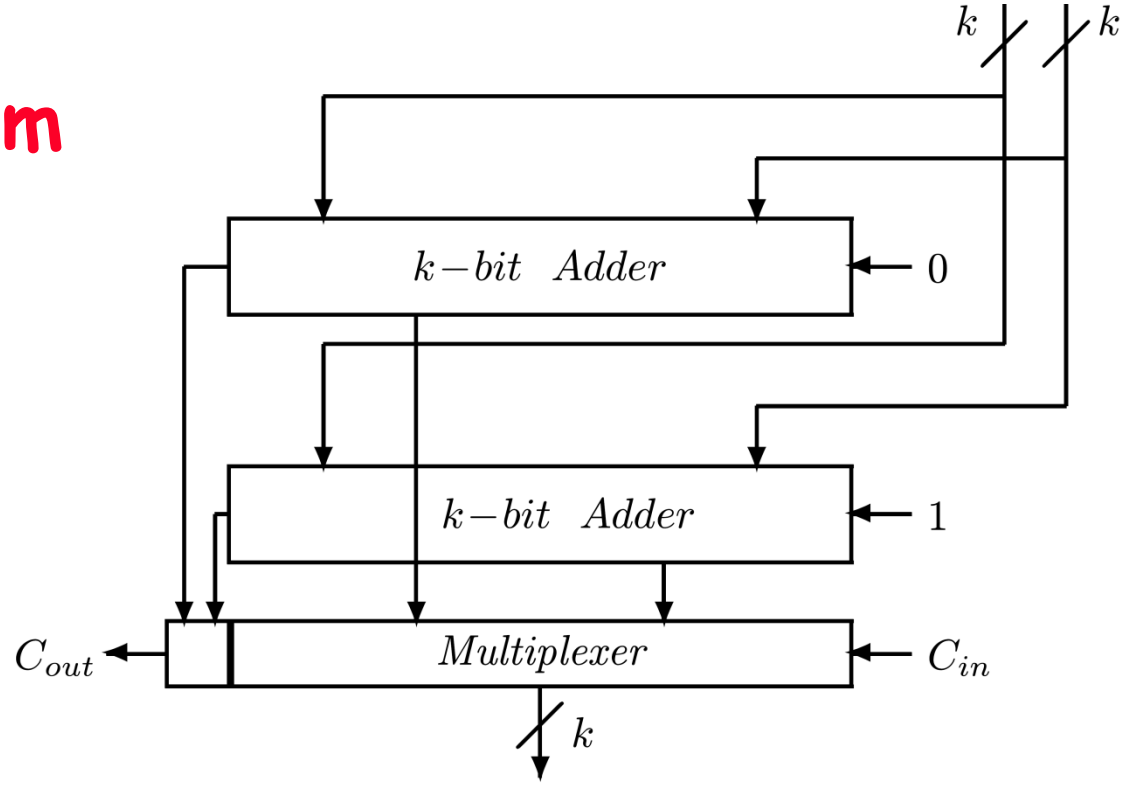
Digital Computer Arithmetic

Part 5
Fast Addition

Soo-Ik Chae

Spring 2009

Conditional Sum Adders



- Logarithmic speed-up of addition
  - O(log n)

- For given k operand bits - generate two outputs - each with k sum bits and an outgoing carry - one for incoming carry 0 and one for 1
- When incoming carry known - select correct output out of two - no waiting for carry to propagate
- Should not apply this idea to all n bits at once

Dividing into Groups

- Divide n bits into smaller groups - apply above to each
- Serial carry-propagation inside groups done in parallel
- Groups can be further divided into subgroups
- Outputs of subgroups combined to generate output of groups
- Natural division of n - two groups of $n/2$ bits each
- Each can be divided into two groups of $n/4$, and so on
- If n power of 2 - last subgroup is of size 1 and $\log_2 n$ steps are needed
- Division not necessarily into equal-sized subgroups - scheme can be applied even if n not a power of 2

Example - Combining Single Bits into Pairs

- s_i^0 / s_i^1 - sum bit at position i under the assumption that incoming carry into currently considered group is 0 /1
- Similarly - outgoing carries (from group) c_{i+1}^0 / c_{i+1}^1
- Step 1 - each bit constitutes a separate group:

i	7	6	
x_i	1	0	
y_i	0	0	
s_i^0	1	0	Assuming incoming carry = 0
c_{i+1}^0	0	0	
s_i^1	0	1	Assuming incoming carry = 1
c_{i+1}^1	1	0	

Example - Step 2

- **Step 2** - two bit positions combined (using data selectors) into one group of size 2
- Carry-out from position 6 becomes internal (to group) carry and appropriate set of outputs for position 7 selected

i	7	6
x_i	1	0
y_i	0	0
s_i^0	1	0
c_{i+1}^0	0	0
s_i^1	0	1
c_{i+1}^1	1	0

$i, i-1$	7, 6	
x_i, x_{i-1}	10	
y_i, y_{i-1}	00	
s_i^0, s_{i-1}^0	10	Assuming incoming carry $= 0$
c_{i+1}^0	0	
s_i^1, s_{i-1}^1	11	Assuming incoming carry $= 1$
c_{i+1}^1	0	

Example - Addition of Two 8-bit Operands

	i	7	6	5	4	3	2	1	0
	x_i	1	0	1	1	0	1	1	0
	y_i	0	0	1	0	1	1	0	1
Step 1	s_i^0	1	0	0	1	1	0	1	1
	c_{i+1}^0	0	0	1	0	0	1	0	0
	s_i^1	0	1	1	0	0	1	0	
	c_{i+1}^1	1	0	1	1	1	1	1	
Step 2	s_i^0	1	0	0	1	0	0	1	1
	c_{i+1}^0	0		1		1		0	
	s_i^1	1	1	1	0	0	1		
	c_{i+1}^1	0		1		1			
Step 3	s_i^0	1	1	0	1	0	0	1	1
	c_{i+1}^0	0				1			
	s_i^1	1	1	1	0				
	c_{i+1}^1	0							
Result		1	1	1	0	0	0	1	1

- Log $_2$8=3 steps
- Forced carry (=0 here) available at start
- Only one set of outputs generated for rightmost group at each step

Carry-Select Adder

- Variation of conditional sum adder
- n bits divided into groups - not necessarily equal
- Each group generates two sets of sum bits and an outgoing carry bit - incoming carry selects one
- Each group is not further divided into subgroups

- Comparing Conditional-sum and Carry-look-ahead
  - Both methods have same speed
  - Design of conditional sum adder less modular (why?)
  - Carry-look-ahead adder more popular

Optimality of Algorithms and Their Implementations

- Numerous algorithms for fast addition proposed - technology keeps changing making new algorithms more suitable
- Performance of algorithm affected by its unique features and number system used to represent operands and results
- Many studies performed to compare performance of different algorithms - preferably independently of implementation technology
- Some studies find the limit (bound) on the performance of any algorithm in executing a given arithmetic operation

Optimal Addition Algorithms

- Execution time reduced by avoiding (or limiting) carry-propagation
- Number systems such as the residue number system and the SD number system have almost carry-free addition - provide fast addition algorithms
- These number systems not frequently used - conversions between number systems needed - may be more complex than addition - not always practical

Lower Bound on Addition Speed

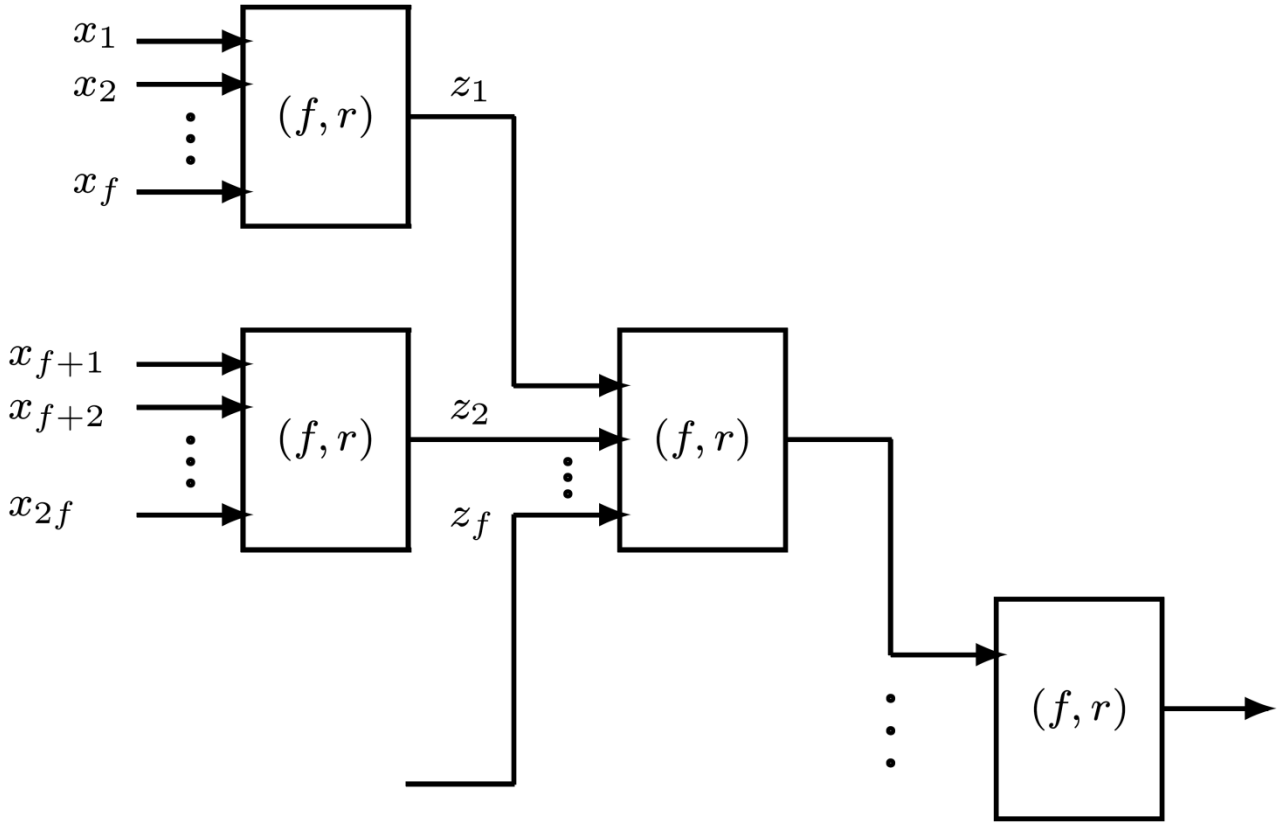
- **Theoretical model** - derives a bound independent of implementation technology
- **Assumptions:**
  - Circuit for addition realized using only one type of gate - (f,r) gate - r is radix of number system used and f is fan-in of gate (maximum number of inputs)
  - All (f,r) gates are capable of computing any r-valued function of f (or less) arguments in exactly the same time
  - This fixed time period is the unit delay - computation time of adder circuit measured in these units
- (f,r) gate can compute any function of f arguments - all we need to find out is how many such gates are required and how many circuit levels are needed in order to properly connect the gates

Lower Bound - Cont.

- A circuit for adding two radix-r operands with n digits each - 2n inputs and n+1 outputs
- Consider output requiring all 2n inputs - can be reduced to a smaller number of arguments by using $\lceil 2n/f \rceil$ such (f,r) gates operating in parallel
- Number of intermediate arguments - $\lceil 2n/f \rceil$ - can be further reduced by a second level of (f,r) gates
- Number of levels in tree - at least $\lceil \log_f 2n \rceil$
- Lower bound - assumes that no argument is needed as input to more than one (f,r) gate
- Lower bound on addition time - measured in units of (f,r) gate delay -

$$T_{add} \geq \lceil \log_f 2n \rceil$$

Circuit Implemented with (f,r) Gates



Limitations of Model

- Only fan-in limitation considered - fan-out ignored
- Fan-out of gate - ability of its output to drive a number of inputs to similar gates in the next level
- In practice fan-out is constrained
- More important - model assumes that any r-valued function of f arguments can be calculated by a single (f,r) gate in one unit delay - not true in practice – O(f)
- Many functions require either a more complex gate (longer delay) or are implemented using several simple gates organized in two or more levels

Improved Bound

- Previous bound assumes at least one output digit that depends on all 2n input digits
- If not - a better (lower) value for the bound exists - smaller trees (with fewer inputs) can be used
- This occurs if carry cannot propagate from least-significant to most-significant position
- Example - only $x_i, y_i, x_{i-1}, y_{i-1}$ needed to determine sum digit s_i -

$$T_{add} \geq \lceil \log_f 4 \rceil$$

- In the binary system - carry can propagate through all n positions -

$$T_{add} \geq \lceil \log_f 2n \rceil$$

- In the two addition algorithms - carry-look-ahead and conditional sum - execution time proportional to log n - previous bound approached

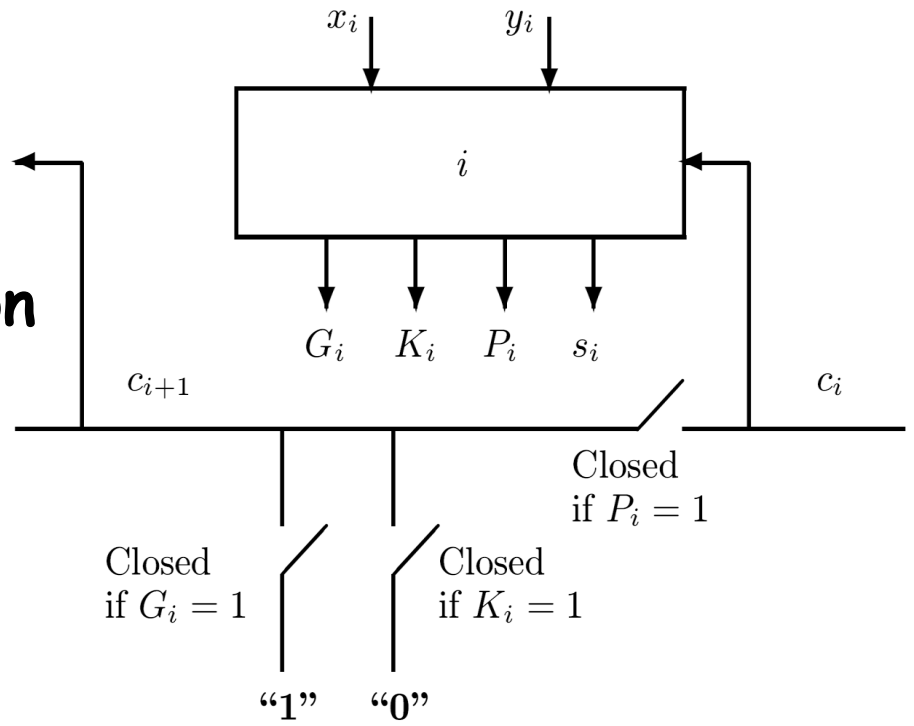
Implementation Cost

- Implementation cost must be considered in addition to execution time
- Implementation cost measure depends on technology
- Example - discrete gates
  - Number of gates measures implementation cost
  - Number of gates along the critical (longest) path (number of circuit levels) determines execution time
- Example - full custom VLSI technology
  - Number of gates - limited effect on implementation cost
  - Regularity of design and length of interconnections more important - affect both silicon area and design time
- Trade-off between implementation cost and addition speed exists

Performance - Cost Trade-off

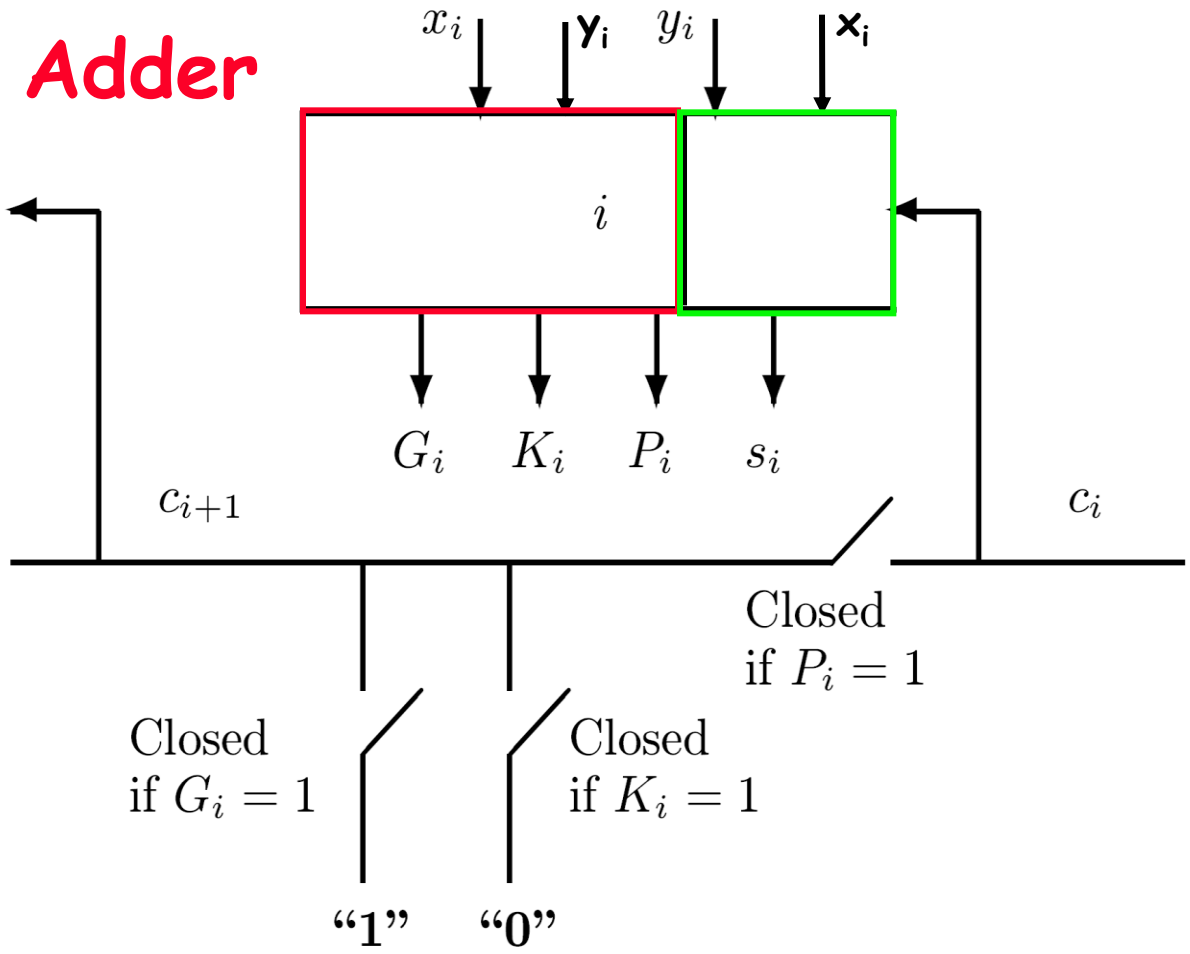
- If performance more important - carry-look-ahead adder preferable
- Implementation cost can be reduced - determined by regularity of design and size of required area
- Taking advantage of the available degree of freedom in design - the blocking factor - bounded by fan-in constraint
- Additional constraints exist - e.g., number of pins
- Highest blocking factor - not necessarily best
- Example - blocking factor of 2 results in a very regular layout of binary trees with up to $\log_2 n$ levels - total area approximately $n \cdot \log_2 n$

Manchester Adder



- If lower cost implementation required, ripple-carry method with speed-up techniques is best
- Manchester adder uses switches that can be realized using pass transistors
  - $P_i = x_i \oplus y_i$ carry-propagate signal
  - $G_i = x_i y_i$ carry-generate signal
  - $K_i = \bar{x}_i \bar{y}_i$ carry-kill signal
- Only one of the switches is closed at any time

Manchester Adder



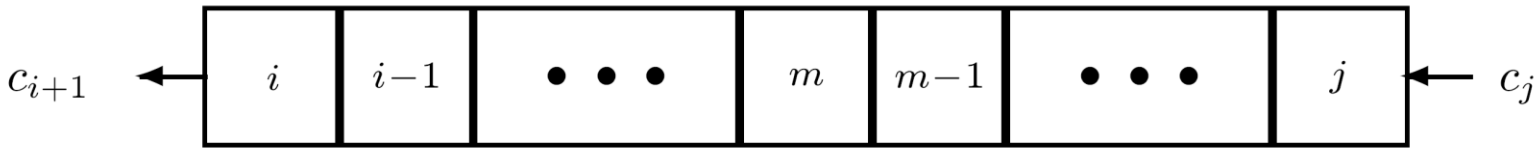
- Pi=xi ⊕ yi used instead of Pi=xi + yi
- If Gi=1 - an outgoing carry is generated always
- If Ki=1 - incoming carry not propagated
- If Pi=1 - incoming carry propagated

Manchester Adder

- Switches in units 0 through n-1 set simultaneously - propagating carry experiences only a single switch delay per stage
- Number of carry-propagate switches that can be cascaded is limited to k, which depends on technology
  - delay per group – $O(k^2)$
- n units partitioned into groups with separating devices (buffers) between them
- In theory - execution time is still linearly proportional to $O(n)$ although it is faster
- In practice - ratio between execution time and that of another adder (e.g., carry-look-ahead) depends on particular technology
- Implementation cost - measured in area and/or design regularity - lower than carry-look-ahead adder

Carry-Look-Ahead Addition Revisited

- Generalizing equations for fast adders - carry-look-ahead, carry-select and carry-skip
- Notation:



  - $P_{i:j}$ - group-propagated carry
  - $G_{i:j}$ - group-generated carry

  for group of bit positions $i, i-1, \ldots, j$ $(i \geq j)$
- $P_{i:j}=1$ when incoming carry into least significant position j, c_j, is allowed to propagate through all $i-j+1$ positions
- $G_{i:j}=1$ when carry is generated in at least one of positions j to i and propagates to $i+1$, $(c_{i+1} = 1)$
  - Generalization of previous equations
  - Special case - single bit-position functions P_i and G_i

Group-Carry Functions

- **Boolean equations**

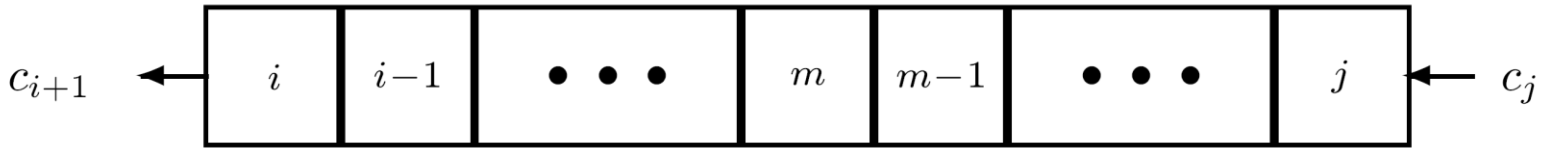
$$P_{i:j} = \begin{cases} P_i & \text{if } i = j \\ P_i \cdot P_{i-1:j} & \text{if } i > j \end{cases}$$

$$G_{i:j} = \begin{cases} G_i & \text{if } i = j \\ G_i + P_i \cdot G_{i-1:j} & \text{if } i > j \end{cases}$$

- **Pi:i ≡ Pi ; Gi:i ≡ Gi**
- **Recursive equations can be generalized (i ≥ m ≥ j+1)**

$$\begin{aligned} P_{i:j} &= P_{i:m} \cdot P_{m-1:j}, \\ G_{i:j} &= G_{i:m} + P_{i:m} \cdot G_{m-1:j} \end{aligned}$$

- **Proof - induction on m**



Fundamental Carry Operator

- Boolean operator - fundamental carry operator - o

$$(P,G)\circ(\widetilde{P},\widetilde{G})=(P\cdot\widetilde{P},G+P\cdot\widetilde{G})$$

- Using the operator o
- $(P_{i:j},G_{i:j})=(P_{i:m},G_{i:m})\circ(P_{m-1:j},G_{m-1:j})\ (i\geq m\geq j+1)$
- Operation is associative

$$((P_{i:m},G_{i:m})\circ(P_{m-1:v},G_{m-1:v}))\circ(P_{v-1:j},G_{v-1:j})$$

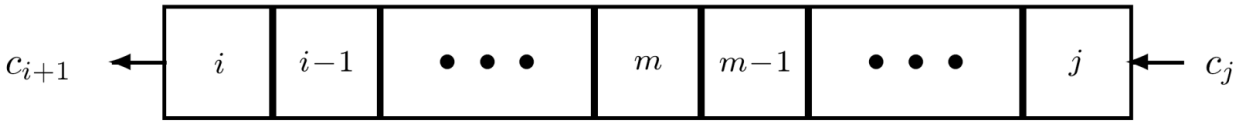
$$=(P_{i:m},G_{i:m})\circ((P_{m-1:v},G_{m-1:v})\circ(P_{v-1:j},G_{v-1:j}))$$

- Operation is idempotent

$$(P,G)\circ(P,G)=(P\cdot P,G+P\cdot G)=(P,G)$$

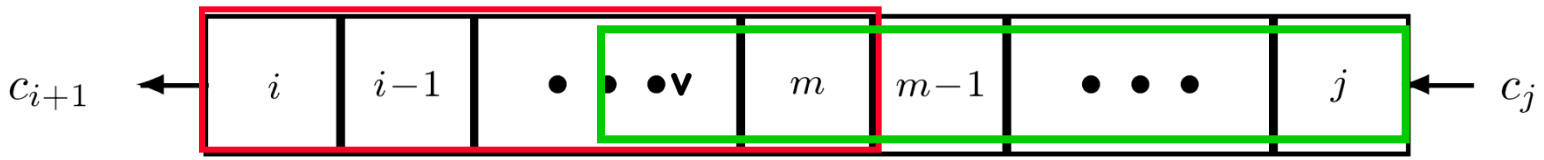
- Therefore

$(P_{i:j},G_{i:j})=(P_{i:m},G_{i:m})\circ(P_{v:j},G_{v:j})$ $\quad i\geq m\ ;\ v\geq j\ ;\ v\geq m-1$



Combining two subgroups

- Group carries $P_{i:j}$ and $G_{i:j}$ calculated from two subgroup carries - subgroups are of arbitrary size and may even overlap
- Group and subgroup carries used to calculate individual bit carries c_{i+1}, c_i,…, c_{j+1}, and sum outputs s_i, s_{i-1},…, s_j
- For the mth bit position, $i \geq m \geq j$
- $(P_{i:j},G_{i:j})=(P_{i:m},G_{i:m}) \circ (P_{v:j},G_{v:j})$ $i \geq m$; $v \geq j$; $v \geq m-1$



Individual Bit Carry & Sum

- Must take into account "external" carry c_j
- For the mth bit position, $i \geq m \geq j$

$$c_m = G_{m-1:j} + P_{m-1:j} \cdot c_j$$

- rewritten as

$$(P_{m-1:j}, G_{m-1:j}) \circ (1, c_j)$$

- If $P_m = x_m \oplus y_m$ then $s_m = c_m \oplus P_m$
- If $P_m = x_m + y_m$ then $s_m = c_m \oplus (x_m \oplus y_m)$

Various Adder Implementations

- Equations can be used to derive various implementations of adders - ripple-carry, carry-look-ahead, carry-select, carry-skip, etc.
- 5-bit ripple-carry adder: All subgroups consist of a single bit position ; computation starts at position 0, proceeds to position 1 and so on

$$(P_4, G_4) \circ \{(P_3, G_3) \circ ((P_2, G_2) \circ [(P_1, G_1) \circ \{(P_0, G_0) \circ (1, c_0)\}])\}$$

- 16-bit carry-look-ahead adder: 4 groups of size 4; ripple-carry among groups

$$(P_{15:12}, G_{15:12}) \circ \{(P_{11:8}, G_{11:8}) \circ [(P_{7:4}, G_{7:4}) \circ \{(P_{3:0}, G_{3:0}) \circ (1, c_0)\}]\}$$

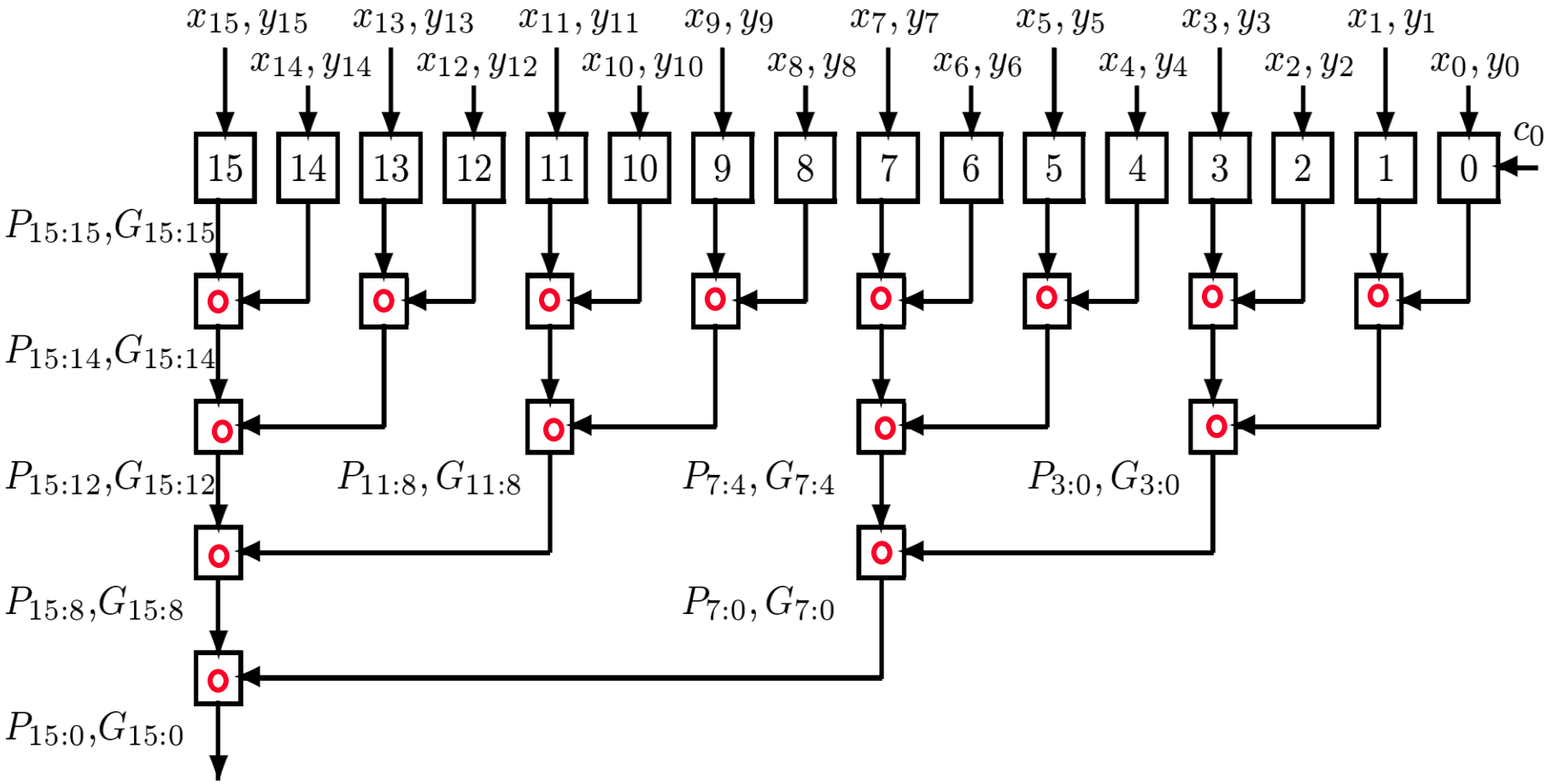
Brent-Kung Adder

- Variant of carry-look-ahead adder - blocking factor of 2 → very regular layout tree with $\log_2 n$ levels - total area ≈ $n \log_2 n$
- Consider c_{16} - incoming carry at stage 16 in a 17-bit (or more) adder and suppose $G_0 = x_0 y_0 + P_0 c_0$
- The part that generates $(P_{7:0}, G_{7:0})$ corresponds to

$$
\begin{aligned}
(P_{7:0}, G_{7:0}) &= (P_{7:4}, G_{7:4}) \circ (P_{3:0}, G_{3:0}) \\
&= \{(P_{7:6}, G_{7:6}) \circ (P_{5:4}, G_{5:4})\} \circ \{(P_{3:2}, G_{3:2}) \circ (P_{1:0}, G_{1:0})\} \\
&= \{[(P_7, G_7) \circ (P_6, G_6)] \circ [(P_5, G_5) \circ (P_4, G_4)]\} \\
&\quad \circ \{[(P_3, G_3) \circ (P_2, G_2)] \circ [(P_1, G_1) \circ (P_0, G_0)]\}
\end{aligned}
$$

- Each line, except c_0, represents two signals - either x_m, y_m or $P_{v:m}, G_{v:m}$

Tree Structure for Calculating C_{16}

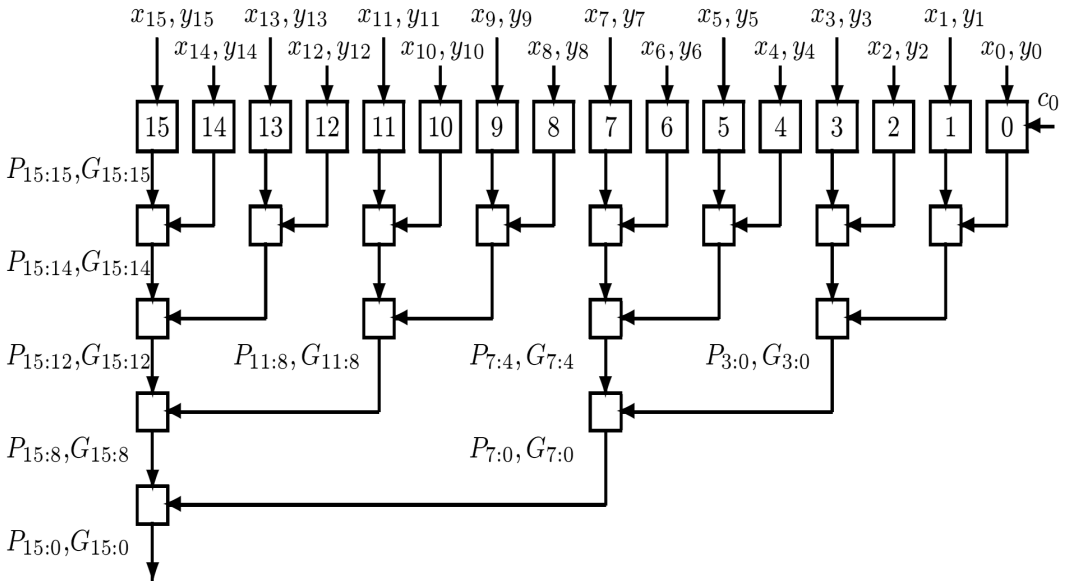


- **Fundamental carry operator -** $\circ$

$$(P, G) \circ (\widetilde{P}, \widetilde{G}) = (P \cdot \widetilde{P}, G + P \cdot \widetilde{G})$$

Carry Calculation

- Circuits in levels 2 to 5 implement fundamental carry op
- $c_{16}=G_{15:0}$; $P_m=x_m \oplus y_m$
  sum: $s_{16}=c_{16} \oplus P_{16}$

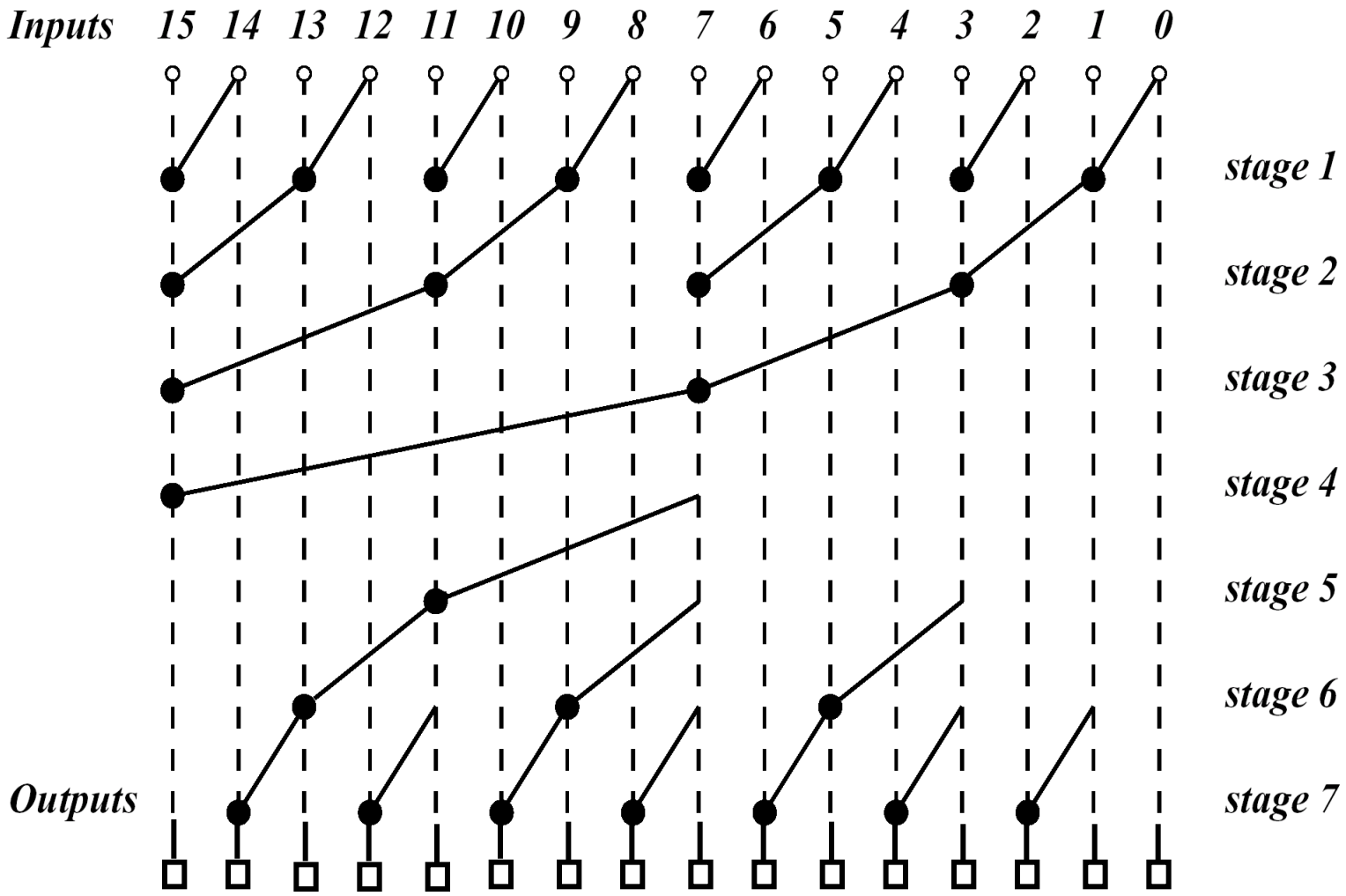


- Tree structure also generates carries c_2, c_4 and c_8
- Carry bits for remaining positions can be calculated through extra subtrees that can be added
- Once all carries are known - corresponding sum bits can be computed
- Above - blocking factor = 2
  - Different factors for different levels may lead to more efficient use of space and/or shorter interconnections

Prefix Adders

- The BK adder is a parallel prefix circuit - a combinational circuit with 2n inputs (P_1, G_1), $(P_2, G_2), \ldots, (P_n, G_n)$
- producing outputs (P_1, G_1), $(P_2, G_2) \circ (P_1, G_1), \ldots,$ $(P_n, G_n) \circ (P_{n-1}, G_{n-1}) \circ \ldots \circ (P_1, G_1)$, where $\circ$ is an associative binary operation
- (before the parallel prefix circuit) First stage of adder generates individual P_i and G_i from xi and yi
- Remaining stages constitute the parallel prefix circuit with fundamental carry operation serving as the $\circ$ associative binary operation
- This part of tree can be designed in different ways

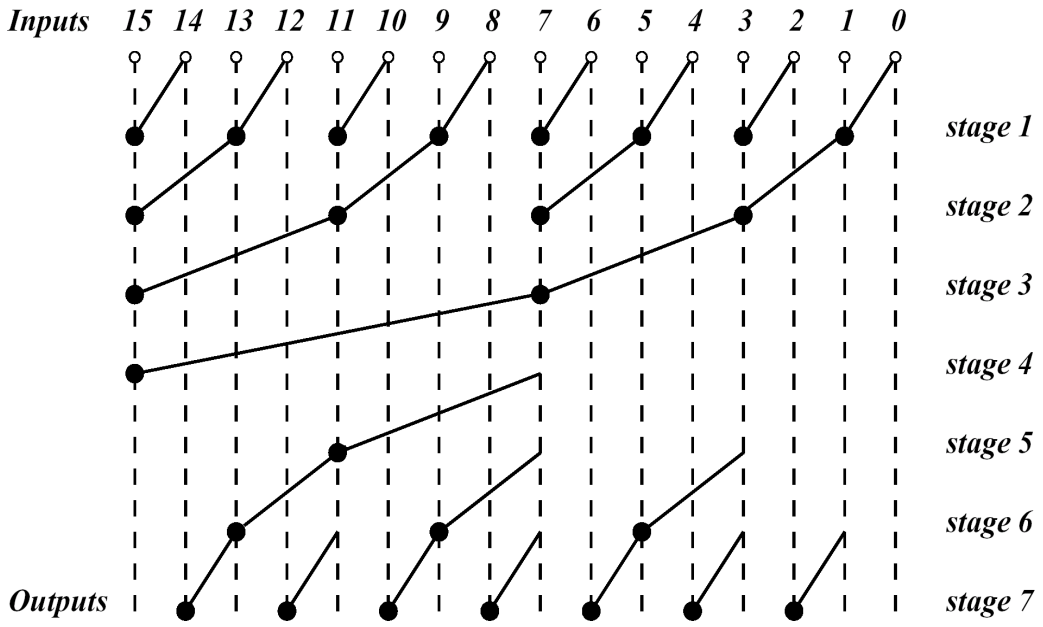
Parallel prefix graph of the 16-bit Brent-Kung Adder



