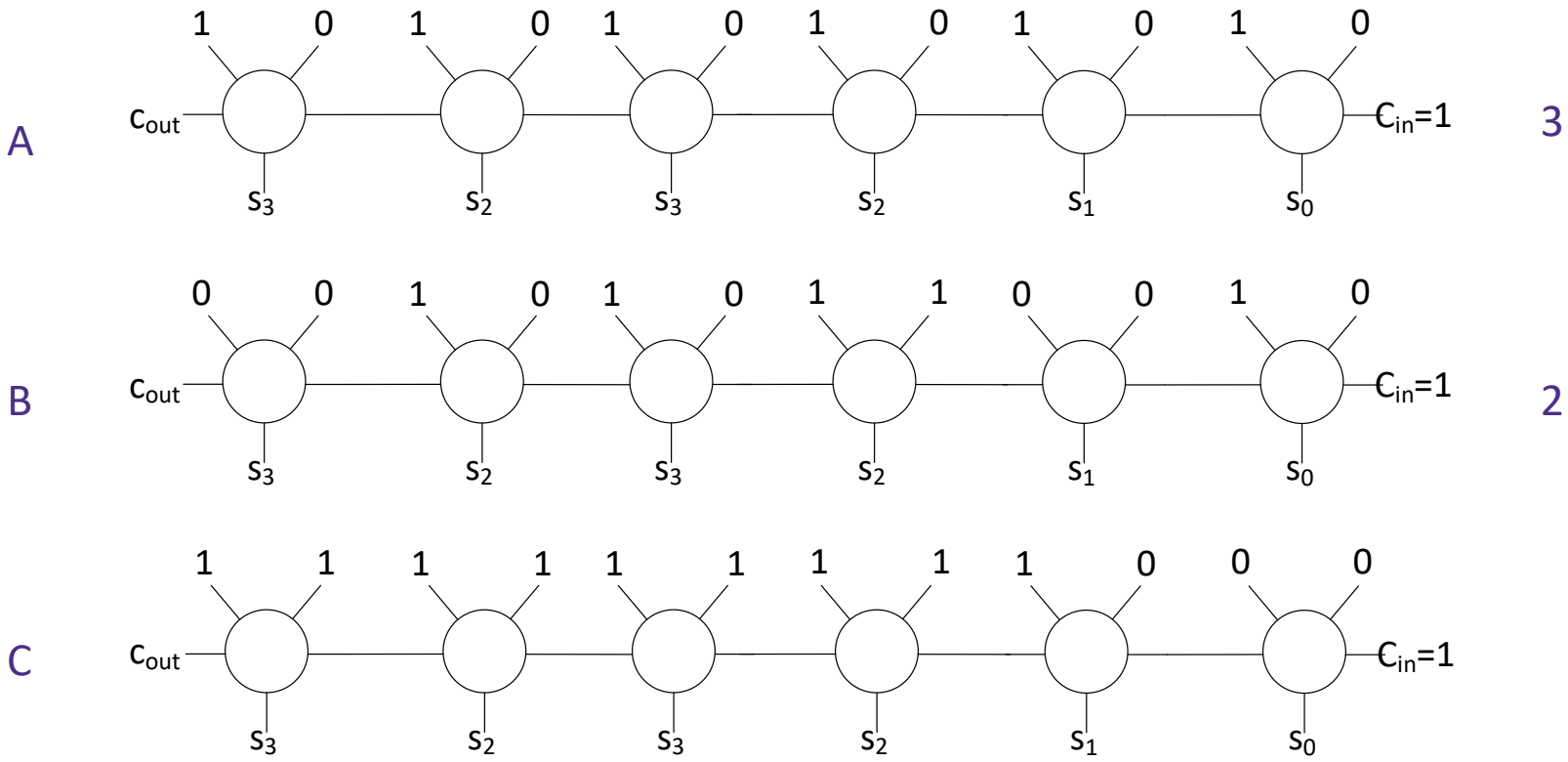
- Full-adder cells only reduce delay of each bit-wise full addition
  - Carry generation/propagation is still occurring serially
- Exercise: Order the above computations. Fastest first. If any delays are equal, mark them.
- Observation: Carry propagation essentially determines latency

Faster Addition Algorithms



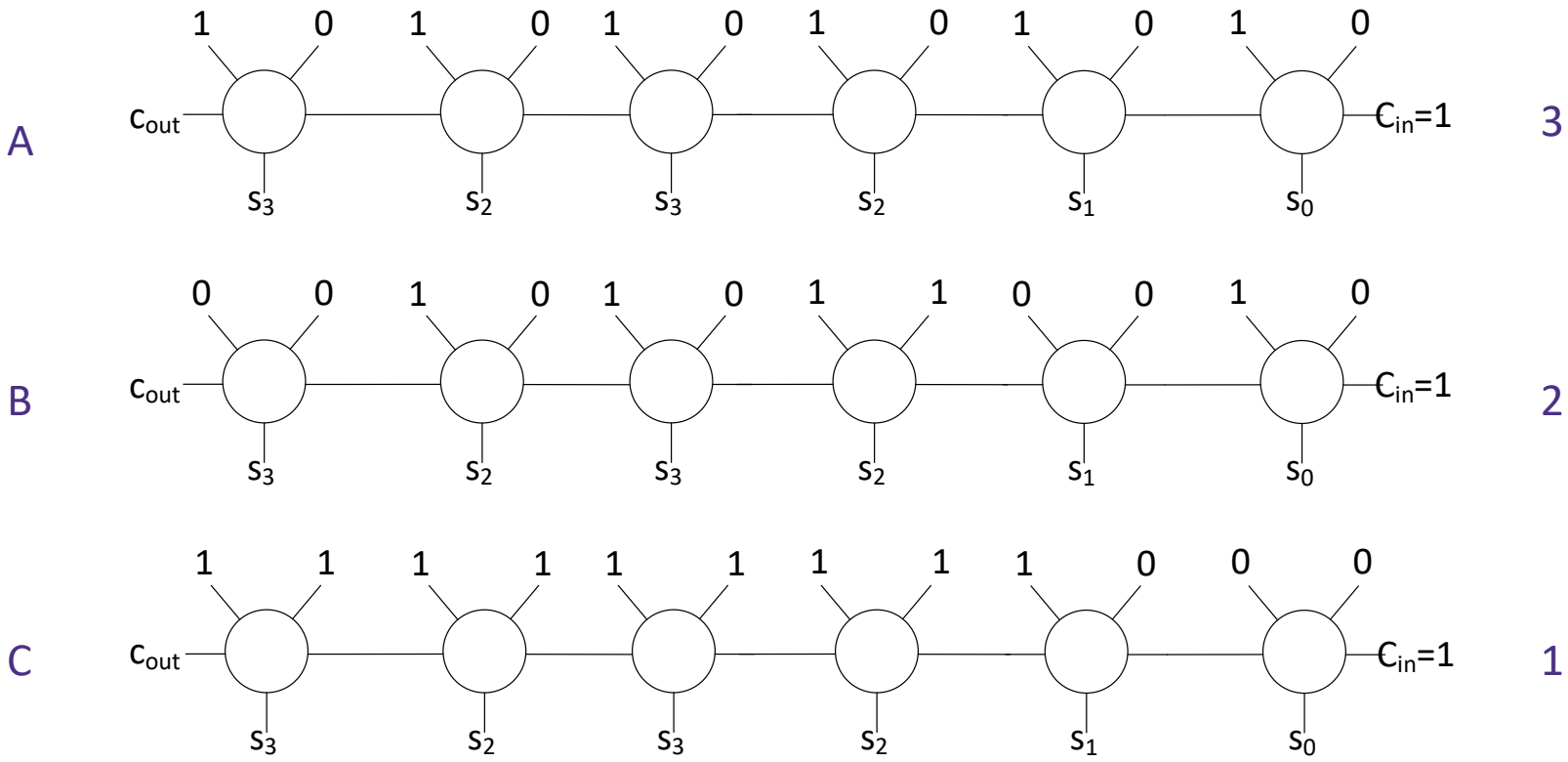
- Full-adder cells only reduce delay of each bit-wise full addition
  - Carry generation/propagation is still occurring serially
- Exercise: Order the above computations. Fastest first. If any delays are equal, mark them.
- Observation: Carry propagation essentially determines latency

Faster Addition Algorithms



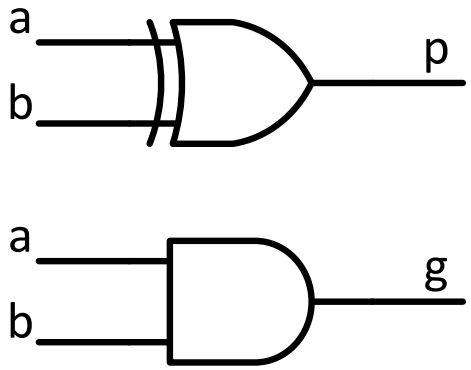
- Full-adder cells only reduce delay of each bit-wise full addition
  - Carry generation/propagation is still occurring serially
- Exercise: Order the above computations. Fastest first. If any delays are equal, mark them.
- Observation: Carry propagation essentially determines latency

Faster Addition Algorithms



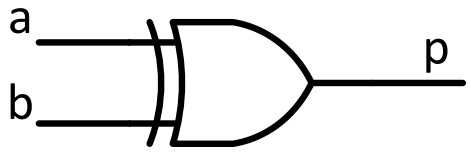
- Full-adder cells only reduce delay of each bit-wise full addition
  - Carry generation/propagation is still occurring serially
- Exercise: Order the above computations. Fastest first. If any delays are equal, mark them.
- Observation: Carry propagation essentially determines latency

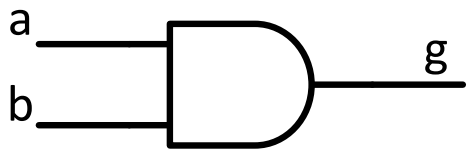
Understanding Carry Behaviour



- GPK formulation : Generate, Propagate, Kill
- At each bit, given inputs A, B
  - Carry-in is generated (regardless of C_{in}) if A&B=1
  - Carry-in is killed (regardless of C_{in}) if A|B=0
  - Carry-in is **propagated*** if A^B=1
- Each bit-stage is said to either Generate, Propagate or Kill the carry
- *Group* generate, *propagate* signals also defined

