Parallel prefix adders

Kostas Vitoroulis, 2006.
Presented to Dr. A. J. Al-Khalili.
Concordia University.

Overview of presentation

- Parallel prefix operations
- Binary addition as a parallel prefix operation
- Prefix graphs
- Adder topologies
- Summary

Parallel Prefix Operation

Terminology background:

- Prefix: The outcome of the operation depends on the initial inputs.
- Parallel: Involves the execution of an operation in parallel. This is done by segmentation into smaller pieces that are computed in parallel.
- Operation: Any arbitrary primitive operator " $\circ$ " that is associative is parallelizable
    - it is fast because the processing is accomplished in a parallel fashion.

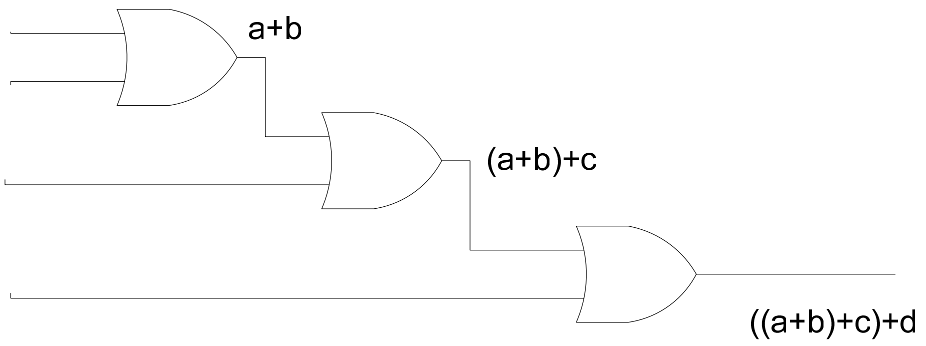
Example: Associative operations are parallelizable

Consider the logical OR operation: a + b

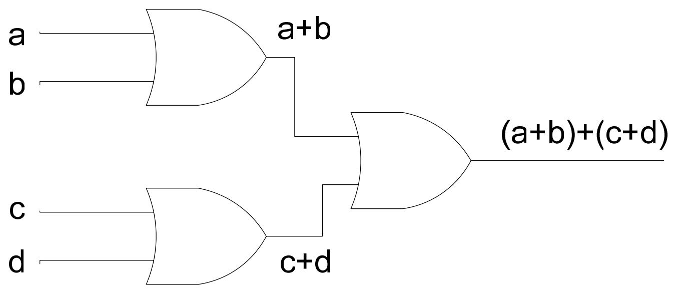
The operation is associative:

a + b + c + d = (((a + b) + c) + d) = ((a + b) + (c + d))

Serial implementation:



Parallel implementation:



Mathematical Formulation: Prefix Sum

- Operator: " $\circ$ "

← this is the unary operator known as "scan" or "prefix sum"

- Input is a vector:

  $A = A_n A_{n-1} \ldots A_1$

- Output is another vector:

  $B = B_n B_{n-1} \ldots B_1$

  where

  $B_1 = A_1$

  $B_2 = A_1 \circ A_2$

  …

  $B_n = A_1 \circ A_2 \ldots \circ A_n$

← B_n represents the operator being applied to all terms of the vector.

Example of prefix sum

Consider the vector: **$A = A_nA_{n-1} \dots A_1$** where element A_i is an integer

The "*" unary operator, defined as:

$$*A = B$$

With

$B = B_nB_{n-1} \dots B_1$

$B_1 = A_1$

$B_2 = A_1 * A_2$

$B_3 = A_1 * A_1 * A_3$

…

and ' * ' here is the integer addition operation.

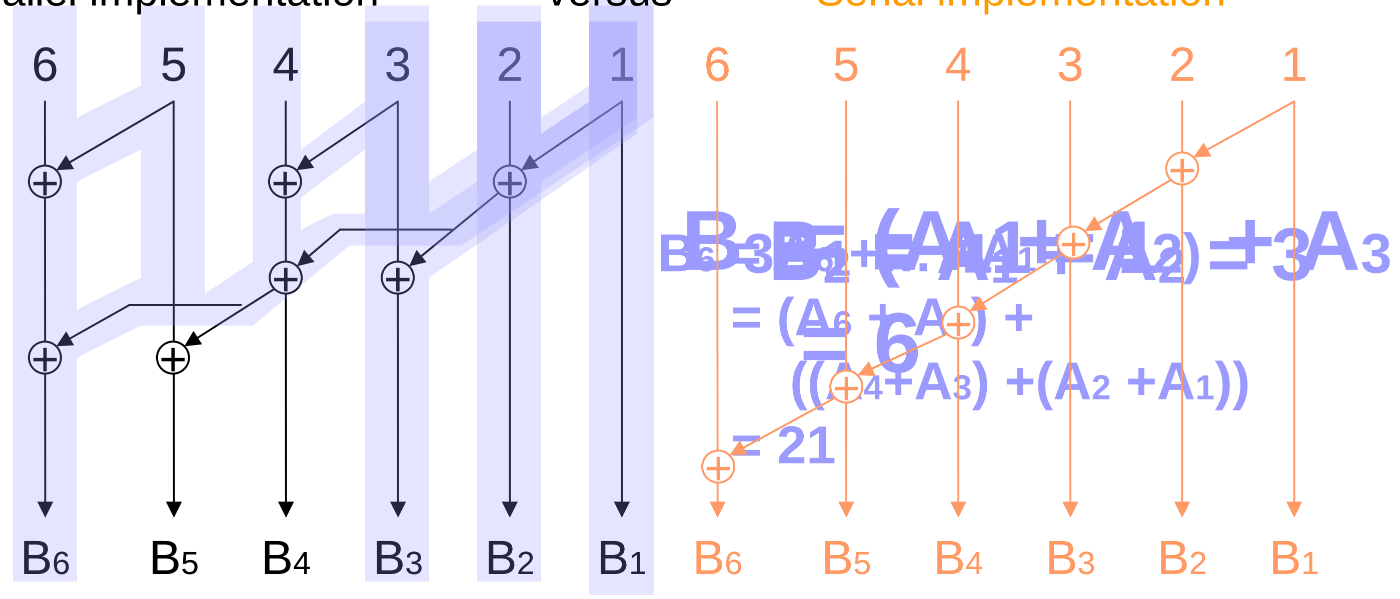
Example of prefix sum

Calculation of *A, where A = 6 5 4 3 2 1 yields:

B = *A = 21 15 10 6 3 1

Because the summation is associative the calculation can be done in parallel in the following manner:

Parallel implementation versus Serial implementation



Binary Addition

$$\begin{array}{cccccc} & & {}^{c_3}\mathbf{x}_3 & {}^{c_2}\mathbf{x}_2 & {}^{c_1}\mathbf{x}_1 & {}^{c_0}\mathbf{x}_0 \\ + & & & & & \\ & & \mathbf{y}_3 & \mathbf{y}_2 & \mathbf{y}_1 & \mathbf{y}_0 \\ \hline & s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

This is the pen and paper addition of two 4-bit binary numbers **x** and **y**. **c** represents the generated carries. **s** represents the produced sum bits.

A **stage** of the addition is the set of **x** and **y** bits being used to produce the appropriate sum and carry bits. For example the highlighted bits $\mathbf{x_2}$, $\mathbf{y_2}$ constitute **stage 2** which generates carry $\mathbf{c_2}$ and sum $\mathbf{s_2}$.

