Arithmetic Circuits

Virendra Singh

Professor

Computer Architecture and Dependable Systems Lab

Department of Computer Science & Engineering, and

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.cse.iitb.ac.in/~viren/

E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

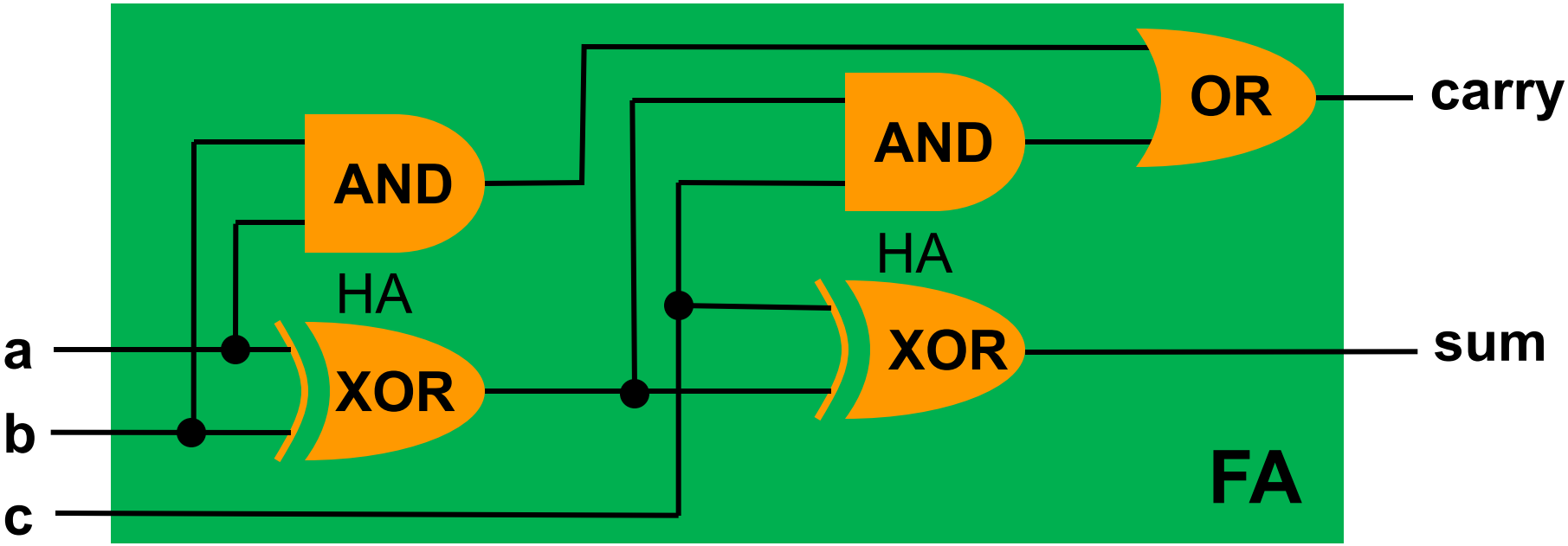
Lecture 13 (03 February 2022)

CADSL

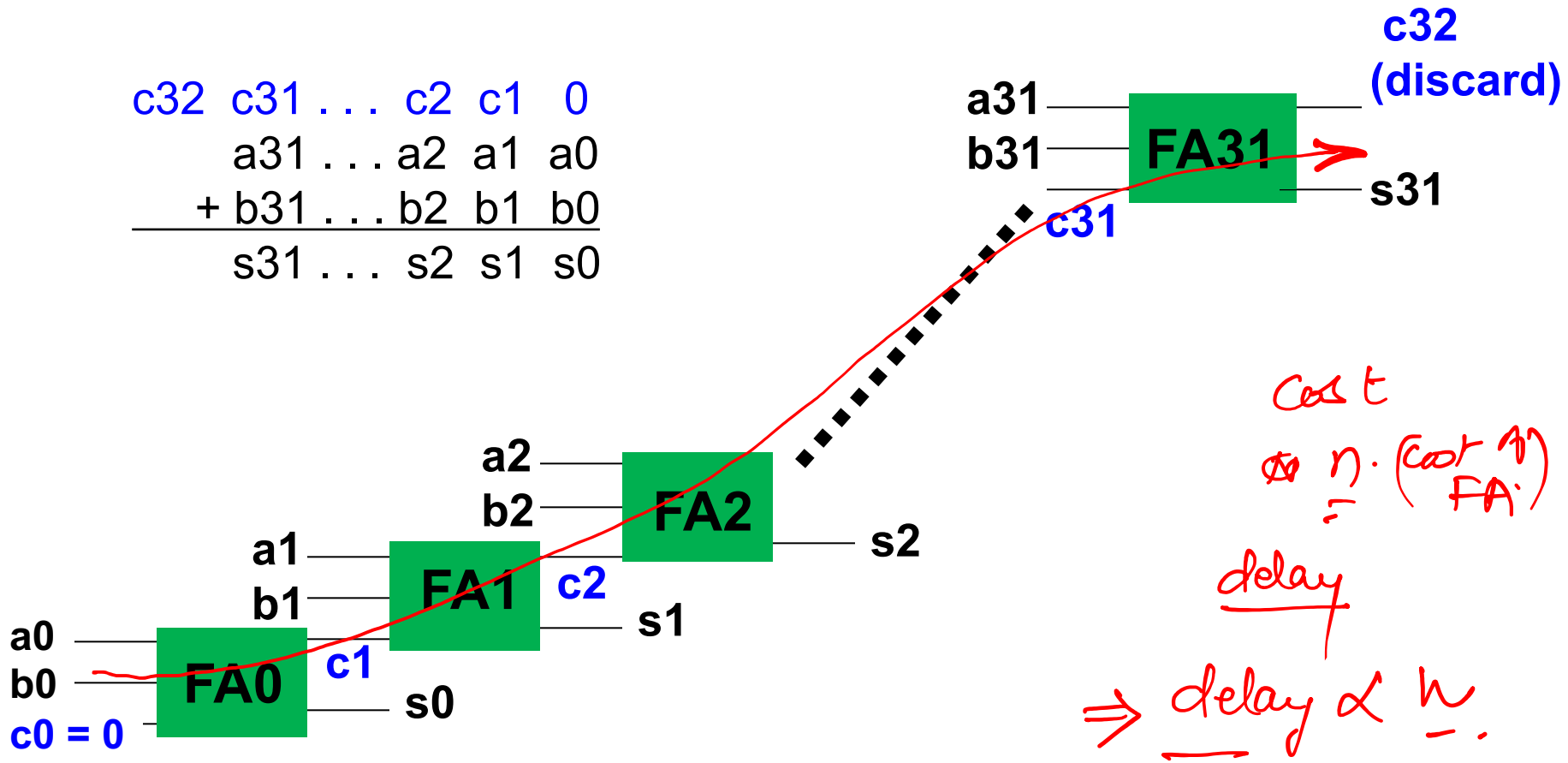
Arithmetic Circuits:

(Adders)