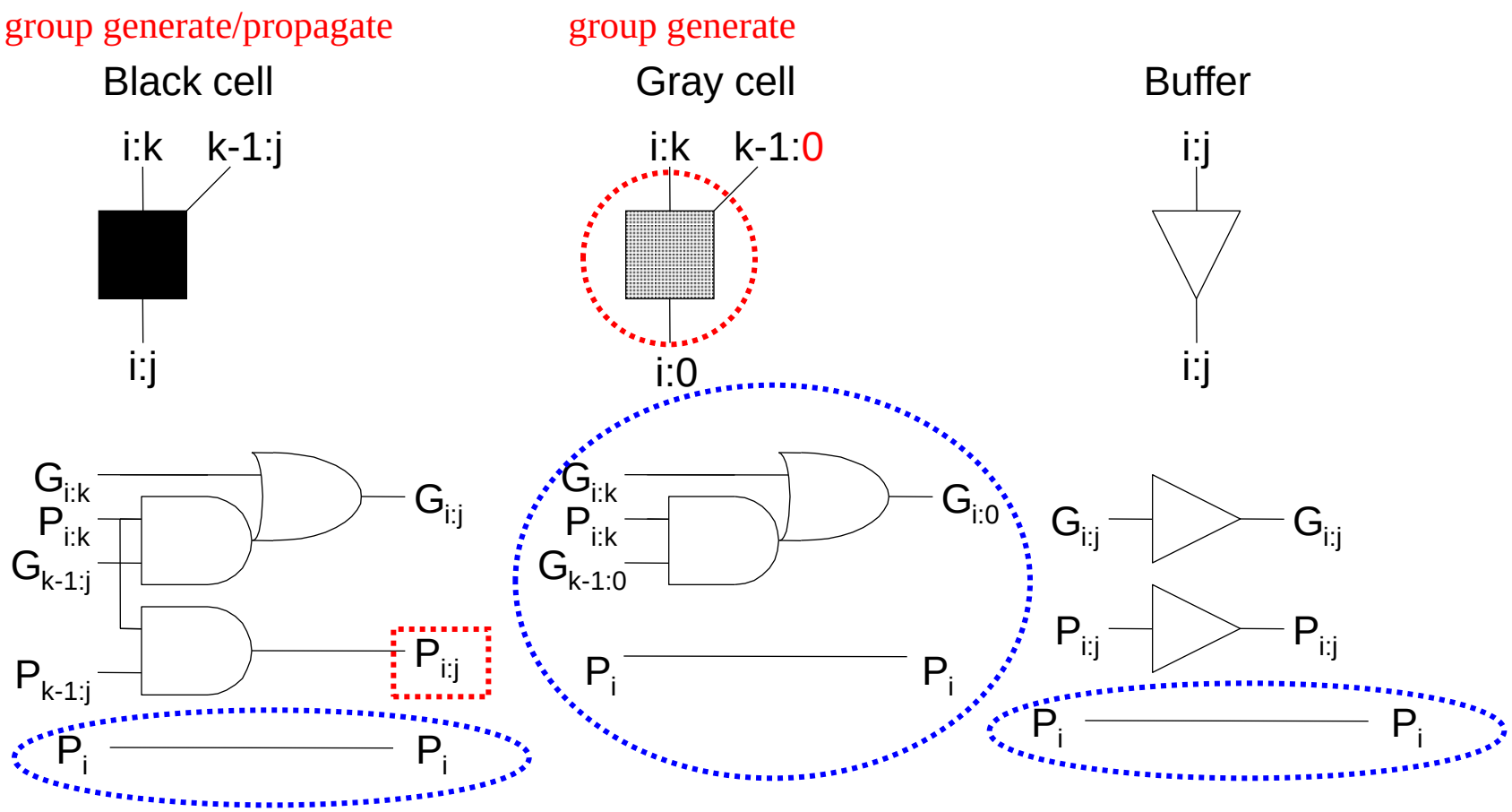
o : p & g generator ● : fundamental carry operation □ : sum generator

Brent-Kung Parallel Prefix Graph

- Bullets implement the fundamental carry operation - empty circles generate individual Pi and Gi
- Number of stages and total delay - can be reduced by modifying structure of parallel prefix graph
- Minimum # of stages = $\log_2 n$
  - 4 for n=16
  - For BK parallel prefix graph = $2\log_2 n - 1$



Prefix Diagram Notation

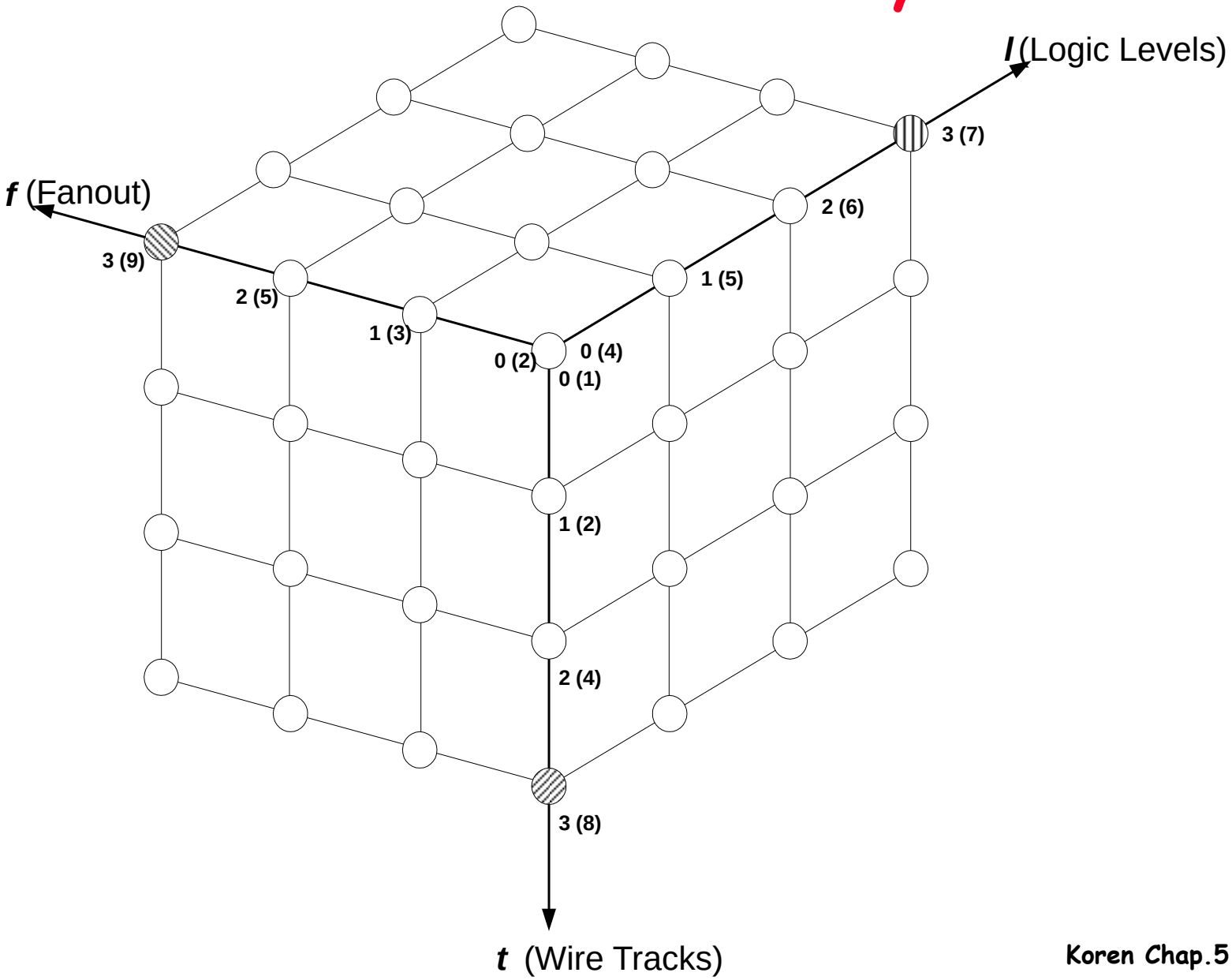


Tree Adder Taxonomy

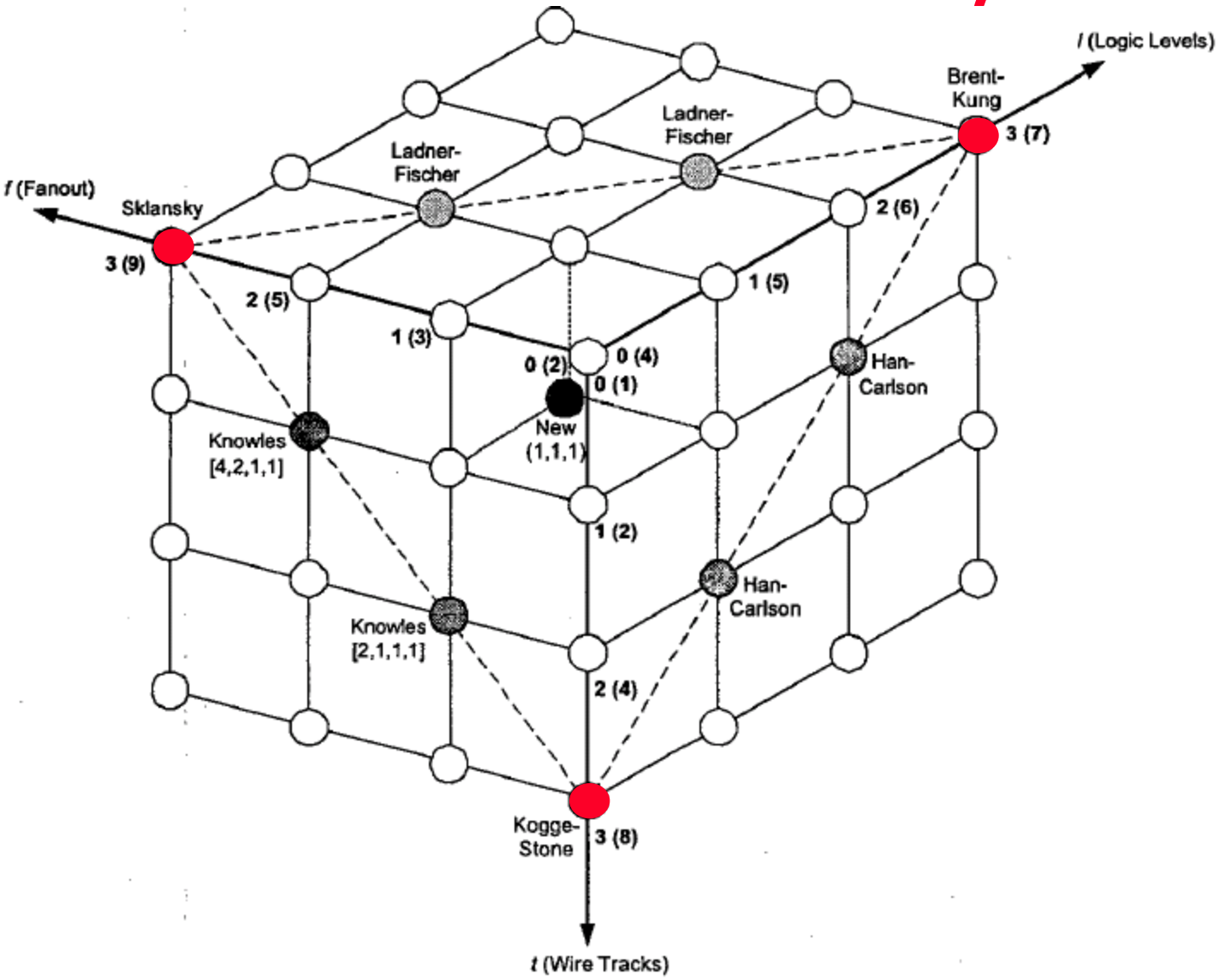
- **Ideal** N-bit tree adder would have
  - $L = \log N$ logic levels
  - Fanout never exceeding 2
  - No more than one wiring track between levels
- Describe adder with 3-D taxonomy (l, f, t)
  - Logic levels: $L + l$
  - Fanout: $2^f + 1$
  - Wiring tracks: 2^t
- **Known tree adders** sit on plane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

$$l + f + t = L - 1$$

Tree Adder Taxonomy

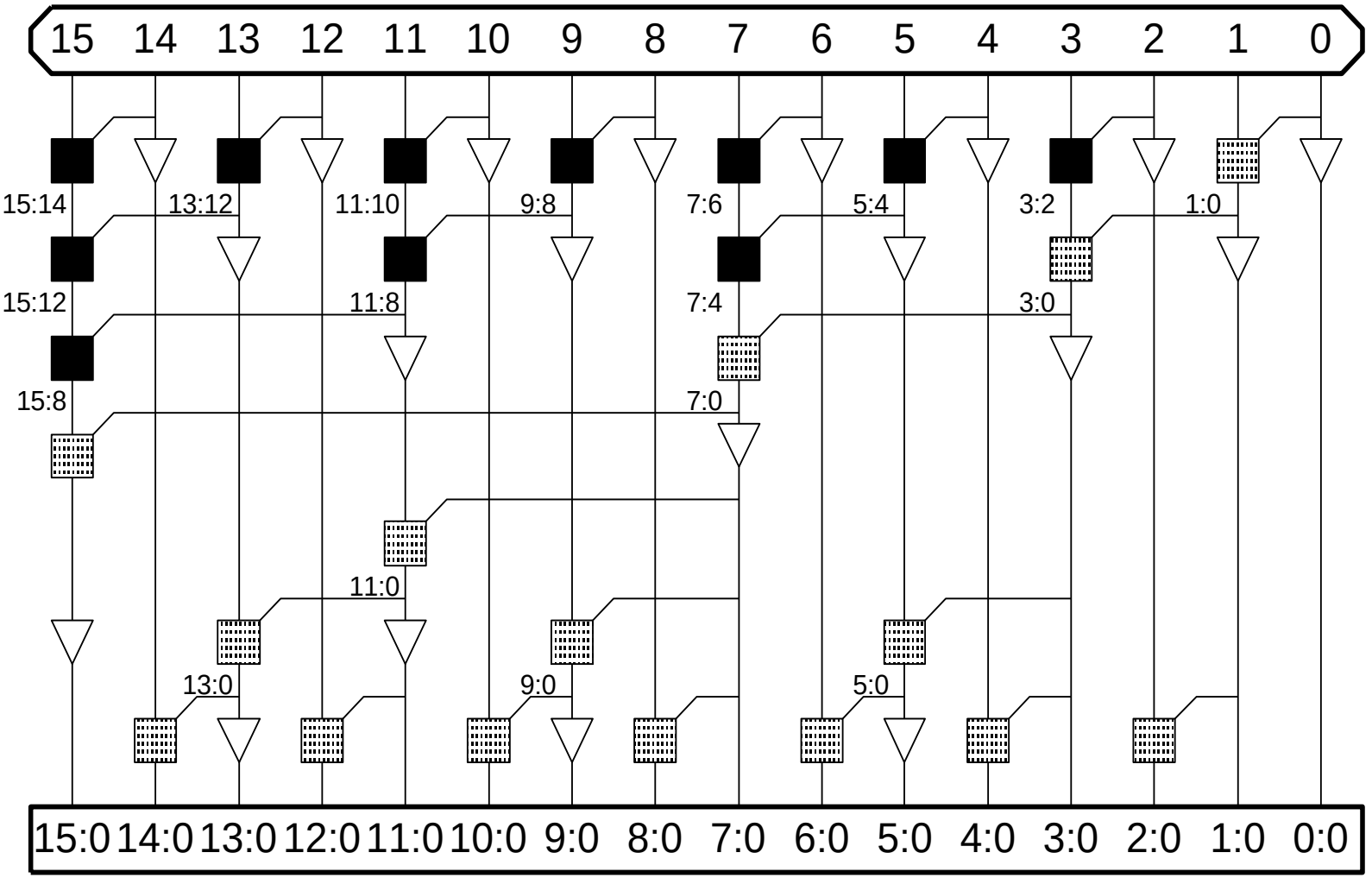


Tree Adder Taxonomy



Brent-Kung Adder

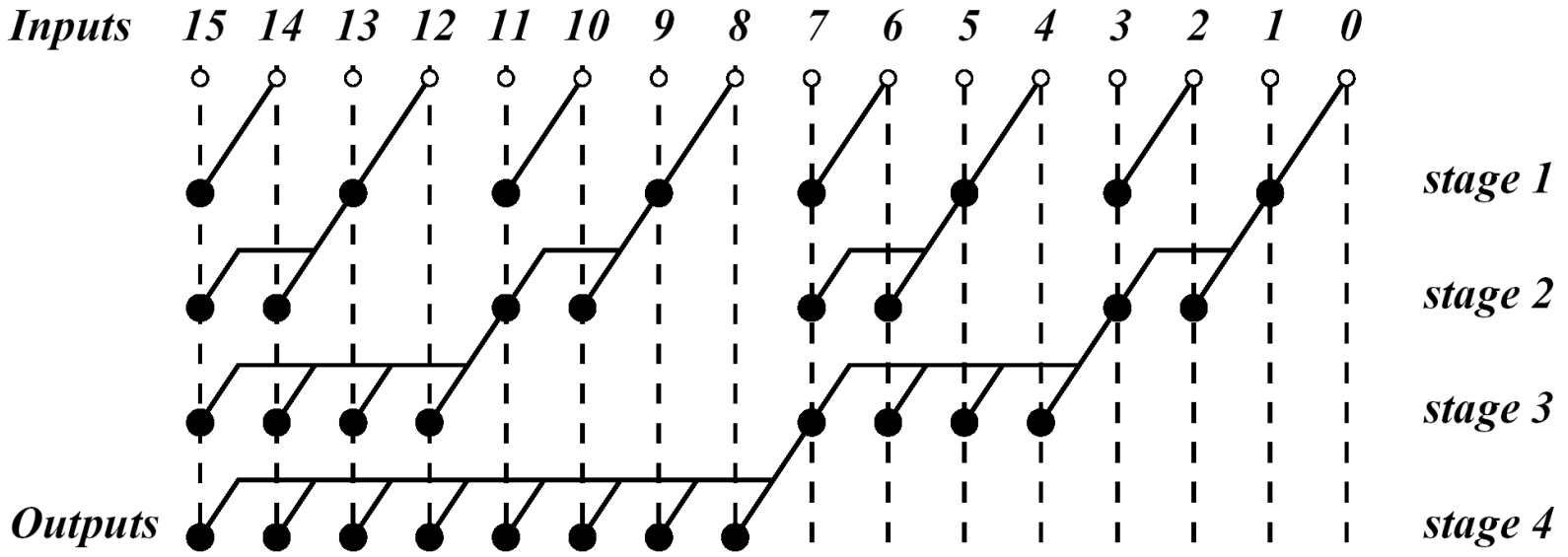
(l,f,t)=(3,0,0)



# of levels = 7; max fanout = 2; max #of tracks = 1

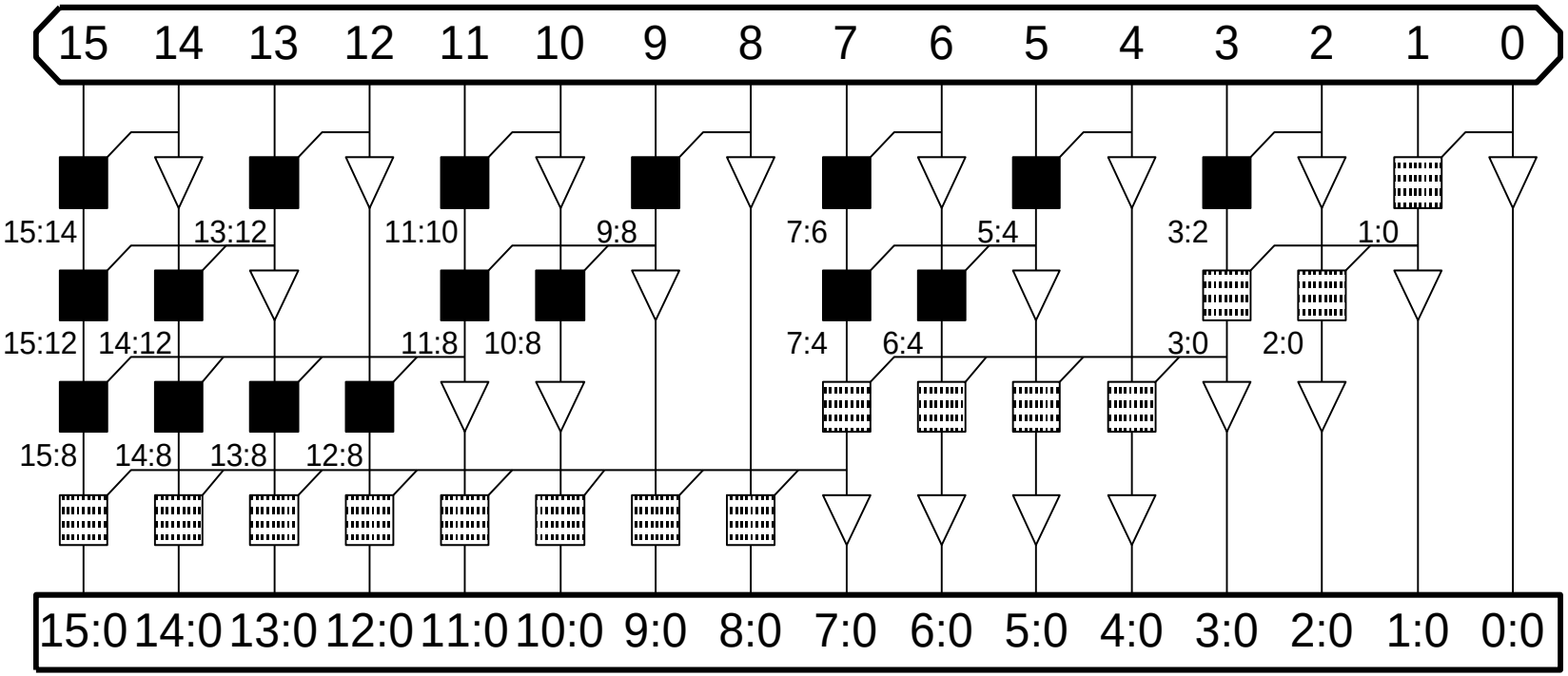
Sklansky Adder

- As a special case of Ladner-Fisher adder
- Implementing a 4-stage parallel prefix graph
- Unlike BK, LF adder employs fundamental carry operators with a fan-out ≥ 2 - blocking factor varies from 2 to n/2
- Fan-out $\leq$ n/2 requiring buffers : adding to overall delay