Understanding Carry Behaviour

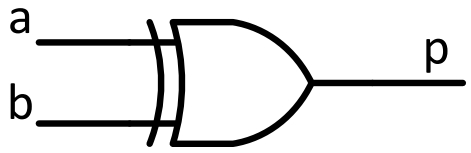


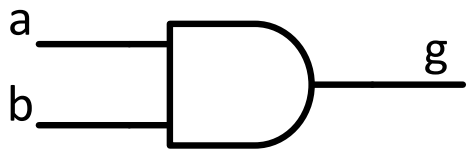


$$k = \overline{g + p}$$

- GPK formulation : Generate, Propagate, Kill
- At each bit, given inputs A, B
  - Carry-in is generated (regardless of C_{in}) if A&B=1
  - Carry-in is killed (regardless of C_{in}) if A|B=0
  - Carry-in is **propagated*** if A^B=1
- Each bit-stage is said to either Generate, Propagate or Kill the carry
- *Group* generate, *propagate* signals also defined

Understanding Carry Behaviour



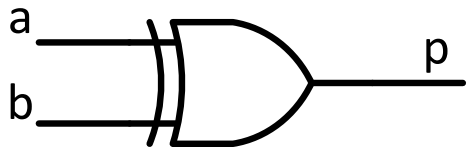


$$G_{i,i} = a \cdot b$$

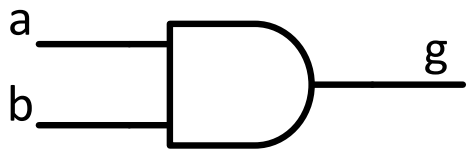
$$k = \overline{g + p}$$

- GPK formulation : Generate, Propagate, Kill
- At each bit, given inputs A, B
  - Carry-in is generated (regardless of C_{in}) if A&B=1
  - Carry-in is killed (regardless of C_{in}) if A|B=0
  - Carry-in is **propagated*** if A^B=1
- Each bit-stage is said to either Generate, Propagate or Kill the carry
- *Group* generate, *propagate* signals also defined

Understanding Carry Behaviour



$$P_{i,i} = a \oplus b$$

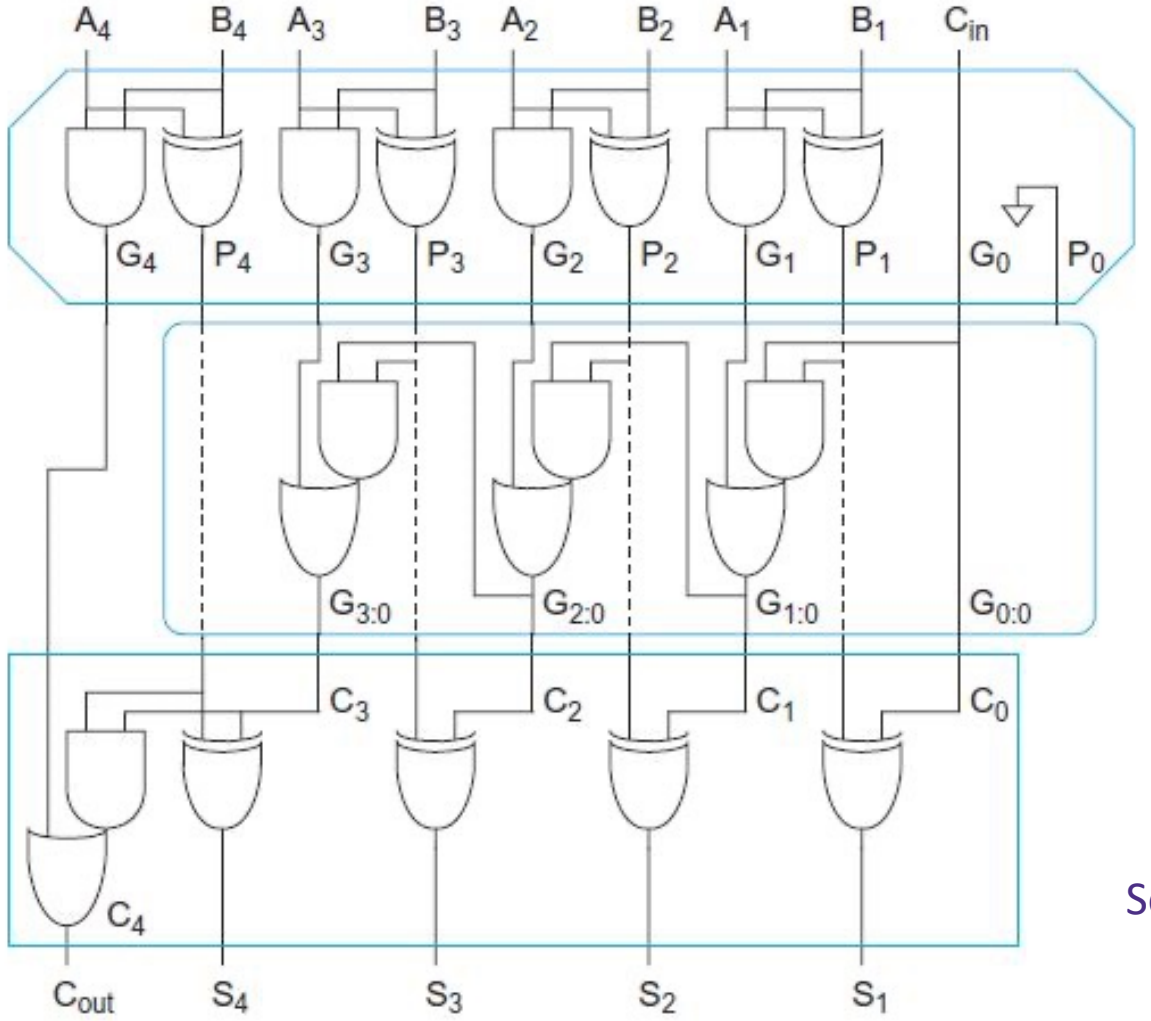


$$G_{i,i} = a \cdot b$$

$$k = \overline{g + p}$$

- GPK formulation : Generate, Propagate, Kill
- At each bit, given inputs A, B
  - Carry-in is generated (regardless of C_{in}) if A&B=1
  - Carry-in is killed (regardless of C_{in}) if A|B=0
  - Carry-in is **propagated*** if A^B=1
- Each bit-stage is said to either Generate, Propagate or Kill the carry
- *Group* generate, *propagate* signals also defined

Using Generate and Propagate



Source: W&H

- 4-bit ripple-carry adder using PG logic
  - Carry ripple occurs now using AOI gate instead of full-adder cell

Group GPK

- Consider equations for C_{out} in terms of g_i, p_i and k_i
  - $C_{out,0} = g_0 + p_0 C_{in}$
  - $C_{out,1} = g_1 + g_0 p_1 + p_0 p_1 C_{in}$
  - $C_{out,2} = g_2 + g_1 p_2 + g_0 p_1 p_2 + p_0 p_1 p_2 C_{in}$
  - $C_{out,2} = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + p_0 p_1 p_2 p_3 C_{in}$

- Define group G,P functions as follows
  - $P_{j,I} = \prod_i^j p_k$
  - $G_{j,i} = g_j + g_{j-1}\, p_j + g_{j-2}\, p_{j-1}\, p_j + \ldots + g_i \prod_i^j p_k$
  - p_i can be computed using OR function instead of XOR

- Some useful results….
  - $C_i = G_{i,0}$ (By definition)
  - **$G_{j,i} = G_{j,k} + G_{k-1,i} P_{j,k}$ (Merging property. Enables parallelism)**
  - $G_{j,k} + G_{k-1,i} P_{j,k} = G_{j,k} + G_{k,i} P_{j,k}$ (Overlaps OK)

Group GPK

- Consider equations for C_{out} in terms of g_i, p_i and k_i
  - $C_{out,0} = g_0 + p_0 C_{in}$
  - $C_{out,1} = g_1 + g_0 p_1 + p_0 p_1 C_{in}$
  - $C_{out,2} = g_2 + g_1 p_2 + g_0 p_1 p_2 + p_0 p_1 p_2 C_{in}$
  - $C_{out,2} = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + p_0 p_1 p_2 p_3 C_{in}$

- Define group G,P functions as follows
  - $P_{j,I} = \prod_i^j p_k$
  - $G_{j,i} = g_j + g_{j-1}\, p_j + g_{j-2}\, p_{j-1}\, p_j + \ldots + g_i \prod_i^j p_k$
  - p_i can be computed using OR function instead of XOR

- Some useful results….
  - $C_i = G_{i,0}$ (By definition)
  - **$G_{j,i} = G_{j,k} + G_{k-1,i} P_{j,k}$ (Merging property. Enables parallelism)** Exercise
  - $G_{j,k} + G_{k-1,i} P_{j,k} = G_{j,k} + G_{k,i} P_{j,k}$ (Overlaps OK)

