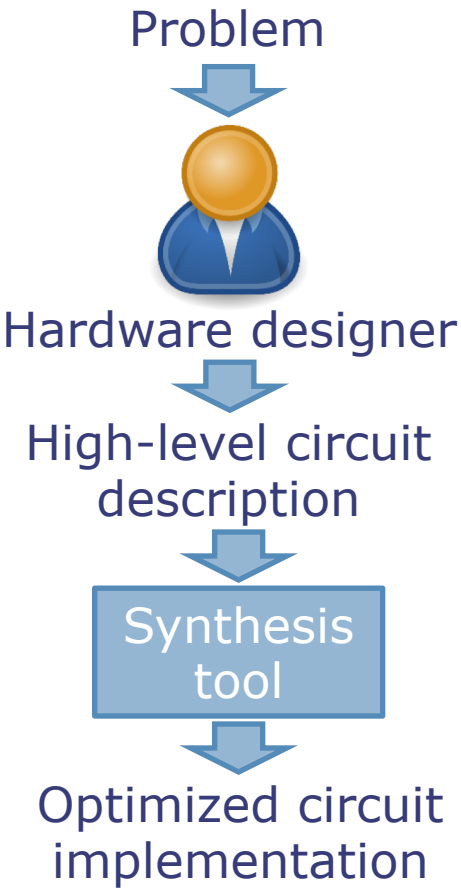
Design Tradeoffs in Arithmetic Circuits

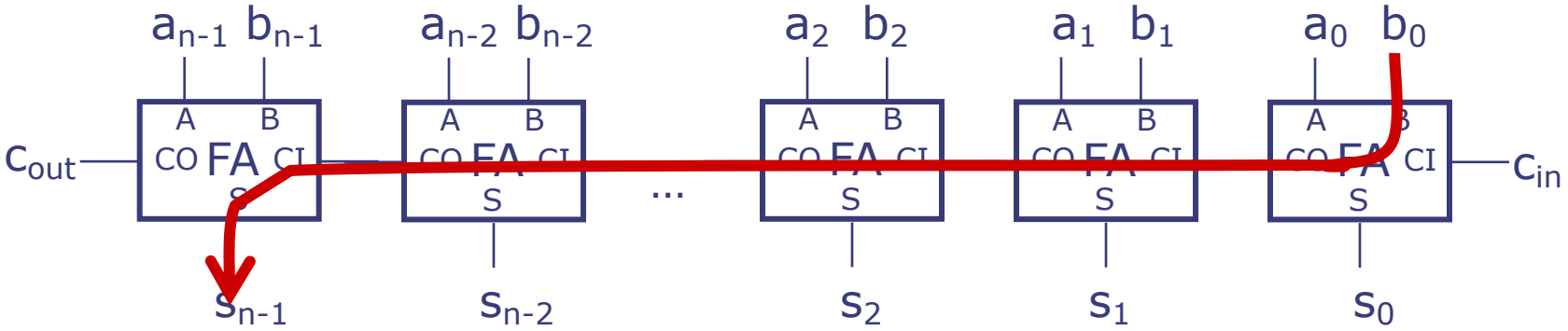
Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs
- Choosing the right algorithms is key to optimizing your design
  - Tools cannot compensate for an inefficient algorithm (in most cases)
  - Just like programming software
- Case study: Building a better adder



Ripple-Carry Adder: Simple but Slow



- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001

$$t_{PD} = (n\text{-}1)*t_{PD,CI\rightarrow CO} + t_{PD,CI\rightarrow S} \quad \approx \Theta(n)$$

- $\Theta(n)$ is read "order n" and tells us that the latency of our adder grows linearly with the number of bits of the operands

Asymptotic Analysis

- Suppose some computation takes n^2+2n+3 steps
- We say it takes $\Theta(n^2)$ (read "is of order n^2") steps
- Why?

  because $2n^2 > n^2+2n+3 > n^2$
  except for a few small integers (1,2 and 3)

