Lecture 4:

Adders

Computer Systems Laboratory
Stanford University
horowitz@stanford.edu

Overview

Reading

- HP Adder paper
- EE271 Adder notes if you have not seen them before
- Chandrakasen -- First part of Chapter 10

Introduction

Fast adders generally use a tree structure to take advantage of parallelism to make the adder go faster. We will talk about a couple of different tree adders in this lecture, and go over in more detail one of the adders. The adder described in the paper is even more complex than the one in the notes, and at the end we will talk a little about what it does.[1]

1. These notes will number the lsb as bit zero, which is the convention that I have used over the past 3 years.

Adders

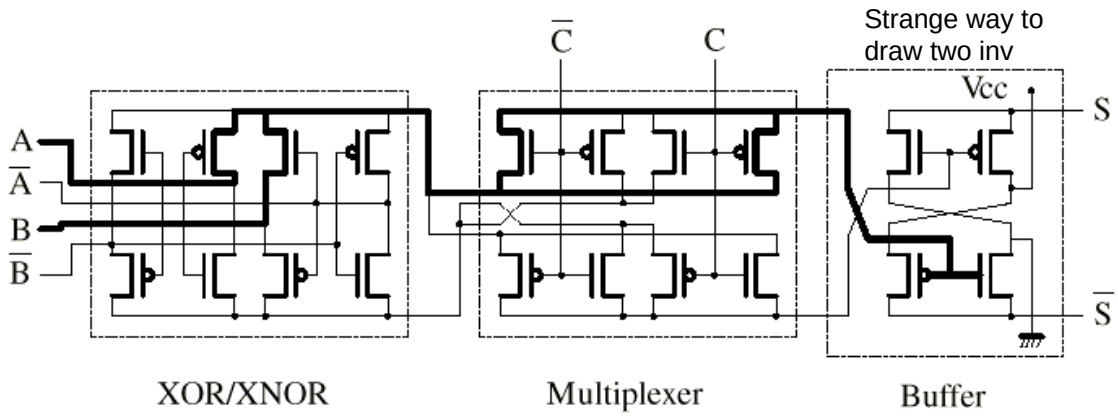
- N-bit adder sums two N-bit quantities (A & B) plus perhaps Cin
  - Sumi = A_i xor B_i xor C_i
  - Fundamental problem is rapidly calculating carry in to bit i (C_i)
  - All carries are dependent on all previous inputs
  - Therefore, least significant input has fanout of N, min delay $\log_4 N$ FO4 delays even if there were no logic.
- Most adders use Generate and Propagate logic to compute C_i
  - G = A AND B (C_{out} forced to be true)
  - P = A + B ($C_{out} = C_{in}$)
  - Also could use P = A XOR B, but OR is faster than XOR
  - Can combine G and P into larger blocks

    $G_{20} = G_1 + G_0P_1$

    $P_{20} = P_1P_0$

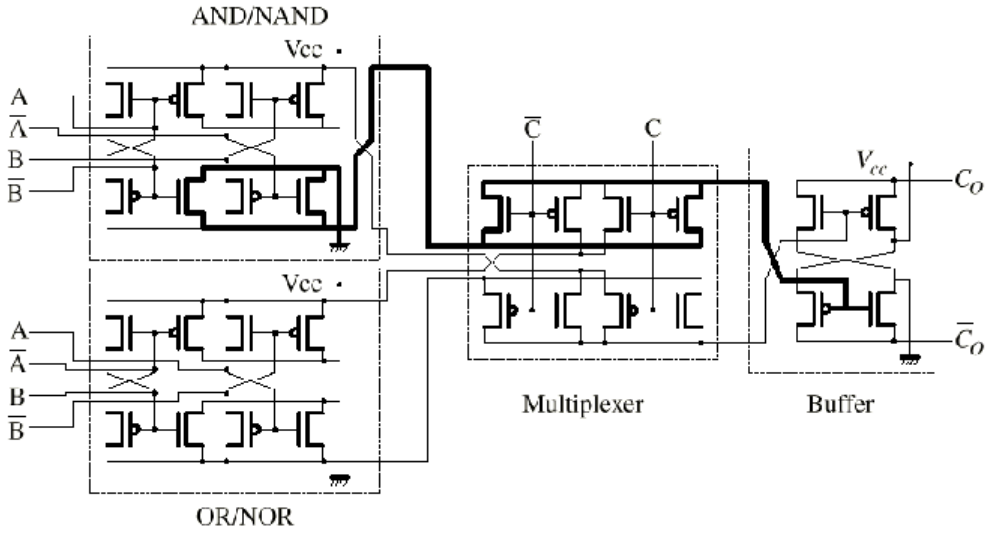
Adder Cell Design

- Lots of XORs which makes it interesting design problem
  - On of few places where pass-gate logic is an attractive



Pass-Gate Logic

- But even here it is still not great



Linear Solutions (see EE271 notes)

- Simplest adder: ripple carry, faster adders: carry look-ahead, carry bypass...
  - All of these work out carry several bits at a time
  - Best designs have around 11 FO4 delays for 64 bits
  - Useful for small adders (up to 16 bits) and moderate performance longer adders
  - Carry Bypass

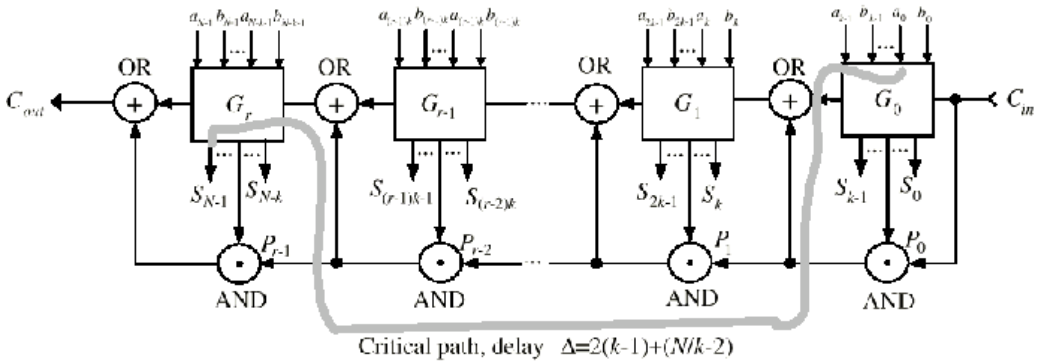
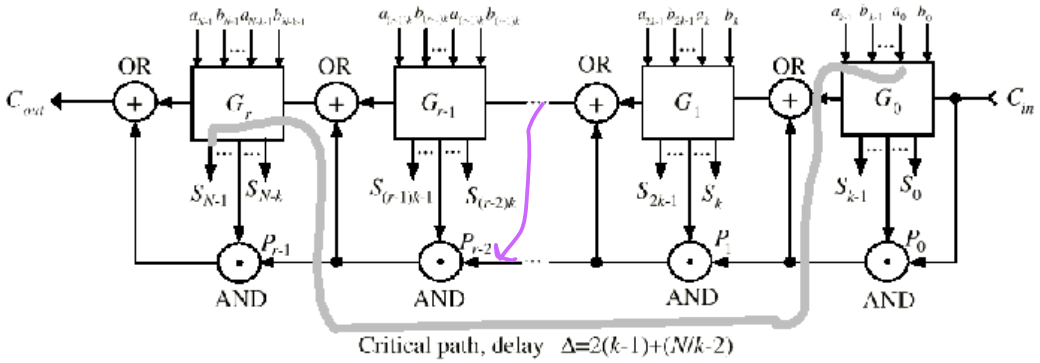