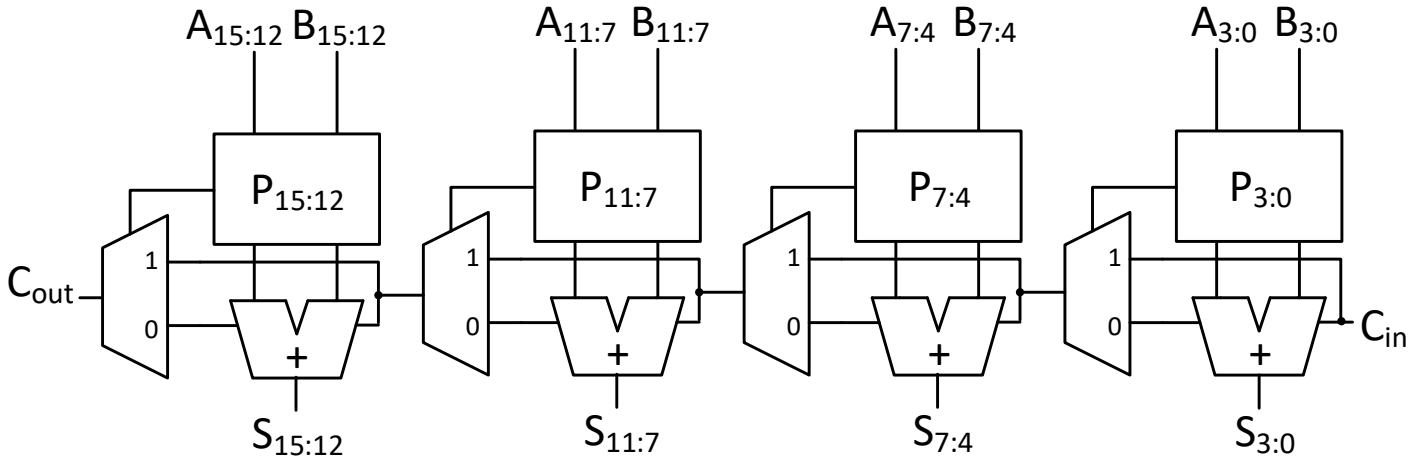
Group GPK

- Consider equations for C_{out} in terms of g_i, p_i and k_i
  - $C_{out,0} = g_0 + p_0 C_{in}$
  - $C_{out,1} = g_1 + g_0 p_1 + p_0 p_1 C_{in}$
  - $C_{out,2} = g_2 + g_1 p_2 + g_0 p_1 p_2 + p_0 p_1 p_2 C_{in}$
  - $C_{out,2} = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + p_0 p_1 p_2 p_3 C_{in}$

- Define group G,P functions as follows
  - $P_{j,I} = \prod_i^j p_k$
  - $G_{j,i} = g_j + g_{j-1}\, p_j + g_{j-2}\, p_{j-1}\, p_j + \ldots + g_i \prod_i^j p_k$
  - p_i can be computed using OR function instead of XOR

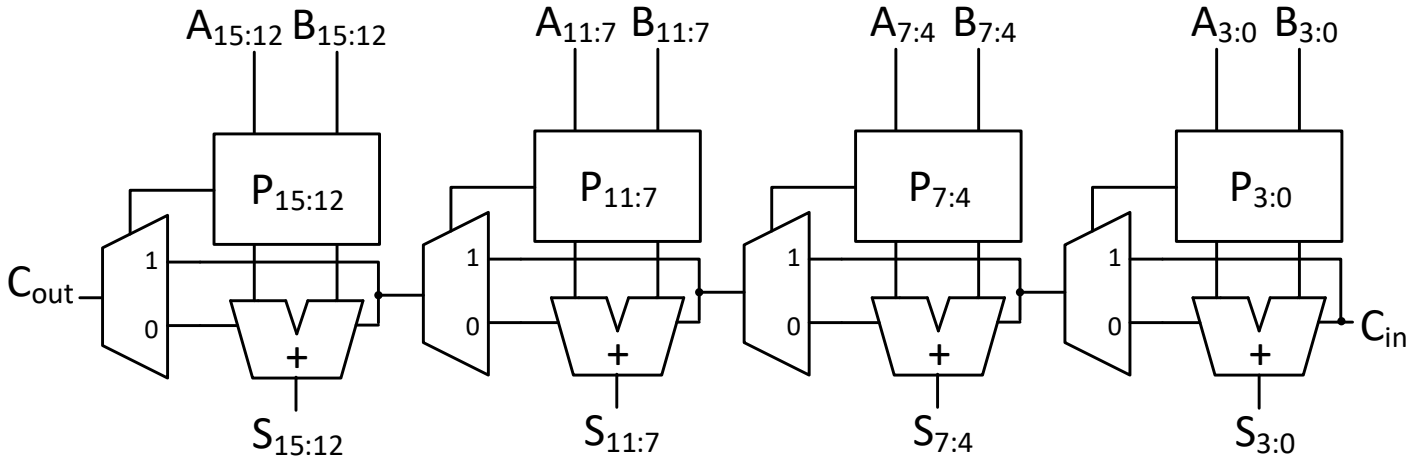
- Some useful results….
  - $C_i = G_{i,0}$ (By definition)
  - **$G_{j,i} = G_{j,k} + G_{k-1,i} P_{j,k}$ (Merging property. Enables parallelism)** Exercise
  - $G_{j,k} + G_{k-1,i} P_{j,k} = G_{j,k} + G_{k,i} P_{j,k}$ (Overlaps OK) Exercise

Carry-Skip Adder



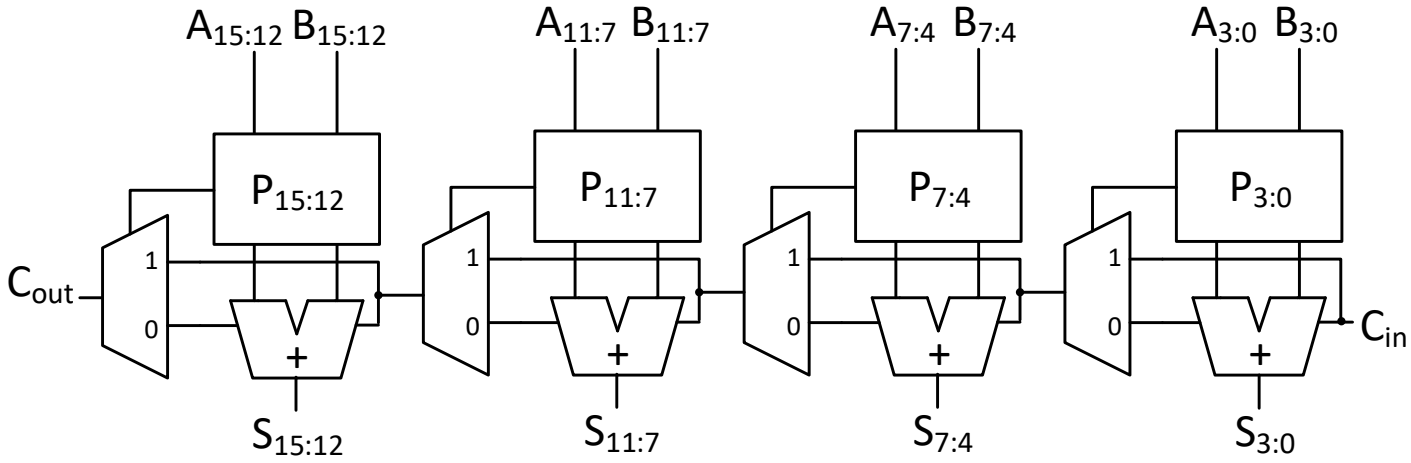
- Recall observation: Delay of a ripple-carry is limited by longest sequence of Propagate stages
- Solution: Detect sequence(group) of propagate stages **in parallel** and "forward" the carry through them
  - Addition computation occurs serially within each group
  - Substantially shorten the sequence of carry propagation stages in the critical path

Carry-Skip Adder



- Recall observation: Delay of a ripple-carry is limited by longest sequence of Propagate stages
- Solution: Detect sequence(group) of propagate stages **in parallel** and "forward" the carry through them
  - Addition computation occurs serially within each group
  - Substantially shorten the sequence of carry propagation stages in the critical path

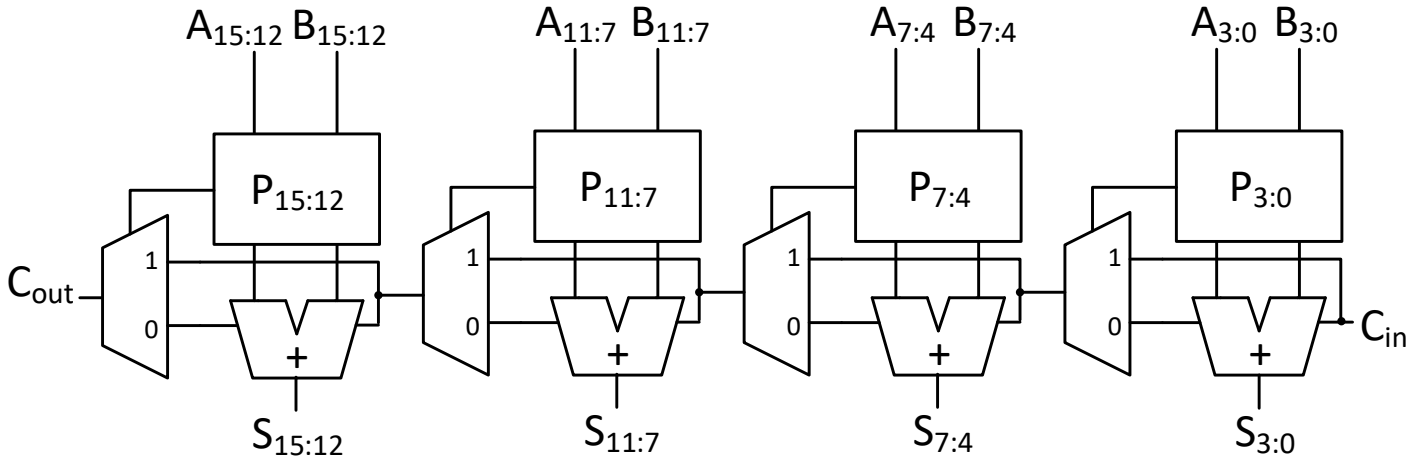
Carry-Skip Adder



Can I ever use an OR function for P?