Sklansky Adder

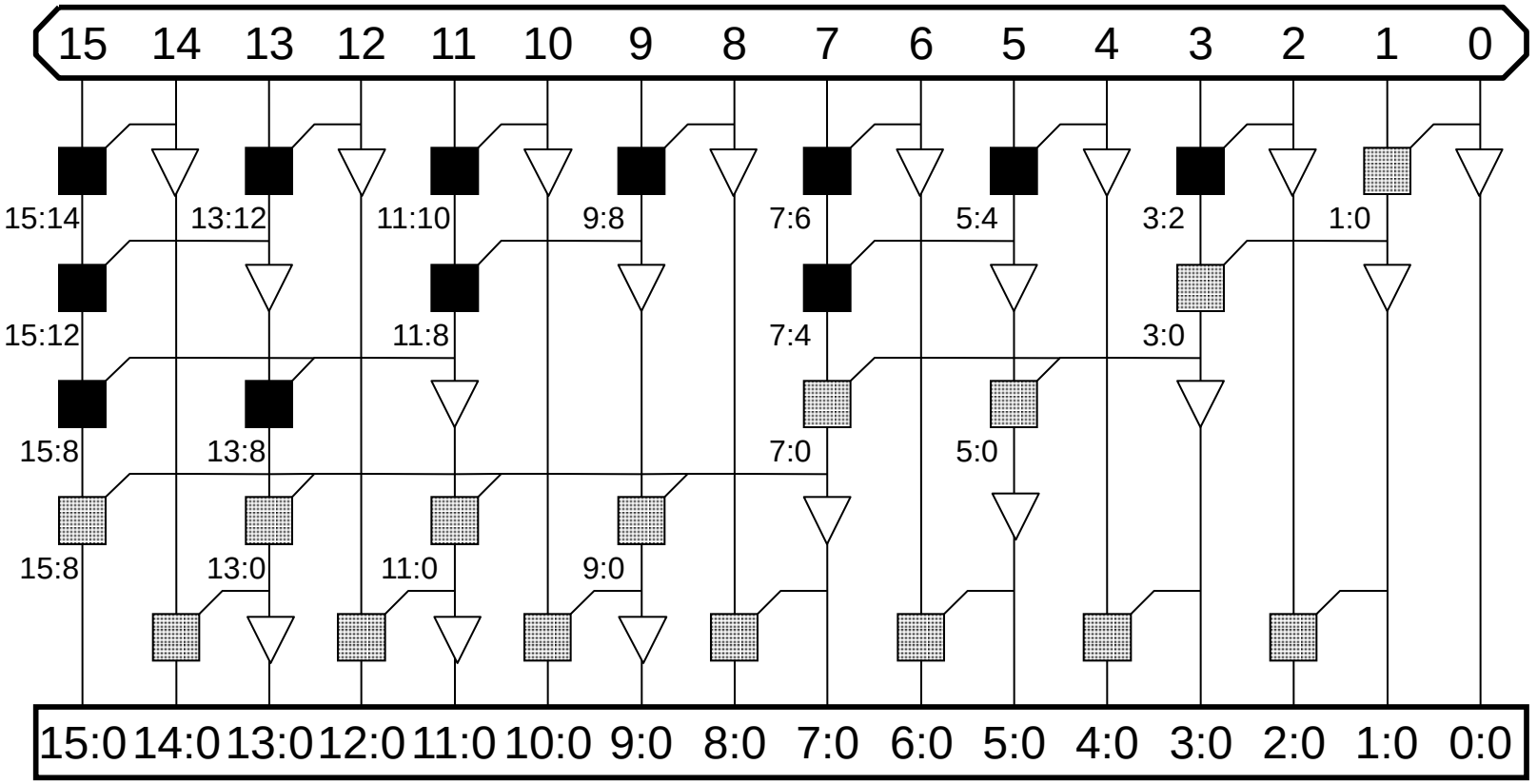
(l,f,t)=(0,3,0)



# of levels = 4; fanout (8,4,2,1); max # of tracks = 1

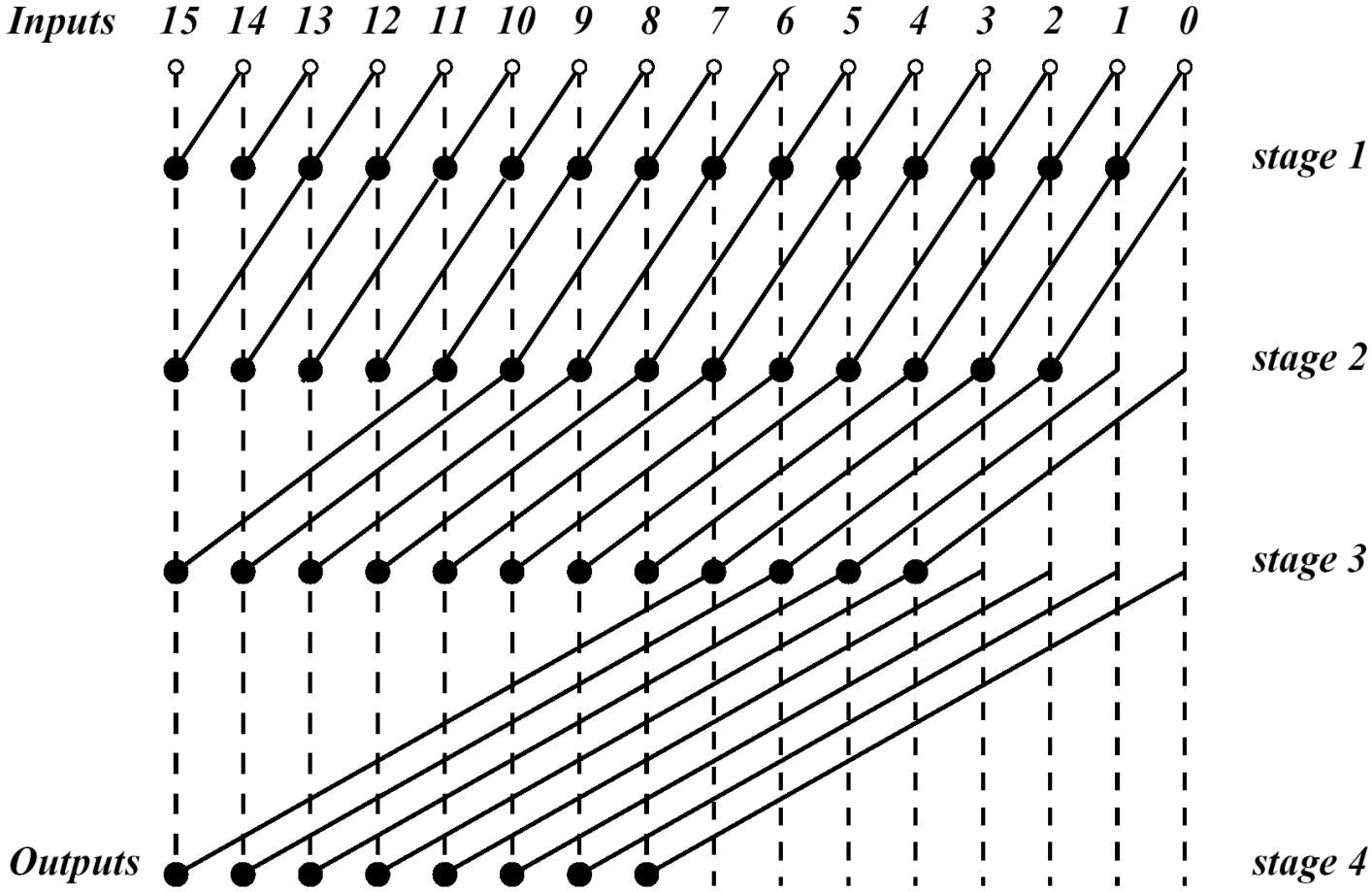
Ladner-Fischer

(l,f,t)=(1,2,0)



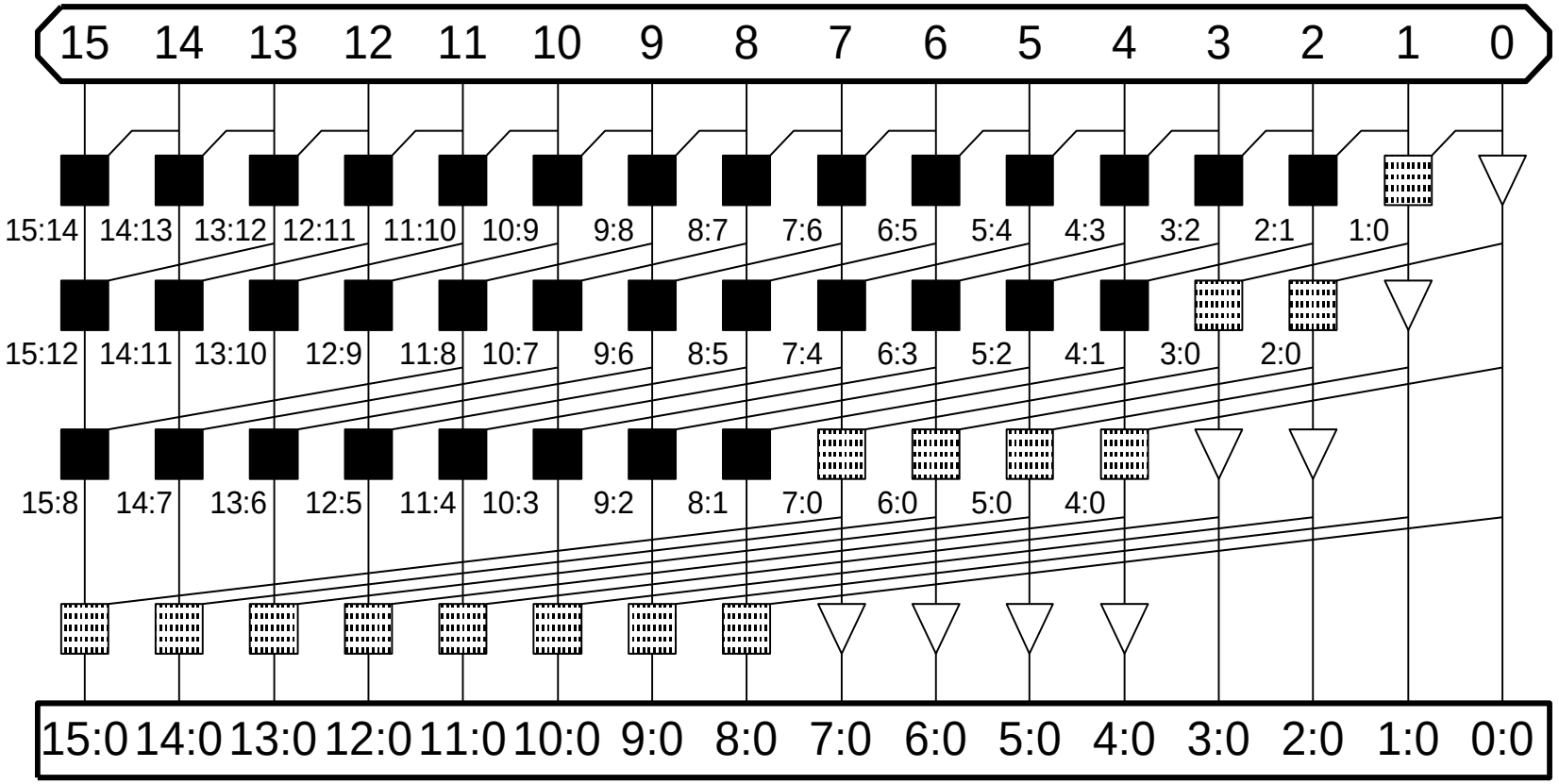
Kogge-Stone Adder

- $\log_2 n$ stages - but lower fan-out
- More lateral wires with long span than BK - requires buffering causing additional delay



Kogge-Stone

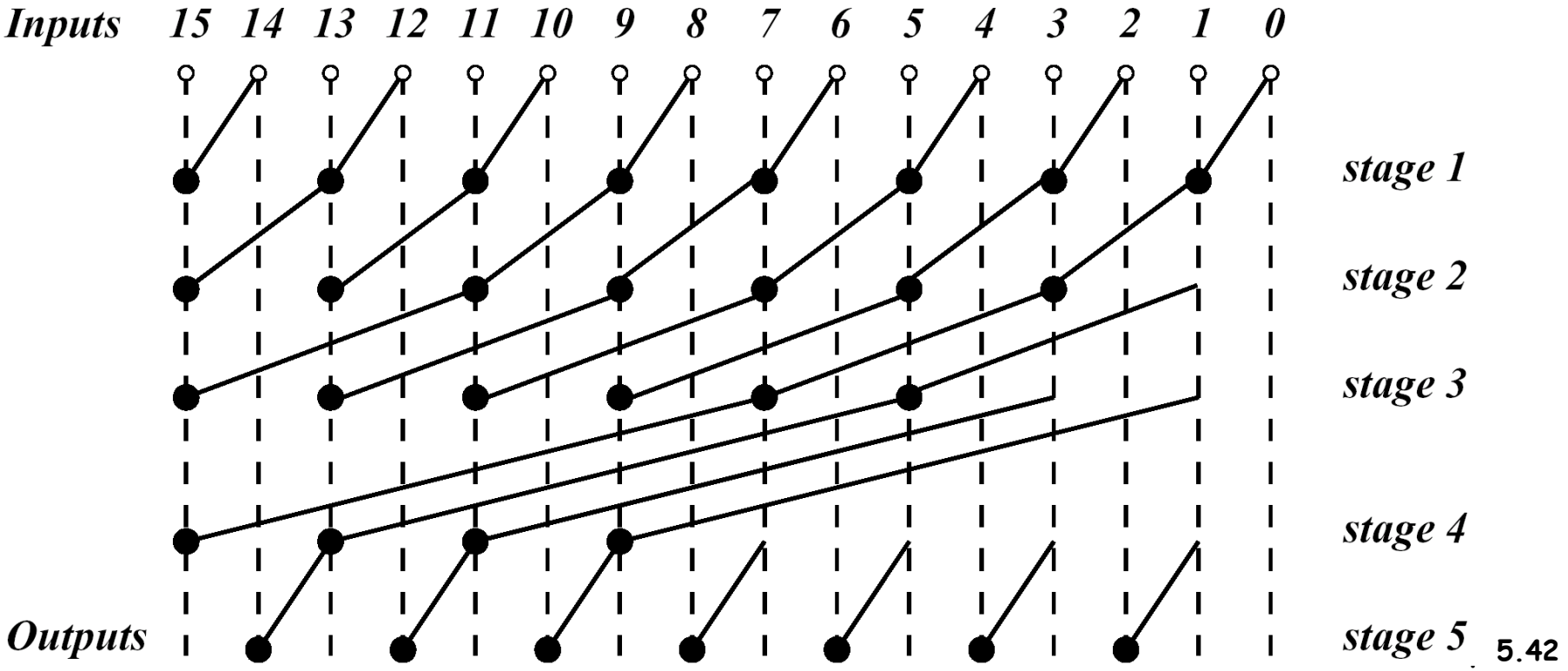
(l,f,t)=(0,0,3)



fanout (1,1,1,1)

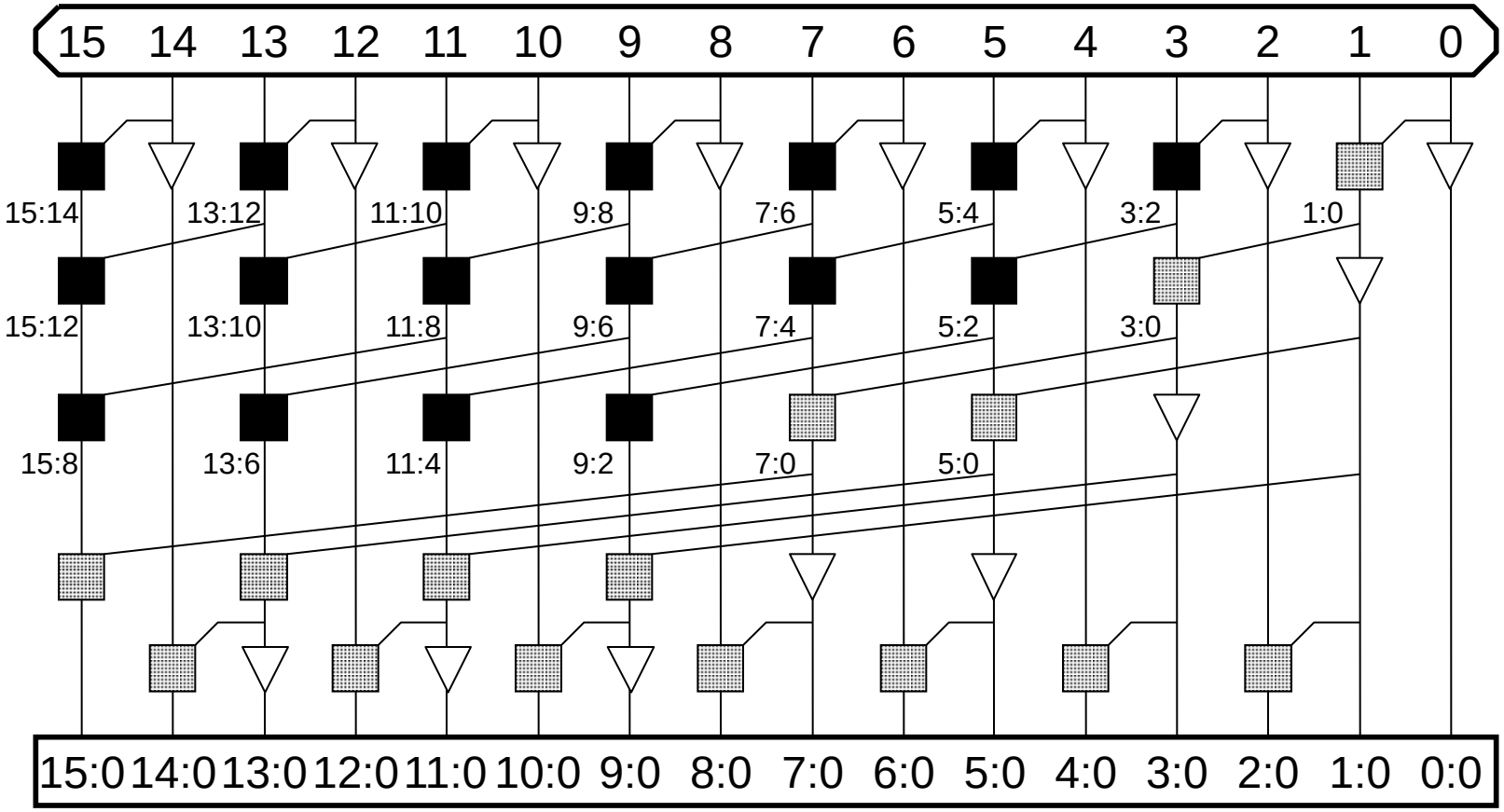
Han-Carlson Adder

- Other variants - small delay in exchange for high overall area and/or power
  - Compromises between wiring simplicity and overall delay
- A hybrid design combining stages from BK and KS
  - 5 stages - middle 3 resembling KS - wires with shorter span than KS



Han-Carlson

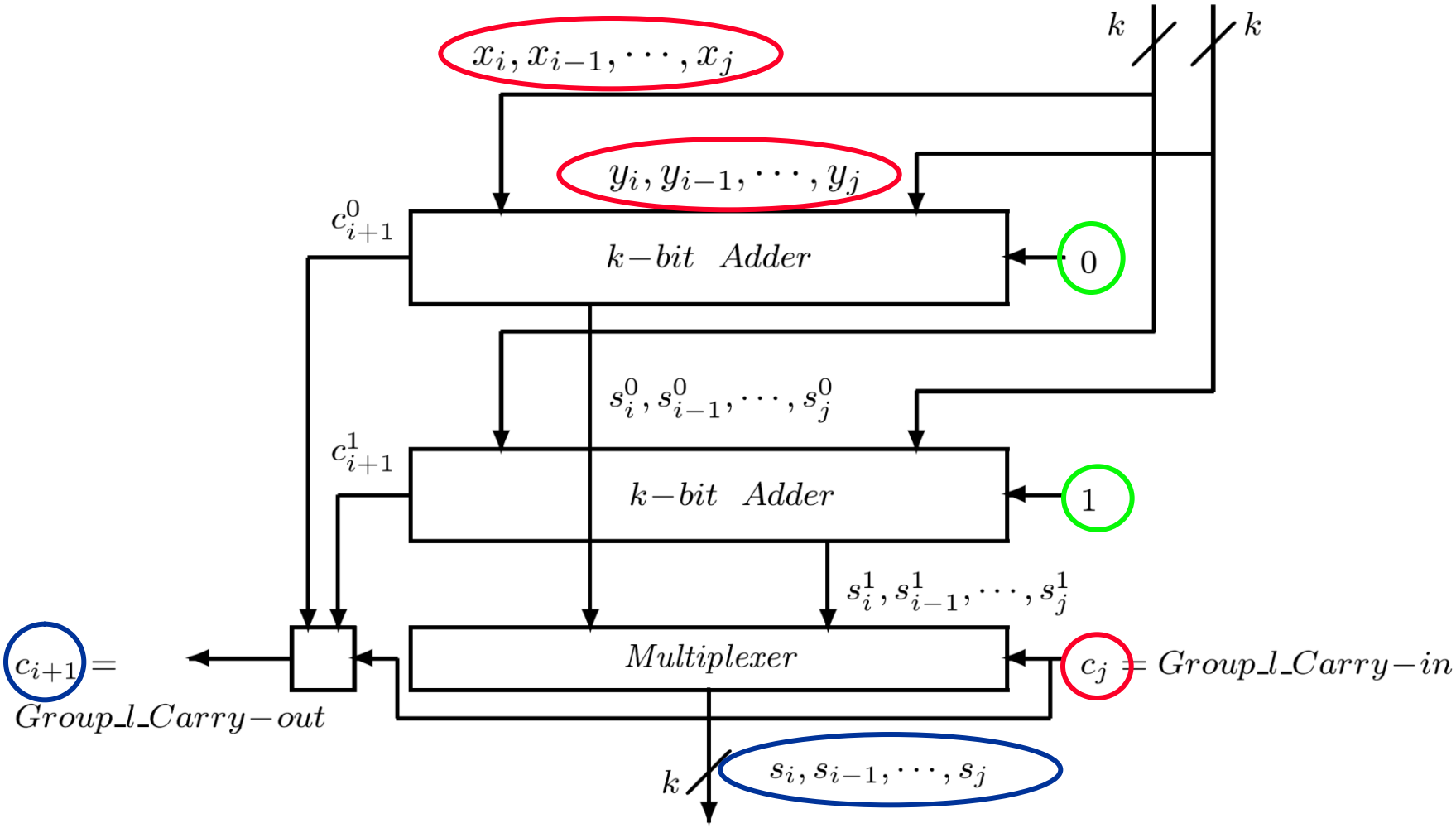
(l,f,t)=(1,0,2)