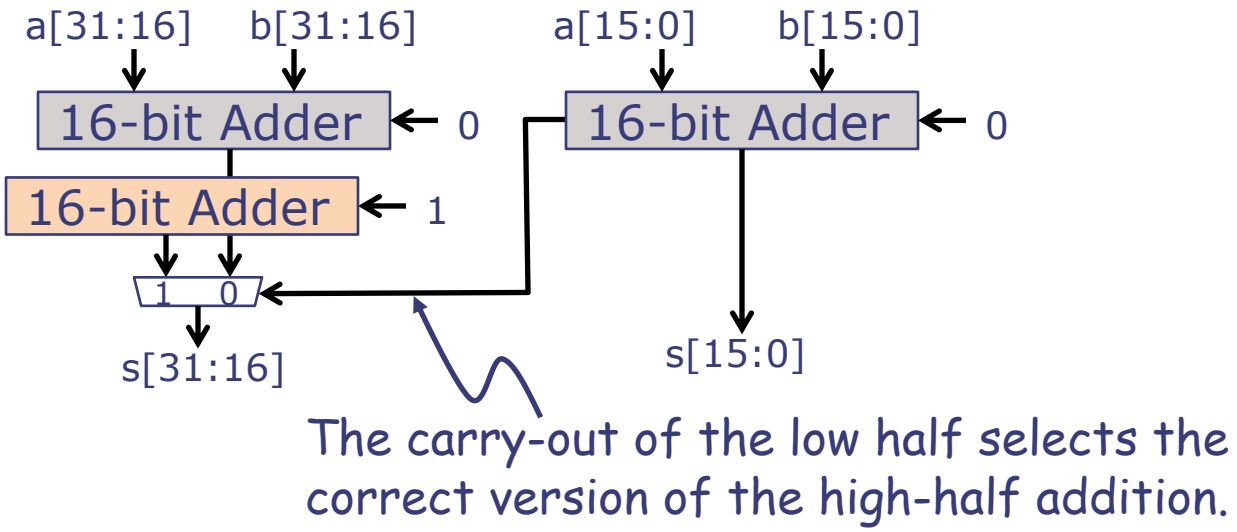
- Formally, $g(n)=\Theta(f(n))$ iff there exist $C_2 \geq C_1 > 0$ such that for all but *finitely many* integers $n \geq 0$,

$$\underbrace{C_2 \cdot f(n) \geq g(n)} \geq C_1 \cdot f(n)$$

$g(n) = O(f(n))$

$\Theta(...)$ implies both inequalities;
$O(...)$ implies only the first.

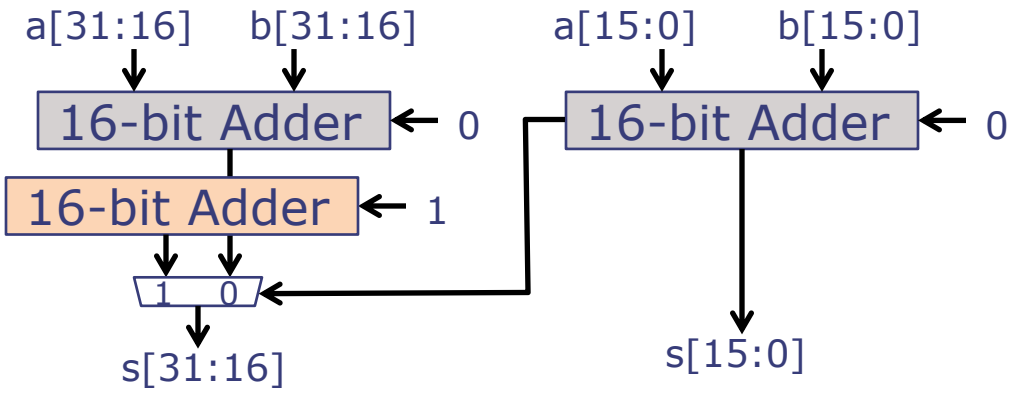
Carry-Select Adder Trades Area for Speed



- Propagation delay: $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder

Drawbacks? *Consumes much more area than ripple-carry adder*
Wide mux adds significant delay

Carry-Select Adder Trades Area for Speed




```
function Bit#(33) csa32(Bit#(32) a, Bit#(32) b, Bit#(1) c);
  let csL = add16(a[15:0], b[15:0], c);
  let csU = (csL[16] == 0) ? add16(a[31:16], b[31:16], 0)
                           : add16(a[31:16], b[31:16], 1);
  return {csU,csL[15:0]};
endfunction
```

We can use any type of adder, including CSA

Recursive Carry-Select Adder

- It is easier to write the code than to draw a picture

```
function Bit#(33) csa32(Bit#(32) a, Bit#(32) b, Bit#(1) c);
  let csL = csa16(a[15:0], b[15:0], c);
  let csU = (csL[16] == 0) ? csa16(a[31:16], b[31:16], 0)
                           : csa16(a[31:16], b[31:16], 1);
  return {csU,csL[15:0]};
endfunction

// following functions can be defined similarly
function Bit#(17) csa16(Bit#(16) a, Bit#(16) b, Bit#(1) c);
function Bit#(9) csa8(Bit#(8) a, Bit#(8) b, Bit#(1) c);
function Bit#(5) csa4(Bit#(4) a, Bit#(4) b, Bit#(1) c);
function Bit#(3) csa2(Bit#(2) a, Bit#(2) b, Bit#(1) c);
// let csa2 use fa instead of csa1
```