- Recall observation: Delay of a ripple-carry is limited by longest sequence of Propagate stages
- Solution: Detect sequence(group) of propagate stages **in parallel** and "forward" the carry through them
  - Addition computation occurs serially within each group
  - Substantially shorten the sequence of carry propagation stages in the critical path

Carry-Skip Adder

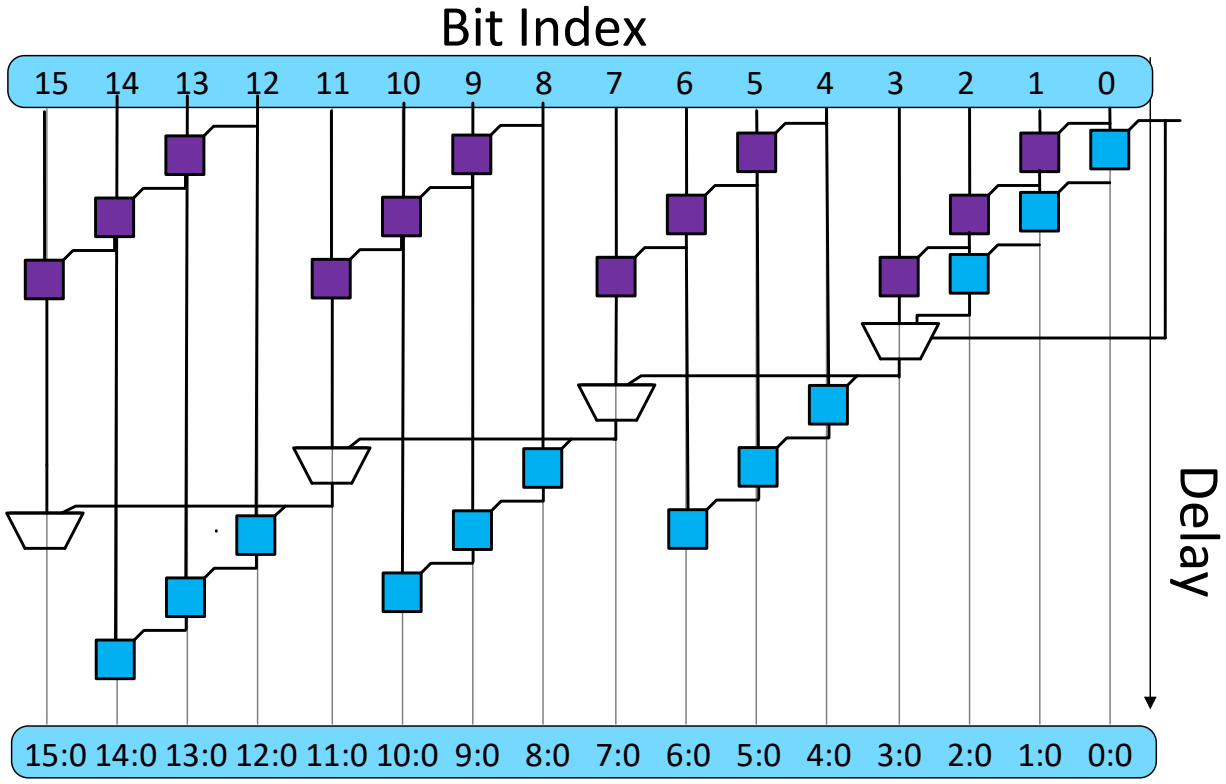


Do I really need an XOR?
Will an OR gate not do?

Can I ever use an OR function for P?

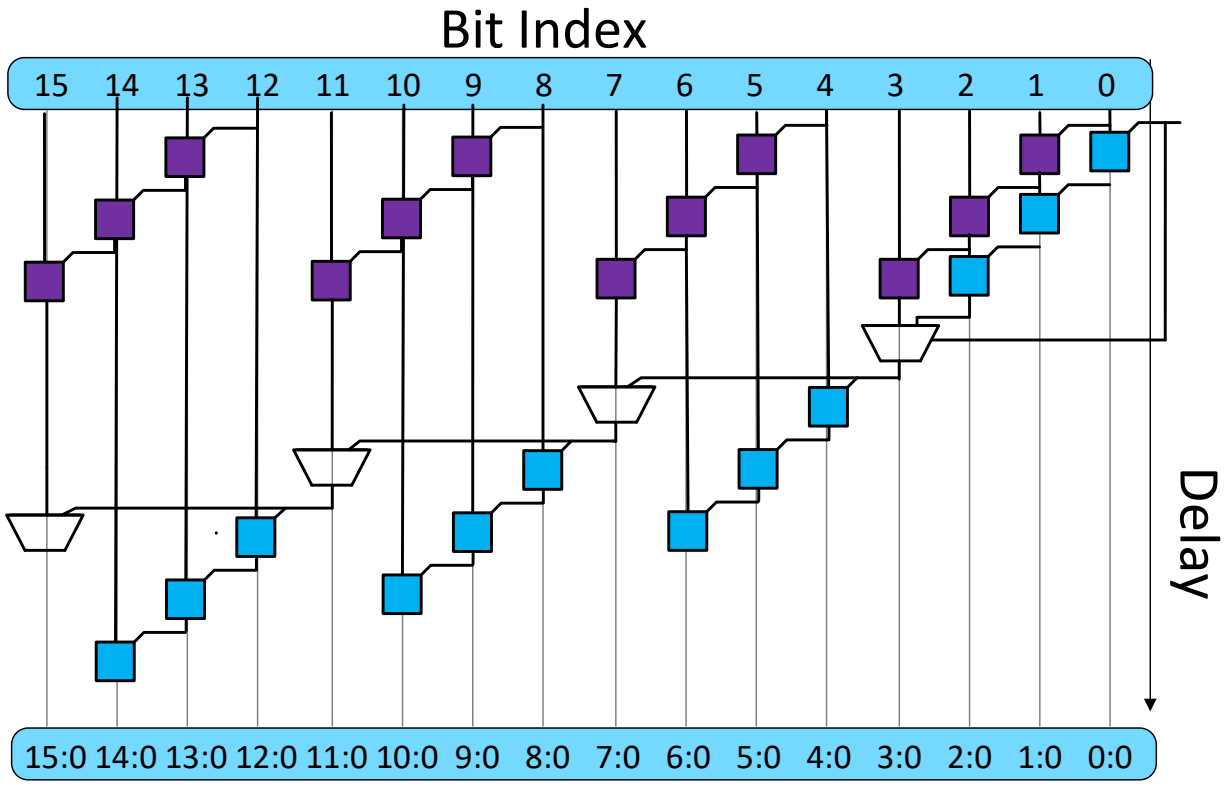
- Recall observation: Delay of a ripple-carry is limited by longest sequence of Propagate stages
- Solution: Detect sequence(group) of propagate stages **in parallel** and "forward" the carry through them
  - Addition computation occurs serially within each group
  - Substantially shorten the sequence of carry propagation stages in the critical path

Carry-Skip Delay Diagram



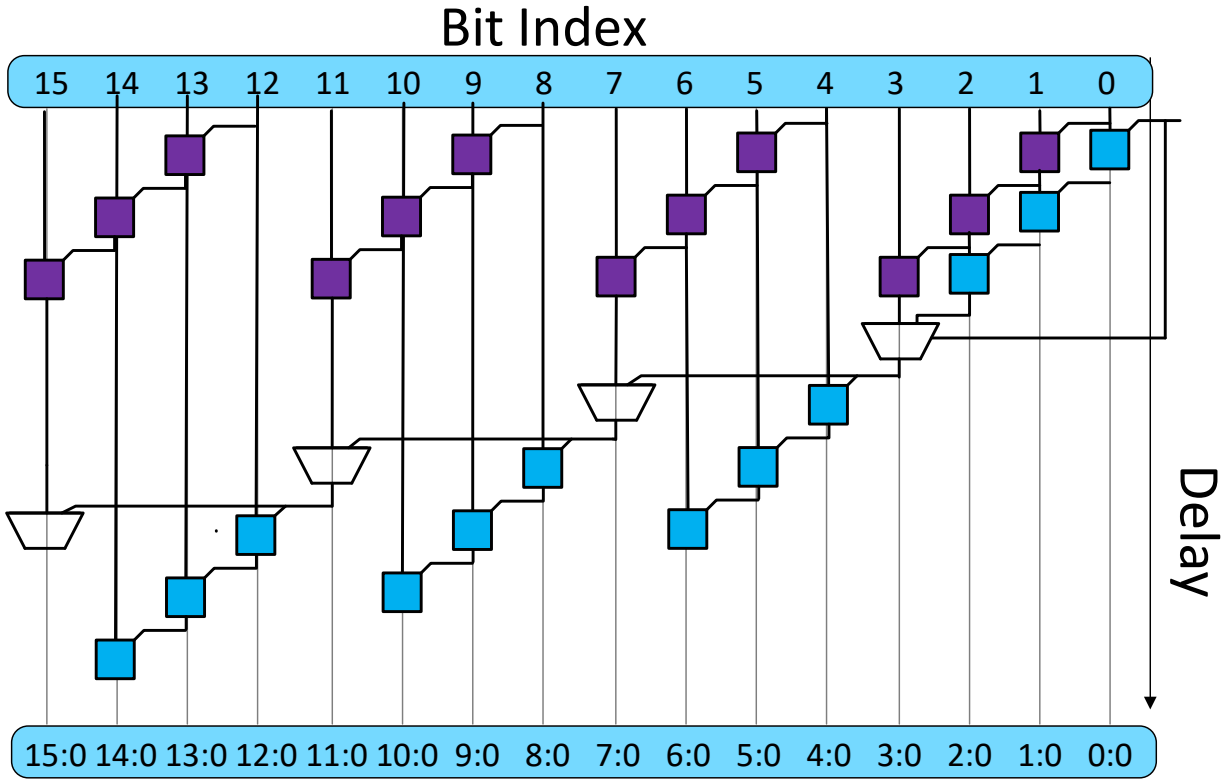
- $T_{skip\text{-}delay} = t_{pg} + (n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + (n\text{-}1)t_{AOI} + t_{XOR}$
  $= t_{pg} + 2(n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + t_{XOR}$
- $t^*_{delay} = \Theta(\sqrt{N})$

Carry-Skip Delay Diagram



- $T_{skip\text{-}delay} = t_{pg} + (n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + (n\text{-}1)t_{AOI} + t_{XOR}$
  $= t_{pg} + 2(n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + t_{XOR}$
- $t^*_{delay} = \Theta(\sqrt{N})$