Full-Adder Adds Three Bits



32-bit Ripple-Carry Adder



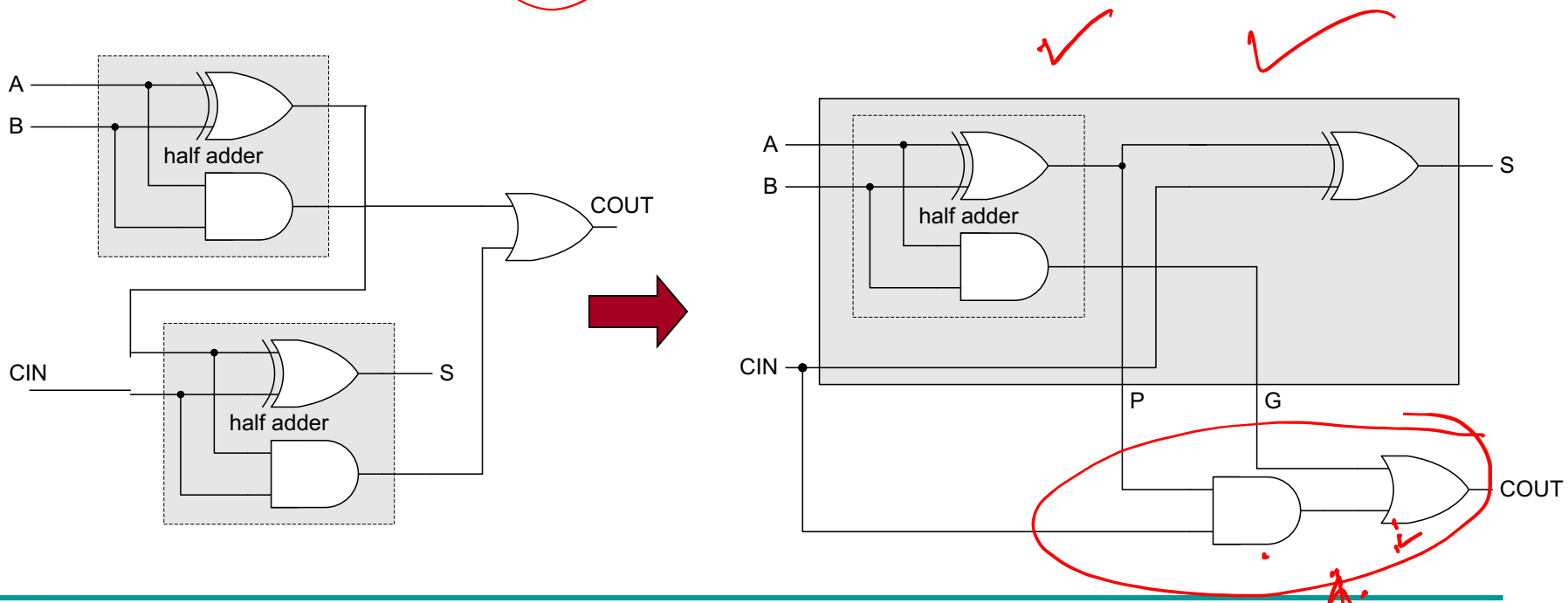
Ripple Carry Adder

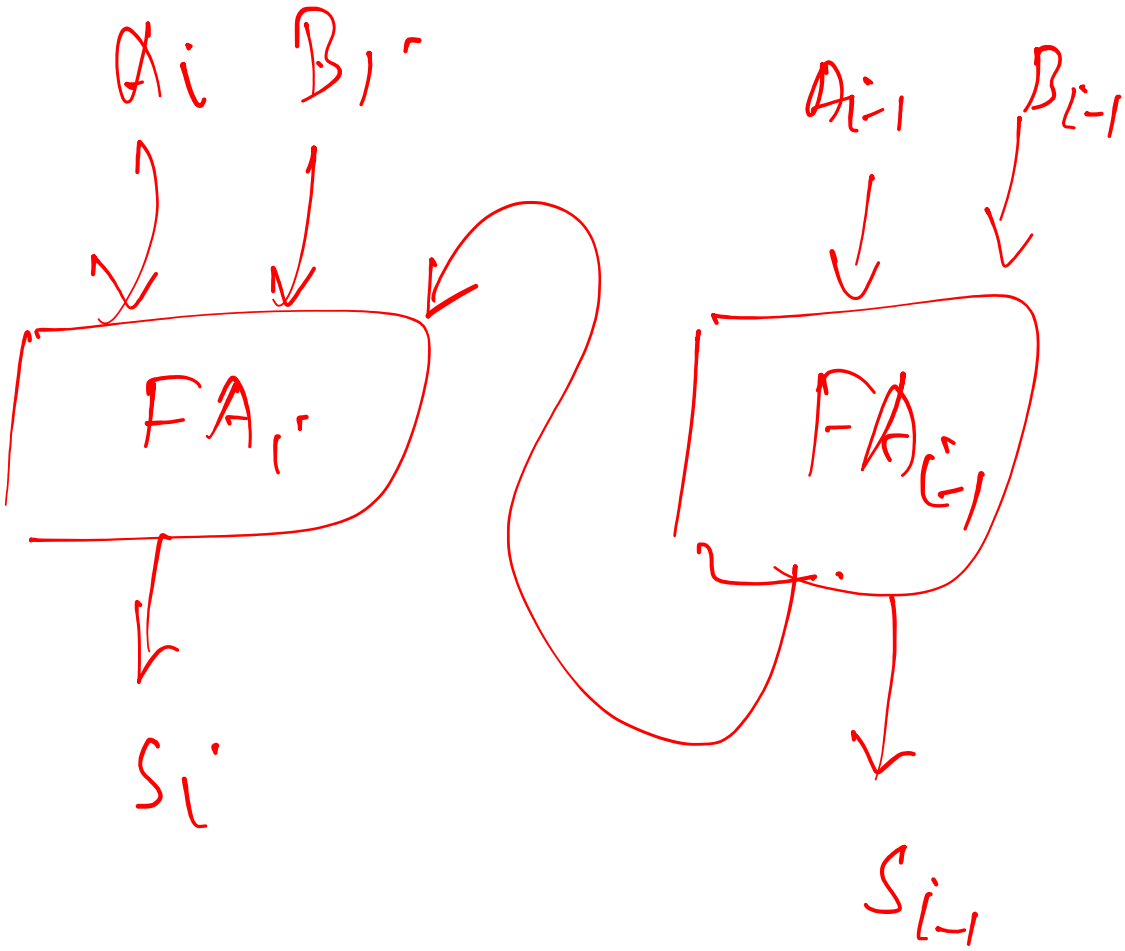
- Easy to create
  - Good hierarchy
  - Tileable structures
- Small hardware
- Slow!
  - Design is limited by the delays in propagating carry through all of the bitwise additions
  - The output bit at position m is not valid until after the carry out of the m-1 position is ready

FAST ADDERS

Faster Addition

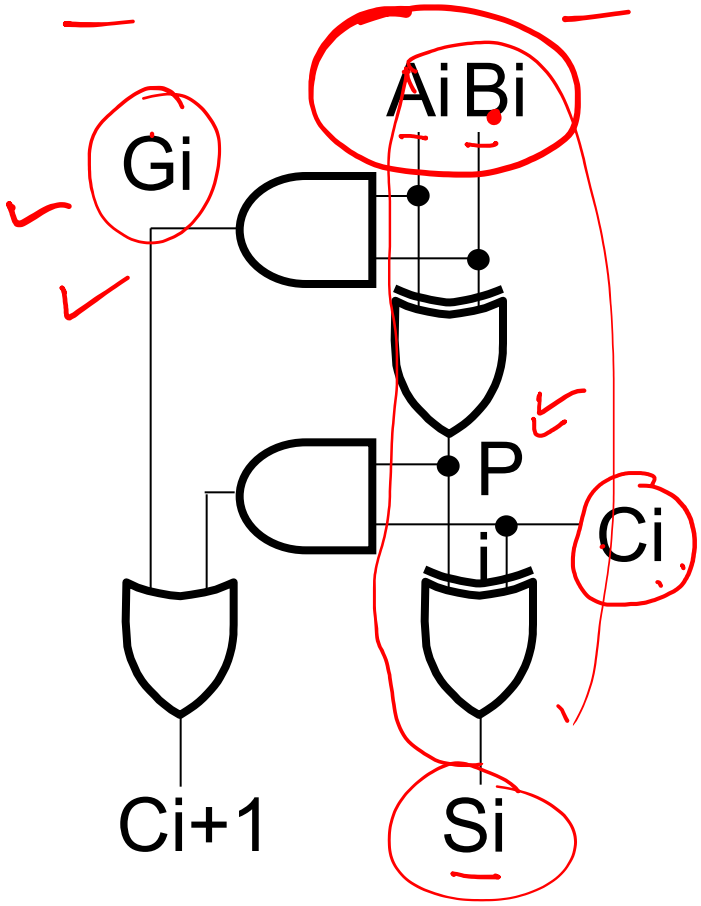
- For big adders, the carry-chain is very long
- Separate the carry chain and sum logic
- Partial Full Adders
  - Contain only the sum part of a FA