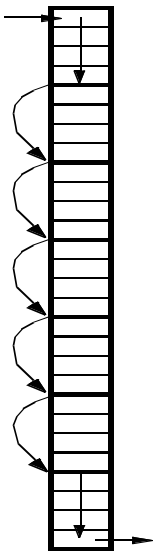
Figure is Not Quite Right

- Need to get group carry outputs into the global chain
  - AND gate for group is really (Pg*Cin + Gg)
    - Groups need to calculate Pg and Gg
  - Output of OR gate is the true input to the next group



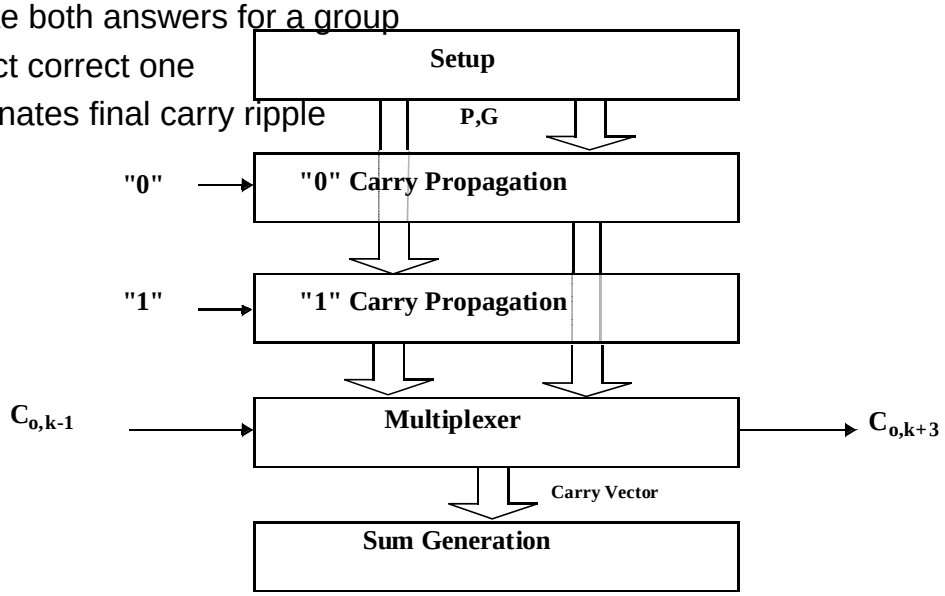
Carry Bypass or Carry Skip Adders

- Generally form multi-level carry chains
  - Break bits into group
  - Ripple carry in all groups in parallel
  - Ripple global carry
- Lots of tricks in choosing the number of bits for each group
  - Equalize the critical path
  - For the least significant ½ of the bits
    - Want group to generate carry out, when C of the global chain will arrive
  - For the most significant ½ of the bits
    - Also want for the last carry ripple in each group to occur at the same time
  - Means the groups should not be the same size



Carry Select

- Calculate both answers for a group
  - Select correct one
  - Eliminates final carry ripple



Many Papers on These Adders

But no one builds them any more

- Why?
  - They are all clever ways of building faster adders
    - While using little addition hardware
      - Bypass logic requires few gates
  - We have too many transistors
    - Don't need to be frugal with transistors
    - Just build the faster adder
- Interesting question
  - As power becomes a larger concern
    - Will these "simpler" adders reappear?

Logarithmic Solutions

- It would seem that to know C_i, we need C_i-1,
  - So delay is linear with N
  - A clever trick called “prefix computation” lets us turn this kind of problem into logarithmic delay since G and P are associative.

- Notation is always an issue
  - I will try to use the following strategy
  - $A\#_i$ means the signal ‘A’ for group size “#” in the i^{th} position
  - P = propagate (A+B)
  - G = generate (AB)
  - C = CarryIn to this bit/Group position

Tree Logic For Logarithm Time Adders

- Compute single bit values: (0 <= i < N)
  - $G_i = A_iB_i$, $P_i = A_i + B_i$
- Compute two bit groups: (0 <= i < N/2)
  - $G2_i = G_{2i+1} + G_{2i}P_{2i+1}$; $P_{2i} = P_{2i+1}P_{2i}$
- Compute four bit groups:
  - $G4_i = G2_{2i+1} + G2_{2i}P2_{2i+1}$ $P4_i = P2_{2i+1}P2_{2i}$ (0 <= i < N/4)
- ...(continue up binary tree to find all generates & propagates)
- ...(work down tree to find carry ins)
- $C4_{2i+1} = G4_{2i} + C8_iP4_{2i}$ $C4_{2i} = C8_i$
- $C2_{2i+1} = G2_{2i} + C4P2_{2i}$ $C2_{2i} = C4_i$
- $C_{2i+1} = G_{2i} + C2_iP_{2i}$ $C_{2i} = C2_i$