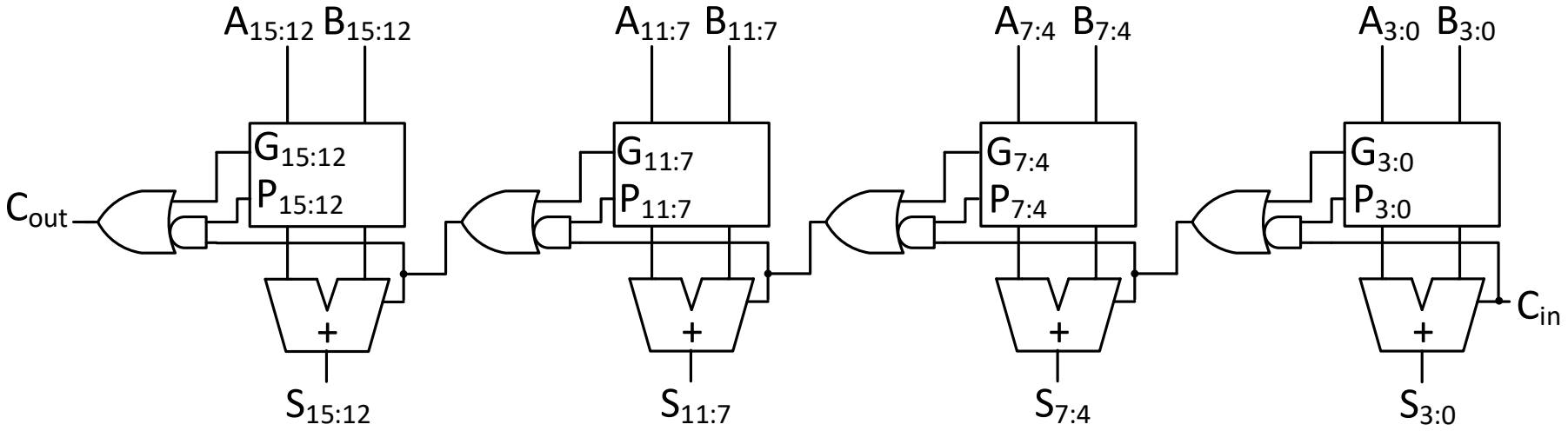
Carry-Skip Delay Diagram



- $T_{skip\text{-}delay} = t_{pg} + (n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + (n\text{-}1)t_{AOI} + t_{XOR}$
  $= t_{pg} + 2(n\text{-}1)t_{AOI} + (k\text{-}1)t_{mux} + t_{XOR}$
- $t^*_{delay} = \Theta(\sqrt{N})$

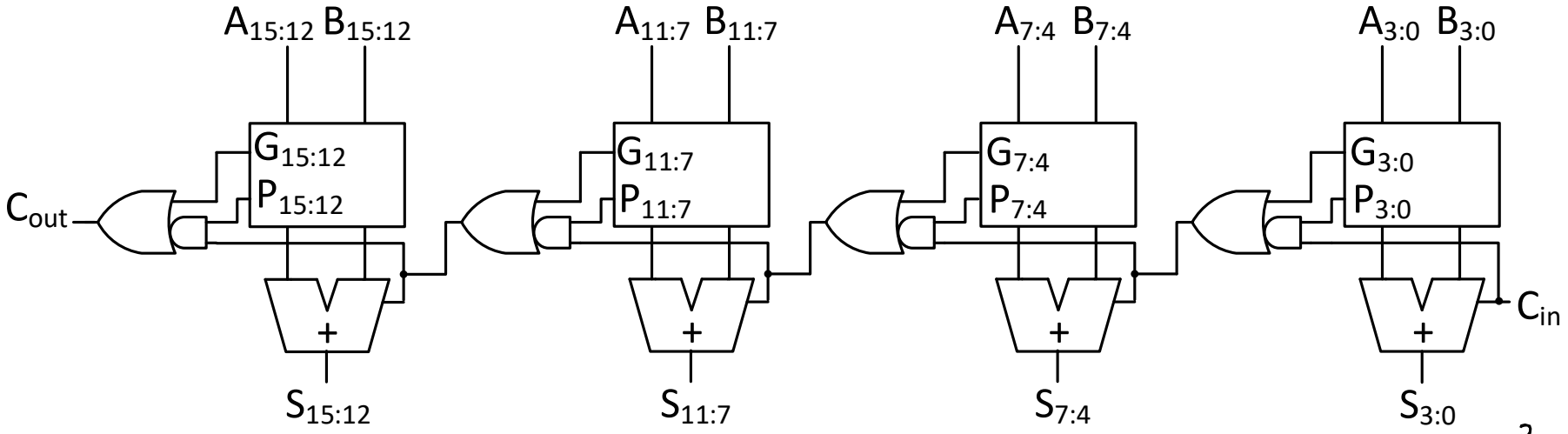
Do I really need a MUX?

Carry-Look Ahead Adder



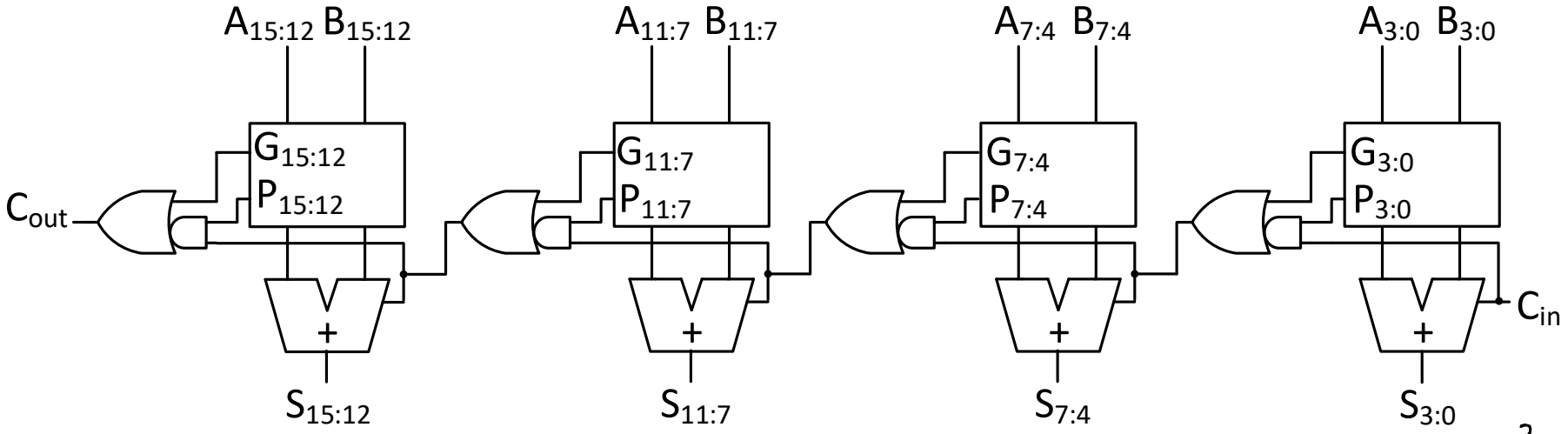
- Compute group G,P signals concurrently
- Merge G, P signals to produce C_i
- Possible to build efficient single-gate implementation of group G, P
  - Gate implementation of $G_{k,k-n}$ ➔ Radix-n implementations
  - Done using dynamic logic (Brief discussion of this at the end of the course)

Carry-Look Ahead Adder



- Compute group G,P signals concurrently
- Merge G, P signals to produce C_i
- Possible to build efficient single-gate implementation of group G, P
  - Gate implementation of $G_{k,k-n}$ ➔ Radix-n implementations
  - Done using dynamic logic (Brief discussion of this at the end of the course)