Carry-Select Adders

- n bits divided into non-overlapping groups of possibly different lengths - similar to conditional-sum adder
- Each group generates two sets of sum and carry; one assumes incoming carry into group is 0, the other 1
- the lth group consists of k bit positions starting with j and ending with i=j+k-1

Carry-Select Adders



Carry-Select Adder - Equations

- Outputs of group: sum bits S_i, S_{i-1}, ... , S_j and group outgoing carry C_{i+1}

$$s_m = s_m^0 \cdot \overline{c_j} + s_m^1 \cdot c_j \; ; \qquad m = j, j+1, \cdots, i$$

$$c_{i+1} = c_{i+1}^0 \cdot \overline{c_j} + c_{i+1}^1 \cdot c_j$$

- Same notation as for conditional-sum adder
- Two sets of outputs can be calculated in a ripple-carry manner

Detailed Expressions

- For bit m - calculate carries from $G^0_{m-1:j}$; $G^1_{m-1:j}$

$$(P_{m-1:j}, G^0_{m-1:j}) = (P_{m-1}, G_{m-1}) \circ (P_{m-2}, G_{m-2}) \circ \cdots \circ (P_j, G_j)$$

$$(P_{m-1:j}, G^1_{m-1:j}) = (P_{m-1:j}, G^0_{m-1:j}) \circ (1,1) = (P_{m-1:j}, G^0_{m-1:j} + P_{m-1:j})$$

- $P_{m-1:j}$ has no superscript - independent of incoming carry
- Once individual carries are calculated - corresponding sum bits are

$$s^0_m = c^0_m \oplus P_m \qquad \text{and} \qquad s^1_m = c^1_m \oplus P_m$$

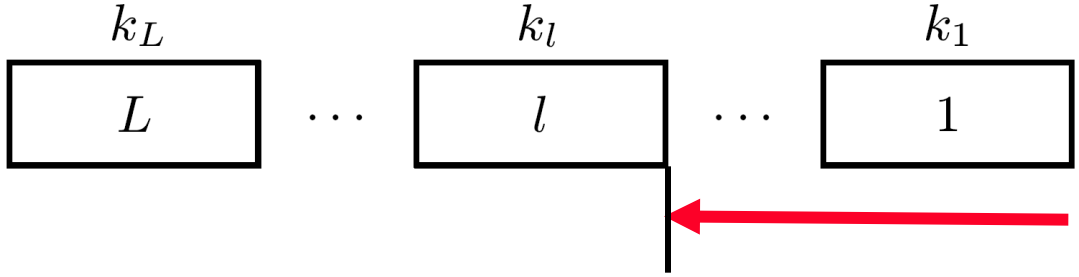
- Since C^0_{i+1} implies C^1_{i+1}, $c_{i+1} = c^0_{i+1} + c^1_{i+1} \cdot c_j$
- Group sizes can be either different

$$c_{i+1} = c^0_{i+1} \cdot \overline{c_j} + c^1_{i+1} \cdot c_j$$

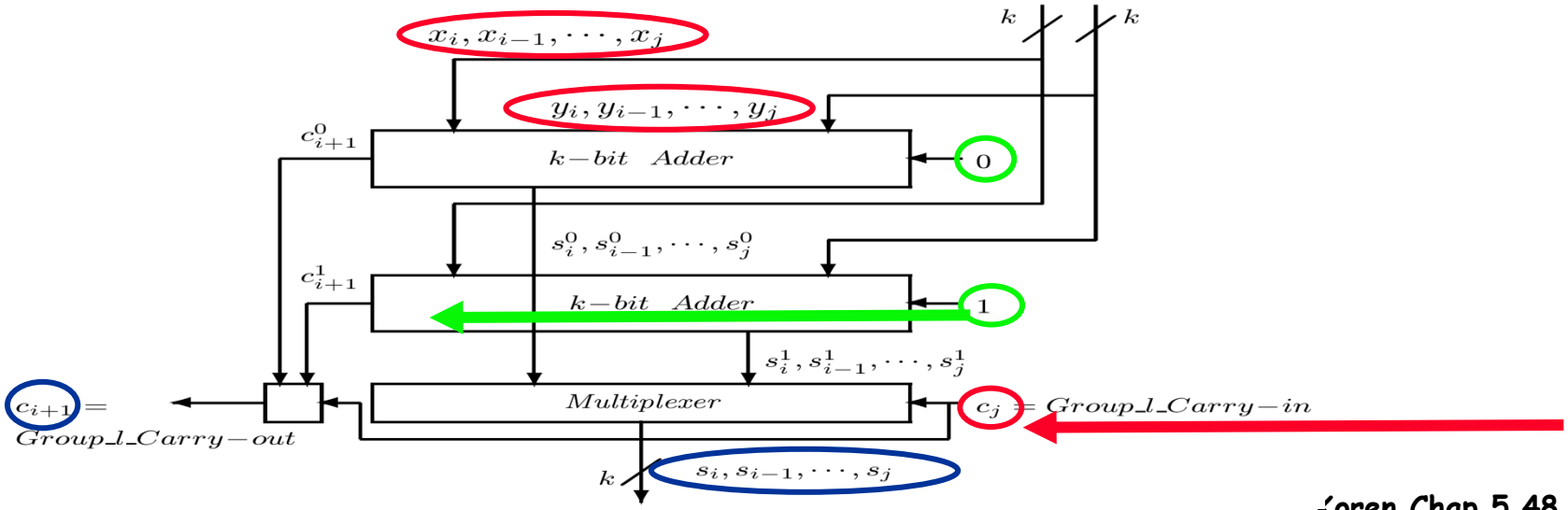
and $C_{i+1} \le C_{i+1}$

Different Group Sizes

- **Notations:**
  - Size of group l - k_l
  - L - number of groups
  - Δ_G - delay of a single gate



- k_l chosen so that delay of ripple-carry within group is equal to delay of carry-select chain from group 1 to l
- Actual delays depend on technology and implementation



Different Group Sizes

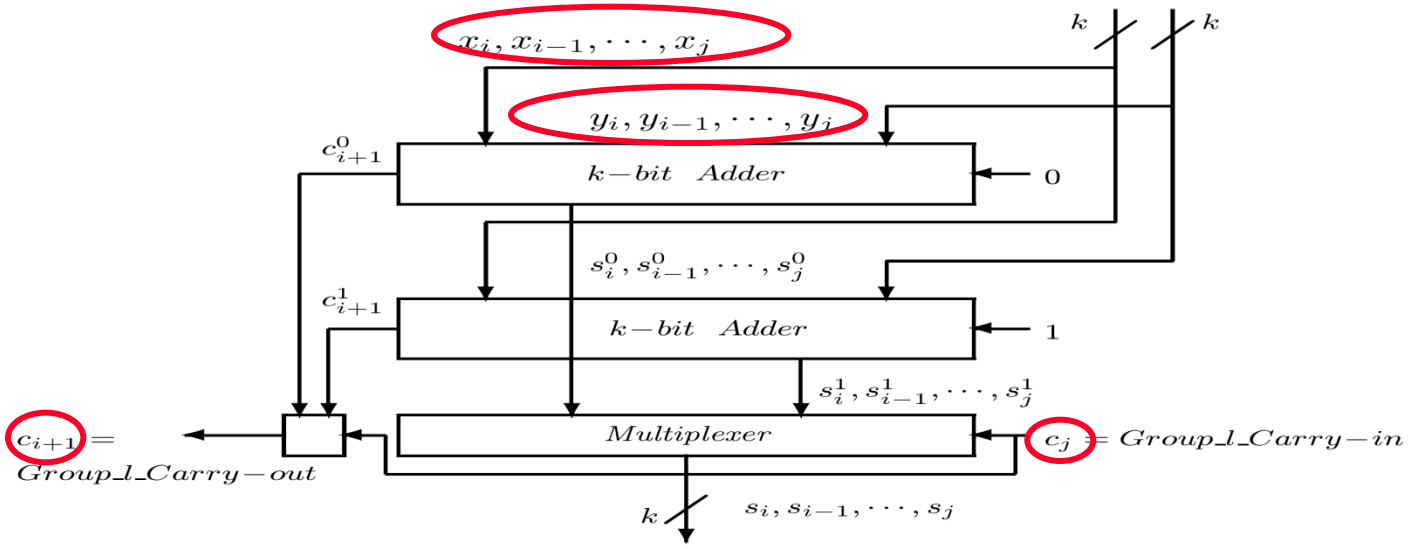
- **Example:** Two-level gate implementation of MUX
  - Delay of carry-select chain through preceding $l-1$ groups - $(l-1)2\Delta_G$
  - Delay of ripple-carry in lth group - $k_l\, 2\Delta_G$
- Equalizing the two - $k_l = l-1$ with $k_l \geq 1$; $l=1,2,\ldots,L$

Different Group Sizes - Cont.

- Resulting group sizes - 1, 1, 2, 3, ...
- Sum of group sizes $\geq n$
- $1+L(L-1)/2 \geq n \rightarrow L(L-1) \geq 2(n-1)$
- Size of largest group and execution time of carry-select adder are of the order of $\sqrt{n}$
- Example: $n=32$, 9 groups required - one possible choice for sizes: 1, 1, 2, 3, 4, 5, 6, 7 & 3
- Total carry propagation time is $18\Delta_G$, instead of $62\Delta_G$ for ripple-carry adder

Different Group Sizes - Cont.

- If sizes of L groups are equal, carry-select chain (i.e., generating Group Carry-Out from Group Carry-In) not necessarily ripple-carry type
- Single or multiple-level carry-look-ahead can be used

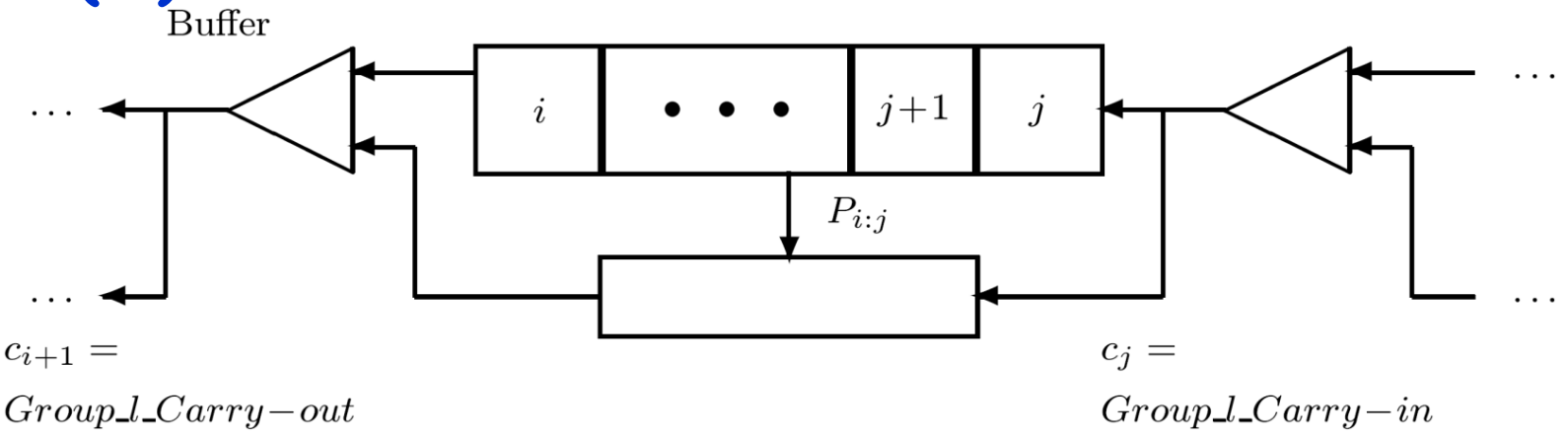


Carry-Skip Adders

- Reduces time needed to propagate carry by skipping over groups of consecutive adder stages
- Generalizes idea behind Manchester Adder
- Illustrates dependence of "optimal" algorithm for addition on available technology
  - Known for many years, only recently became popular
- In VLSI - speed comparable to carry look-ahead (for commonly used word lengths - not asymptotically)
- Requires less chip area and consumes less power
- Based on following observation:
- Carry propagation process can skip any adder stage for which $x_m \neq y_m$ (or, $P_m = x_m \oplus y_m = 1$)
- Several consecutive stages can be skipped if all satisfy $x_m \neq y_m$

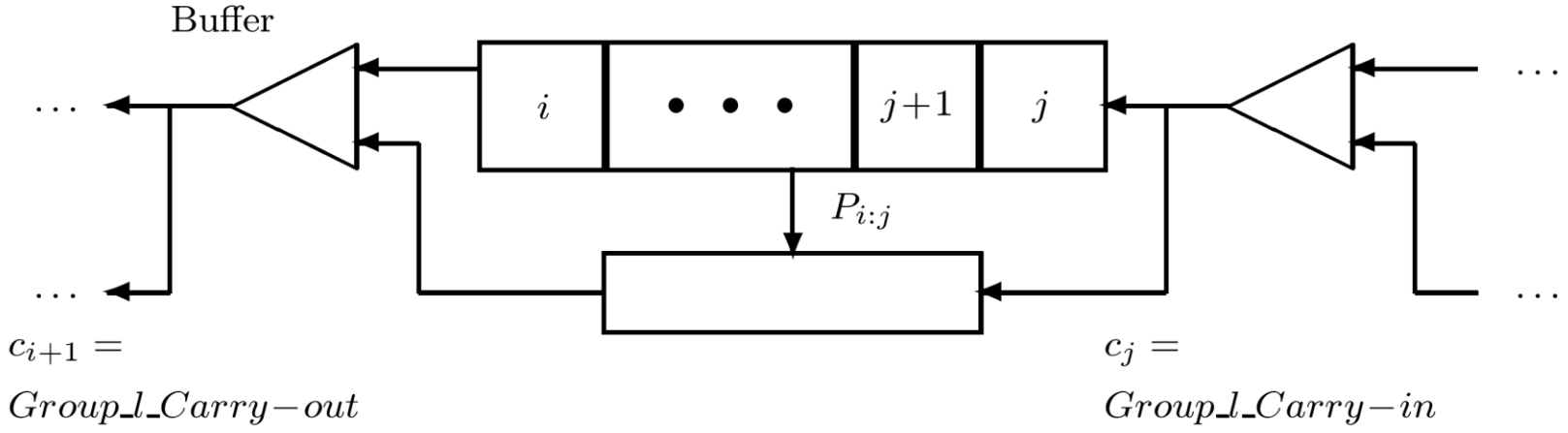
Carry-Skip Adder - Structure

- n stages divided into groups of consecutive stages with simple ripple-carry used in each group
- Group generates a group-carry-propagate signal that equals 1 if for all internal stages $P_m=1$
- Signal allows an incoming carry into group to "skip" all stages within group and generate a group-carry-out
- Group l consists of k bit positions j,j+1,…,j+k-1(=i)



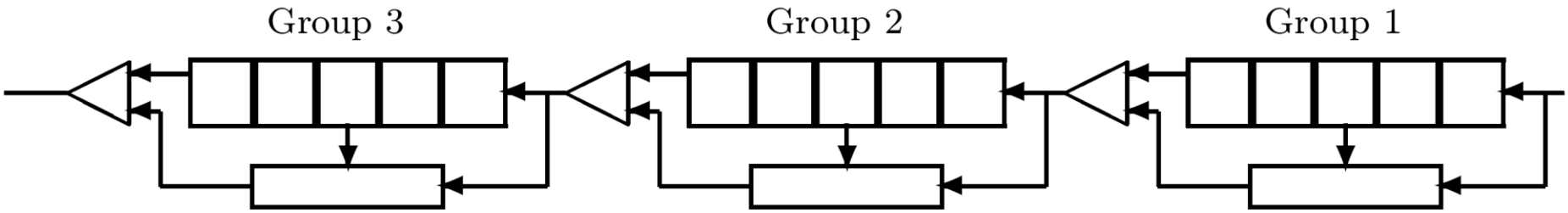
Carry Skip Adder - Structure

- Group_l_Carry-out = Gi:j + Pi:j Group_l_Carry-in
- Gi:j = 1 when a carry is generated internal to group and allowed to propagate through all remaining bit positions including i
- Pi:j = 1 when k=i-j+1 bit positions allow incoming carry cj to propagate to next position i+1
- Buffers realize the OR operation



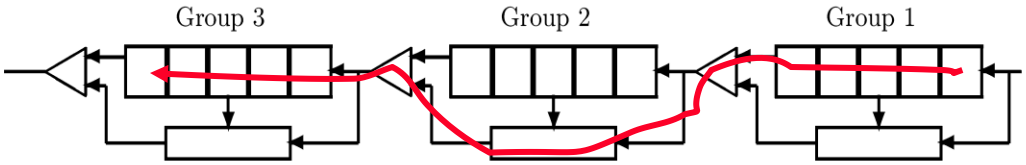
Example - 15-bit carry-skip adder

- Consisting of 3 groups of size 5 each
- $P_{i:j}$ for all groups can be generated simultaneously allowing a fast skip of groups which satisfy $P_{i:j}=1$



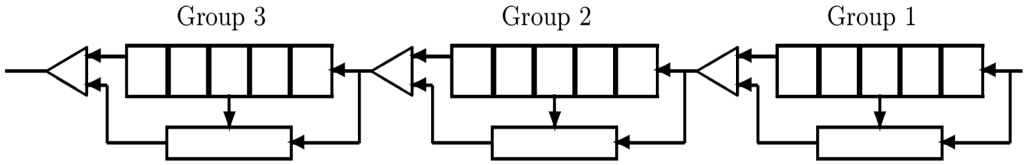
Determining Optimal Group Size k

- **Assumption**: Groups have equal size k - n/k integer
- k selected to minimize time for longest carry-propagation chain
- **Notations**:
  - tr - carry-ripple time through a single stage
  - ts(k) - time to skip a group of size k (for most implementations - independent of k)
  - tb - delay of buffer (implements OR) between two groups
  - Tcarry - overall carry-propagation time - occurs when a carry is generated in stage 0 and propagates to stage n-1
- Carry will ripple through stages 1,2, … ,k-1 within group 1, skip groups 2,3, … , (n/k-1), then ripple through group n/k



Determining Optimal k - Cont.