Each stage *i* adds bits a_i, b_i, c_{i-1} and produces bits s_i, c_i

The following hold:

a_i	b_i	c_i	Comment:	Formal definition:	
0	0	0	The stage “kills” an incoming carry.	“Kill” bit:	$k_i = \overline{x_i + y_i}$
0	1	c_{i-1}	The stage “propagates” an incoming carry	“Propagate” bit:	$p_i = x_i \oplus y_i$
1	0	c_{i-1}	The stage “propagates” an incoming carry		
1	1	1	The stage “generates” a carry out	“Generate” bit:	$g_i = x_i \bullet y_i$

Binary Addition

a_i	b_i	c_i	Comment:	Formal definition:	
0	0	0	The stage “kills” an incoming carry.	“Kill” bit:	$k_i = \overline{x_i + y_i}$
0	1	c_{i-1}	The stage “propagates” an incoming carry	“Propagate” bit:	$p_i = x_i \oplus y_i$
1	0	c_{i-1}	The stage “propagates” an incoming carry		
1	1	1	The stage “generates” a carry out	“Generate” bit:	$g_i = x_i \bullet y_i$

The carry c_i generated by a stage *i* is given by the equation:

$$c_i = g_i + p_i \cdot c_{i-1} = x_i \cdot y_i + \left(x_i \oplus y_i\right) \cdot c_{i-1}$$

This equation can be simplified to:

$$c_i = x_i \cdot y_i + \left(x_i + y_i\right) \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

The “a_i” term in the equation being the “alive” bit.

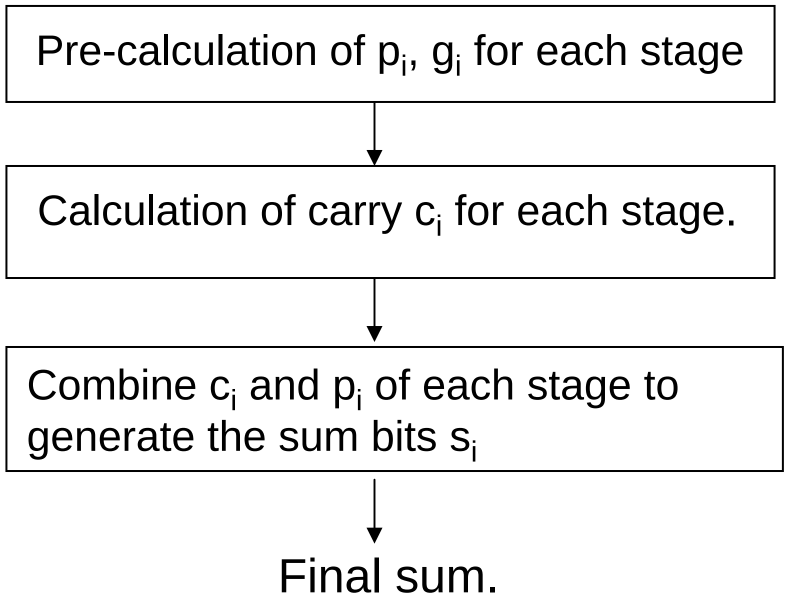
The later form of the equation uses an OR gate instead of an XOR which is a more efficient gate when implemented in CMOS technology. Note that:

$$a_i = \overline{k_i}$$

Where k_i is the “kill” bit defined in the table above.

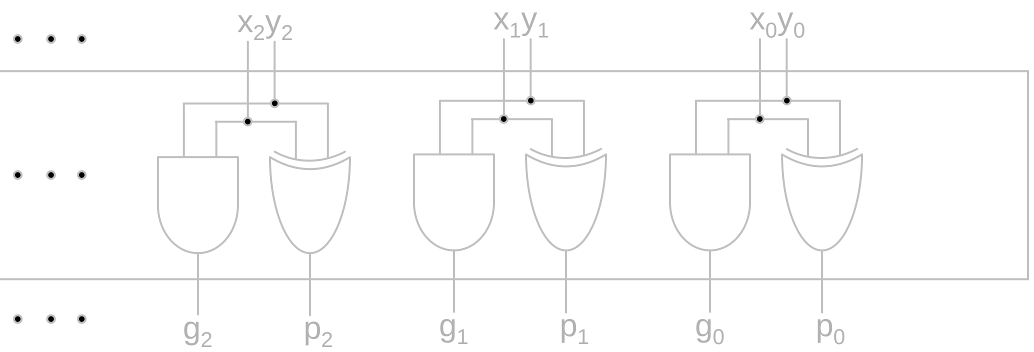
Carry Look Ahead adders

The CLA adder has the following 3-stage structure:

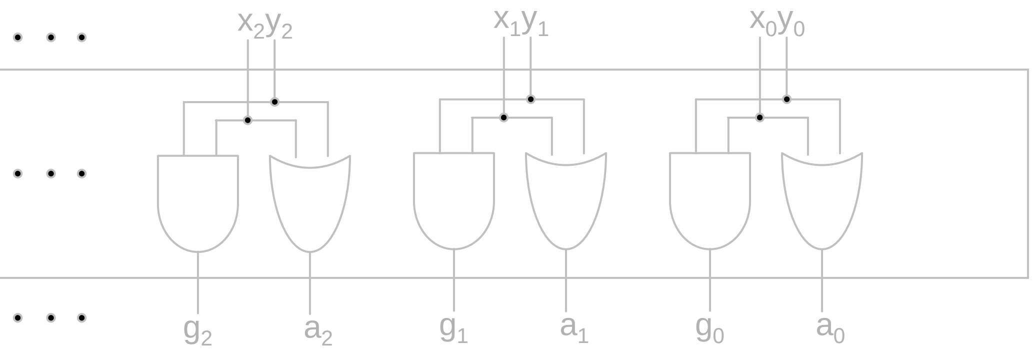

Pre-calculation of p_i, g_i for each stage
Calculation of carry c_i for each stage.
Combine c_i and p_i of each stage to generate the sum bits s_i
Final sum.

Carry Look Ahead adders

- The pre-calculation stage is implemented using the equations for p_i, g_i shown at a previous slide:



- Alternatively using the “alive” bit:



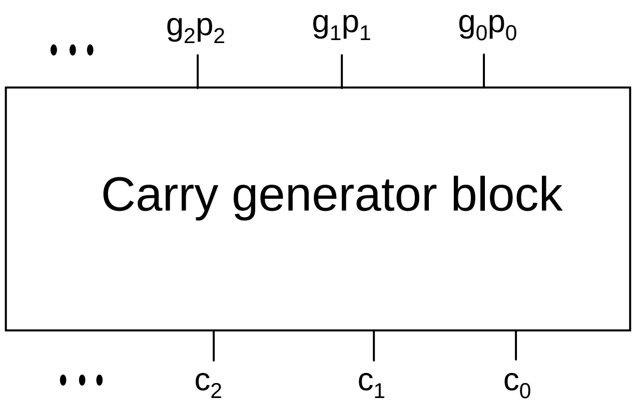
- Note the symmetry when we use the “propagate” or the “alive” bit… We can use them interchangeably in the equations!