Ai Bi
Ai-1 Bi-1
FAi
FAi-1
Si
Si-1

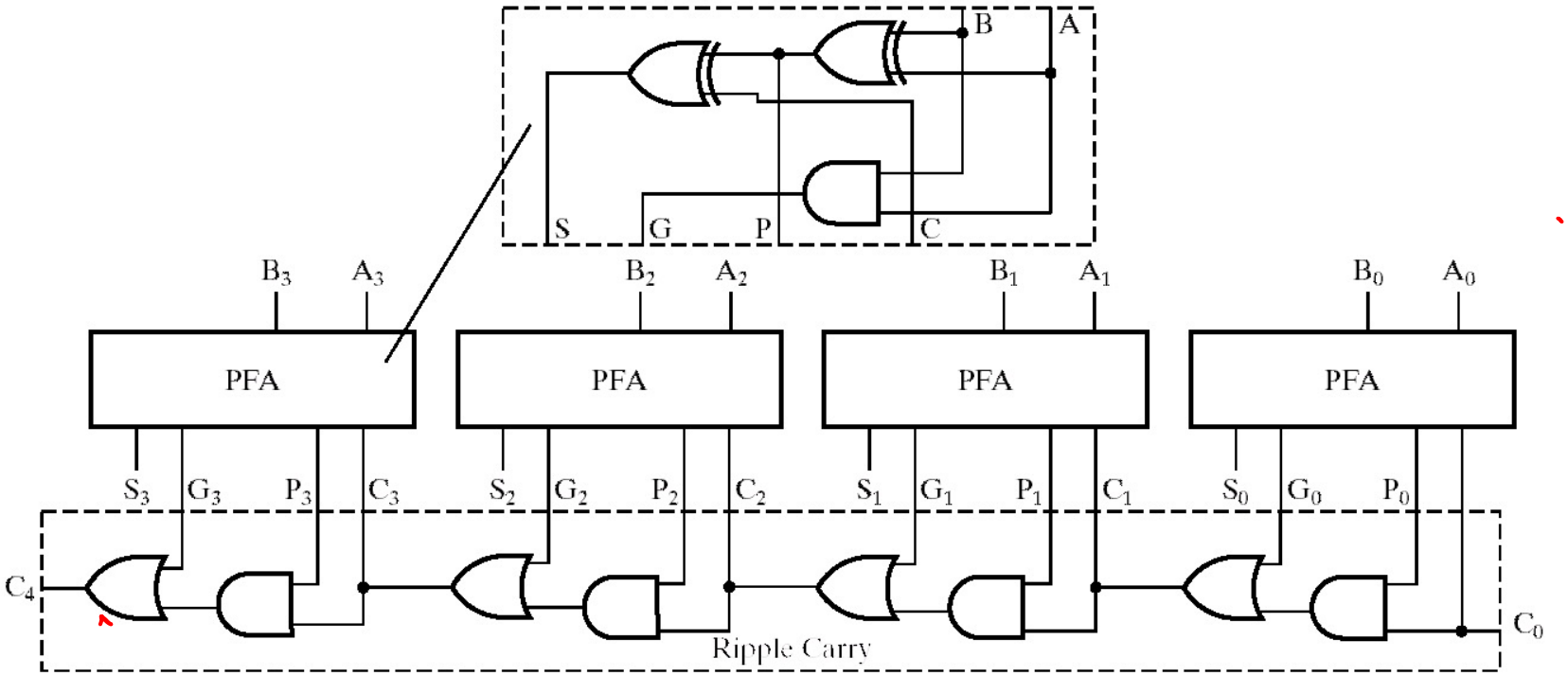
Carry Lookahead

- Given Stage i from a Full Adder, we know that there will be a carry generated when $A_i = B_i =$ "1", whether or not there is a carry-in.
- Alternately, there will be a carry propagated if the "half-sum" is "1" and a carry-in, C_i occurs.
- These two signal conditions are called *generate*, denoted as G_i, and *propagate*, denoted as P_i respectively and are identified in the circuit:



Reassembling RCA Using PFAs

- Can create ripple-carry adder with PFAs
  - G: **Generates** a carry at this position
  - P: **Propagates** a carry through this position



Carry Lookahead (continued)

- In the ripple carry adder:
  - G_i, P_i, and S_i are local to each cell of the adder
  - C_i is also local each cell
- In the carry lookahead adder, in order to reduce the length of the carry chain, C_i is changed to a more global function spanning multiple cells
- Defining the equations for the Full Adder in term of the P_i and G_i:

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

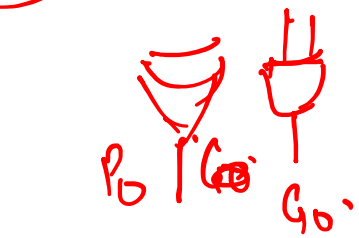
Carry Lookahead Development

- Flatten equations for carry using G_i and P_i terms for less significant bits
- Beginning at the cell 0 with carry in C_0:

$$C1 = G0 + P0\ C0$$

$$C2 = G1 + P1\ C1 = G1 + P1(G0 + P0\ C0)$$

$$= G1 + P1G0 + P1P0\ C0$$



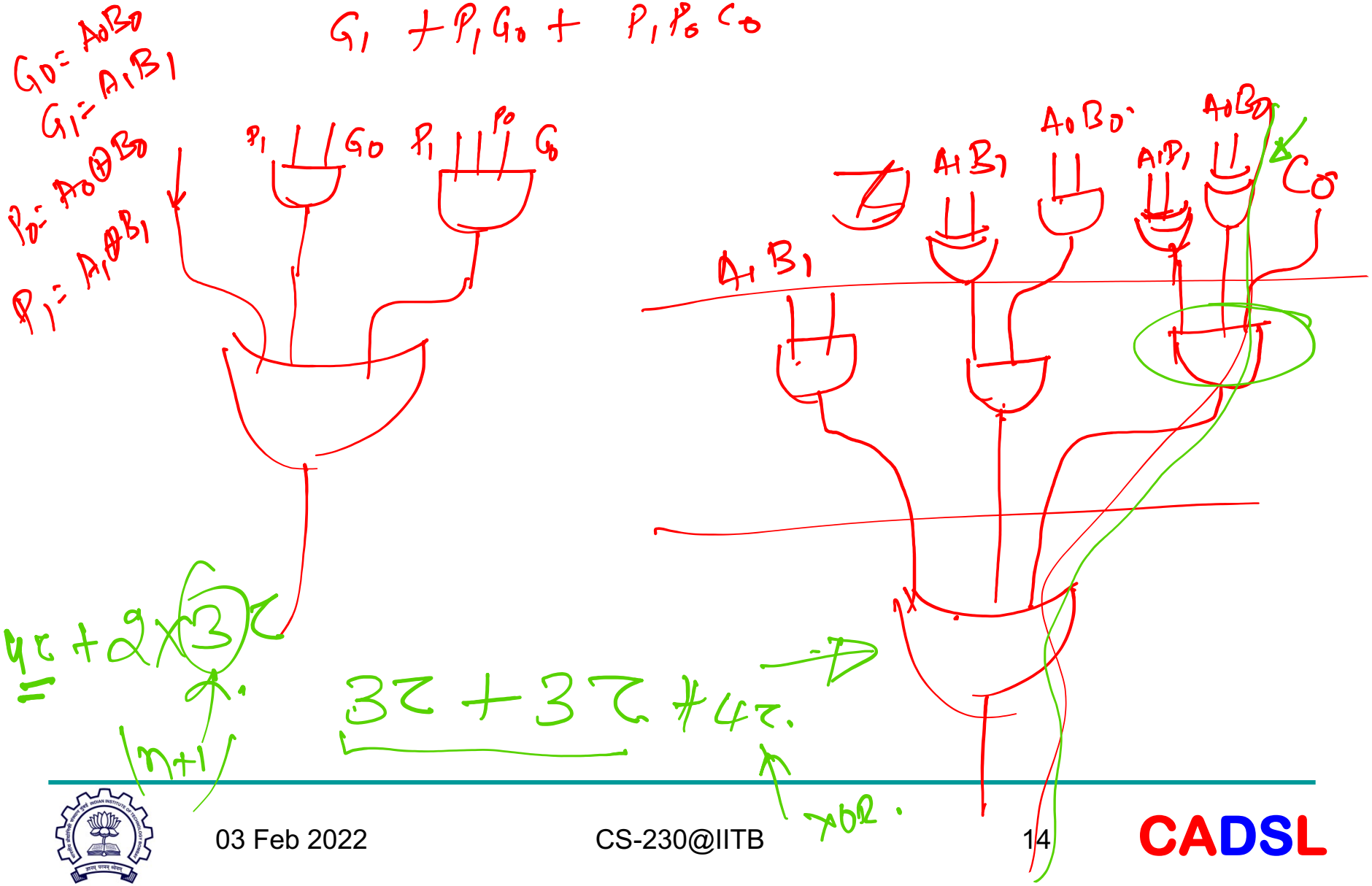
Carry Lookahead Development

$$C3 = G2 + P2\,C2 = G2 + P2(G1 + P1G0 + P1P0\,C0)$$

$$= G2 + P2G1 + P2P1G0 + P2P1P0\,C0$$

$$C4 = G3 + P3\,C3 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0\,C0$$

Carry Lookahead Adder



Carry Lookahead Adder

delay $\propto n$

↑ CMOS

$4\tau + (n+1)\tau$

$= 5\tau + n\tau$

BJT ⇐ delay is independent of input

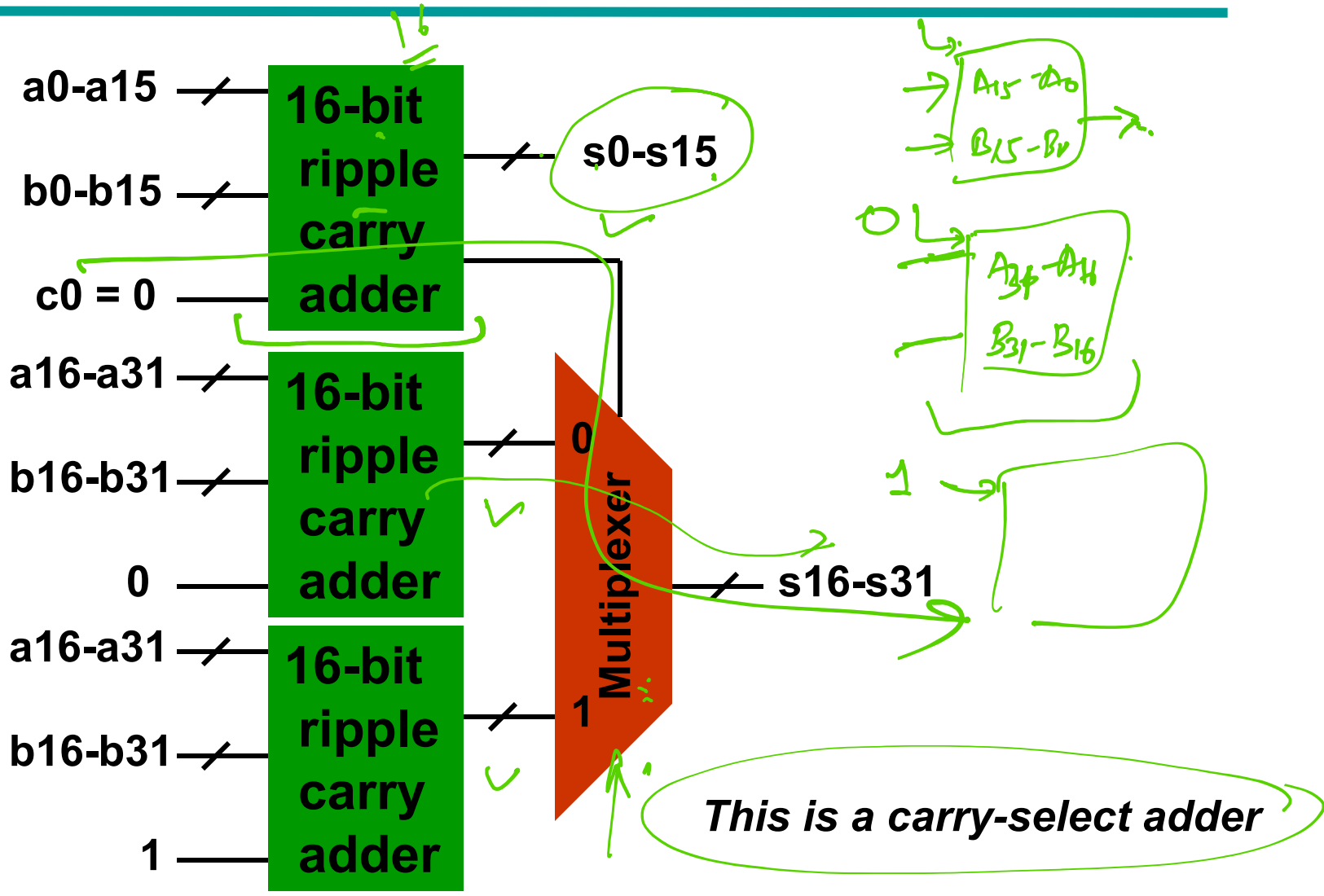
↳ TTL ⇐ ⇑ constant delay.

Carry Look-Ahead Adder

- As usual, can trade area/power for speed
  - Multi-level logic (ripple carry) reduces area
  - Flattening carry logic increases speed
- Sometimes called CLA

Break carry chain

Speeding Up the Adder

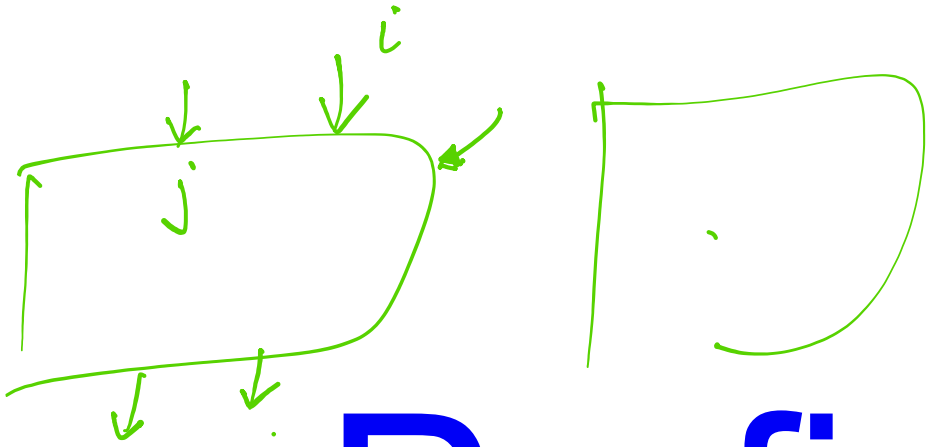


Fast Adders

- In general, any output of a 32-bit adder can be evaluated as a logic expression in terms of all 65 inputs.

- Number of levels of logic can be reduced to $\log_2 N$ for N-bit adder. Ripple-carry has N levels.

- More gates are needed, about $\log_2 N$ times that of ripple-carry design.



Prefix Computation

Key Architectures for Carry Calculation

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

Other parallel adder architectures:

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

Binary Addition

- **Input:** two *n*-bit binary numbers $a_{n-1}\ldots a_1 a_0$ and $b_{n-1}\ldots b_1 b_0$, one bit carry-in c_0
- **Output:** n-bit sum $s_{n-1}\ldots s_1 s_0$ and one bit carry out c_n
- Prefix Addition: Carry generation & propagation

$$\text{Generate}: \quad g_i = a_i b_i$$

$$\text{Propagate}: \quad p_i = a_i \oplus b_i$$

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

$$s_i = c_i \oplus (a_i \oplus b_i)$$

Binary Addition as a prefix sum problem.

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

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

With:

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

Where:

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

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

We have:

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

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

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

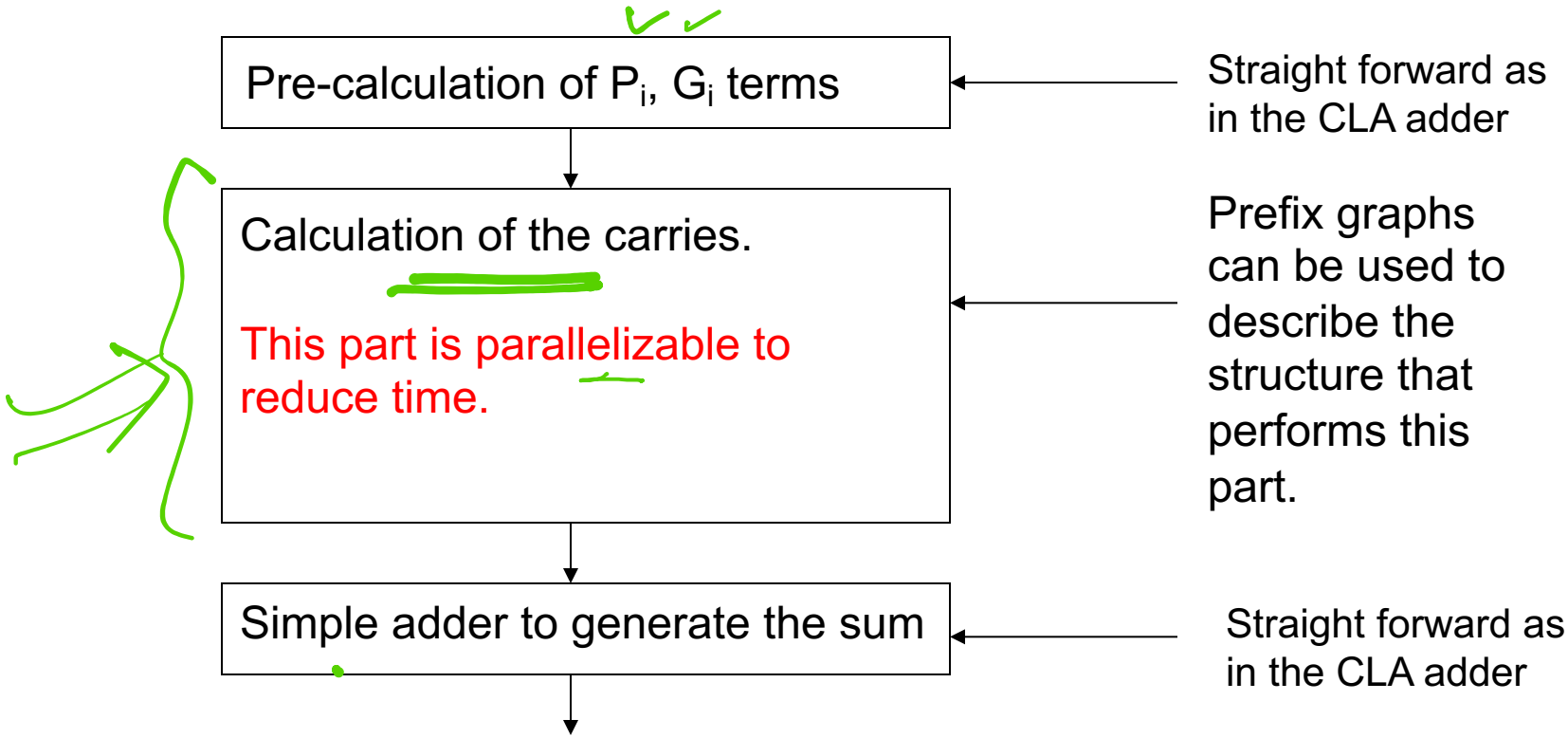
$$= (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1), p_3 \cdot p_2 \cdot p_1)$$

etc…

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

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:



Prefix Addition – Formulation

Pre-processing:

$$g_i = a_i b_i \qquad p_i = a_i \oplus b_i$$

Prefix Computation:

$$G_{[i:k]} = G_{[i:j]} + P_{[i:j]} G_{[j-1:k]}$$

$$P_{[i:k]} = P_{[i:j]} P_{[j-1:k]}$$

Post-processing:

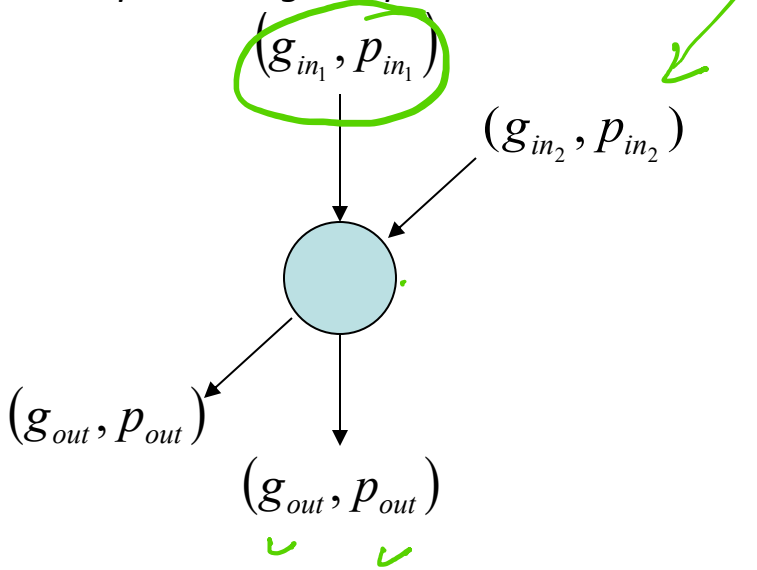
$$c_{i+1} = G_{[i:0]} + P_{[i:0]} \cdot c_0$$

$$s_i = p_i \oplus c_i$$

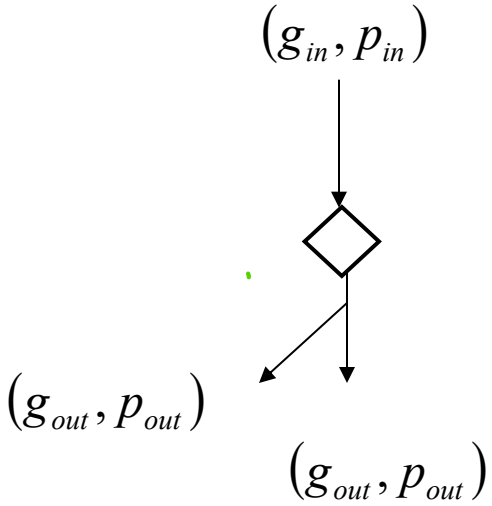
Computation of Carries – Prefix Graphs

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

processing component:



buffer component:

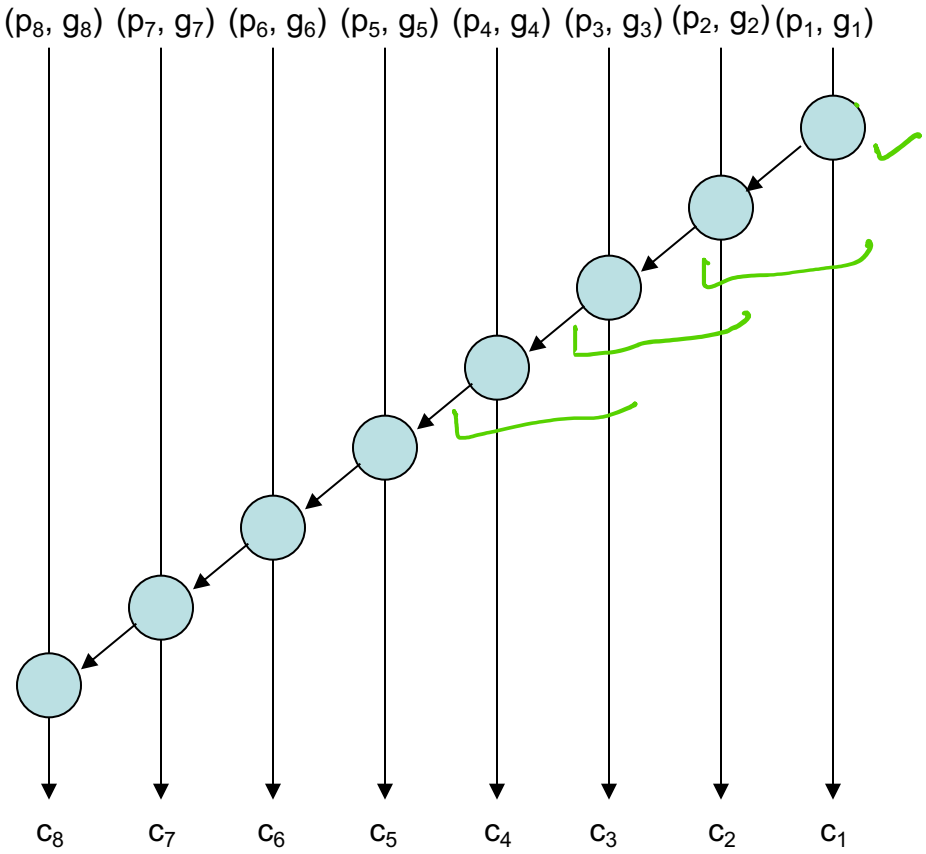


$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2})$$

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

Prefix graphs for representation of Prefix addition

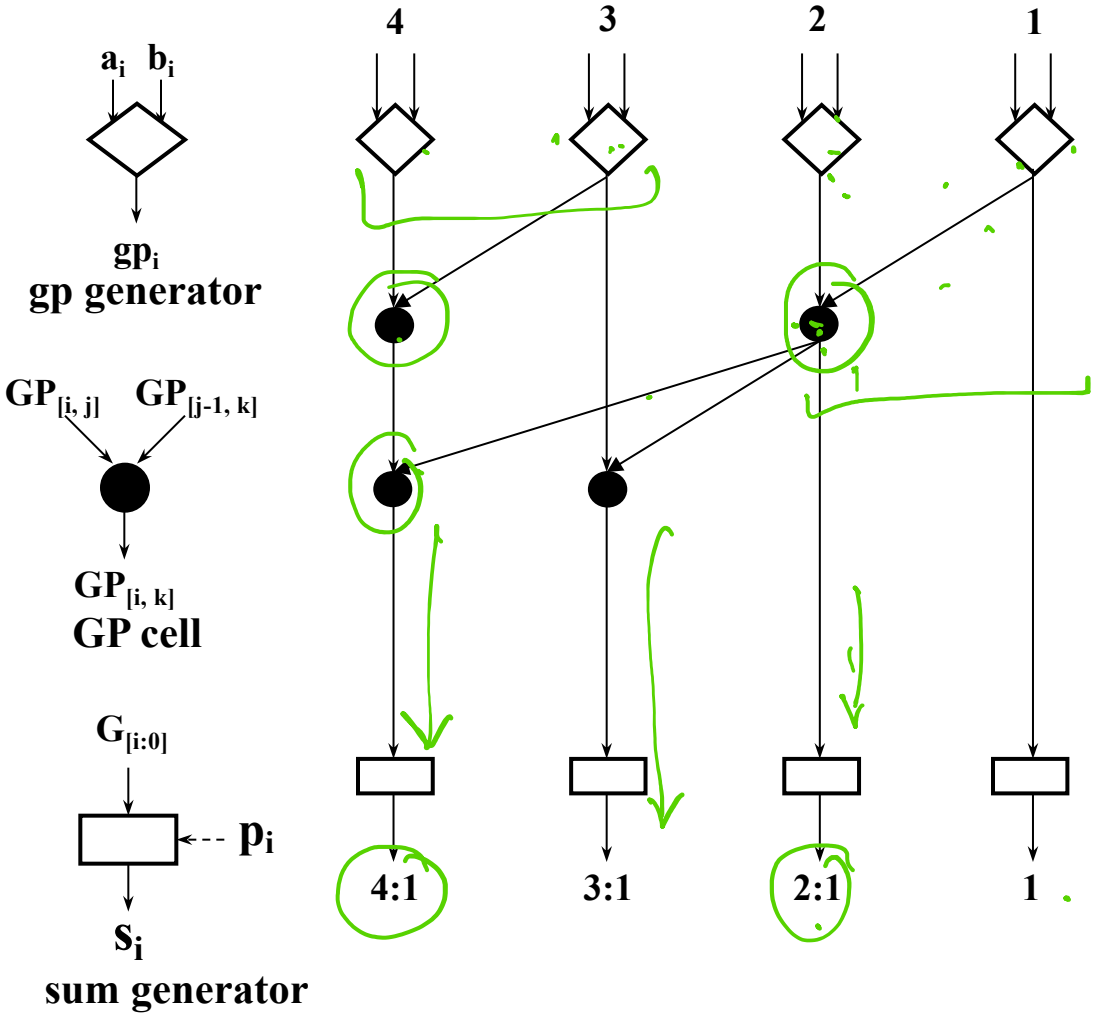
- Serial adder carry generation represented by prefix graphs



RCA

CADSL

Prefix Adder – Prefix Structure Graph

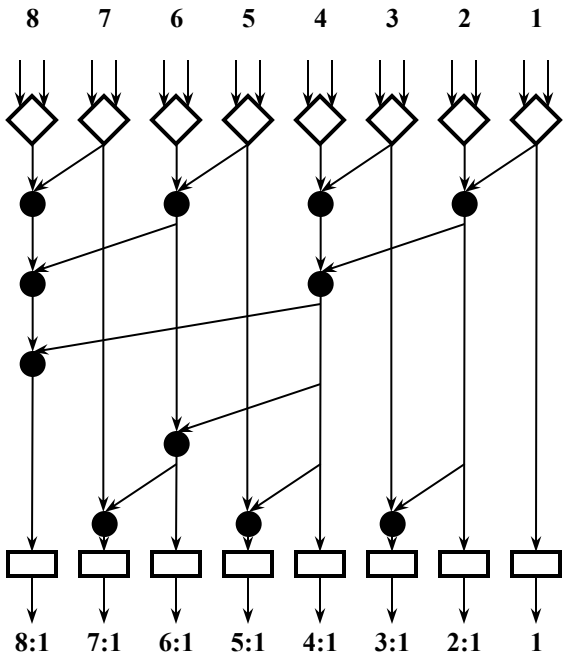


Pre-processing

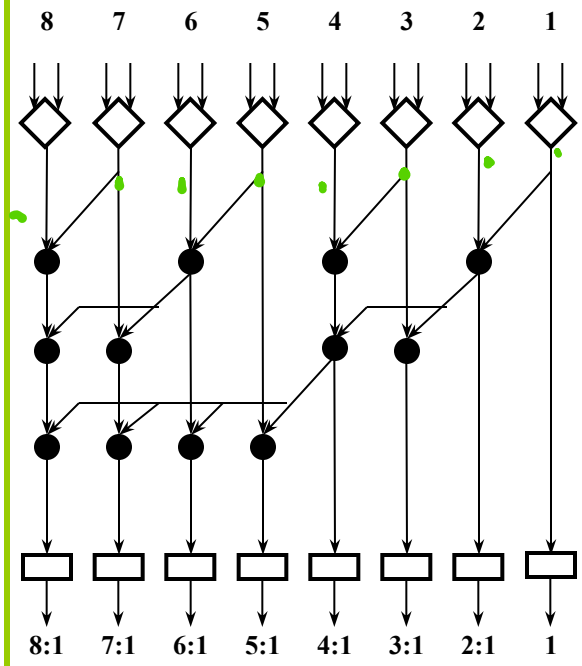
Prefix Computation

Post-processing

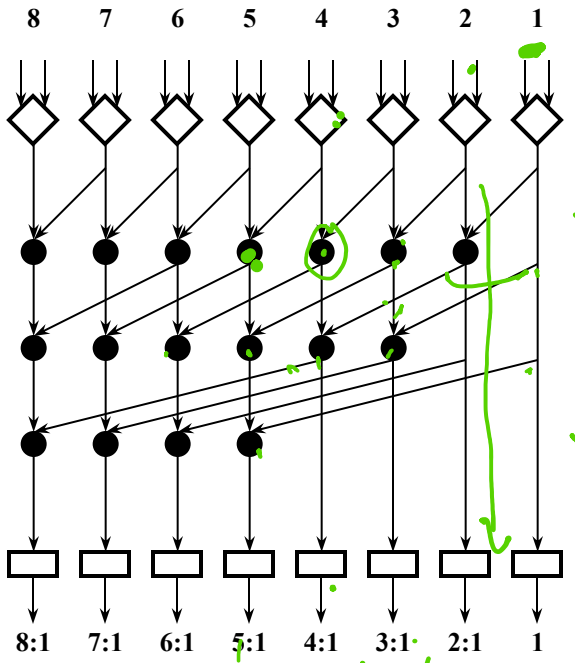
Classical Prefix Adders



Brent-Kung:
Logical levels: $2\log_2 n - 1$
Max fanouts: 2

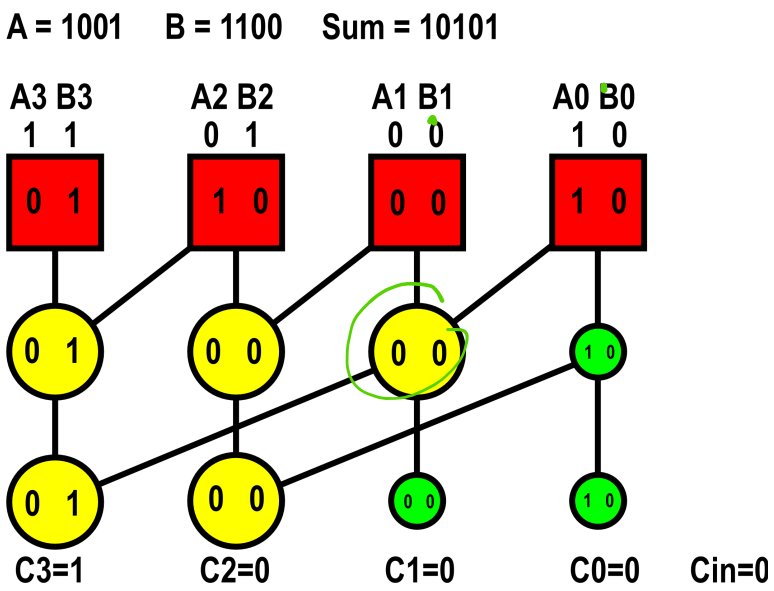


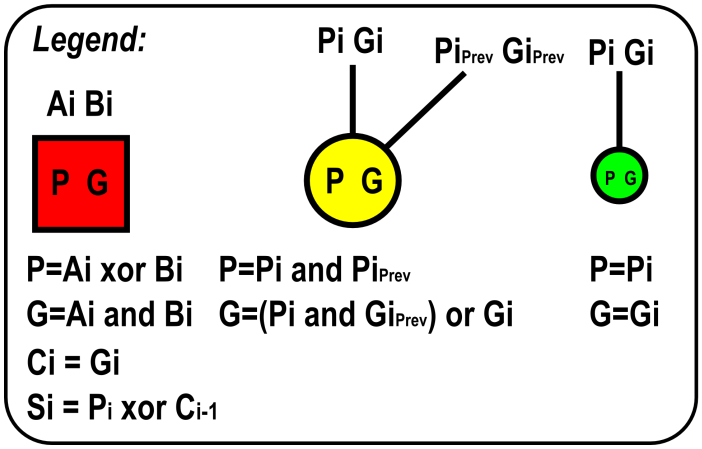
Sklansky:
Logical levels: $\log_2 n$
Max fanouts: n/2