- $T_{carry}=(k-1)t_r+t_b+(n/k-2)(t_s+t_b)+(k-1)t_r$
- Example - two-level gate implementation used for ripple-carry and carry-skip circuits
  - $t_r = t_s+t_b=2\Delta_G$
  - $T_{carry}=(4k+2n/k-7)\ \Delta_G$



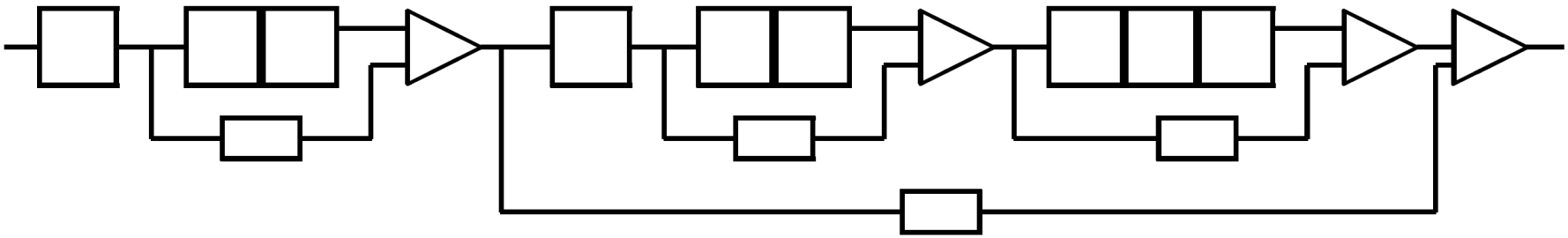
- Differentiating T_{carry} with respect to k and equating to 0 -

$$k_{opt} = \sqrt{n/2}$$

- Group size and carry propagation time proportional to $\sqrt{n}$ - same as for carry-select adder
- Example: $n=32$, 8 groups of size $k_{opt} = 4$ is best
- $T_{opt}=25\Delta_G$ instead of $62\Delta_G$ for ripple-carry adder

Further Speedup

- Size of first and last groups smaller than fixed size k ⇒ ripple-carry delay through these is reduced
- Size of center groups increased - since skip time is usually independent of group size
- Another approach: add second level to allow skipping two or more groups in one step (more levels possible)
- Algorithms exist for deriving optimal group sizes for different technologies and implementations (i.e., different values of ratio (ts+tb)/tr)



Variable-Size Groups

- Unlike equal-sized group case - cannot restrict to analysis of worst case for carry propagation
- This may lead to trivial conclusion: first and last groups consisting of a single stage - remaining n-2 stages constituting a single center group
- Carry generated at the beginning of center group may ripple through all other n-3 stages - becoming the worst case
- Must consider all possible carry chains starting at arbitrary bit position a (with $x_a=y_a$) and stopping at b ($x_b=y_b$) where a new carry chain (independent of previous) may start

Optimizing Different Size Groups

- $k_1, k_2, \ldots, k_L$ - sizes of L groups $\sum_{i=1}^{L} k_i = n$
- General case: Chain starts within group u, ends within group v, skips groups u+1, u+2, … ,v-1
- Worst case - carry generated in first position within u and stops in last position within v
- Overall carry-propagation time is

$$T_{carry}(u,v) = (k_u - 1)\cdot t_r + t_b + \sum_{l=u+1}^{v-1} (t_s(k_l) + t_b) + (k_v - 1)\cdot t_r$$

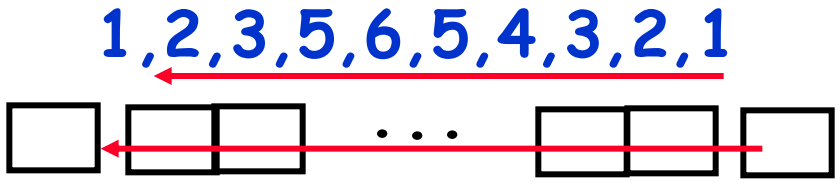
- Number of groups L and sizes $k_1, k_2, \ldots, k_L$ selected so that longest carry-propagation chain is minimized -

$$minimize\left[\max_{1\le u\le v\le L} T_{carry}(u,v)\right]$$

- Solution algorithms developed - geometrical interpretations or dynamic programming

Optimization - Example

- 32-bit adder with single level carry-skip
- $t_s + t_b = t_r$
- Optimal organization: L=10 groups with sizes $k_1, k_2, \ldots, k_{10} = 1,2,3,4,5,6,5,3,2,1$
- Resulting in $T_{carry} \leq 9\ t_r$
- If $t_r = 2\ \Delta_G$ - $T_{carry} \leq 18\ \Delta_G$ instead of $25\ \Delta_G$ in equal-size group case
- Exercise: Show that any two bit positions in any two groups u and v ($1 \leq u \leq v \leq 10$) satisfy $T_{carry}(u,v) \leq 9\ t_r$



Carry-skip vs. Carry-select Adder

- Strategies behind two schemes sound different
- Equations relating group-carry-out with group-carry-in are variations of same basic equation
- Both have execution time proportional to $\sqrt{n}$
- Only details of implementation vary, in particular calculation of sum bits
- Even this difference is reduced when the multiplexing circuitry is merged into summation logic

Hybrid Adders

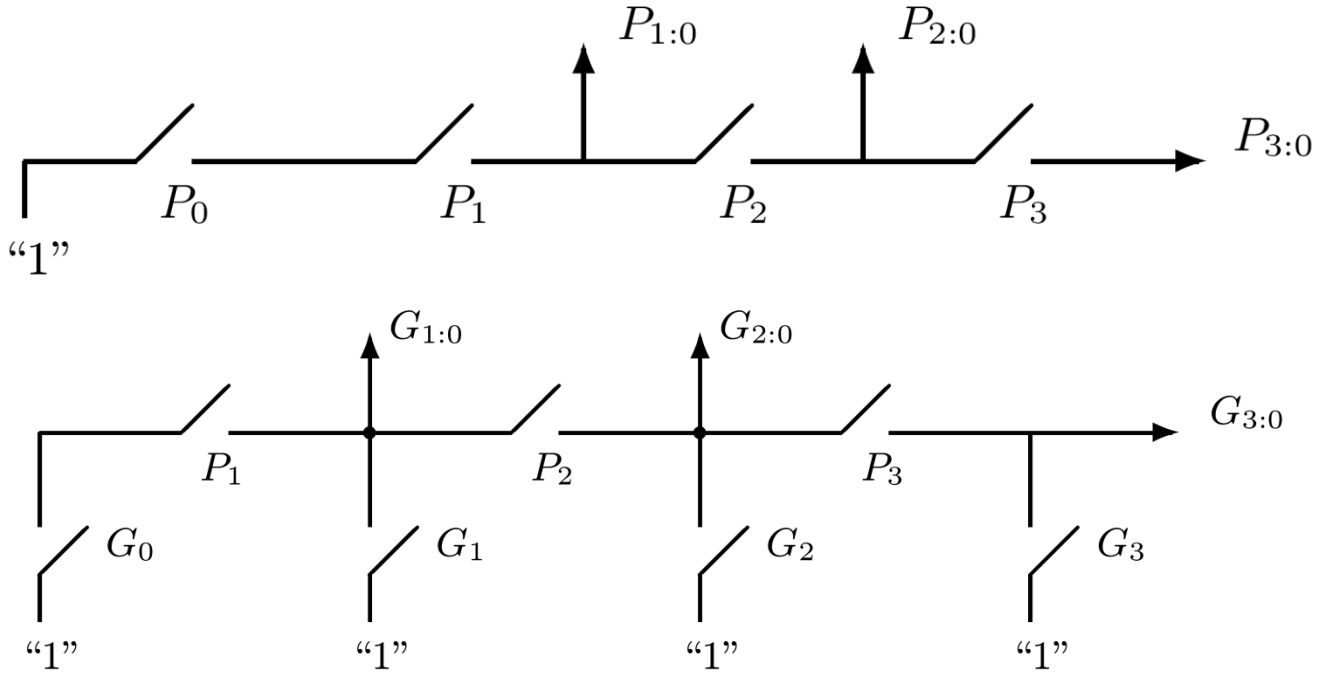
- Combination of two or more addition methods
- Common approach: one method for carry,
  another for sum
- Two hybrid adders combining variation of
  - a carry-select for sum
  - modified Manchester carry for carry
- Both divide operands into equal groups - 8 bits each
  - First - uses carry-select for sum for each group of 8 bits separately
  - Second - uses a variant of conditional-sum
- Group carry-in signal into 8-bit groups, which selects one out of two sets of sum bits, is generated by a carry-look-ahead tree
- 64-bit adder - carries are $C_8, C_{16}, C_{24}, C_{32}, C_{40}, C_{48}, C_{56}$

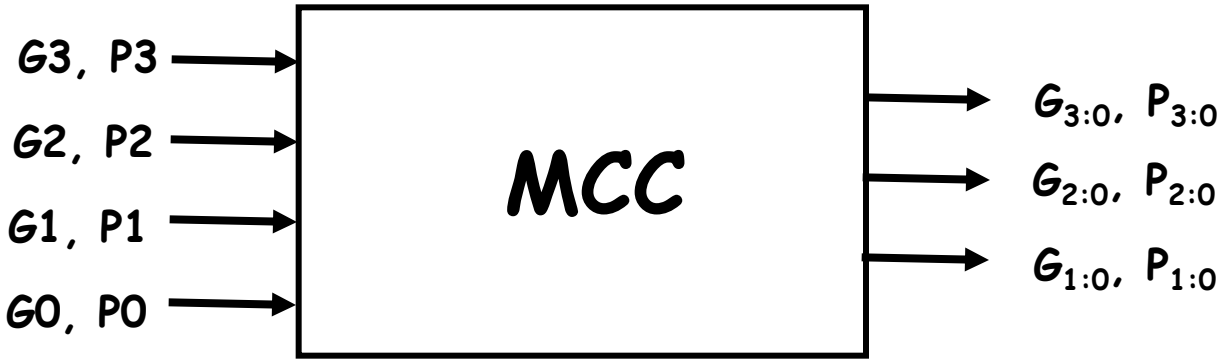
Blocking Factor in Carry Tree

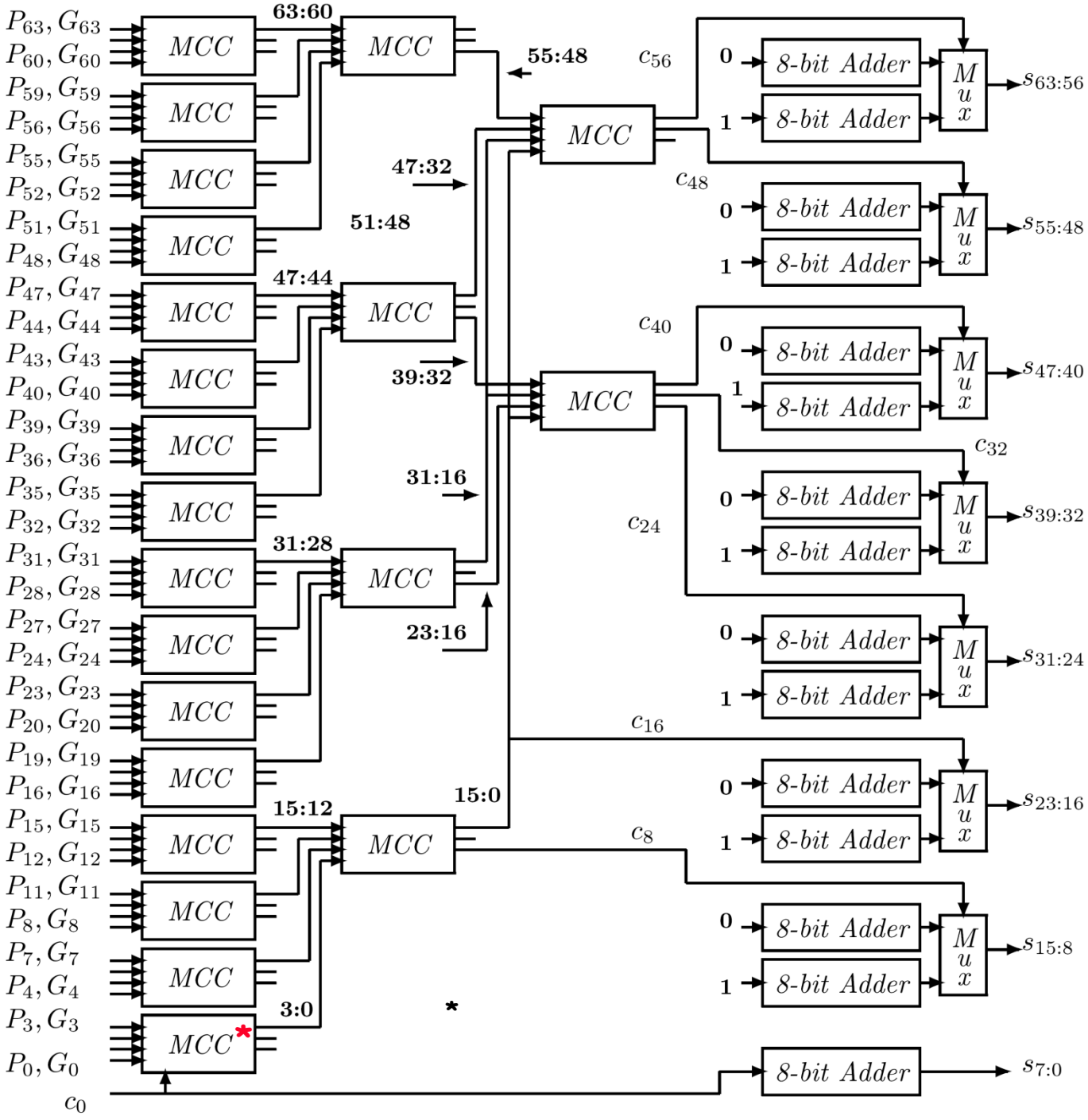
- Structure of carry-look-ahead tree for generating carries similar to those seen before
- Differences - variations in blocking factor at each level and exact implementation of fundamental carry operator
- Restricting to a fixed blocking factor - natural choices include 2, 4 or 8
  - 2 - largest number of levels in tree, vs.
  - 8 - complex modules for fundamental carry operator with high delay
- Factor of 4 - a reasonable compromise
- A Manchester carry chain (MCC), which generates both carry generate/propagate bits, with a blocking factor of 4

$$c_m = G_{m-1:0} + P_{m-1:0} \cdot c_0$$

Manchester Carry Module







64-bit Hybrid Adder

55:48 - 55:0 ⇒ C56
47:32 - 47:0 ⇒ C48
31:16 - 31:0
15:0 - x

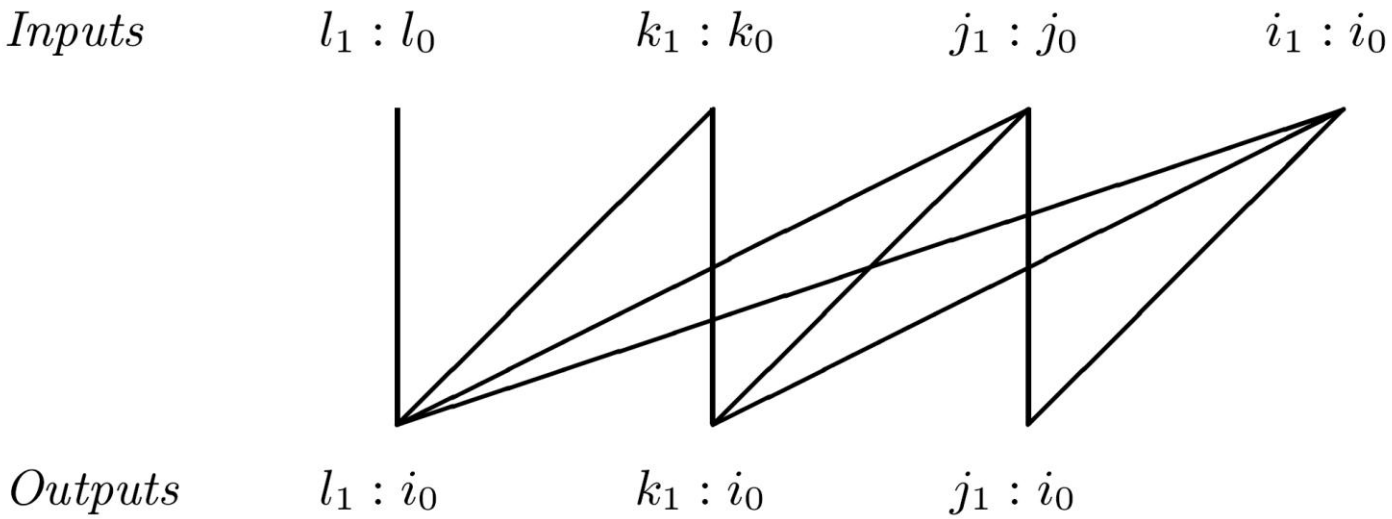
38:32 - 39:0 ⇒ C40
31:16 - 31:0 ⇒ C32
23:16 - 23:0 ⇒ C24
15:0 - x

15:12 - 15:0 ⇒ C16
11:8 - 11:0
7:4 - 7:0 ⇒ C8
3:0 - x

Koren Chap.5.66

MCC - General Case

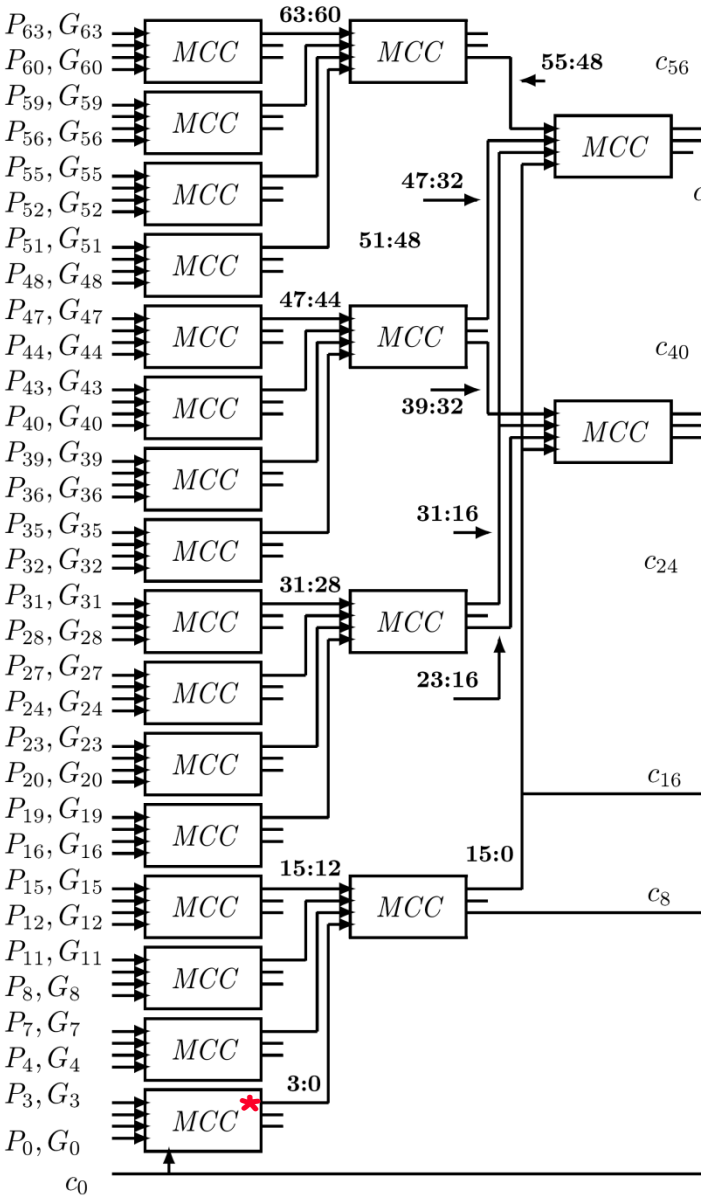
- MCC accepts 4 pairs of inputs:
- $(P_{i1:i0}, G_{i1:i0}), (P_{j1:j0}, G_{j1:j0}), (P_{k1:k0}, G_{k1:k0}), (P_{l1:l0}, G_{l1:l0})$
- where $i_1 \geq i_0$, $j_1 \geq j_0$, $k_1 \geq k_0$, $l_1 \geq l_0$
- Produces 3 pairs of outputs:
- $(P_{j1:i0}, G_{j1:i0}), (P_{k1:i0}, G_{k1:i0}), (P_{l1:i0}, G_{l1:i0})$
- where $i_1 \geq j_0-1$, $j_1 \geq k_0-1$, $k_1 \geq l_0-1$
- Allows overlap among input subgroups



Carry Tree

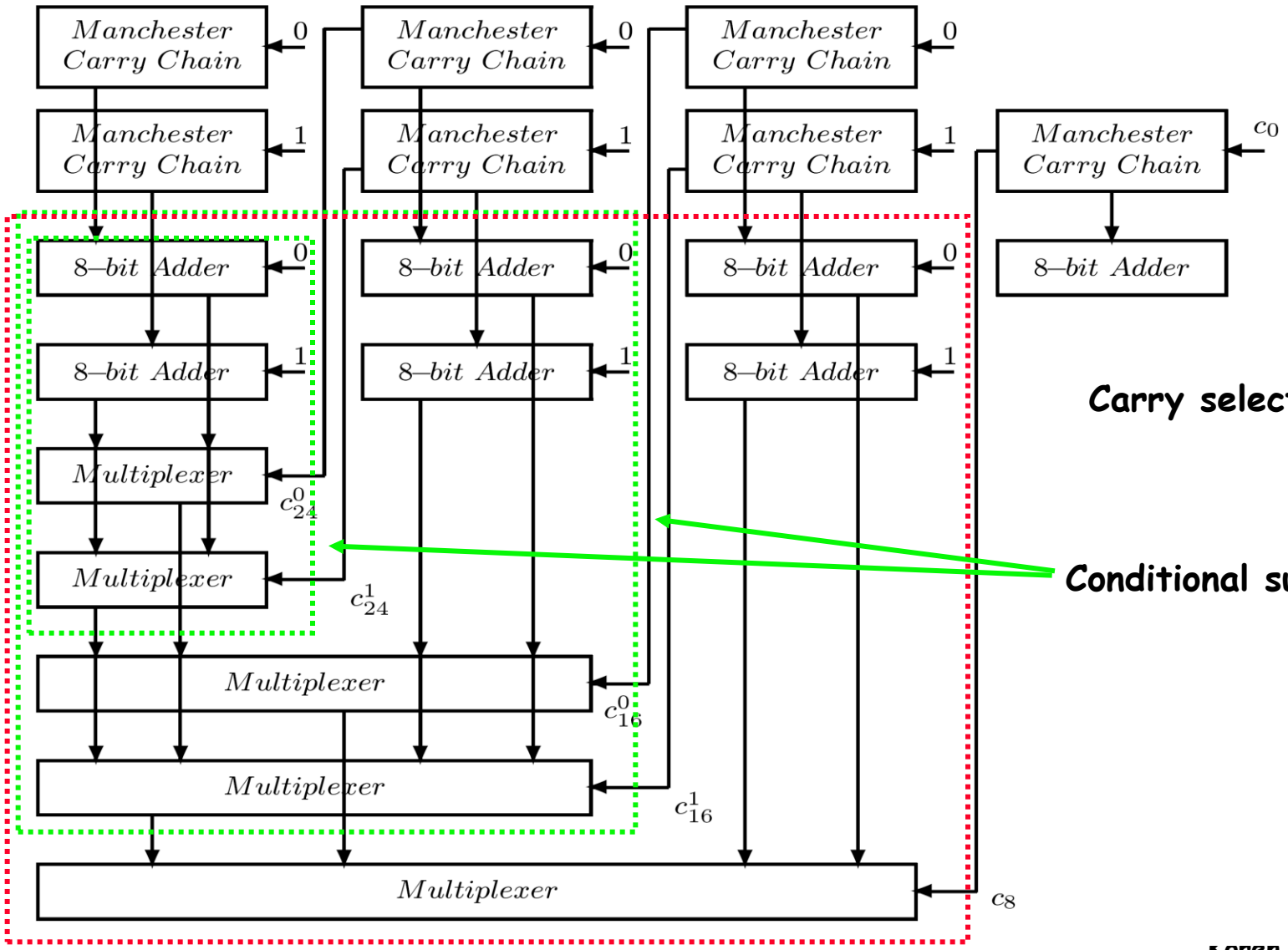
- First level - 14 MCCs calculating (P3:0,G3:0),…,(P55:52,G55:52)
  - * only outputs P3:0 and G3:0 are utilized
- Second level: each MCC generates 2 pairs (P3:0, G3:0),(P1:0, G1:0)
- Providing (P7:0,G7:0),(P15:0,G15:0), (P23:16,G23:16),(P31:16,G31:16), (P39:32,G39:32),(P47:32,G47:32), (P55:48,G55:48)
- Generates C8 & C16 - G7:0 & G15:0
- C0 is incorporated into MCC*

for (P3:0, G3:0)