Carry Look Ahead adders

- The carry calculation stage is implemented using the equations produced when unfolding the recursive equation:

$$c_i = g_i + p_i \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

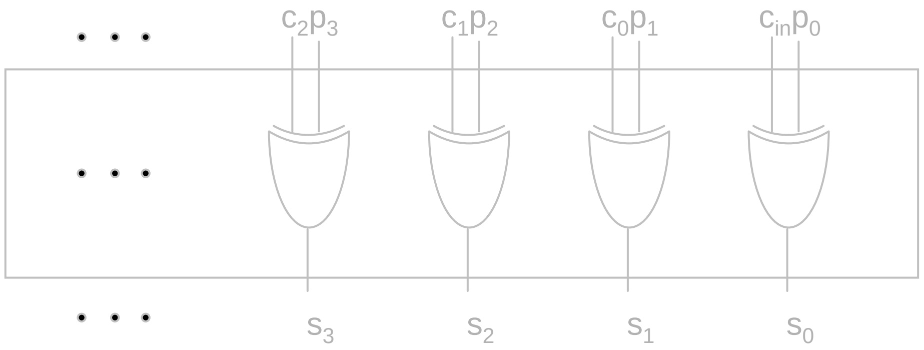
$$\begin{aligned} c_0 &= g_0 \\ c_1 &= g_1 + p_1 \cdot g_0 \\ c_2 &= g_2 + p_2 \cdot c_1 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0) \\ &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 \\ &\quad etc\ldots \end{aligned}$$



Carry Look Ahead adders

- The final sum calculation stage is implemented using the carry and propagate bits c_i,p_i:

$$s_i = p_i \oplus c_{i-1}, \quad with\ p_i = x_i \oplus y_i$$
$$Note:$$
$$s_i = g_i + a_i \cdot c_{i-1}, \quad with\ a_i = x_i + y_i$$



- If the ‘alive’ bit a_i is used the final sum stage becomes more complex as implied by the equations above.

Binary addition as a prefix sum problem.

- We define a new operator: " $\circ$ "
- Input is a vector of pairs of 'propagate' and 'generate' bits:

$$(g_n, p_n)(g_{n-1}, p_{n-1})\ldots(g_0, p_0)$$

- Output is a new vector of pairs:

$$(G_n, P_n)(G_{n-1}, P_{n-1})\ldots(G_0, P_0)$$

- Each pair of the output vector is calculated by the following definition:

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$

$$\textit{Where}:$$

$$(G_0, P_0) = (g_0, p_0)$$

$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y)$$

$$\textit{with} \quad +,\cdot \quad \textit{being the OR, AND operations}$$

Binary addition as a prefix sum problem.

- Properties of operator " ∘ ":
  - Associativity (hence parallelization)
    - Easy to prove based on the fact that the logical AND, OR operations are associative.
  - With the definition:

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$
$$\text{Where} \quad (G_1, P_1) = (g_1, p_1)$$

G_i becomes the carry signal at stage i of an adder. *Illustration on next slide.*

  - The operation is idempotent

$$(g_x, p_x) \circ (g_x, p_x) = (g_x + p_x \cdot g_x, p_x \cdot p_x) = (g_x, p_x)$$

    - Which implies

$$(G_{i:j}, P_{i:j}) = (G_{i:n}, P_{i:n}) \circ (G_{m:j}, P_{m:j})$$
$$\text{Where} \quad i \geq j \quad \text{and} \quad m \geq n$$

Binary Addition as a prefix sum problem.

$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline \end{array}$$

A stage i will generate a carry if

$g_i = a_i b_i$

and propagate a carry if

$p_i = XOR(a_i, b_i)$

Hence for stage i:

$c_i = g_i + p_i c_{i-1}$

$$\textit{With:} \qquad\qquad \textit{Where:}$$

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1}) \qquad (G_0, P_0) = (g_0, p_0)$$

$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y)$$

$$\textit{We have:}$$

$$(G_1, P_1) = (g_1, p_1)$$

$$(G_2, P_2) = (g_2, p_2) \circ (G_1, P_1) = (g_2 + p_2 \cdot g_1, p_2 \cdot p_1)$$

$$(G_3, P_3) = (g_3, p_3) \circ (G_2, P_2) = (g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1), p_3 \cdot p_2 \cdot p_1)$$

$$= (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1), p_3 \cdot p_2 \cdot p_1)$$

$$etc \ldots$$

… The familiar carry bit generating equations for stage *i* in a CLA adder.

Addition as a prefix sum problem.

Conclusion:

The equations of the well known CLA adder can be formulated as a parallel prefix problem by employing a special operator “ ° ”.

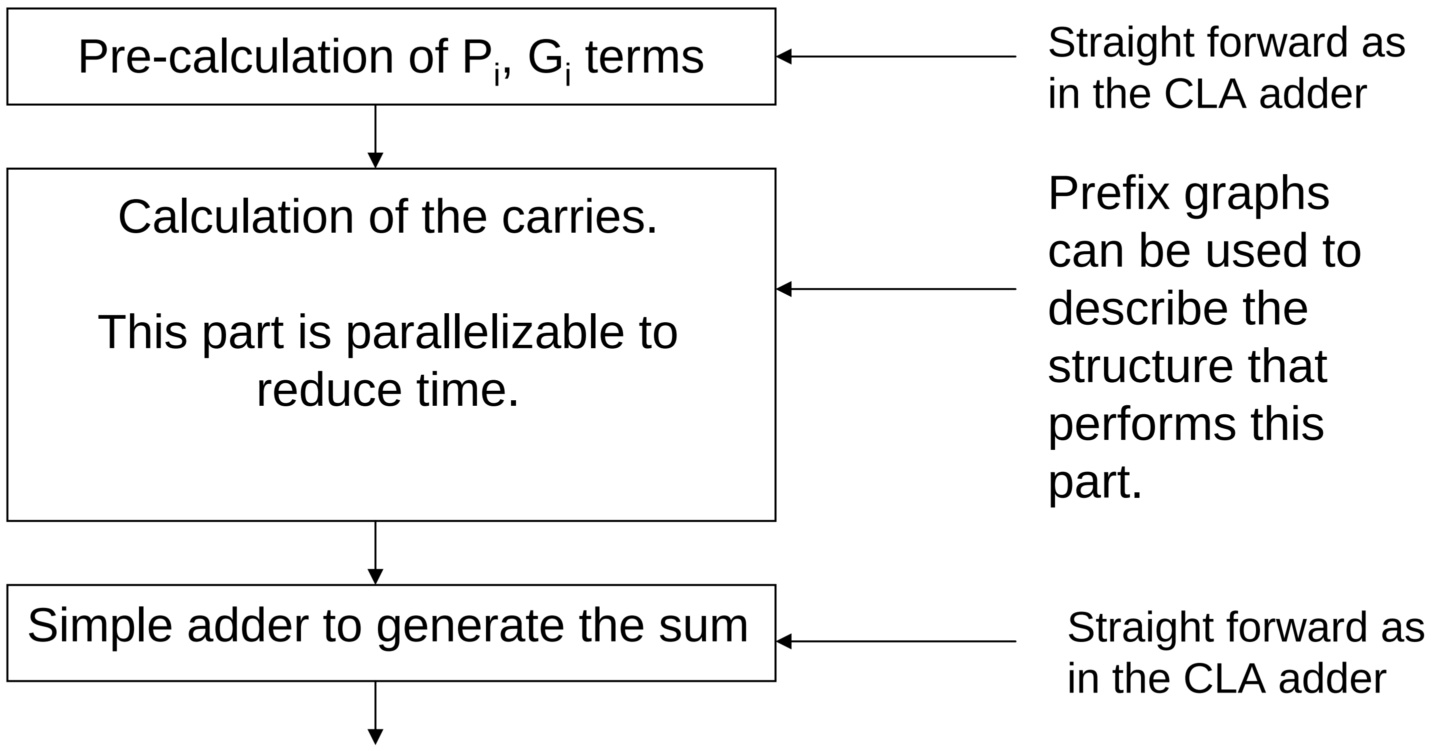
This operator is associative hence it can be implemented in a parallel fashion.