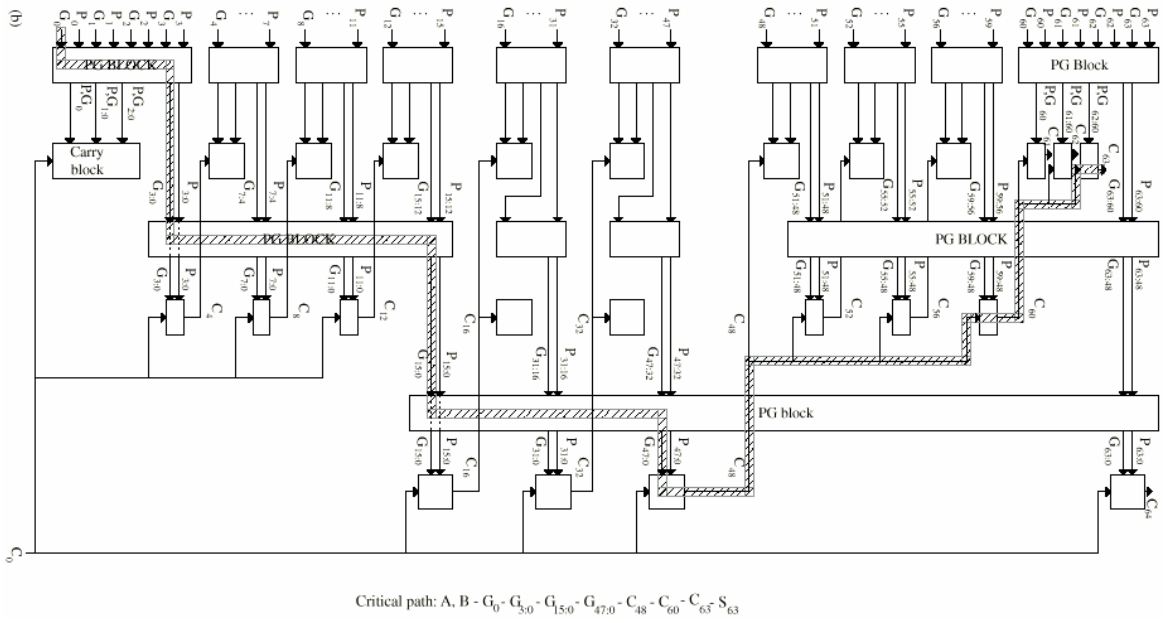
Many Types of Tree Adders

Two basic parameters

- Radix
- Density

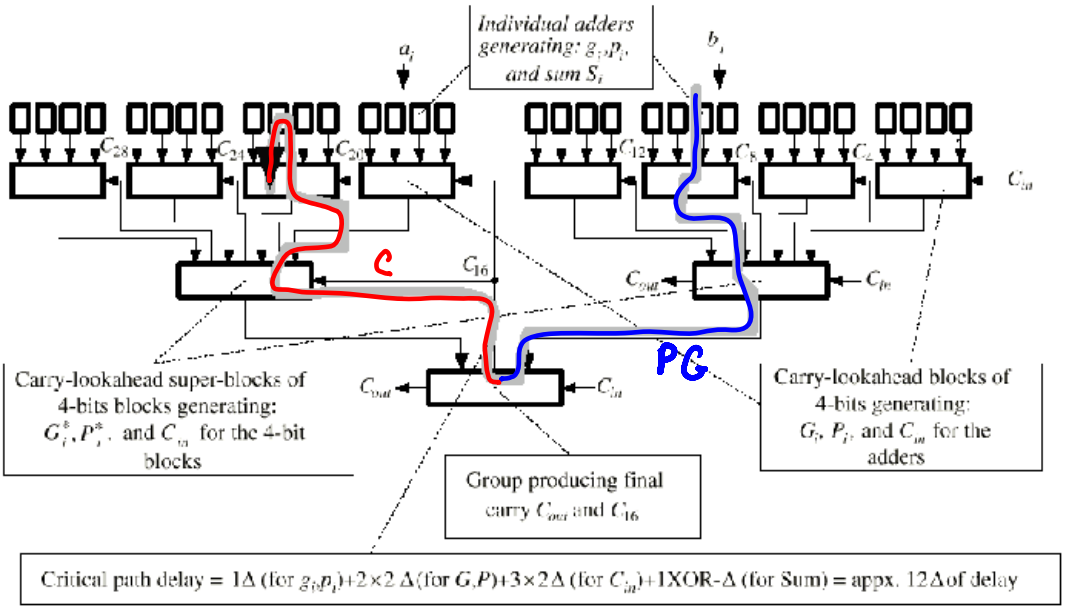
- Radix
  - Previous section showed a binary tree, radix 2
  - Can build other size trees, but generally it is 2 or 4
    - But not always!
    - Not always the same radix at each level
- Tree density
  - Previous slide actually needed two trees
    - One to generate PG information
    - One to distribute this information to all the consumers
  - Needed second tree since it did not compute all PGs it could

Radix 4 PG and Carry Trees

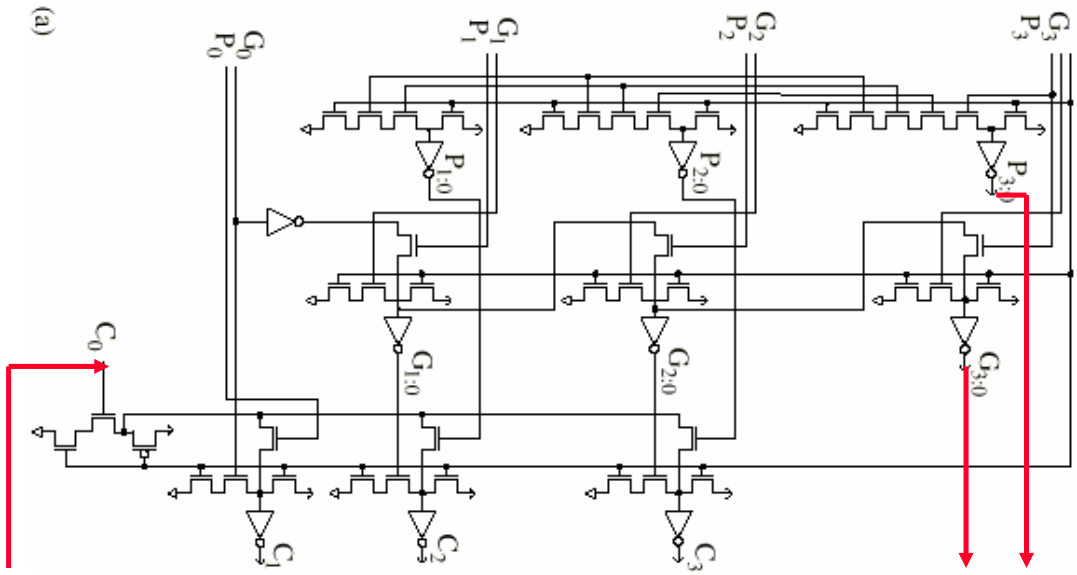


Critical path: A, B - G_0 - $G_{3:0}$ - $G_{15:0}$ - $G_{47:0}$ - C_{48} - C_{60} - C_{63} - S_{63}