Koren Chap.5.68

A Schematic Diagram of a 32-bit Hybrid Adder

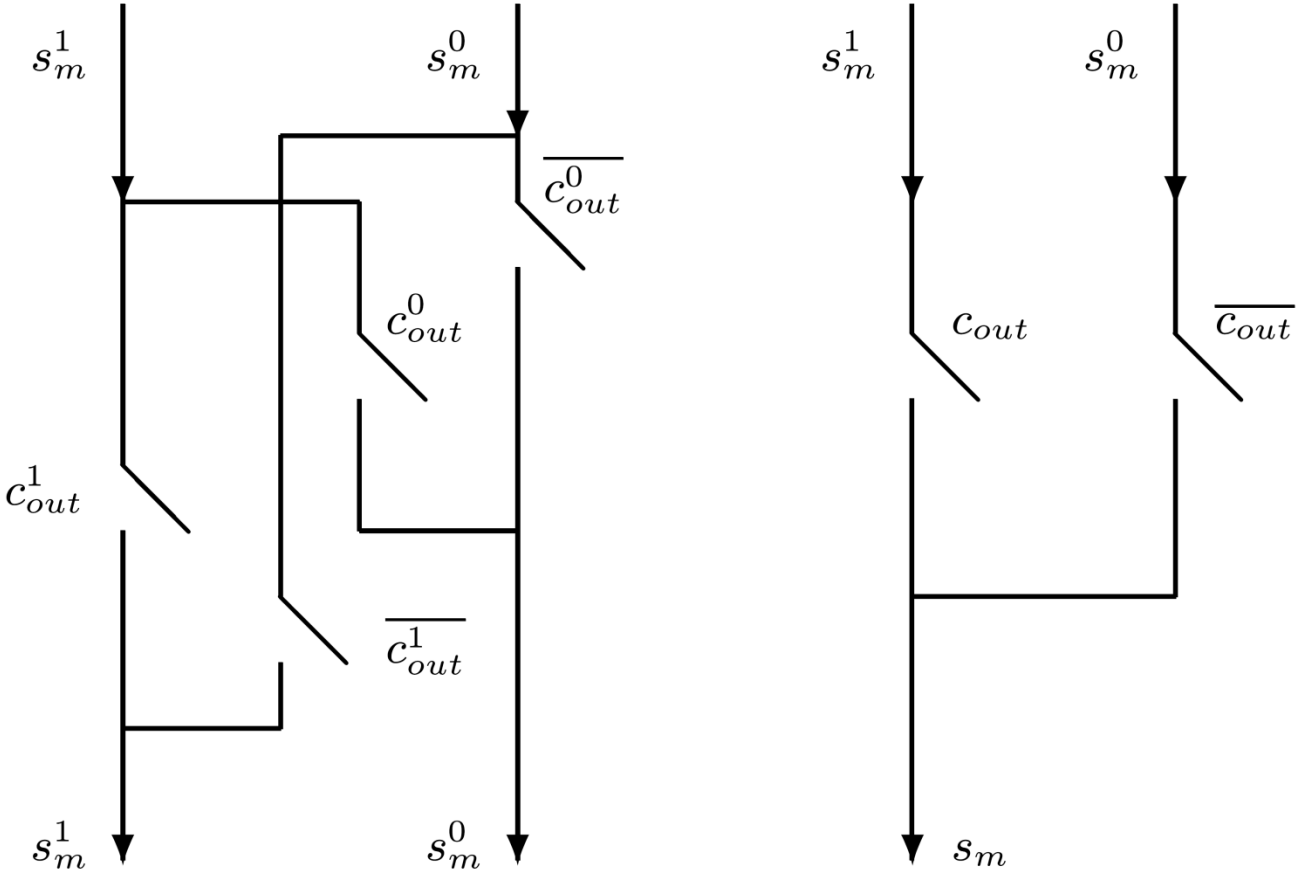


Grouping of Bits in a 64-bit Adder

- 64 bits divided into two sets of 32 bits, each set further divided into 4 groups of 8 bits
- For every group of 8 bits - 2 sets of conditional sum outputs generated separately
- Two most significant groups combined into group of size 16
- Further combined with next group of 8 to form group of 24 bits and so on
  - principle of conditional-sum addition
  - However, the way input carries for basic 8-bit groups are generated is differently with MCC
- MCC generates P_m, G_m and K_m and C_{out}^0 , C_{out}^1 for assumed incoming carries of 0 and 1
- Conditional carry-out signals control multiplexers

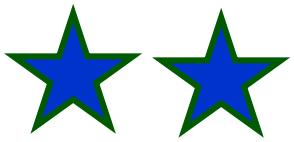
Dual and Regular Multiplexer

- Two sets of dual multiplexers (of size 8 and 16)
- Single regular multiplexer of size 24



High-Order Half of 64-bit Adder

- Similar structure but incoming carry C_{32} calculated by separate carry-look-ahead circuit
- Inputs are conditional carry-out signals generated by 4 MCCs
- Allows operation of high-order half to overlap operation of low-order half
- Summary: combines variants of 3 different techniques for fast addition: Manchester carry generation, carry-select, conditional-sum
- Other designs of hybrid adders exist - e.g., groups with unequal number of bits
- "Optimality" of hybrid adders depends on technology and delay parameters



Carry-Save Adders (CSAs)

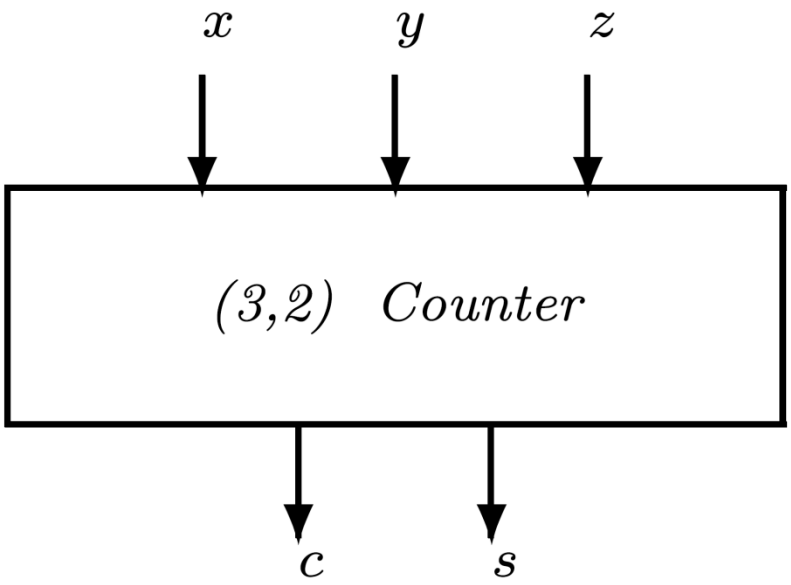
- 3 or more operands added simultaneously (e.g., in multiplication) using 2-operand adders
- Time-consuming carry-propagation must be repeated several times: k operands - k-1 propagations
- Techniques for lowering this penalty exist - most commonly used - carry-save addition
- Carry propagates only in last step - other steps generate partial sum and sequence of carries
- Basic CSA accepts 3 n-bit operands; generates 2 n-bit results: n-bit partial sum, n-bit carry
- Second CSA accepts the 2 sequences and another input operand, generates new partial sum and carry
- CSA reduces number of operands to be added from 3 to 2 without carry propagation

Implementing Carry Save Adders

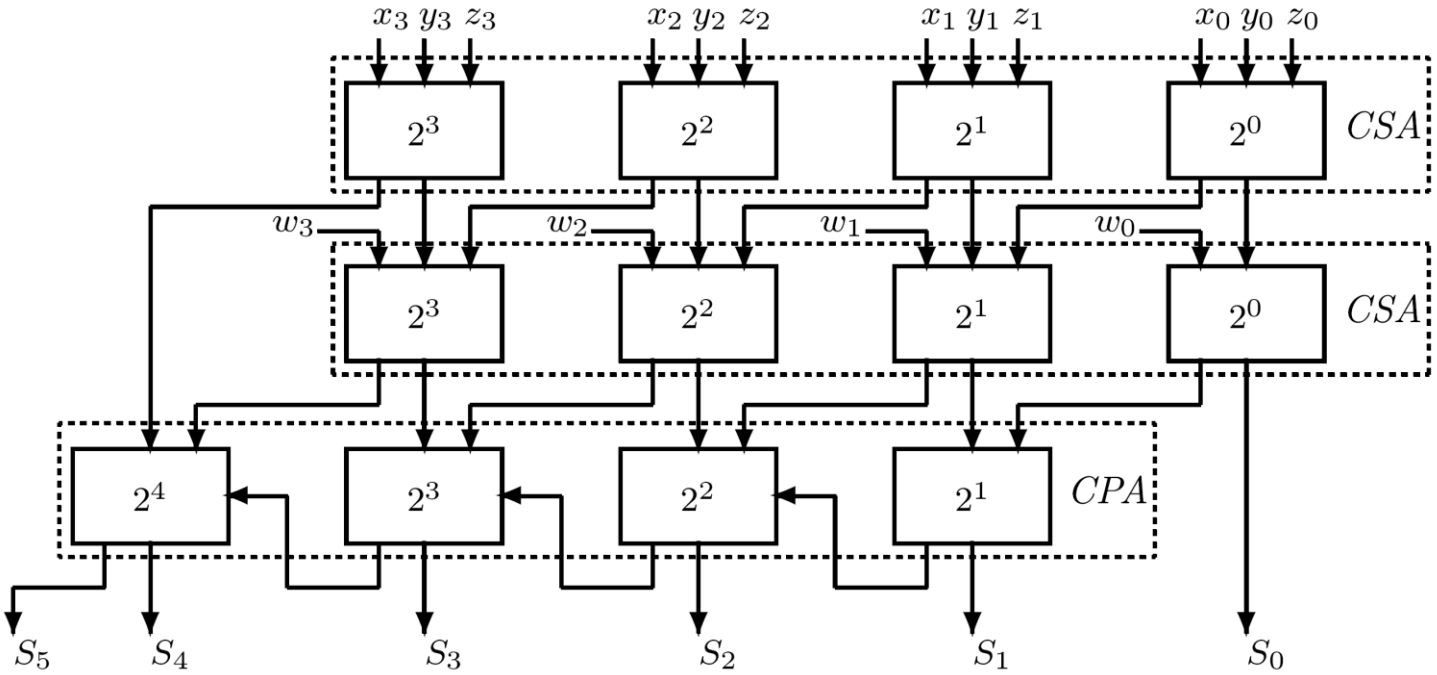
- Simplest implementation - full adder (FA) with 3 inputs x,y,z
- x+y+z=2c+s (s,c - sum and carry outputs)

$$s = (x + y + z) \; mod \; 2 \qquad \text{and} \qquad c = \frac{(x + y + z) - s}{2}.$$

- Outputs - weighted binary representation of number of 1's in inputs
- FA called a (3,2) counter
- n-bit CSA: n (3,2) counters in parallel with no carry links

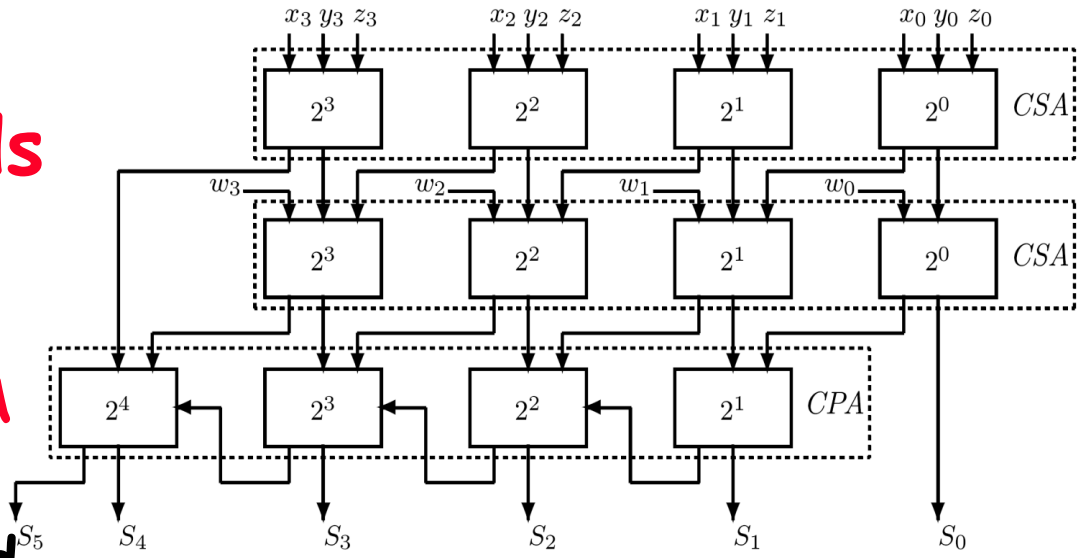


Carry-Save Adder for four 4-bit Operands



* Upper 2 levels - 4-bit CSAs
* 3rd level - 4-bit carry-propagating adder (CPA)
* Ripple-carry adder - can be replaced by a carry-look-ahead adder or any other fast CPA
* Partial sum bits and carry bits interconnected to guarantee that only bits having same weight are added by any (3,2) counter

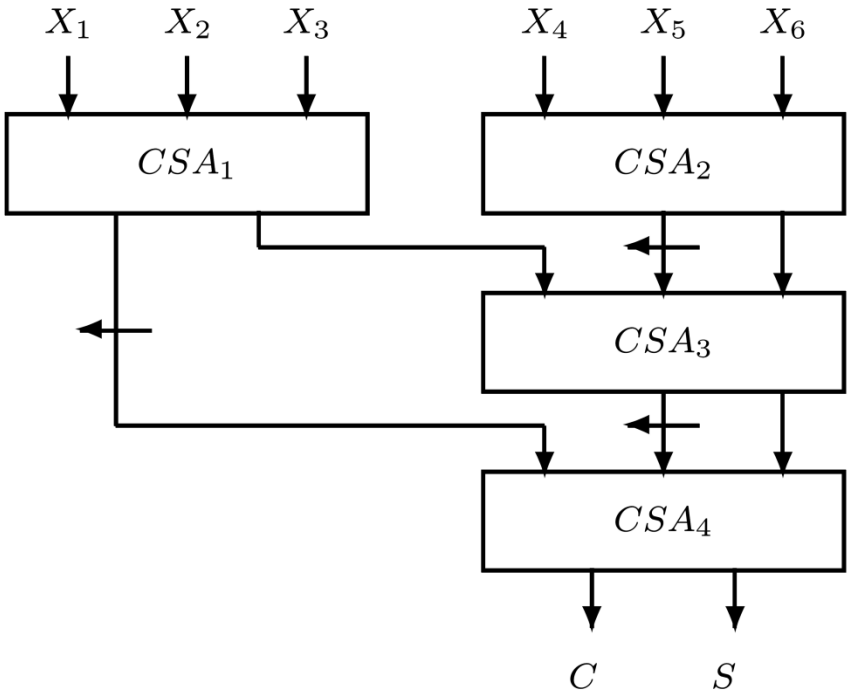
Adding k Operands

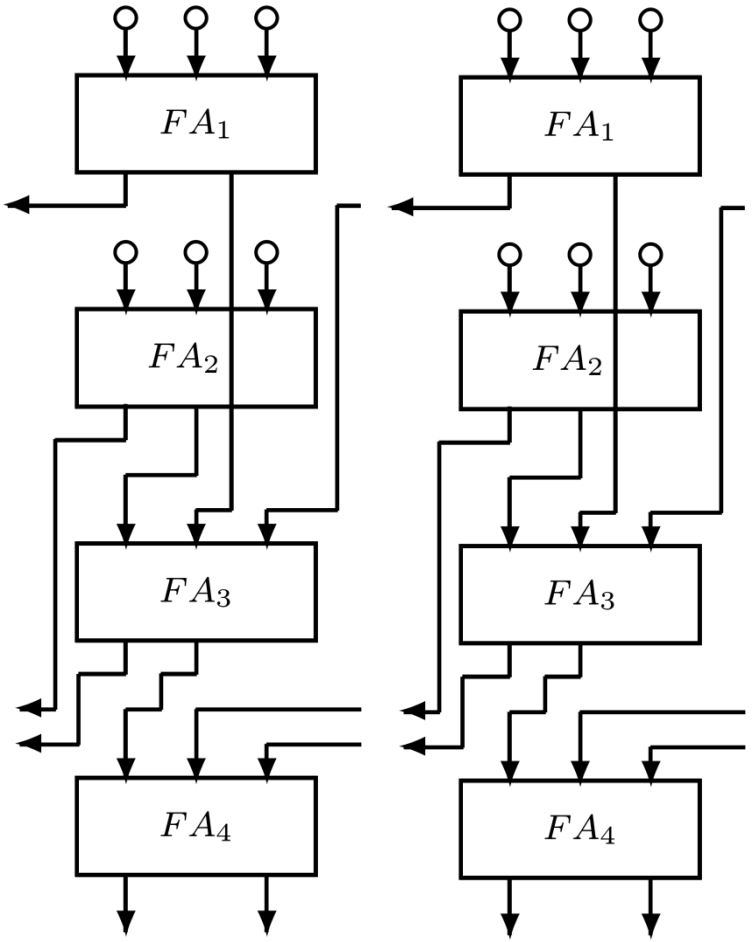


- (k-2) CSAs + one CPA
- If CSAs arranged in cascade - time to add k operands is $(k-2)T_{CSA} + T_{CPA}$
- T_{CPA} ; T_{CSA} - operation time of CPA ; CSA
- ΔG ; ΔFA delay of a single gate ; full adder
- $T_{CSA} = \Delta FA \geq 2\ \Delta G$
- Sum of k operands of size n bits each can be as large as $k(2^n-1)$
- Final addition result may reach a length of $n+\lceil \log_2 k \rceil$ bits

Six-operand Wallace Tree

- Better organization for CSAs - faster operation time





Number of Levels in Wallace Tree

- Number of operands reduced by a factor of 2/3 at each level - $k \cdot (\frac{2}{3})^l \leq 2$ (l - number of levels)
- Consequently, l = Number of levels $\approx \frac{log\ (k/2)}{log\ (3/2)}$
- Only an estimate of l - number of operands at each level must be an integer
- N_i - number of operands at level i
- N_{i+1} - at most $\lfloor 3/2\ N_i \rfloor$ ($\lfloor x \rfloor$ - largest integer smaller than or equal to x)
- Bottom level (0) has 2 - maximum at level 1 is 3 - maximum at level 2 is $\lfloor 9/2 \rfloor = 4$
- Resulting sequence: 2,3,4,6,9,13,19,28,…
- For 5 operands - still 3 levels

Number of Levels in a CSA Tree for k operands

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

- Example: k=12 - 5 levels - delay of 5T_{CSA} instead of 10T_{CSA} in a linear cascade of 10 CSAs

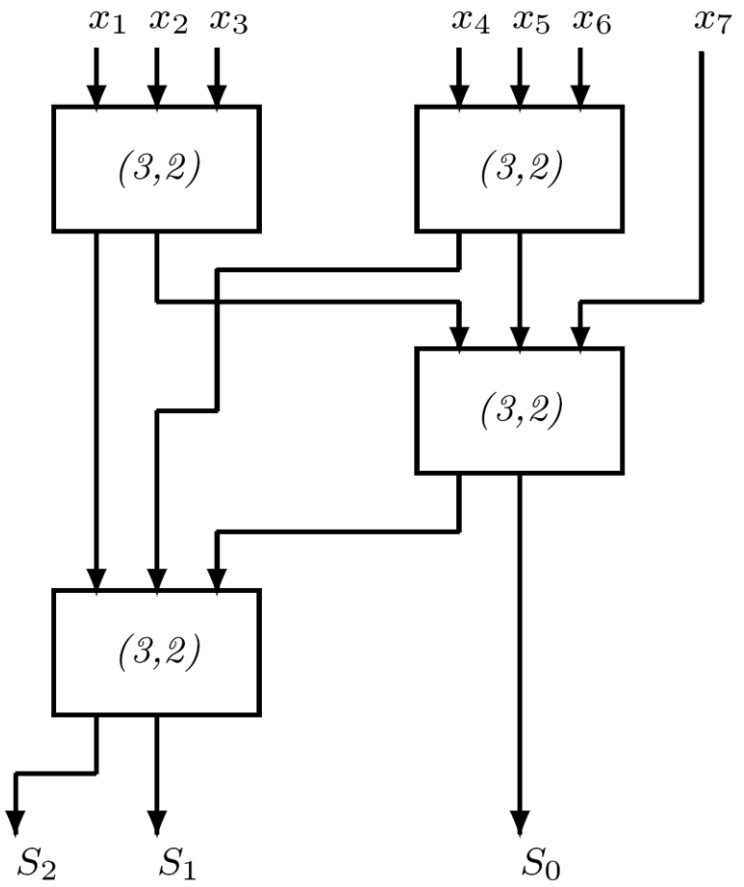
Most Economical Implementation (Fewer CSAs)