A Parallel Prefix Adder (PPA) is equivalent to the CLA adder… The two differ in the way their carry generation block is implemented.

In subsequent slides we will see different topologies for the parallel generation of carries. Adders that use these topologies are called *Parallel Prefix Adders*.

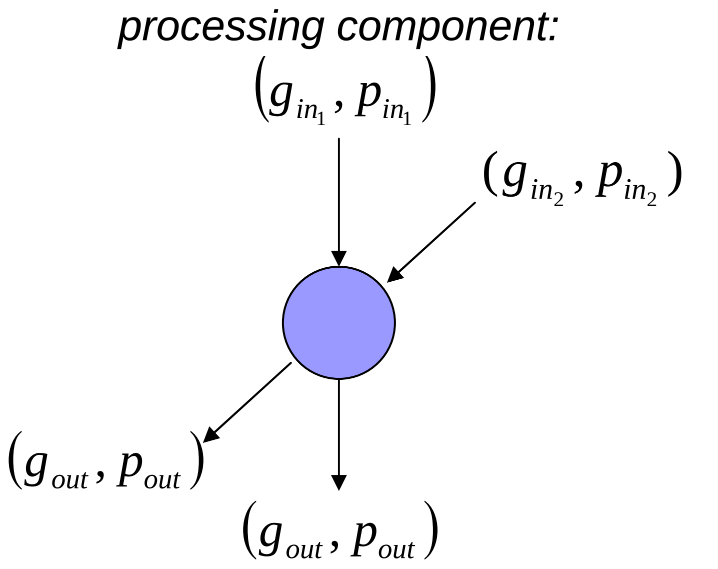
Parallel Prefix Adders

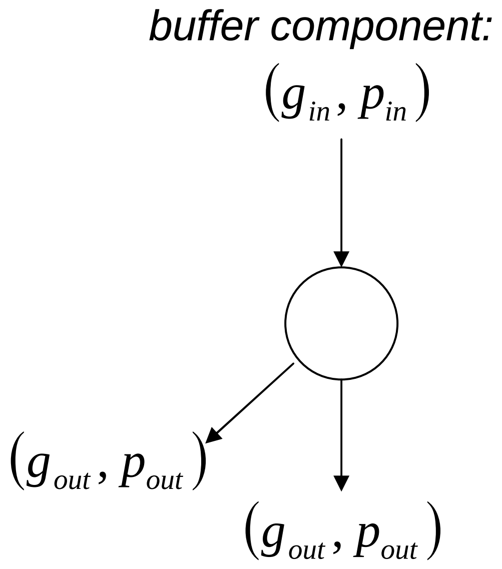
- The parallel prefix adder employs the 3-stage structure of the CLA adder. The improvement is in the carry generation stage which is the most intensive one:


Pre-calculation of P_i, G_i terms
Calculation of the carries.
This part is parallelizable to reduce time.
Simple adder to generate the sum
Straight forward as in the CLA adder
Prefix graphs can be used to describe the structure that performs this part.
Straight forward as in the CLA adder

Calculation of carries – Prefix Graphs

The components usually seen in a prefix graph are the following:



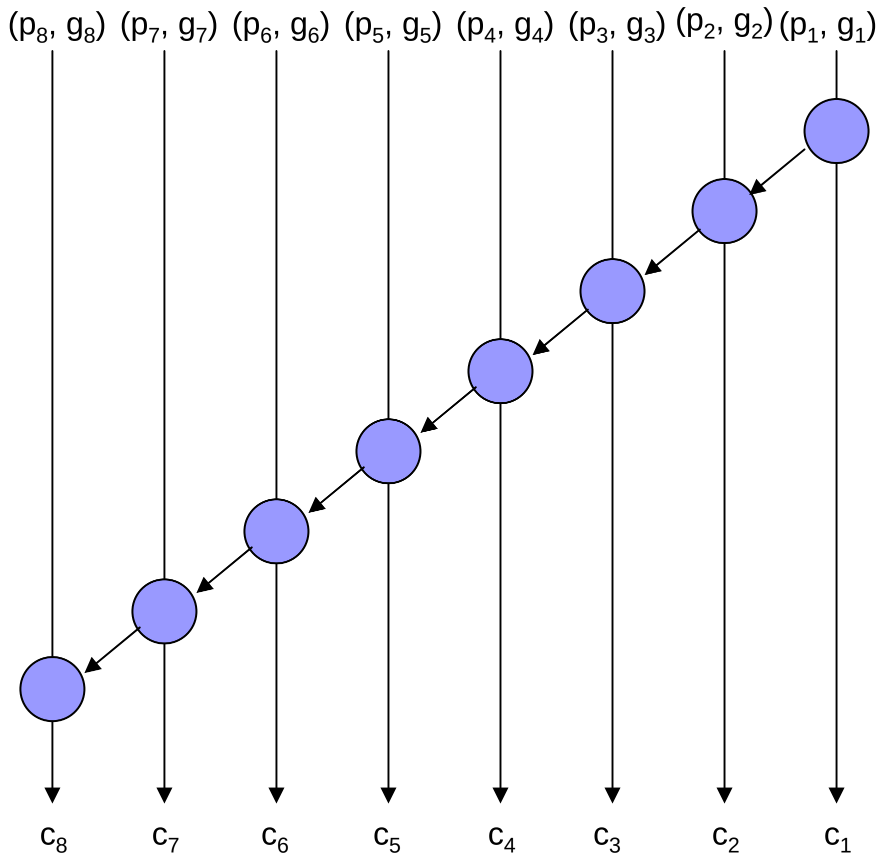


$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2})$$

$$(g_{out}, p_{out}) = (g_{in}, p_{in})$$

Prefix graphs for representation of Prefix addition

- Example: serial adder carry generation represented by prefix graphs



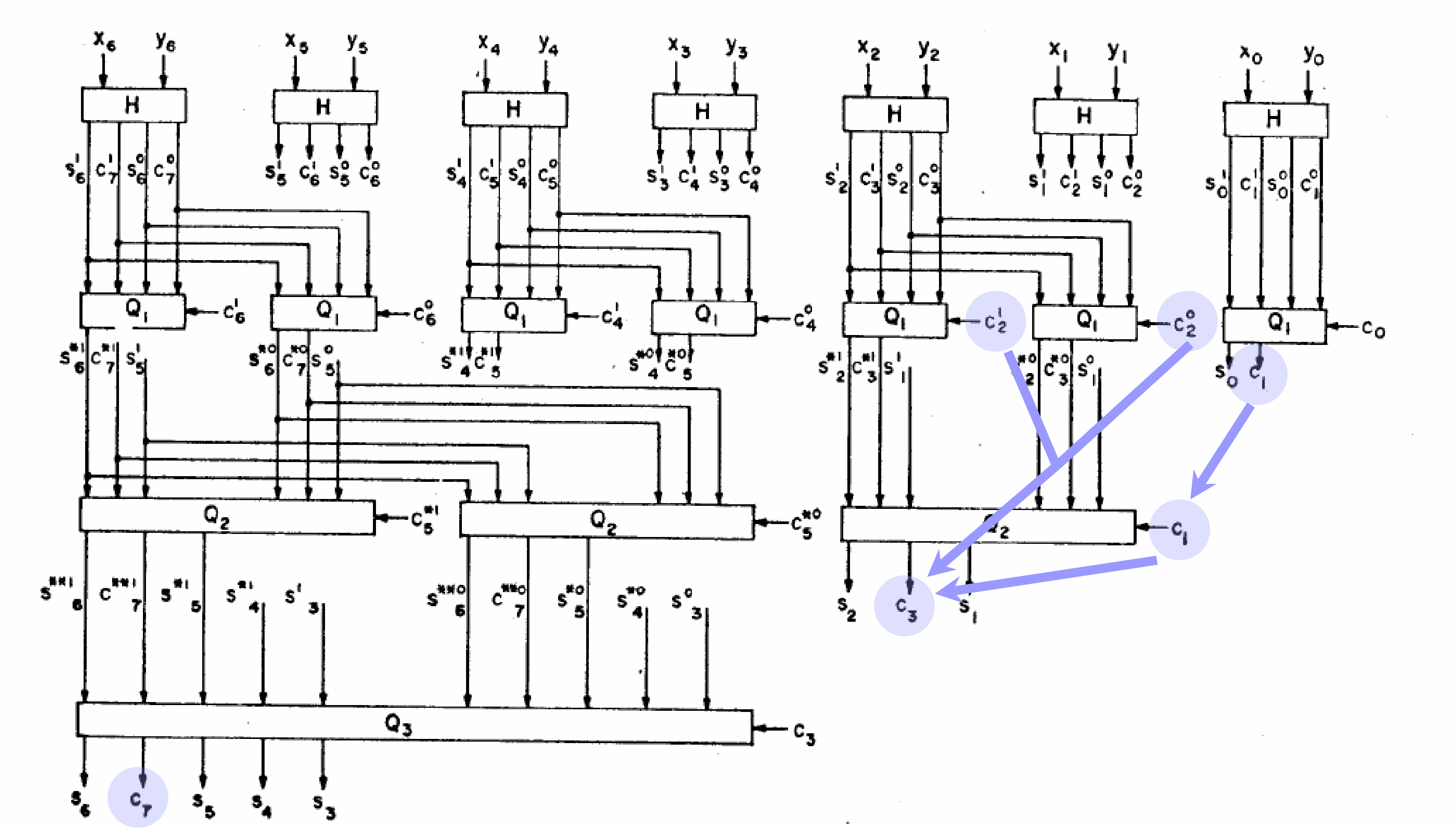
Key architectures for carry calculation:

- 1960: J. Sklansky – conditional adder
- 1973: Kogge-Stone adder
- 1980: Ladner-Fisher adder
- 1982: Brent-Kung adder
- 1987: Han Carlson adder
- 1999: S. Knowles

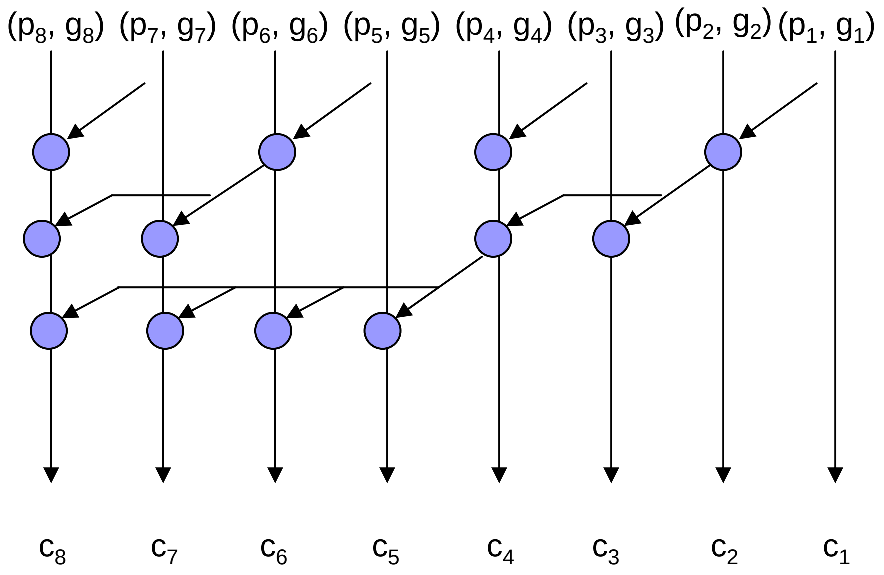
Other parallel adder architectures:

- 1981: H. Ling adder
- 2001: Beaumont-Smith

1960: J. Sklansky – conditional adder

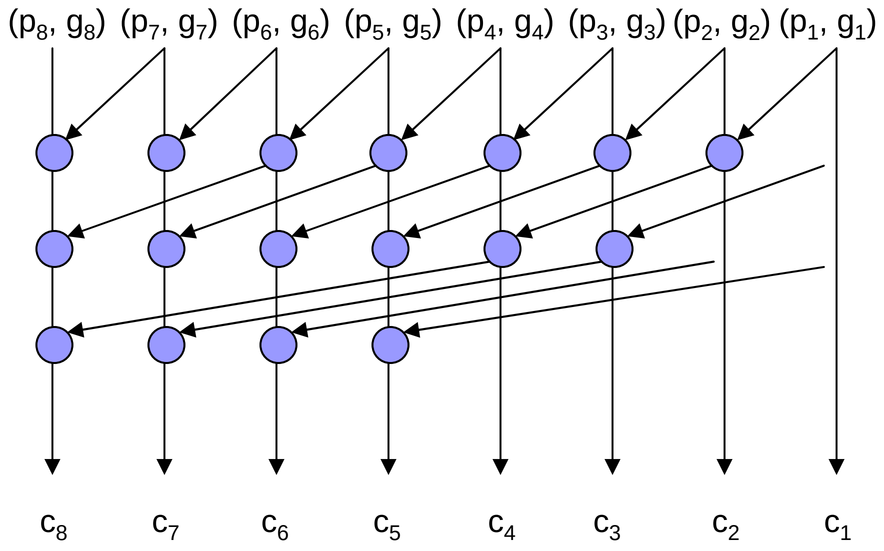

x_6 y_6 x_5 y_5 x_4 y_4 x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0
H
Q_1
Q_2
Q_3
c_0
c_1
c_2^1
c_2^0
c_3
s_6 c_7 s_5 s_4 s_3
s_2 c_3 s_1 s_0 c_1