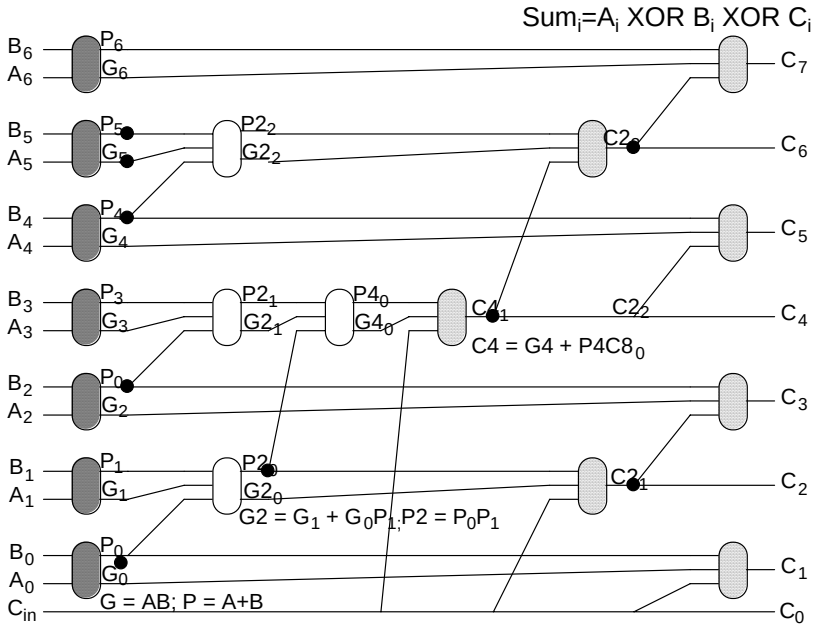
CLA Adder – Folded Tree



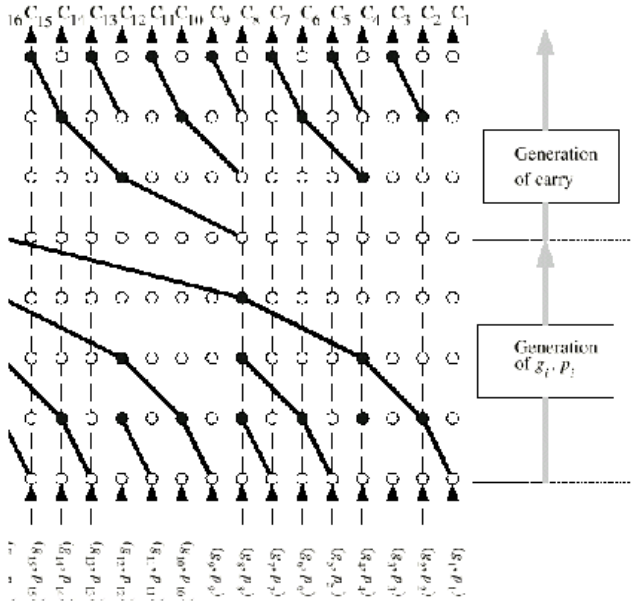
Dynamic Logic for 4 Bit CLA Block



8 Bit Binary Tree Adder



Tree Adder

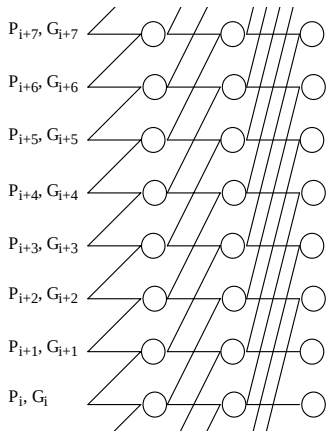


Tree Depth and Density

- In addition to changing the radix of the adder
  - Which we have seen in our examples
- The other main variable is depth and density of the trees
  - The difference between ‘n’ and ln’n’ is small for small numbers
    - Often the bottom of the Carry trees are chopped
    - Replaced with linear structures
  - Can compute more PG results going up the tree
    - Forest of trees
    - Don’t need a very large (or any C tree)
- In chopping trees
  - Many adders use carry select at the final stage
  - Compute two results, and use C4 or C8 to select right result
    - Group size in the select is small for fanout reasons

Binary Tree Adder

- Can eliminate the carry out tree by computing a group for each bit position:

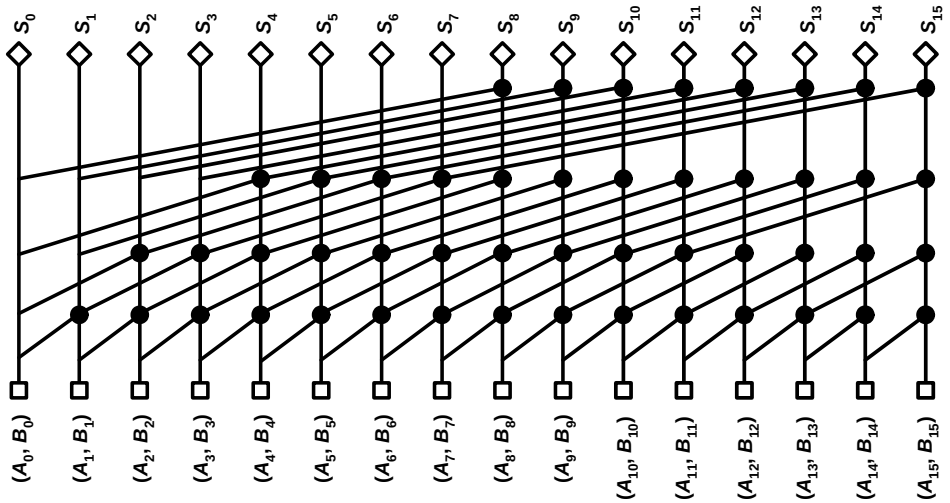


- Each circle has two gates, one that computes P for the group and one that computes G. This adder has a large number of wires, but can be very fast. In fact it provides a way to estimate what the min delay of an adder would be.