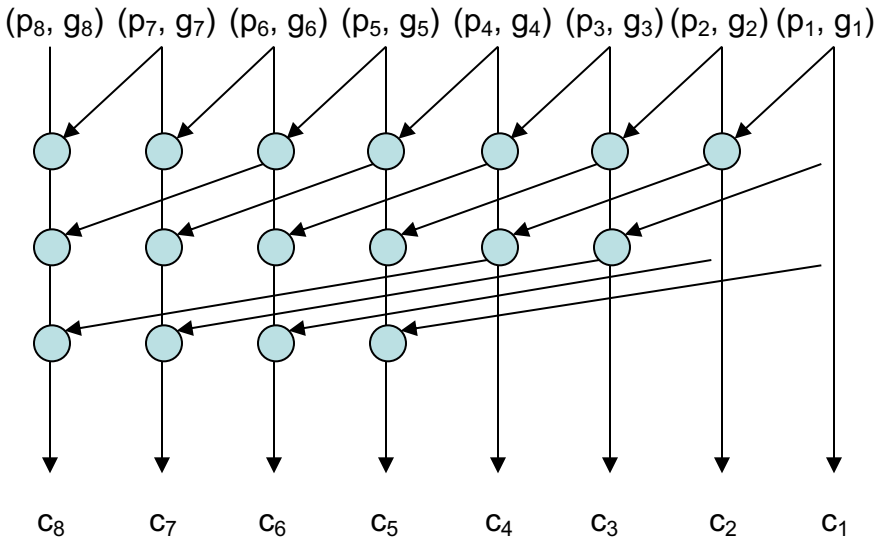
Kogge-Stone:
Logical levels: $\log_2 n$
Max fanouts: 2

Kogge Stone Adder



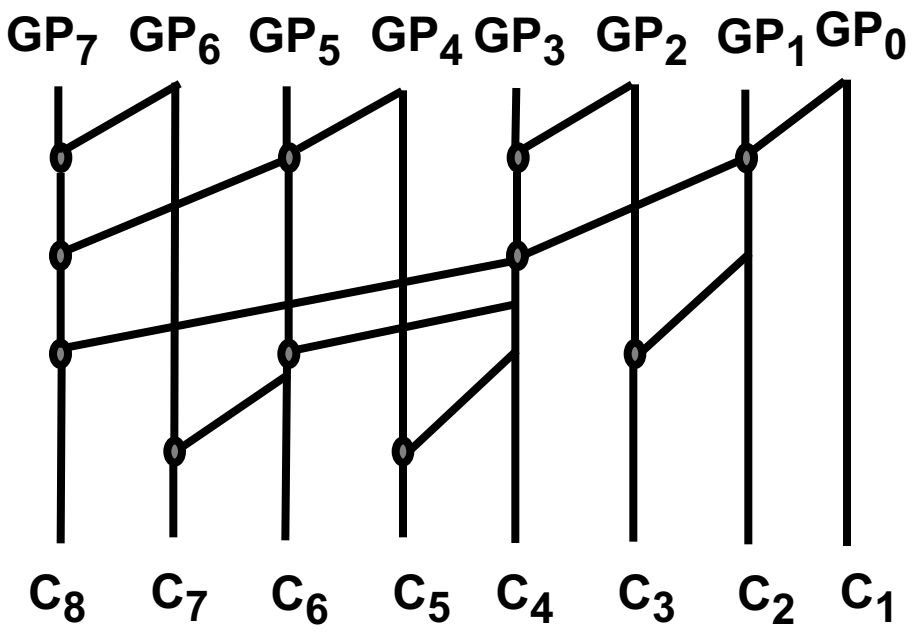


1973: Kogge-Stone adder



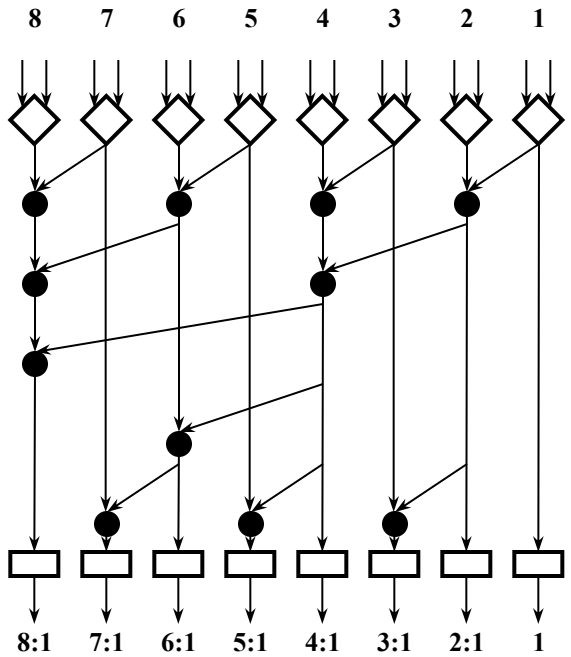
- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

1982: Brent-Kung (BK) Adder



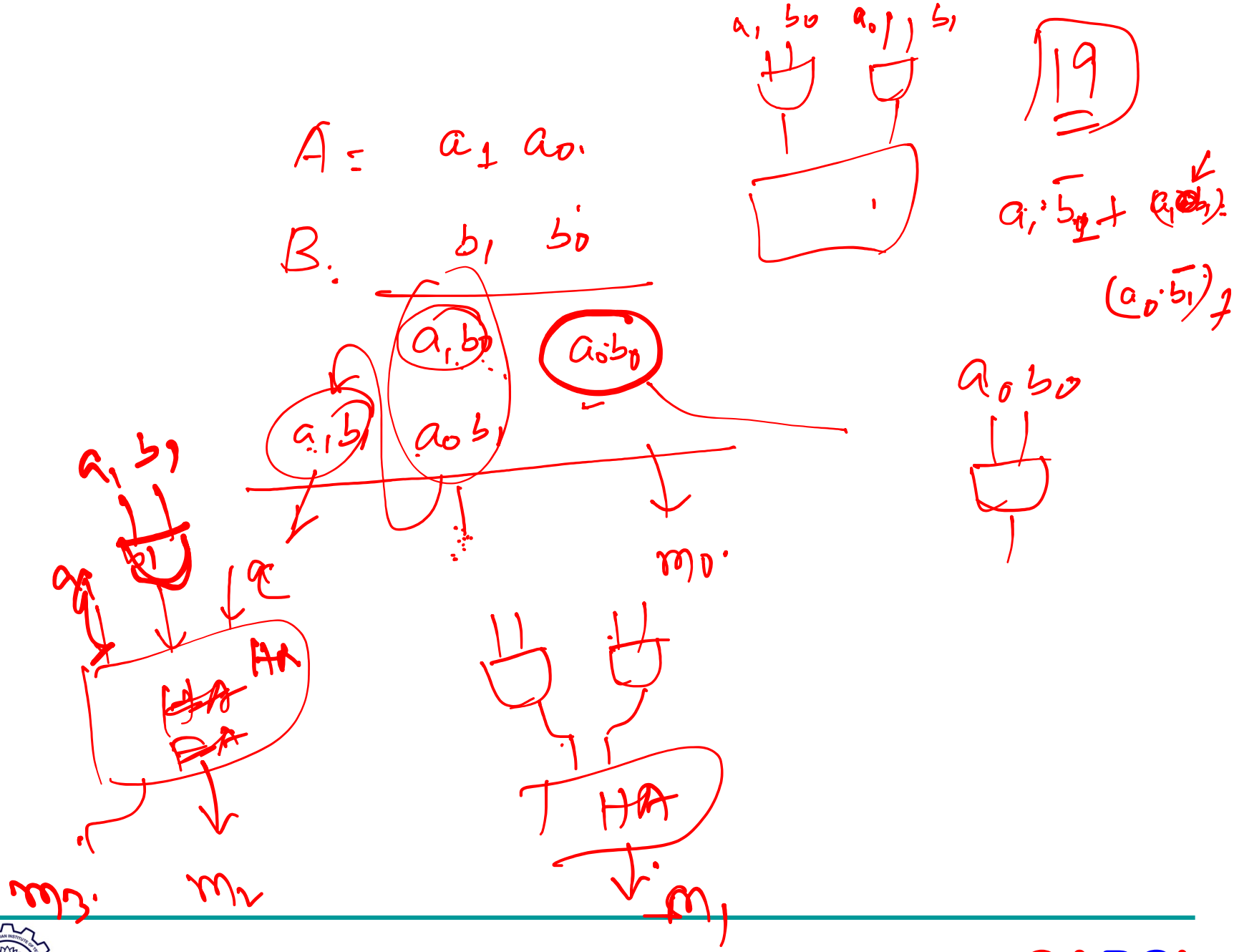
Brent and Kung, “A regular layout for parallel adders”, In IEEE transaction for Computers, 1982