- Achieved when number of operands is element of 3,4,6,9,13,19,28,…
- If given number of operands, k, not in sequence - use only enough CSAs to reduce k to closest (smaller than k) element
- Example: k=27, use 8 CSAs (24 inputs) rather than 9, in top level - number of operands in next level is 8×2+3=19
- Remaining part of tree will follow the series

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

(7,3) and Other Counters

- (7,3) counter: 3 outputs - represent number of 1's in 7 inputs
- Another example: (15,4) counter
- In general: (k,m) counter - k and m satisfy $2^m - 1 \geq k$ or $m \geq \lfloor \log_2 (k+1) \rfloor$
- (7,3) counter using (3,2) counters:
- Requires 4 (3,2)'s in 3 levels - no speed-up



(7,3) Counters

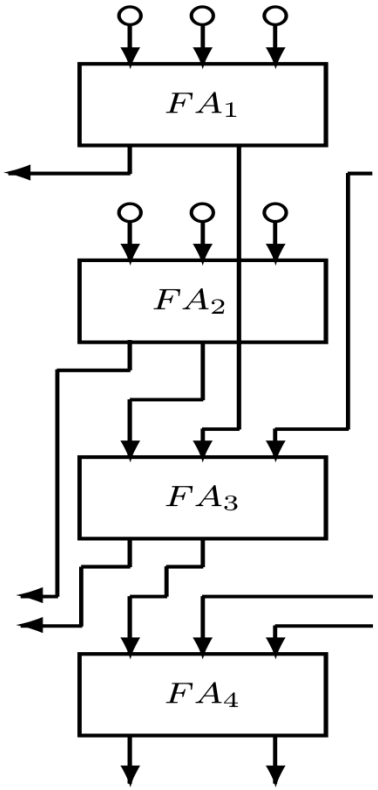
- (7,3) can be implemented as a multilevel circuit - may have smaller delay
- Number of interconnections affects silicon area - (7,3) preferrable to (3,2)
  - (7,3) has 10 connections and removes 4 bits
  - (3,2) has 5 connections and removes only 1 bit
- Another implementation of (7,3) - ROM of size $2^7 \times 3 = 128 \times 3$ bits
- Access time of ROM unlikely to be small enough
- Speed-up may be achieved for ROM implementation of (k,m) counter with higher values of k

Avoiding Second Level of Counters

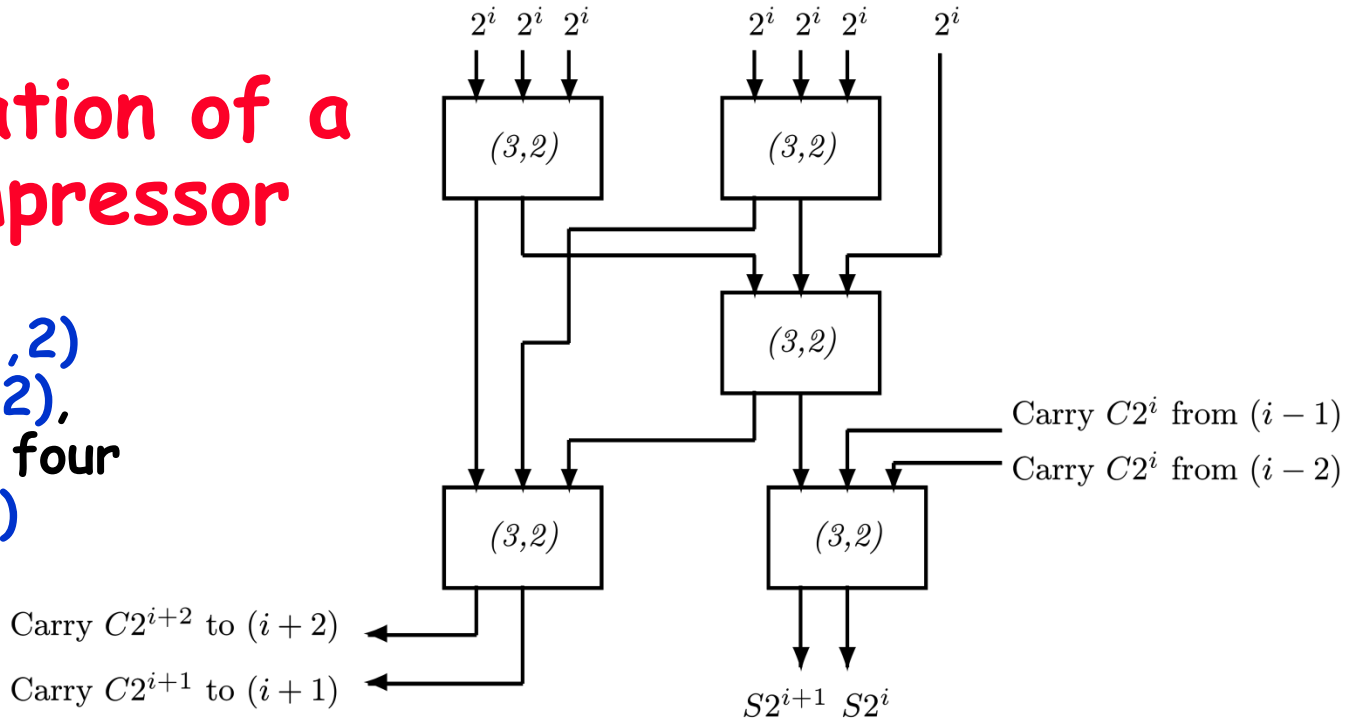
- Several (7,3) counters (in parallel) are used to add 7 operands - 3 results obtained
- Second level of (3,2) counters needed to reduce the 3 to 2 results (sum and carry) added by a CPA
- Similarly - when (15,4) or more complex counters are used - more than two results generated
- In some cases - additional level of counters can be combined with first level - more convenient implementation
- When combining a (7,3) counter with a (3,2) counter - combined counter called a (7;2) compressor

(k;m) Compressor

- Variant of a counter with k primary inputs, all of weight 2^i , and m primary outputs of weights $2^i, 2^{i+1}, \ldots, 2^{i+m-1}$
- Compressor has several incoming carries of weight 2^i from previous compressors, and several outgoing carries of weights 2^{i+1} and up
- Trivial example of a (6;2) compressor:
- All outgoing carries have weight 2^{i+1}
- Number of outgoing carries = number of incoming carries = k-3 (in general)



Implementation of a (7;2) Compressor



* Bottom right (3,2) - additional (3,2), while remaining four - ordinary (7,3) counter

* 7 primary inputs of weight 2^i and 2 carry inputs from columns i-1 and i-2
* 2 primary outputs, $S2^i$ and $S2^{i+1}$, and 2 outgoing carries $C2^{i+1}$, $C2^{i+2}$, to columns i+1 and i+2
* Input carries do not participate in generation of output carries - avoids slow carry-propagation
* Not a (9,4) counter - 2 outputs with same weight
* Above implementation does not offer any speedup
* Multilevel implementation may yield smaller delay as long as outgoing carries remain independent of incoming carries

multiple-column counters

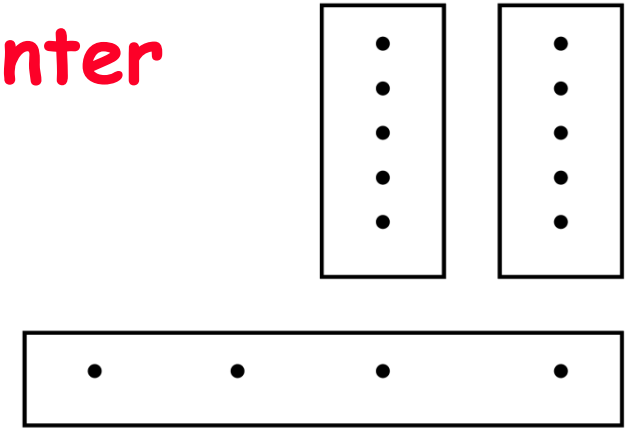
- Generalized parallel counter: add l input columns and produce m-bit output - $(k_{l-1}, k_{l-2}, \ldots, k_0, m)$
- k_i - number of input bits in i-th column with weight 2^i
- (k,m) counter - a special case
- Number of outputs m must satisfy

$$2^m - 1 \geq \sum_{i=0}^{l-1} k_i 2^i$$

- If all l columns have same height k - $(k_0 = k_1 = \ldots = k_{l-1} = k)$ -

  $2^m - 1 \geq k(2^l - 1)$

Example - (5,5,4) Counter



- $k=5, l=2, m=4$
- $2^m - 1 = k(2^l - 1)$ - all 16 combinations of output bits are useful
- (5,5,4) counters can be used to reduce 5 operands (of any length) to 2 results that can then be added with one CPA
- Length of operands determines number of (5,5,4) counters in parallel
- Reasonable implementation - using ROMs
- For (5,5,4) - $2^{5+5}x4$ (=1024x4) ROM

Number of Results of General Counters

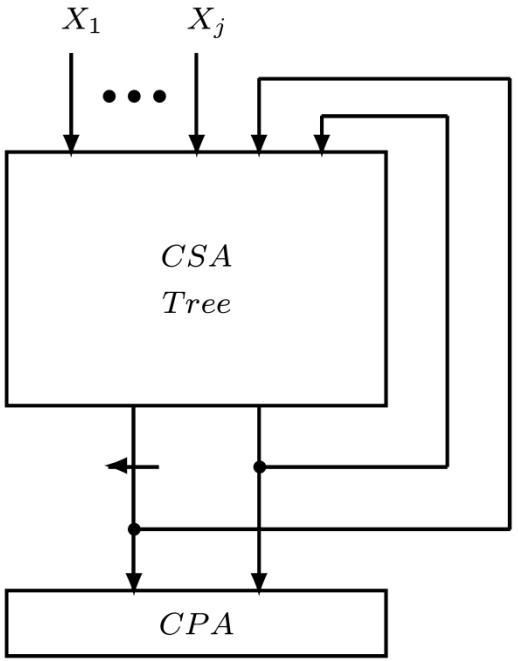
- String of (k,k,…,k,m) counters may generate more than 2 intermediate results
  - requiring additional reduction before CPA
- Number of intermediate results:
- A set of (k,k,…,k,m) counters, with l columns each, produces m-bit outputs at intervals of l bits
- Any column has at most $\lceil m/l \rceil$ output bits
- k operands can be reduced to $s = \lceil m/l \rceil$ operands
  - If s=2 - a single CPA can generate final sum
  - Otherwise, reduction from s to 2 needed

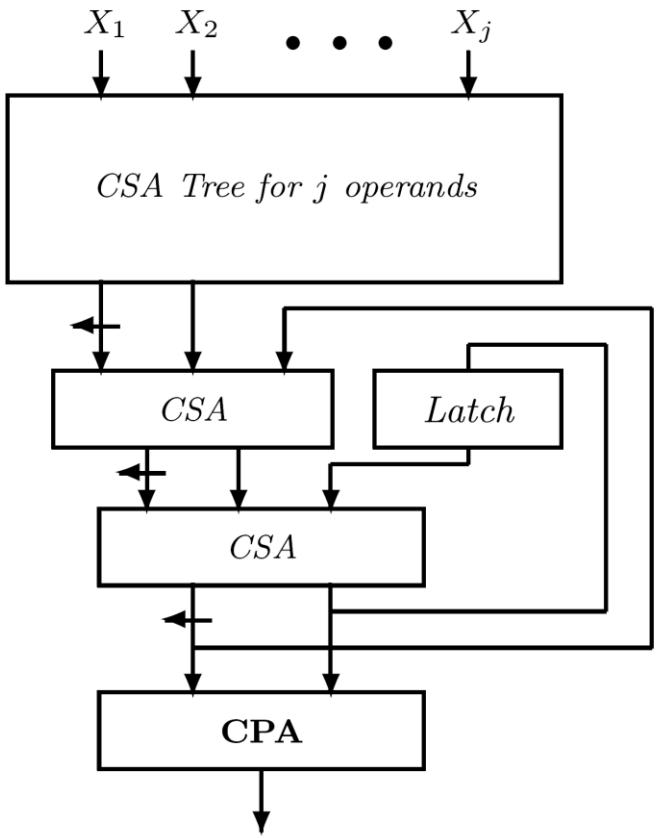
Example

- Number of bits per column in a 2-column counter (k,k,m) is increased beyond 5 - $m \geq 5$ and $s = \lceil m/2 \rceil > 2$
- For k=7, $2^m - 1 \geq 7 \times 3 = 21 \Rightarrow m=5$
- (7,7,5) counters generate s=3 operands - another set of (3,2) counters is needed to reduce number of operands to 2

Reducing Hardware Complexity of CSA Tree

- Design a smaller carry-save tree - use it iteratively
- n operands divided into ⌈n/j⌉ groups of j operands - design a tree for j+2 operands and a CPA



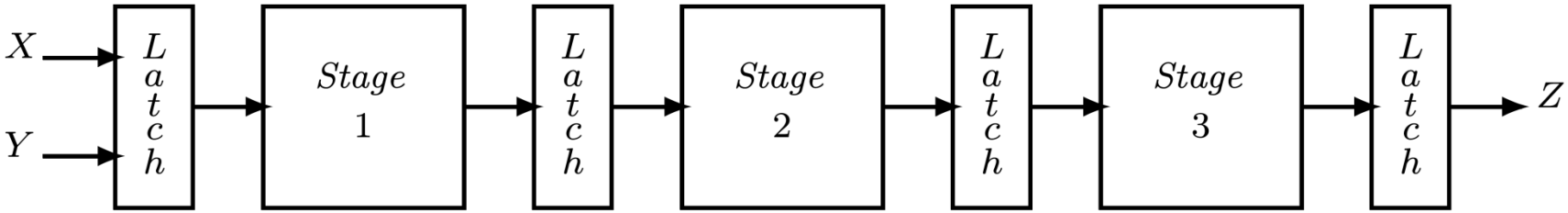


- Feedback paths - must complete first pass through CSA tree before second set of j operands is applied
- Execution slowed down - pipelining not possible

Pipelining of Arithmetic Operations

- Pipelining - well known technique for accelerating execution of successive identical operations
- Circuit partitioned into several subcircuits that can operate independently on consecutive sets of operands
- Executions of several successive operations overlap - results produced at higher rate
- Algorithm divided into several steps - a suitable circuit designed for each step
- Pipeline stages operate independently on different sets of operands
- Storage elements - latches - added between adjacent stages - when a stage works on one set of operands, preceding stage can work on next set of operands

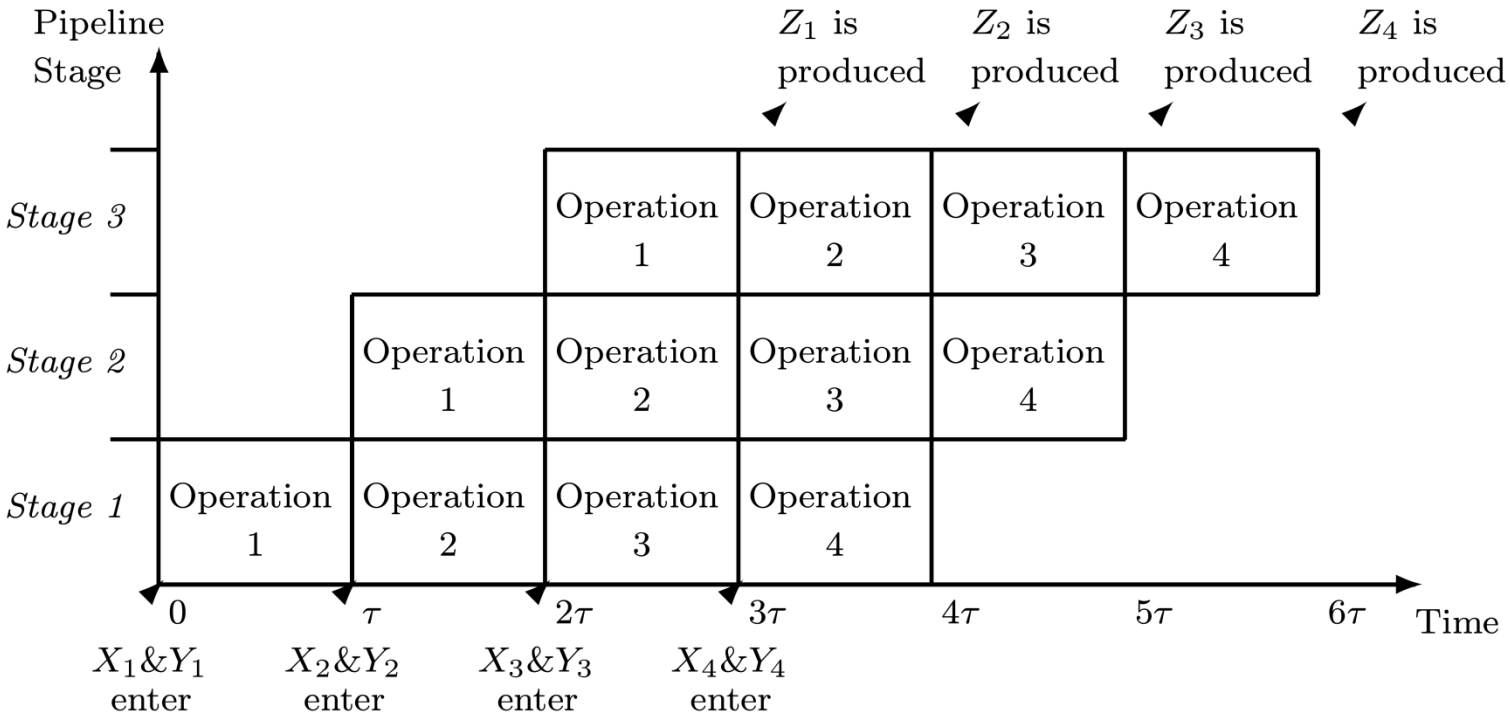
Pipelining - Example



- Addition of 2 operands X,Y performed in 3 steps
- Latches between stages 1 and 2 store intermediate results of step 1
- Used by stage 2 to execute step 2 of algorithm
- Stage 1 starts executing step 1 on next set of operands X,Y

Pipelining Timing Diagram

- 4 successive additions with operands X_1 & Y_1, X_2 & Y_2, X_3 & Y_3, X_4 & Y_4 producing results Z_1, Z_2, Z_3, Z_4



Pipeline Rate

- τ_i - execution time of stage i
- τ_l - time needed to store new data into latch
- Delays of different stages not identical - faster stages wait for slowest before switching to next task
- τ - time interval between two successive results being produced by pipeline: $\tau = \max_{1 \le i \le k} \{\tau_i\} + \tau_l$
- k - number of stages
- τ - pipeline period ; $1/\tau$ - pipeline rate or bandwidth
- Clock period $\geq \tau$
- After latency of 3τ, new results produced at rate $1/\tau$

Design Decisions

- Partitioning of given algorithm into steps to be executed by separate stages
  - Steps should have similar execution times - pipeline rate determined by slowest step
- Number of steps
  - As this number increases, pipeline period decreases, but number of latches (implementation cost) and latency go up
- Latency - time elapsed until first result produced
  - Especially important when only a single pass through pipeline required
- Tradeoff between latency and implementation cost on one hand and pipeline rate on the other hand
- Extra delay due to latches, τ_l , can be lowered by using special circuits like Earl latch

Pipelining of Two-Operand Adders

- Two-operand adders - usually not pipelined
- Pipelining justified with many successive additions
- Conditional-sum adder - easily pipelined
- $\log_2 n$ stages corresponding to $\log_2 n$ steps - execution of up to $\log_2 n$ additions can be overlapped
- Required number of latches may be excessive
- Combining several steps to one stage reduces latches' overhead and latency
- Carry-look-ahead adder cannot be pipelined - some carry signals must propagate backward
- Different designs can be pipelined - final carries and carry-propagate signals (implemented as $P_i=x_i \oplus y_i$) used to calculate sum bits - no need for feedback connections

Pipelining in Multiple-Operand Adders

- Pipelining more beneficial in multiple-operand adders like carry-save adders
- Modifying implementation of CSA trees to form a pipeline is straightforward - requires only addition of latches
- Can be added at each level of tree if maximum bandwidth is desired
- Or - two (or more) levels of tree can be combined to form a single stage, reducing overall number of latches and pipeline latency

Partial Tree

- Reduced hardware complexity of CSA tree - partial tree
- Two feedback connections prevent pipelining
- Modification - intermediate results of CSA tree connected to bottom level of tree
- Smaller tree with j inputs, 2 separate CSAs, and a set of latches at the bottom
- CSAs and latches form a pipeline stage
- Top CSA tree for j operands can be pipelined too - overall time reduced