64 Bit Adder Delay

- Assume:
  - The output load is equal to the load on each input.
  - Using static gates (inverter is the fastest gate)
- Find delay from the effective fanout:
  - Simple approximation:
    - Must compute A_i xor B_i xor C_i
    - Cin (or LSB) must fanout to all bits (FO 64)
    - Total effective fanout 64 * 2 (for some logic in chain) (3.5 FO4 delays)
- More complex
  - Look at effective fanout of the path through adder
  - P gates drive 3 gates, G gates drive 2. Effective fanout is about 3.5/stage
  - 1.5 (first NAND/NOR) 3.5^6 * 1ish (final Mux for Sum optimize for late select)
  - 5.7 FO4 (not really accounting for parasitic delay correctly

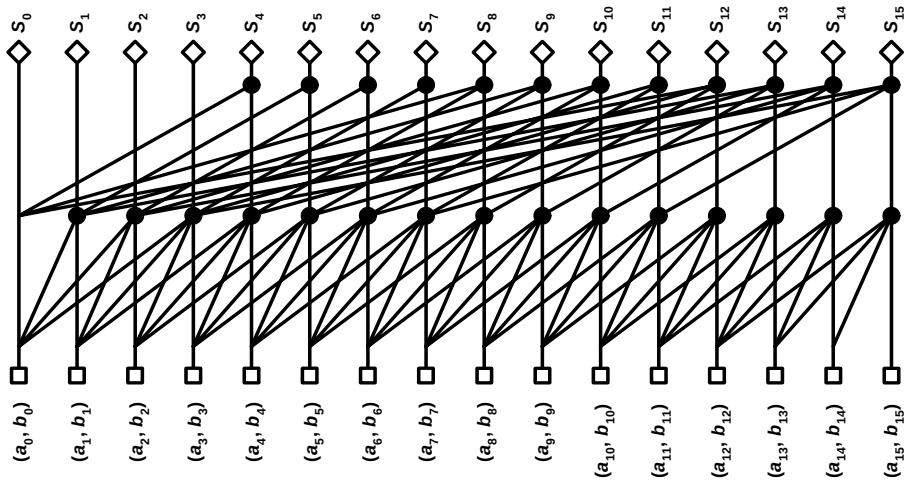
Tree Adders: Radix 2



16-bit radix-2 Kogge-Stone Tree

From Bora Nikolic

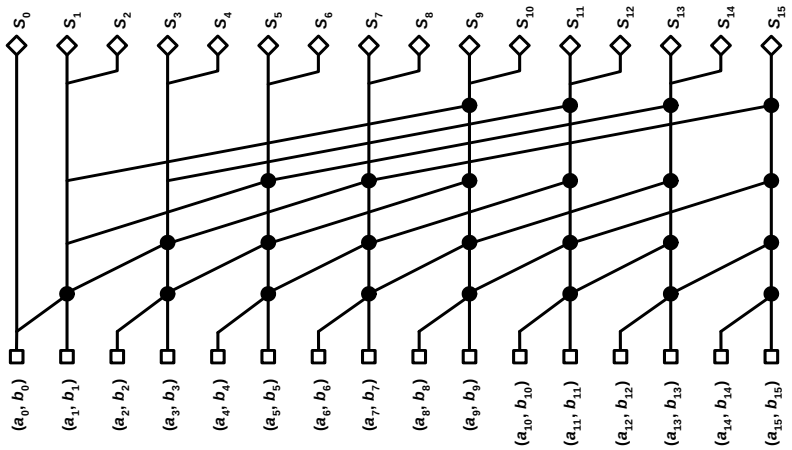
Tree Adders: Radix 4



From Bora Nikolic

16-bit radix-4 Kogge-Stone Tree

Sparse Trees

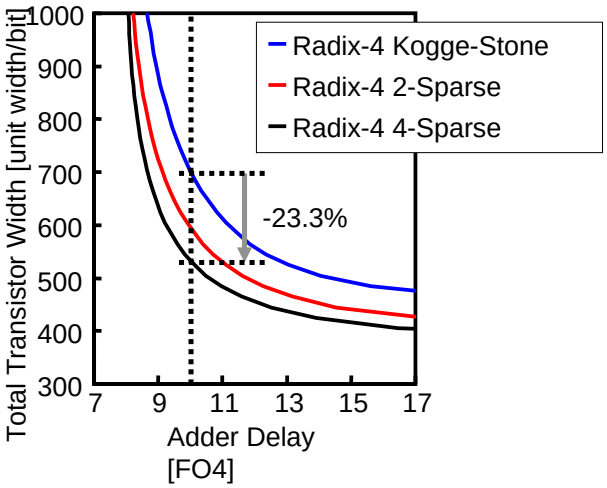


From Bora Nikolic

16-bit radix-2 sparse tree with sparseness of 2 (Han-Carlson)

Full vs. Sparse Trees

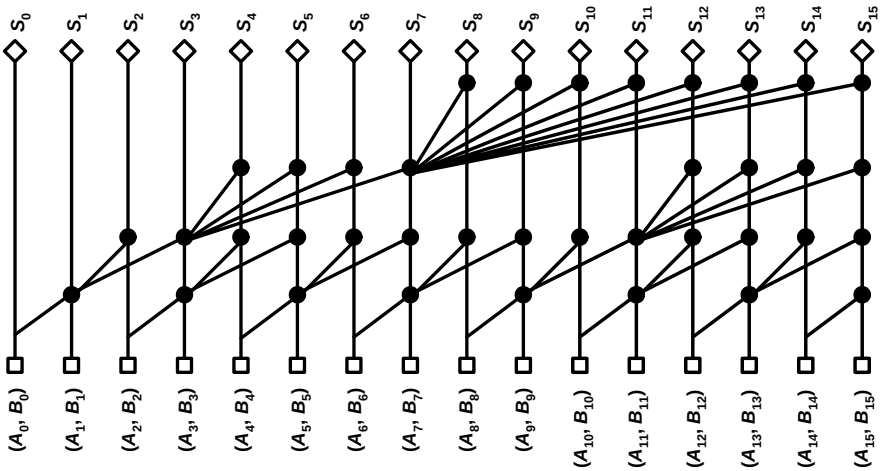
- Sparse trees have less transistors, wires
  - Less power
- Less input loading
- Recovering missing carries
  - Ripple (extra gate delay)
  - Precompute (extra fanout)
- Complex precompute can get into the critical path