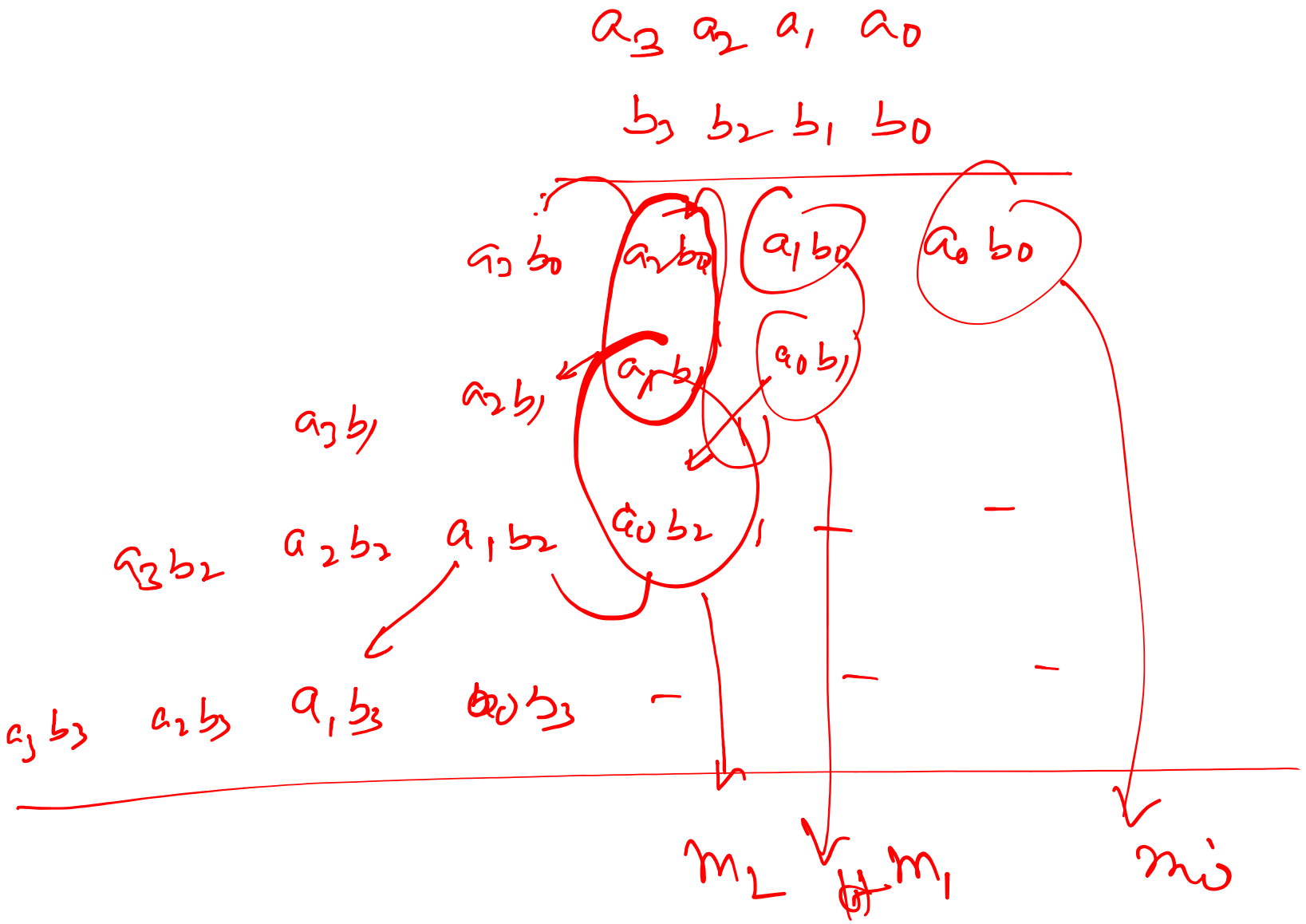
Brent-Kung (BK) Adder

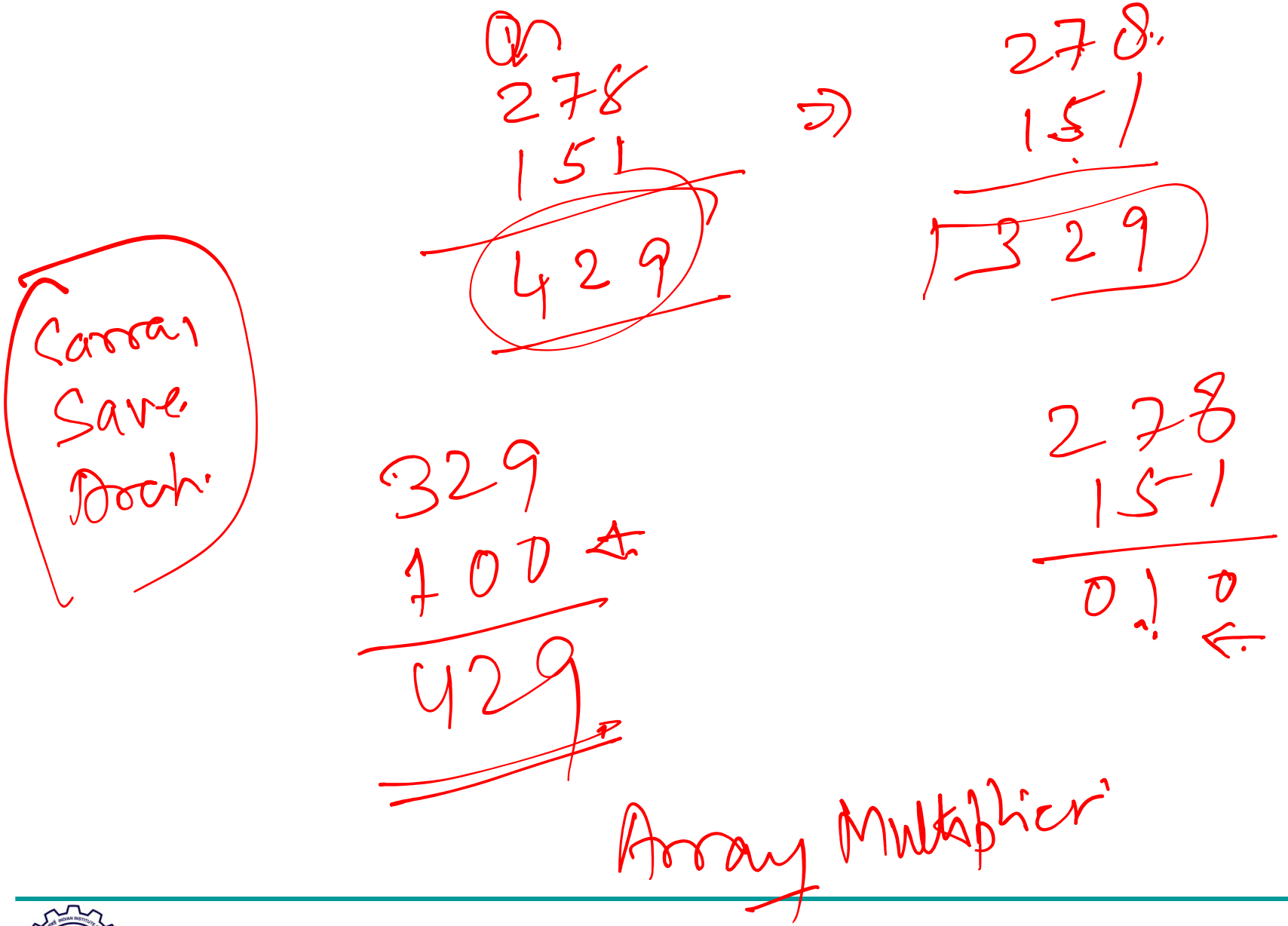


multiplier

A_i


A = a1 a0
B: b1 b0
a1 b0
a0 b1
a1 b1
a0 b0
m0
m3
m2
m1
HA
HA
HA
a1 b1
a0 b0
19
a1 · b0 + (a1 ⊕ b1) ·
(a0 · b1)'


$a_3\ a_2\ a_1\ a_0$
$b_3\ b_2\ b_1\ b_0$
a_3b_0
a_2b_0
a_1b_0
a_0b_0
a_2b_1
a_1b_1
a_0b_1
a_3b_1
a_3b_2
a_2b_2
a_1b_2
a_0b_2
a_3b_3
a_2b_3
a_1b_3
a_0b_3
m_2
m_1
m_0


278
151
429
278
151
329
Carry Save Book
329
100
429
278
151
010
Array Multiplier

Thank You