1960: J. Sklansky – conditional adder



- The Sklansky adder has:
  - Minimal depth
  - High fan-out nodes

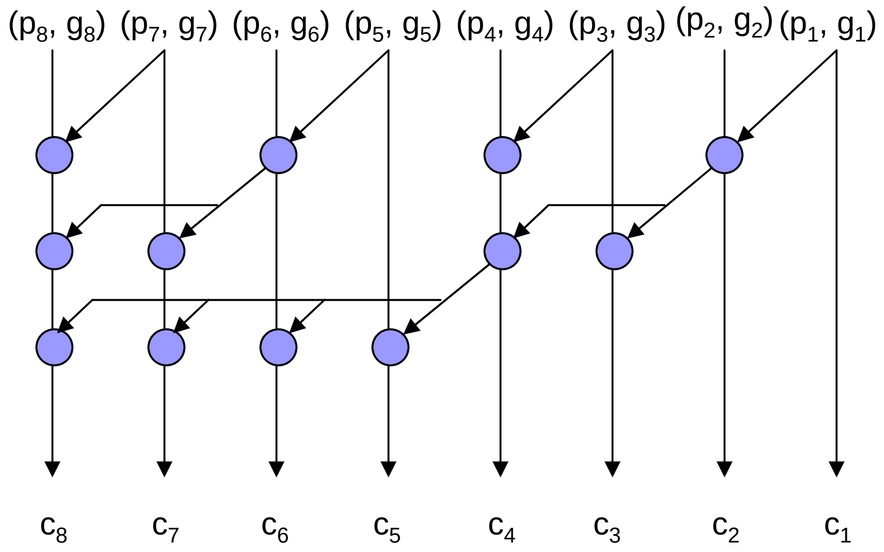
1973: Kogge-Stone adder



(p_8, g_8) (p_7, g_7) (p_6, g_6) (p_5, g_5) (p_4, g_4) (p_3, g_3) (p_2, g_2) (p_1, g_1)
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1

- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

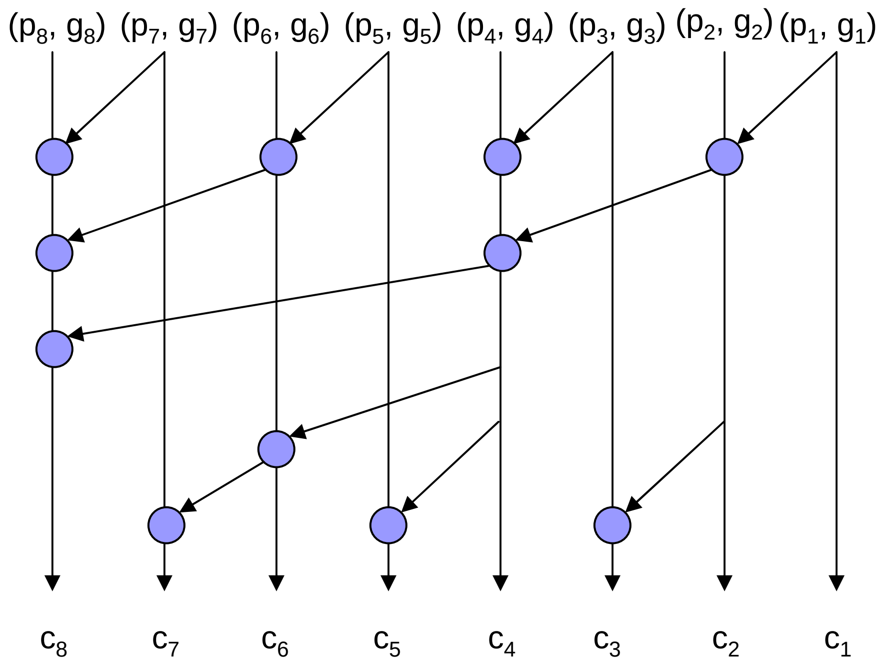
1980: Ladner-Fischer adder



(p_8, g_8) (p_7, g_7) (p_6, g_6) (p_5, g_5) (p_4, g_4) (p_3, g_3) (p_2, g_2) (p_1, g_1)
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1

- The Ladner-Fischer adder has:
  - Low depth
  - High fan-out nodes
  - This adder topology appears the same as the Schlanskly conditional sum adder. Ladner-Fischer formulated a parallel prefix network design space which included this minimal depth case. The actual adder they included as an application to their work had a structure that was slightly different than the above.

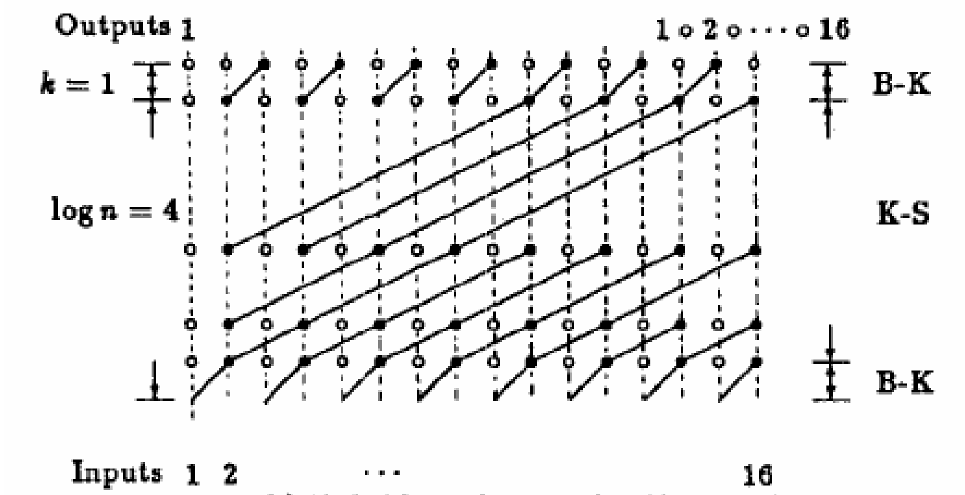
1982: Brent-Kung adder



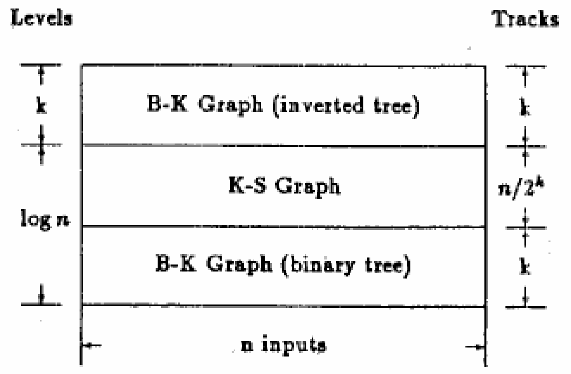
(p_8, g_8) (p_7, g_7) (p_6, g_6) (p_5, g_5) (p_4, g_4) (p_3, g_3) (p_2, g_2) (p_1, g_1)
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1

- The Brent-Kung adder is the *extreme* boundary case of:
  - Maximum logic depth in PP adders (implies longer calculation time).
  - Minimum number of nodes (implies minimum area).