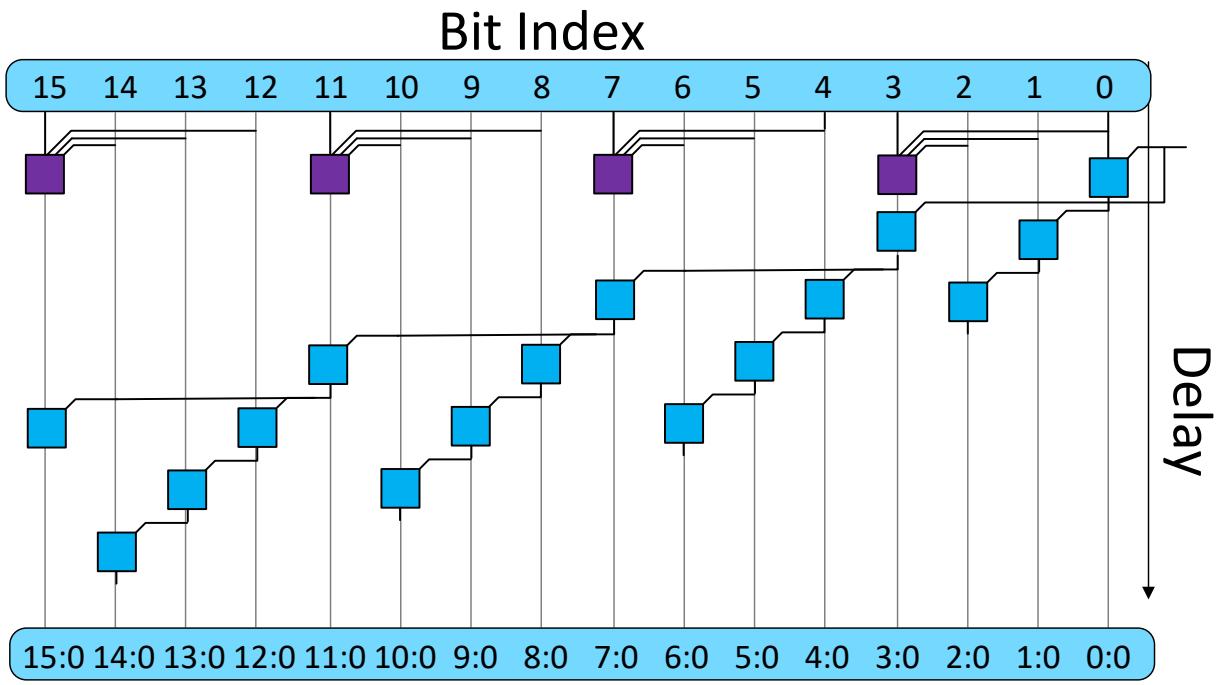
Carry-Look Ahead Adder



Wait, how come I get to use an AOI here and not in carry-skip

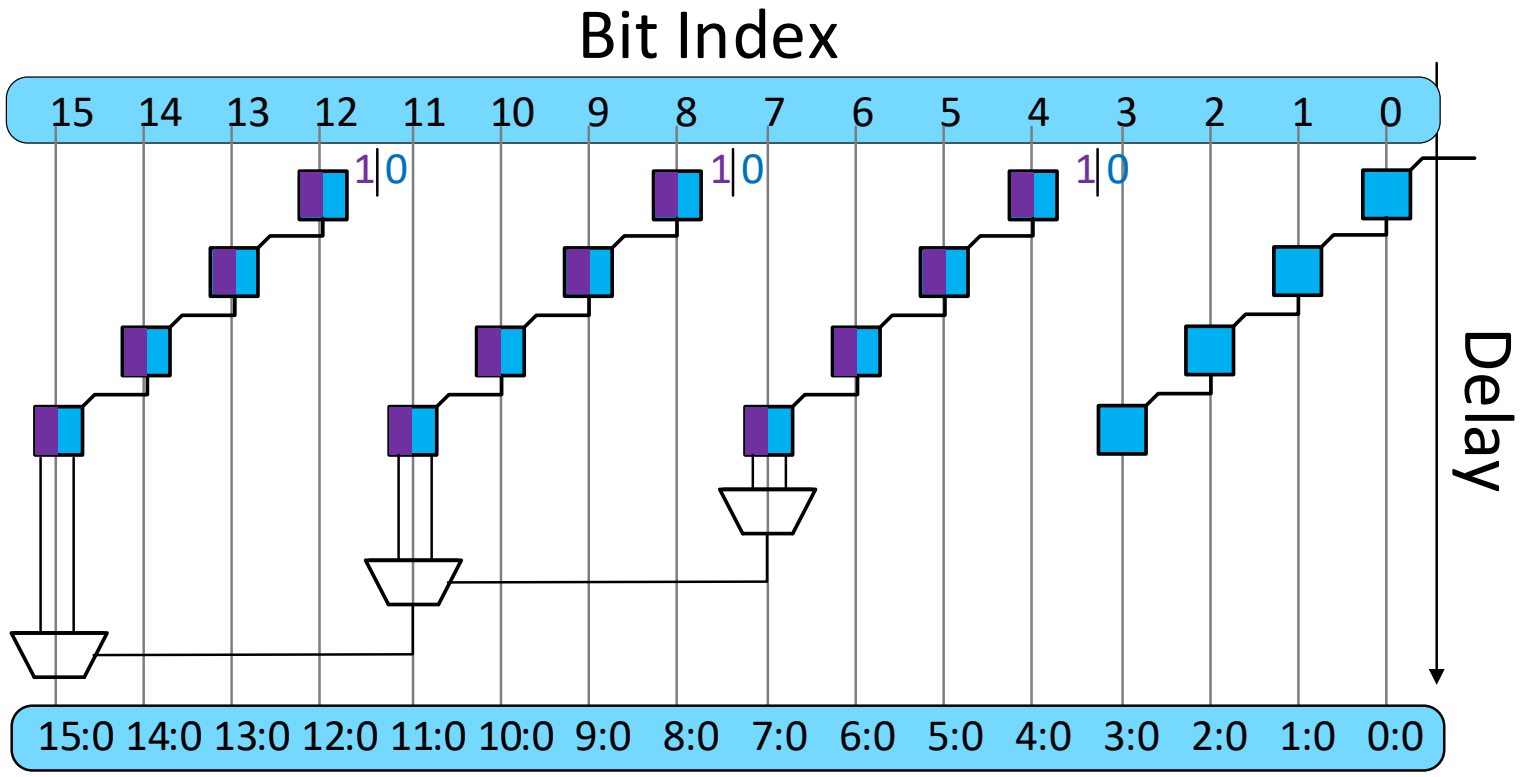
- Compute group G,P signals concurrently
- Merge G, P signals to produce C_i
- Possible to build efficient single-gate implementation of group G, P
  - Gate implementation of $G_{k,k\text{-}n}$ ➔ Radix-n implementations
  - Done using dynamic logic (Brief discussion of this at the end of the course)

CLA Delay Diagram



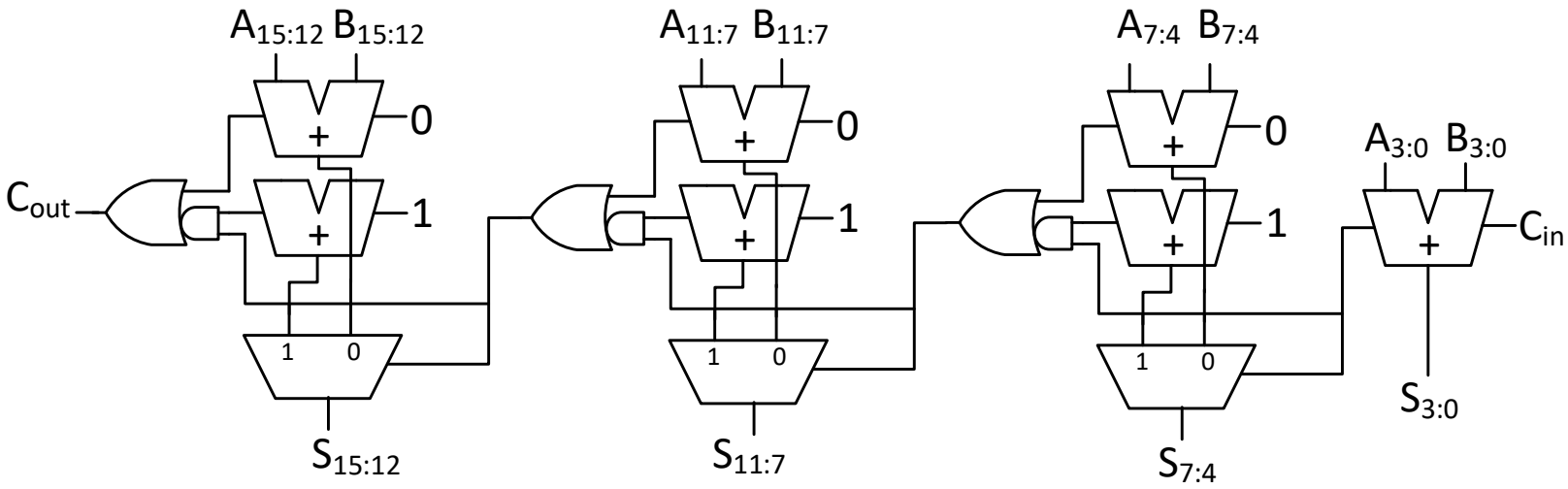
- $T_{cla\text{-}delay} = t_{pg} + t_{group\text{-}pg} + (k\text{-}1)t_{AOI} + (n\text{-}1)t_{AOI} + t_{XOR}$
- $t^*_{delay} = \Theta(\sqrt{N})$
- Note: Essentially identical to carry-skip

Speculating the Incoming Carry



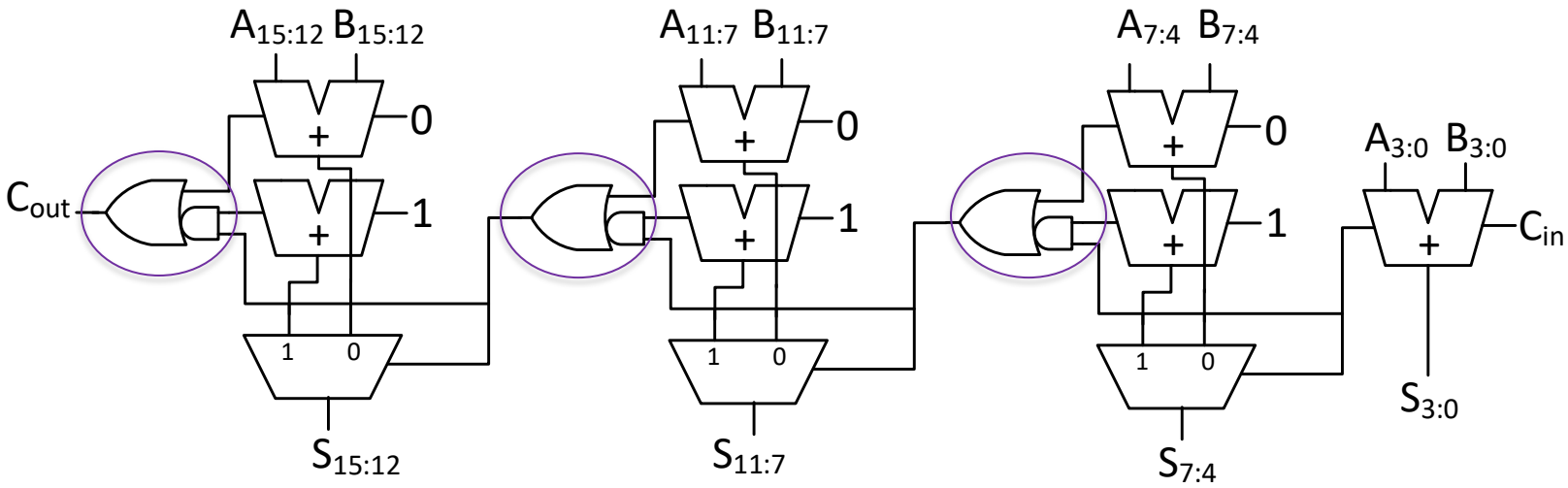
- Note: Speculate on the carry. Make a late selection
- $T_{csa\text{-}delay} = t_{pg} + (k\text{-}1)t_{mux} + nt_{AOI}$
- $t^*_{delay} = \Theta(\sqrt{N})$