From Bora Nikolic

Tree Adder with Growing Fanout

- Ladner-Fischer



From Bora Nikolic

Other Sparse Trees

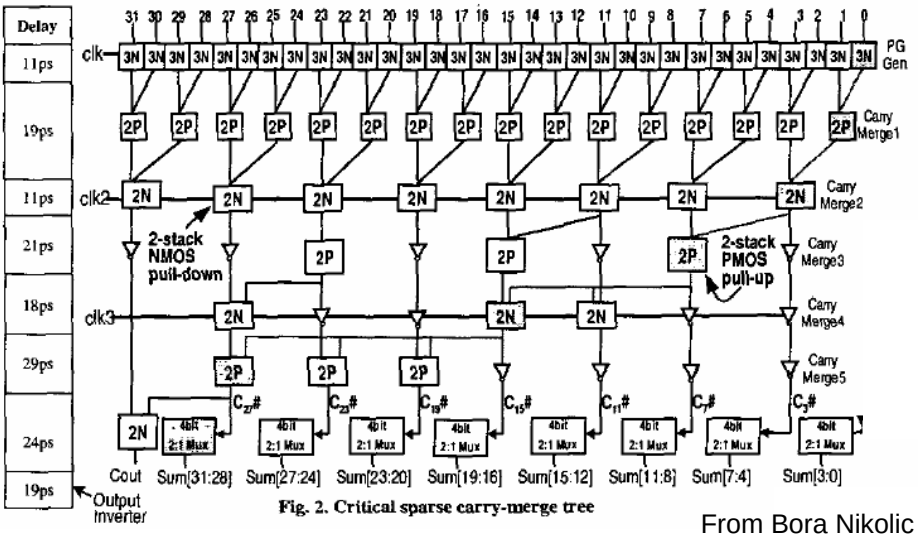


Fig. 2. Critical sparse carry-merge tree

From Bora Nikolic

Mathew, VLSI'02

Ling Adder

Variation of CLA

$$p_i = a_i \oplus b_i$$

$$g_i = a_i \cdot b_i$$

$$G_i = g_i + p_i \cdot G_{i-1}$$

$$S_i = p_i \oplus G_{i-1}$$

Ling's equations

$$t_i = a_i + b_i$$

$$g_i = a_i \cdot b_i$$

$$H_i = g_i + t_{i-1} \cdot H_{i-1}$$

$$S_i = t_i \oplus H_i + g_i t_{i-1} H_{i-1}$$

Ling, IBM J. Res. Dev, 5/81

From Bora Nikolic

Ling Adder

Conventional CLA:

$$G_i = g_i + p_i \cdot G_{i-1}$$

Also:

$$G_i = g_i + t_i \cdot G_{i-1}$$

Ling's equation shifts the index of pseudo carry

$$H_i = g_i + t_{i-1} \cdot G_{i-1}$$

Propagates information on two bits

From Bora Nikolic

Doran, Trans on Comp 9/88

Ling Adder

Conventional radix-4

$$G_3 = g_3 + t_3 g_2 + t_3 t_2 g_1 + t_3 t_2 t_1 g_0$$

Ling radix-4

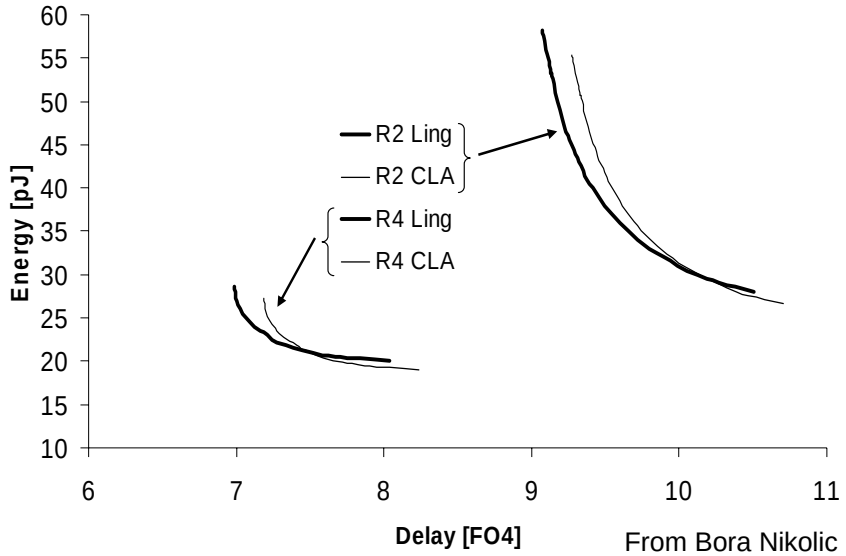
$$H_3 = g_3 + t_2 g_2 + t_2 t_1 g_1 + t_2 t_1 t_0 g_0$$