1987: Han Carlson adder



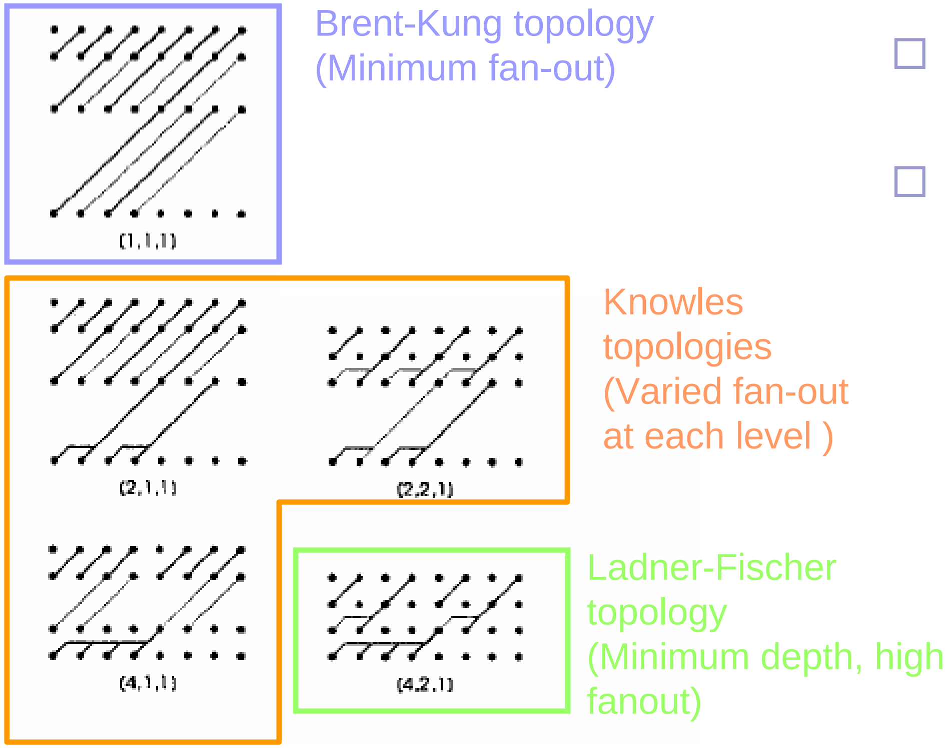
b) Hybrid Prefix Graph of k=1 and n=16



a) Block Diagram of HPC

- The Han-Carlson adder combines the Brent-Kung and Kogge-Stone structures into a hybrid structure.
  - Efficient
  - Suitable for VLSI implementation.

1999: S. Knowles


(1,1,1)
Brent-Kung topology
(Minimum fan-out)
(2,1,1)
(2,2,1)
Knowles topologies (Varied fan-out at each level)
(4,1,1)
(4,2,1)
Ladner-Fischer topology (Minimum depth, high fanout)

- Knowles proposed adders that trade off:
  - Depth, interconnect, area.
  - These adders are bound by the Lander-Fischer (minimum depth) and Brent-Kung (minimum fanout) topologies.

An interesting taxonomy:

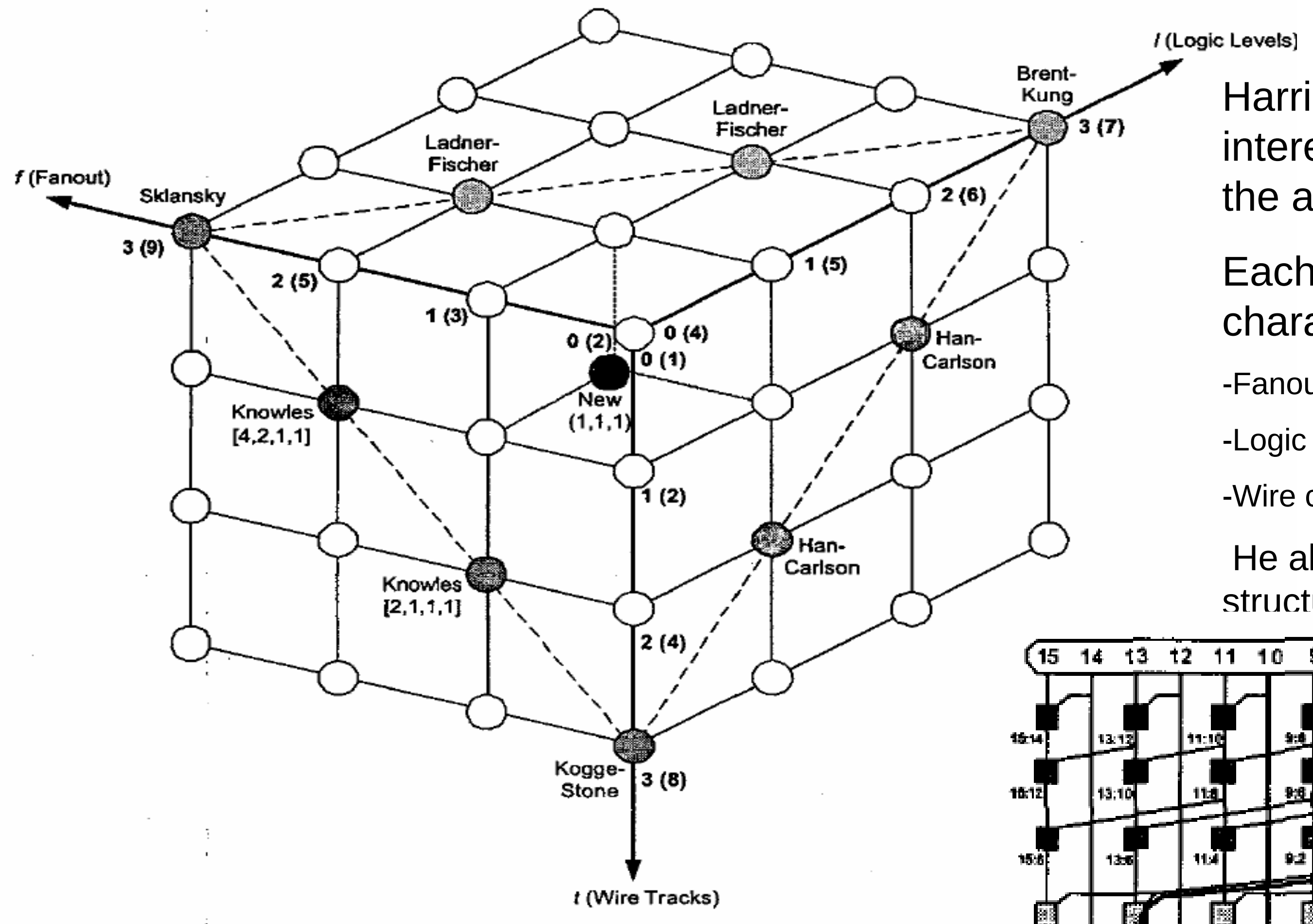


Fig. 3. Taxonomy of prefix graphs

Harris[2003] presented an interesting 3-D taxonomy of the adders presented so far.