Carry-Select Adder



- Basic Idea: Speculate on the value of the carry per segment.
  - Compute C_{out} in each case
  - Optionally compute the sum output for each segment
- When the actual carry arrives
  - Select the C_{out} corresponding to the real C_{in}
  - Select the output sums for the segment based on C_{in}

Carry-Select Adder

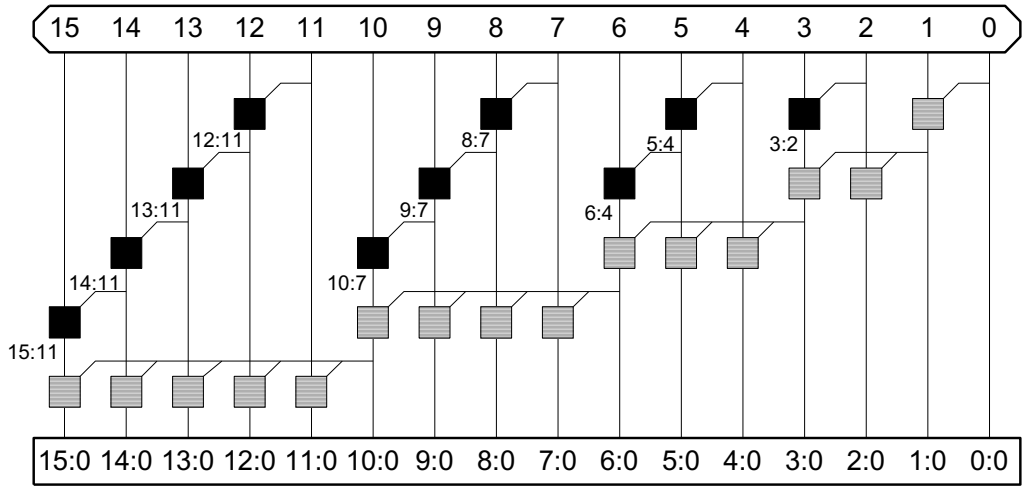


- Basic Idea: Speculate on the value of the carry per segment.
  - Compute C_{out} in each case
  - Optionally compute the sum output for each segment
- When the actual carry arrives
  - Select the C_{out} corresponding to the real C_{in}
  - Select the output sums for the segment based on C_{in}

Exploiting Non-uniform Group Size

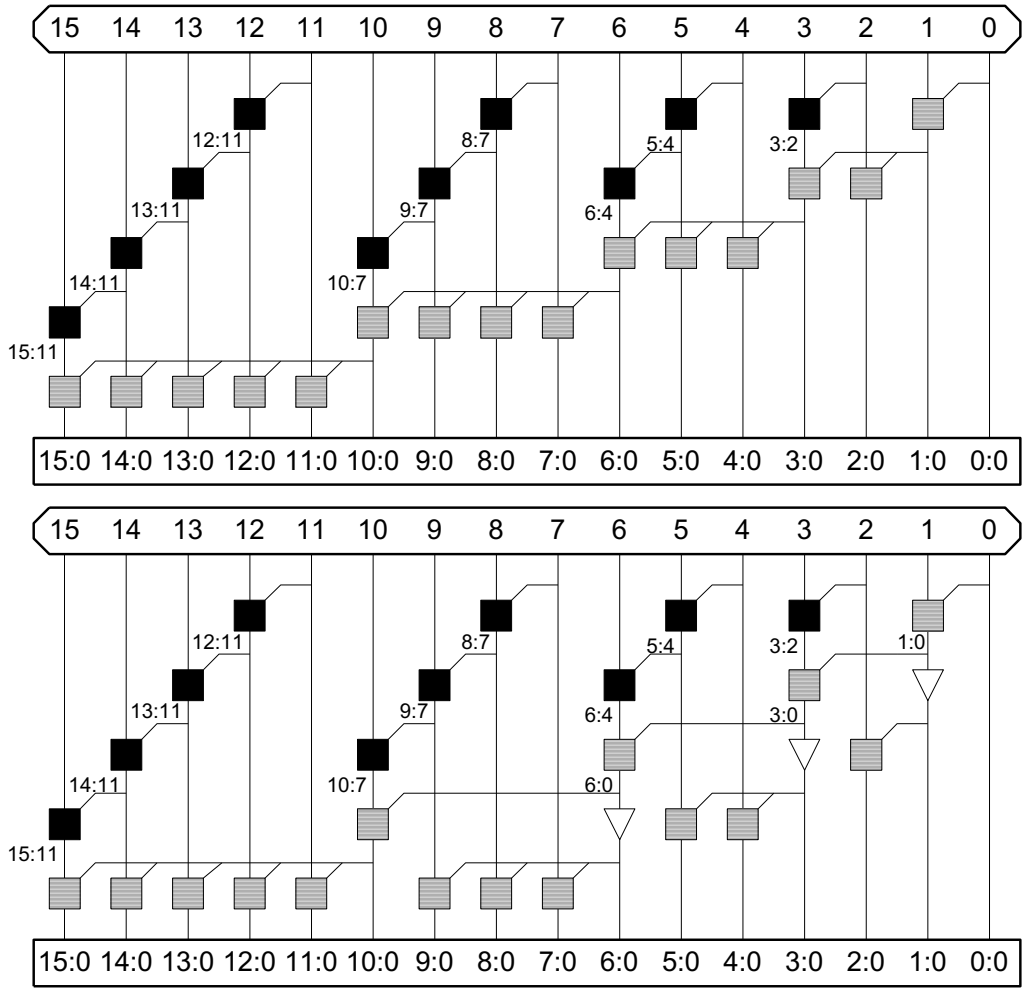
- Exercise. Consider an alternative partition of each group of the adder bits to build a faster carry-select adder

Exploiting Non-uniform Group Size



- Exercise. Consider an alternative partition of each group of the adder bits to build a faster carry-select adder

Exploiting Non-uniform Group Size

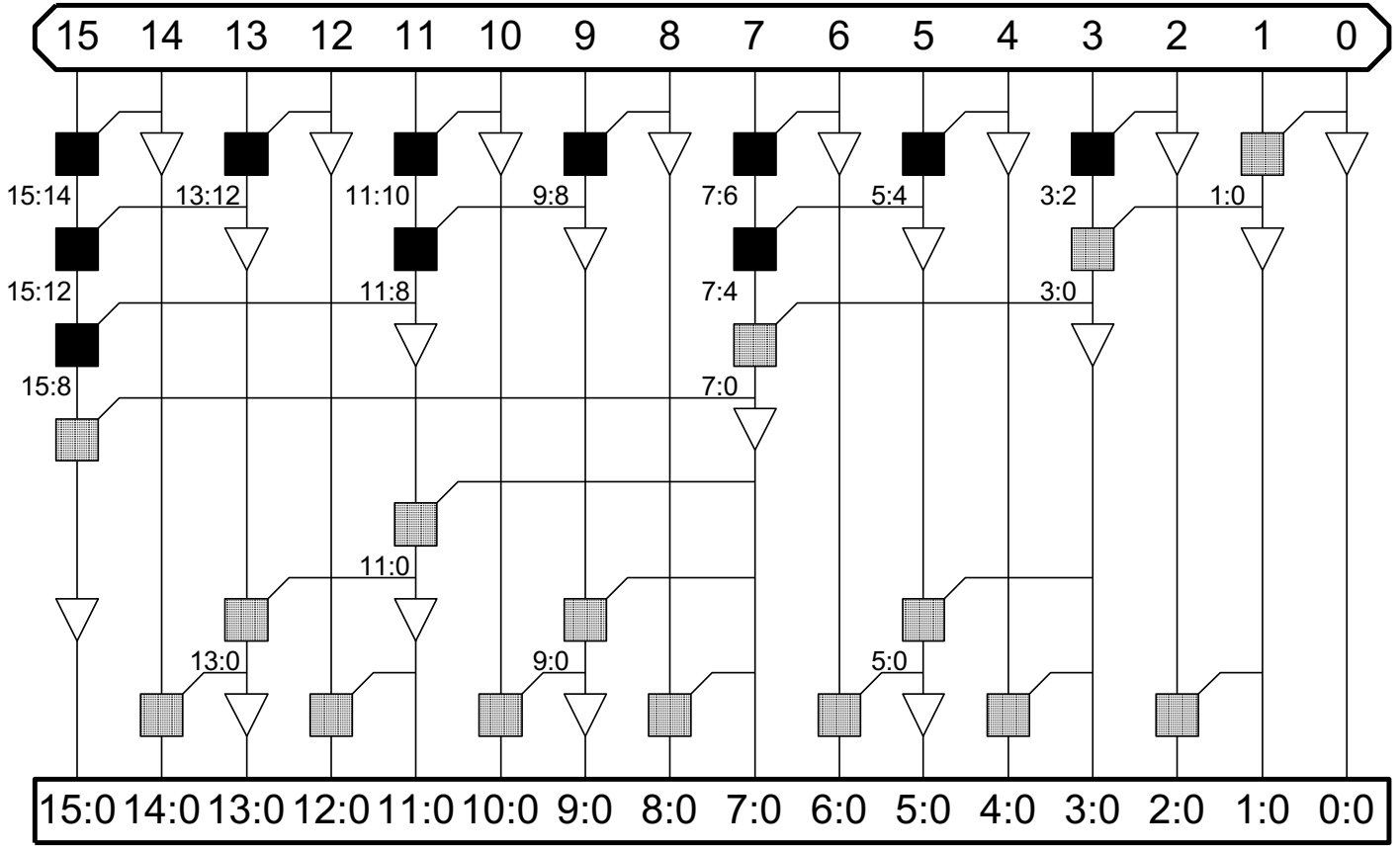


- Exercise. Consider an alternative partition of each group of the adder bits to build a faster carry-select adder

Tree Adders

- Carry-lookahead. Fast, but still merges groups linearly
- Key enabling observations:
  - Carry-out can be written in terms of group propagates and generates
  - Group propagates and generates are associative (can be computed in parallel)
- Large variety of tree adders
  - Trade-off Latency, Fanout and Routing Tracks
  - We consider two tree adder topologies
    - Brent-Kung
    - Kogge-Stone
- Weste-Harris' Textbook has an excellent treatment of classifying several popular tree adders and the trade-offs they make (Examined in closer detail in EE477. **Optional for EE476**)
- High-quality digital systems all use tree adder variants of some kind
  - Higher radix
  - Dynamic logic or Mixed logic (Static+Dynamic) in high perf. systems
  - Combination of Tree adders with sum/carry select