$$= g_3 + g_2 + t_2 g_1 + t_2 t_1 g_0$$

Reduces the stack height (or width)
Reduces input loading

From Bora Nikolic

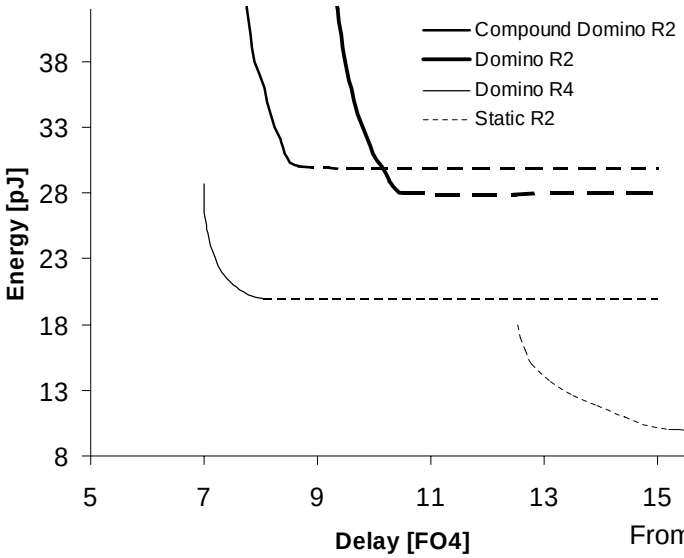
Ling vs. CLA



From Bora Nikolic

R. Zlatanovici, ESSCIRC'03

Static vs. Dynamic



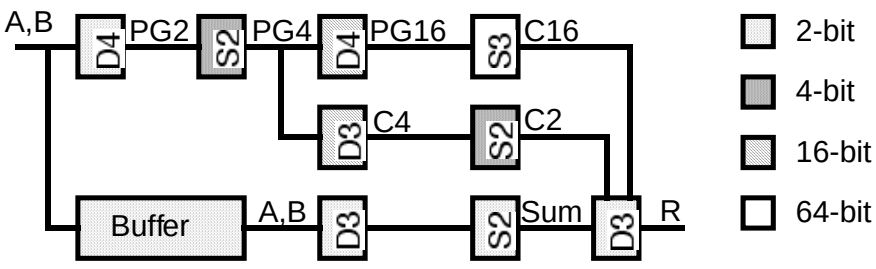
From Bora Nikolic

Example

- Look at a real design of a 64-bit adder (by David Harris)
  - Dual-level carry-select adder, radix 4 (kind of)
  - Fully dual-rail domino
  - Performs only addition (input inverted in bypass network for subtract)
  - Delay (simulated) = 6.4 FO4 delays
- Design Issues
  - Adder architecture
  - Gate sizing

Architecture

- Logarithmic adder must generate up tree, then send carry down tree
- Carry Select adder cuts delay by only finding carry into block of 16
  - Meanwhile, sums must be computed for block assuming C_{in} = 0, 1
  - This also requires a very fast 16 bit adder
- Reuse the same trick: build a 16 bit carry select adder
  - 16 bit adder can share generate and propagate logic with main adder



2-Bit Logic

- 2-bit Propagates & Generates
  - P2 = P_0P_1 =
  - G2 = $G_1+G_0P_1$ =
- Speculative Sums to bit b assuming carry in c: $sumc_b$
  - $sum0_0$ =
  - $sum1_0$ =
  - $sum0_1$ =
  - $sum1_1$ =
- Final Result
  - R_0 =
  - R_1 =

4-bit Logic

- Carry in to 2 bit block b assuming cin to 16 bit block is c: $cin2c_b$
  - $cin20_0$ =
  - $cin21_0$ =
  - $cin20_1$ =
  - $cin11_1$ =
- 4-bit Propagates and Generates
  - g4 =
  - p4 =

16-bit Logic

- Carry in to 4 bit block b assuming cin to 16 bit block is c: $cin4c_b$
  - $cin40_0$ =
  - $cin41_0$ =
  - $cin40_1$ =
  - $cin41_1$ =
  - $cin40_2$ =
  - $cin41_2$ =
  - $cin40_3$ =
  - $cin41_3$ =
- 16-bit Propagates and Generates
  - g16 =
  - p16 =

64-bit Logic

- Carry in to 16 bit block b assuming cin to 16 bit block is c: $cin16_b$[1]
- $cin16_0$ =
- $cin16_1$ =
- $cin16_2$ =
- $cin16_3$ =

1. Assumes no carry in to 64 bit adder. This might be relaxed to handle subtraction at the expense of one more series transistor in the critical path.

Domino Implementation

- Notice that all the P, G, and C terms are monotonic!
- Only the sum select mux needs complementary inputs
  - Option 1: Generate single rail C, use static mux to select sum
    - Requires static mux plus inverter for C, C_b mux select
    - Good for area
    - Delay = 6.7 - 7.3 FO4 delays
  - Option 2: Generate dual rail P, G, C, use domino mux to select sum
    - Requires twice as much hardware
    - Good for speed
    - Can produce dual-rail output in domino form for later circuits
    - Delay = 6.4 FO4 delays

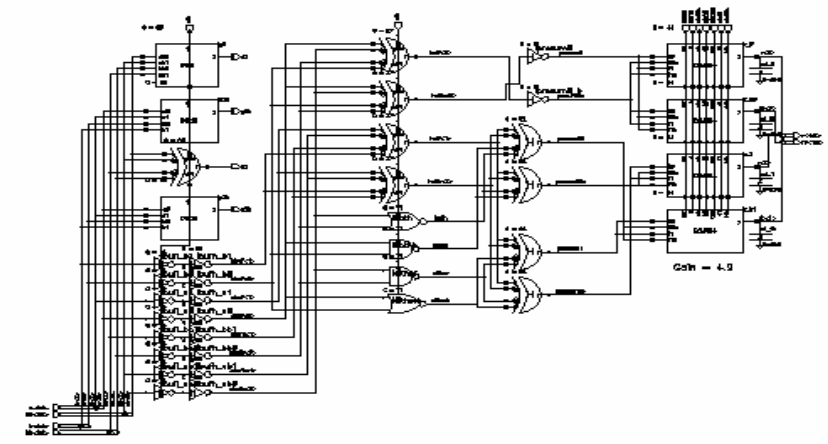
Domino Tricks

- A bunch of domino tricks were used for maximum speed
  - Full utilization of static as well as domino gates
  - No clocked pulldowns on some domino gates
    - Needs delayed precharge clocks, so make sure it is worth it
  - Precharge necessary internal nodes
    - Predischarge other internal nodes when all transistors are off
    - Another speed hack that might not be worth it
  - No keepers
    - Living dangerously, probably not a good idea
- Skewed for fast eval at cost of slow precharge
- Some of these conflict with robust design principles

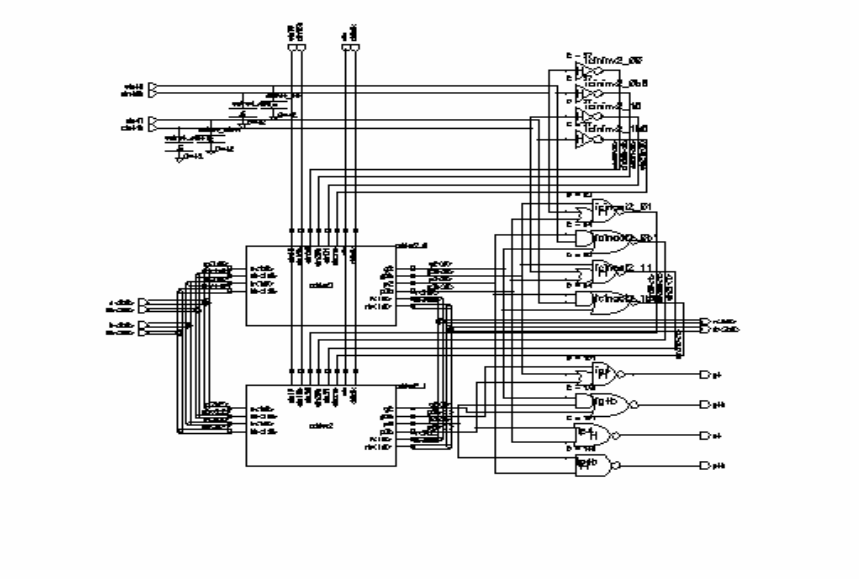
Naming Conventions

- Very important to name all gates and nodes
  - Node names necessary when debugging internal signals
  - Gates needed for same reason
  - Gate types:
    - D1 = Domino with evaluation transistor
    - D2 = Domino with no evaluation transistor
    - H = High skew CMOS
    - L = Low skew CMOS
  - Clocks:
    - clk: normal clock
    - dlclk: delayed low (falling) clock (for D2 delayed precharge)
    - ddlclk: doubly delayed low clock