Each axis represents a characteristic of the adders:

-Fanout

-Logic depth

-Wire connections

He also proposed the following structure:

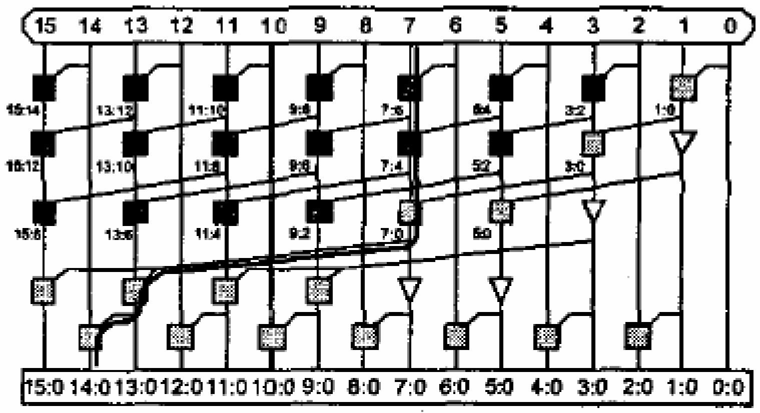


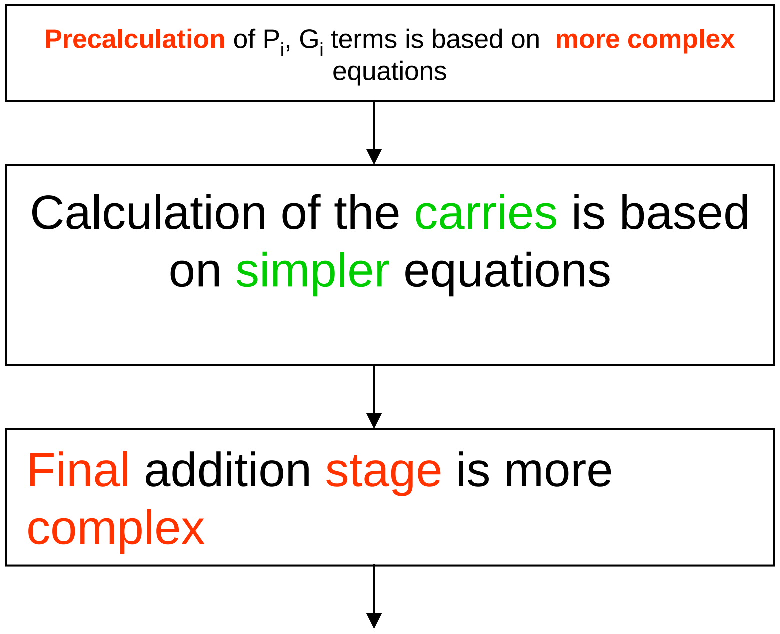
Fig. 4. New (1,1,1) parallel prefix network

1981: H. Ling adder

Ling Adders are a different family of adders.
They can still be formulated as prefix adders.

Ling adders differ from the “traditional” PP adders in that:

- They are based on a different set of equations.
- The new set of equations introduces the following tradeoffs:

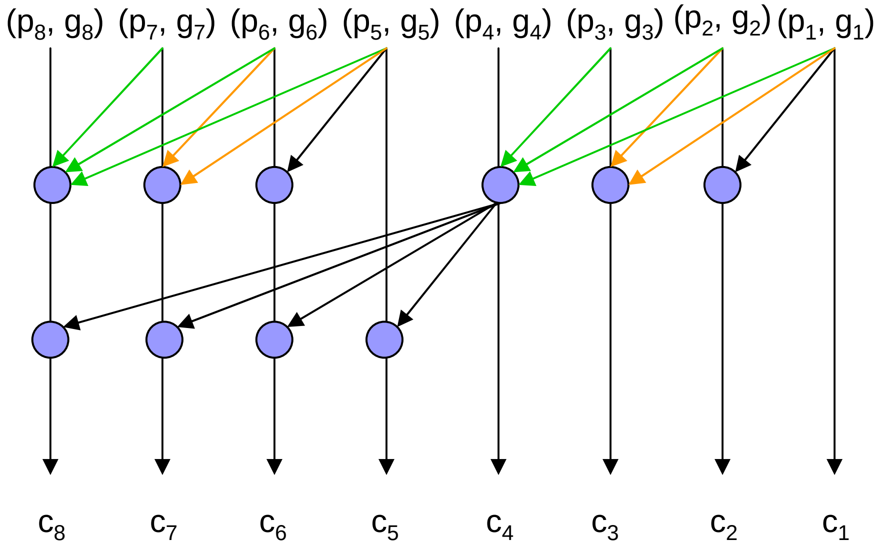


Precalculation of P_i, G_i terms is based on **more complex** equations

Calculation of the carries is based on simpler equations

Final addition stage is more complex

2001: Beaumont-Smith



(p_8, g_8) (p_7, g_7) (p_6, g_6) (p_5, g_5) (p_4, g_4) (p_3, g_3) (p_2, g_2) (p_1, g_1)
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1

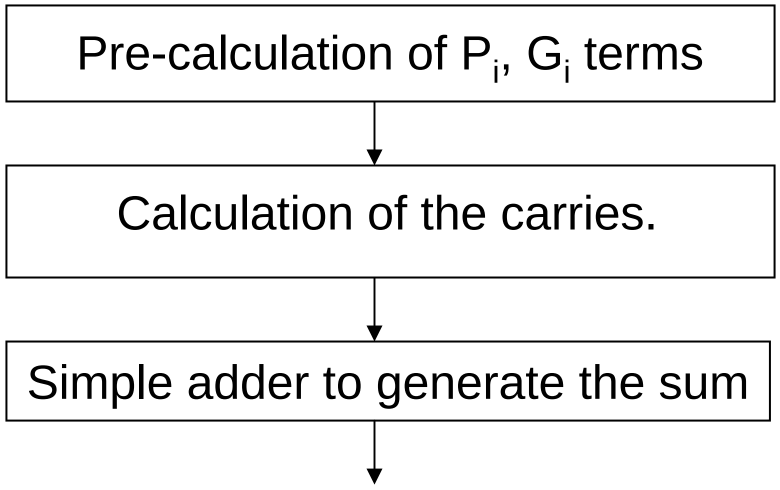
- The Beaumont-Smith adders incorporate nodes that can accept more than a pair of inputs and produce the carry calculation.
- These ‘higher valency’ nodes are optimized circuits for a specific technology (CMOS).
- The above topology is a Beaumont-Smith tree based on the Kogge-Stone architecture

Summary (1/3)

- The parallel prefix formulation of binary addition is a very convenient way to formally describe an entire family of parallel binary adders.

Summary (2/3)

- A parallel prefix adder can be seen as a 3-stage process:


Pre-calculation of P_i, G_i terms
Calculation of the carries.
Simple adder to generate the sum

- There exist various architectures for the carry calculation part.
- Trade-offs in these architectures involve the
  - area of the adder
  - its depth
  - the fan-out of the nodes
  - the overall wiring network.

Summary (3/3)

- Variations of parallel adders have been proposed. These variations are based on:
  - Modifying the carry generation equations and reformulating the prefix definition (Ling)
  - Restructuring the carry calculation trees based by optimizing for a specific technology (Beaumond-Smith)
  - Other optimizations.

References:

Beaumont-Smith, Cheng-Chew Lim, “Parallel Prefix Adder Design”, IEEE, 2001

Han, Carlson, “Fast Area-Efficient VLSI Adders, IEEE, 1987

Dimitrakopoulos, Nikolos, “High-Speed Parallel-Prefix VLSI Ling Adders”, IEEE 2005

Kogge, Stone, “A Parallel Algorithm for the Efficient solution of a General Class of Recurrence equations”, IEEE, 1973

Simon Knowles, “A Family of adders”, IEEE, 2001

Ladner, Fischer, “Parallel Prefix Computation”, ACM, 1980

Brent, Kung, “A regular Layout for Parallel Adders”, IEEE, 1982

H. Ling, “High-Speed Binary Adder”, IBM J. Res. And Dev., 1980

J. Sklansky, “Conditional-Sum Addition Logic”, IRE transactions on computers, 1960

D. Harris, “A Taxonomy of Parallel Prefix Networks”, IEEE, 2003