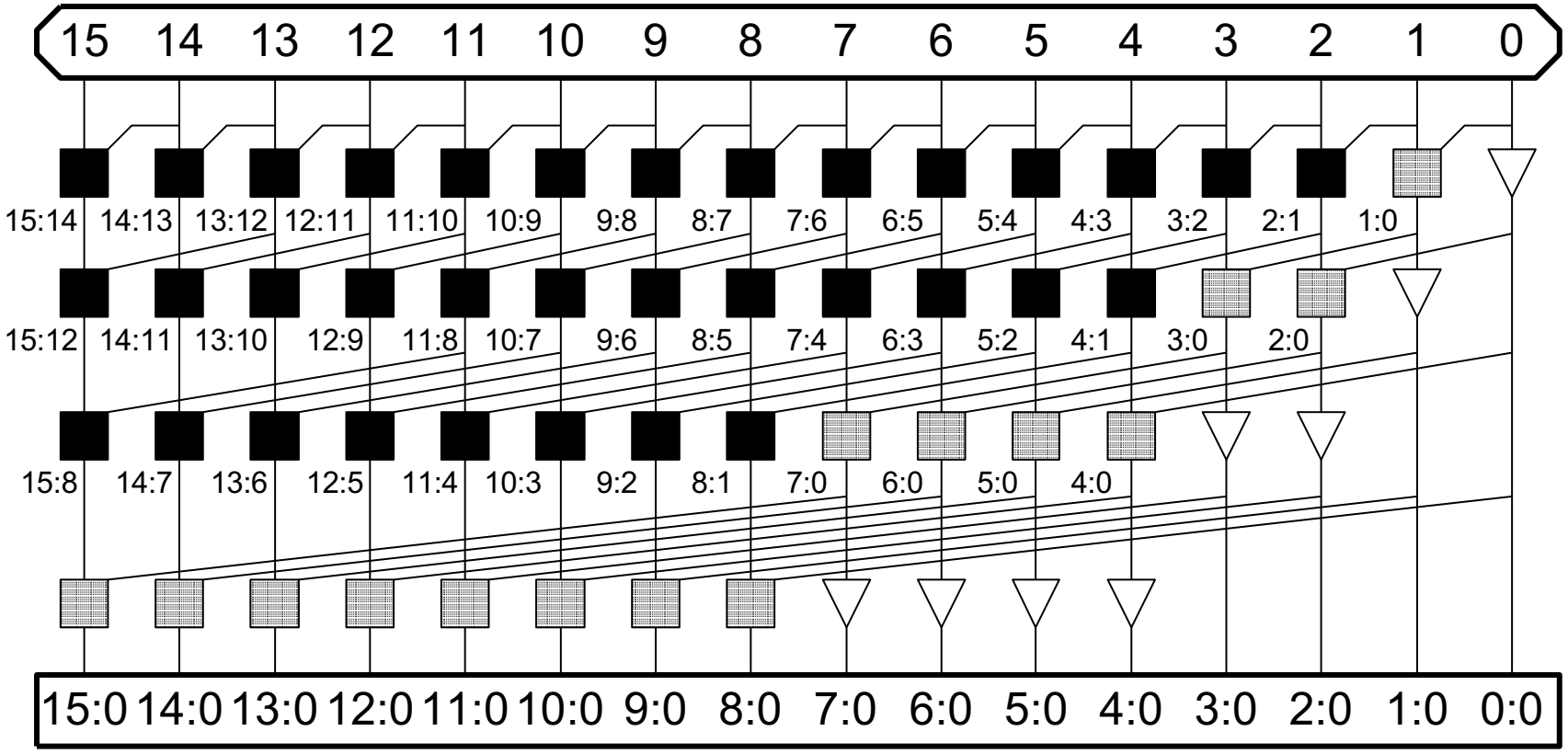
Brent-Kung Adder



Source: W&H

- "Baseline" Tree adder
  - Need to do 2 tree traversals to get C14 (crit. path)
  - Fanout at each gate is managed

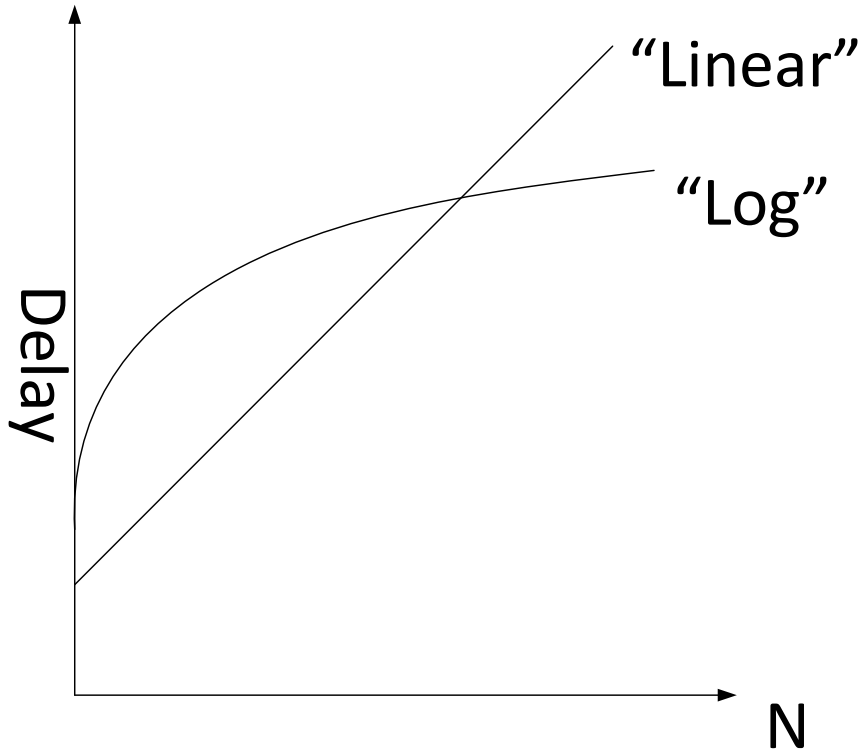
Kogge-Stone Adder



Source: W&H

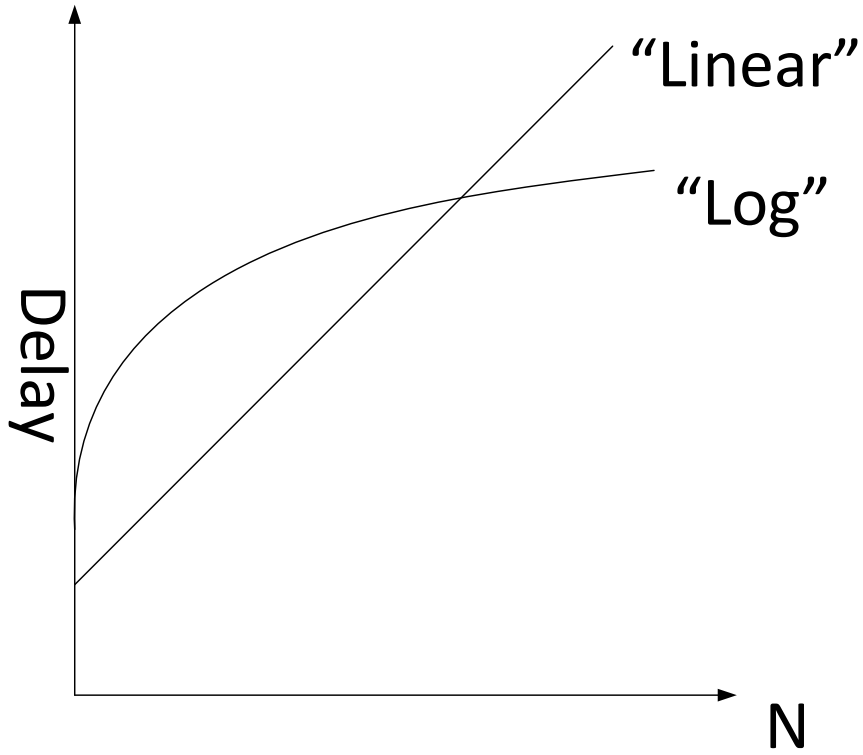
- Lowest number of delay levels ($\log_2 n$)
  - Fanout managed well (compared to Sklansky)
  - Relatively more regular layout (but wire parasitics, track-allocation tricky)

Algorithmic Complexity and VLSI



- General rule of thumb: Alorithmic complexity is a good indicator of latency of VLSI computation
- However, VLSI implementations often fall in the “middle zone”
  - N is large enough that poor complexity results in slow implementations
  - N is not so large that overhead can be ignored…

Algorithmic Complexity and VLSI



Multiply two nxn matrices:

- Definition: $O(n^3)$
- Fastest known: $O(n^{2.37188})$

Duan, Ran; Wu, Hongxun; Zhou, Renfei (Oct. 2022)

- General rule of thumb: Alorithmic complexity is a good indicator of latency of VLSI computation
- However, VLSI implementations often fall in the "middle zone"
  - N is large enough that poor complexity results in slow implementations
  - N is not so large that overhead can be ignored…

Overflow

- Signed and unsigned arithmetic results can exceed bounds
  - 4 bit unsigned computation.
    - 12 + 12 = 8
  - 4-bit signed computation
    - 7+7=-2
    - -8 -5 = 3

- Nomenclature adopted in microprocessors: Carry flag to detect overflow in unsigned arithmetic, Overflow flag for signed arithmetic
  - Carry flag. Set when C_{out} of n-bit computation, $C_{out,n}$ is 1
  - Overflow flag. Set as $C_{out,n} \wedge C_{out,n-1}$ (Prove this using Eqn 1 or circle-plot)

Reading

- Required
  - W&H : 11.2-11.2.2.7
- Optional
  - W&H: 11.2.2.8 – 11.2.2.10