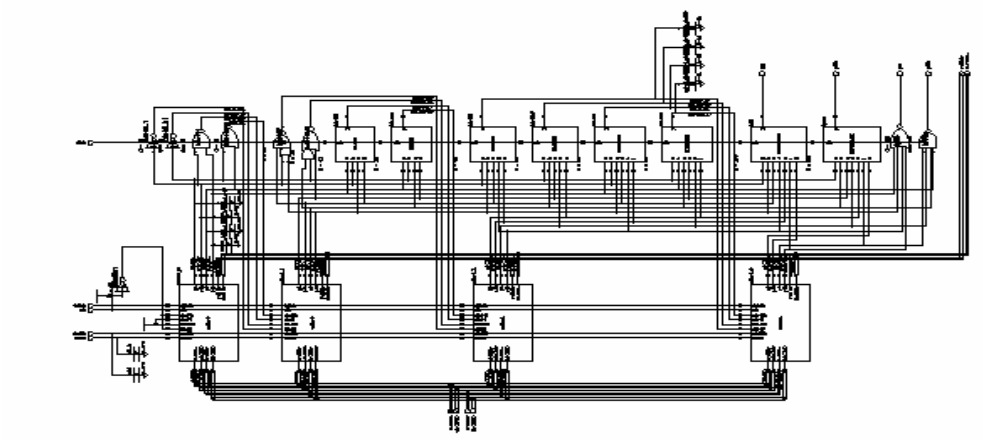
2 Bit Adder Block



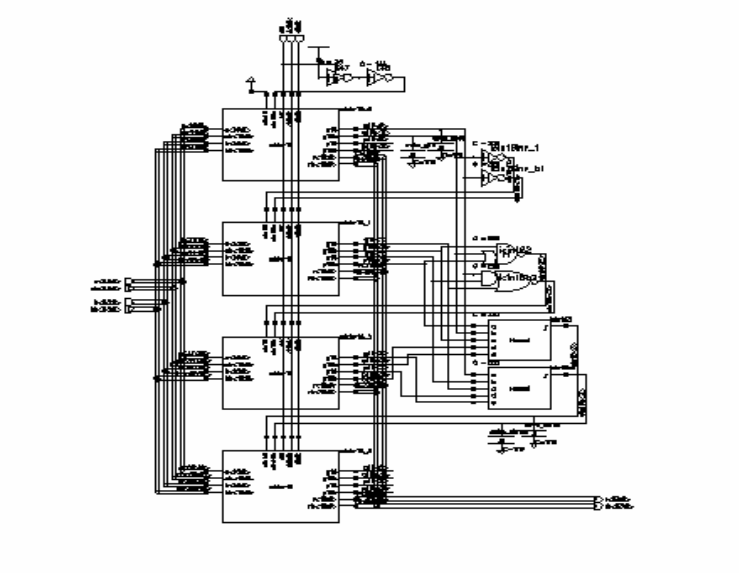
4 Bit Adder Block



16 Bit Adder Block



64 Bit Adder Block



2-Bit Logic

- 2-bit Propagates & Generates
  - $P2 = P_0P_1 = (A_0+B_0)(A_1+B_1)$
  - $G2 = G_1+G_0P_1 = A_1B_1 + A_0B_0\ (A_1+B_1)$
- Speculative Sums to bit b assuming carry in c: $sumc_b$
  - $sum0_0 = A_0$ xor B_0
  - $sum1_0 = \sim(A_0$ xor $B_0)$
  - $sum0_1 = A_1$ xor B_1 xor (A_0B_0)
  - $sum1_1 = A_1$ xor B_1 xor $(A_0 + B_0)$
- Final Result
  - R_0 = cin16 ? (cin21 ? $sum1_0$: $sum0_0$) : (cin20 ? $sum1_0$: $sum0_0$)
  - R_1 = cin16 ? (cin21 ? $sum1_1$: $sum0_1$) : (cin20 ? $sum1_1$: $sum0_1$)

4-bit Logic

- Carry in to 2 bit block b assuming cin to 16 bit block is c: $cin2c_b$
  - $cin20_0 = cin40$
  - $cin21_0 = cin41$
  - $cin20_1 = g2_0 + p2_0 cin40$
  - $cin11_1 = g2_0 + p2_0 cin41$
- 4-bit Propagates and Generates
  - $g4 = g2_1 + p2_1 g2_0$
  - $p4 = p2_0 p2_1$

16-bit Logic

- Carry in to 4 bit block b assuming cin to 16 bit block is c: $cin4c_b$
  - $cin40_0 = 0$
  - $cin41_0 = 1$
  - $cin40_1 = g4_0$
  - $cin41_1 = g4_0 + p4_0$
  - $cin40_2 = g4_1 + p4_1 (g4_0)$
  - $cin41_2 = g4_1 + p4_1 (g4_0 + p4_0)$
  - $cin40_3 = g4_2 + p4_2 (g4_1 + p4_1 (g4_0))$
  - $cin41_3 = g4_2 + p4_2 (g4_1 + p4_1 (g4_0 + p4_0))$
- 16-bit Propagates and Generates
  - $g16 = g4_3 + p4_3 (g4_2 + p4_2 (g4_1 + p4_1 g4_0))$
  - $p16 = p2_0 p2_1 p2_2 p2_3$

64-bit Logic

- Carry in to 16 bit block b assuming cin to 16 bit block is c: $cin16_b$
  - $cin16_0 = 0$
  - $cin16_1 = g16_0$
  - $cin16_2 = g16_1 + p16_1 (g16_0)$
  - $cin16_3 = g16_2 + p16_2 (g16_1 + p16_1 (g16_0))$