CSA analysis

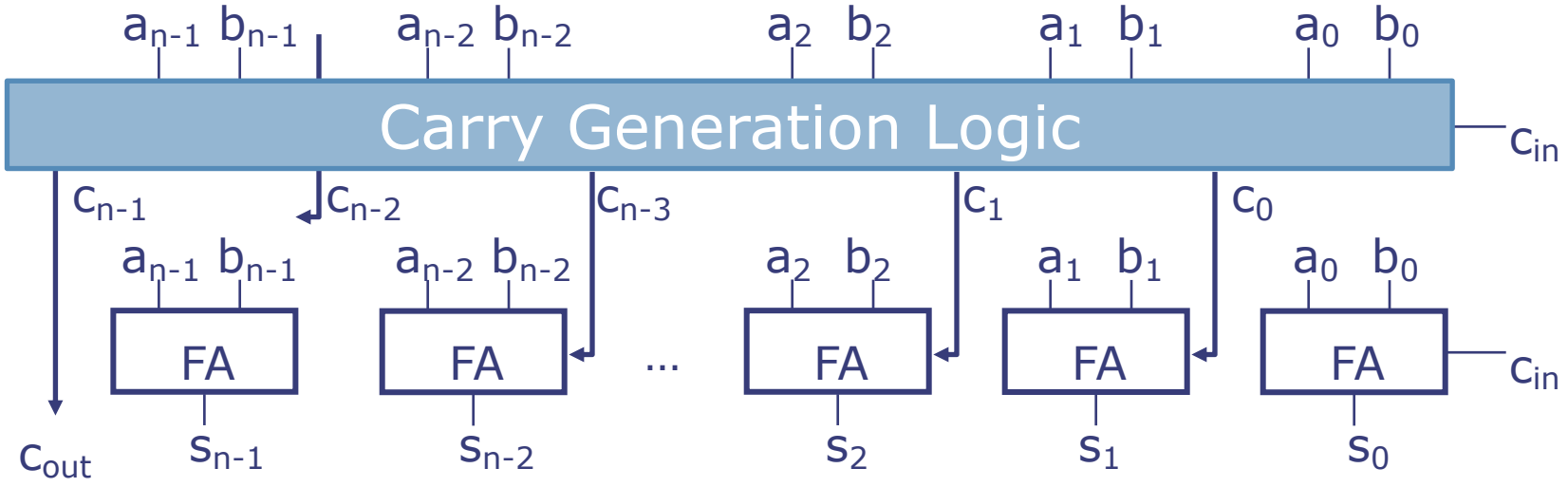
- Carry delay $t_{PD,n}=\Theta(\log n)$ but lots of extra hardware
- Area calculation
  - csa32 = 3 csa16 + mux17
  - csa16 = 3 csa8 + mux9
  - csa8 = 3 csa4 + mux5
  - csa4 = 3 csa2 + mux3
  - csa2 = 3 fa + mux2
  - fa = 5 gates
  - muxn = 3n+1 gates

32-RCA has an area of 32 fa = 32X5 = 160 gates

Wait until the end for some synthesis results

- Total gates in csa32
  - 243 fa + 81 mux2 + 27 mux3 + 9 mux5 + 3 mux9 + mux17
  - = 2339 gates
- Lots of circuits are duplicated but Boolean optimizations by the compiler may reduce the size of the circuit
  - Many of the 243 fa's have the same inputs

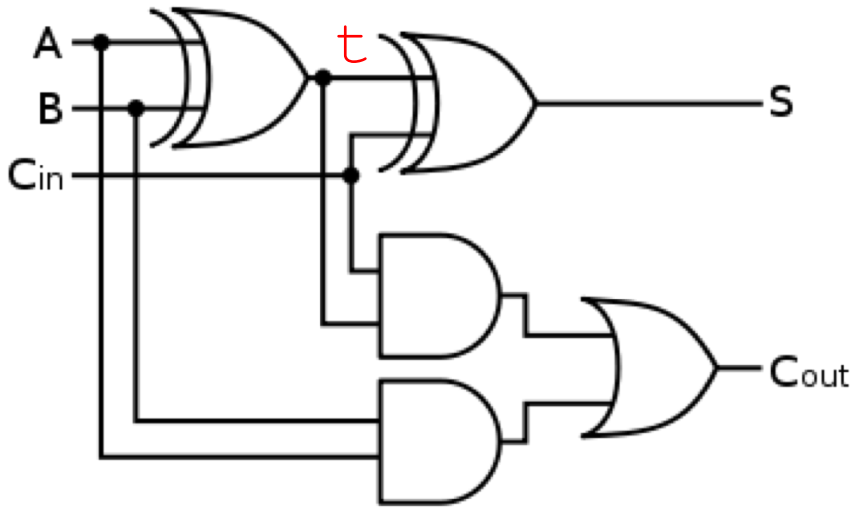
Carry-Lookahead Adders (CLAs)



- CLAs compute all carry bits in Θ(log n) delay
- Key idea: Transform chain of carry computations into a tree
  - Transforming a chain of associative operations (e.g., AND, OR, XOR) into a tree is easy
  - But how to do this with carries?

Full Adder

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



Boolean equations

$$s = (\sim a \cdot \sim b \cdot c_{in}) + (\sim a \cdot b \cdot \sim c_{in}) + (a \cdot \sim b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$

$$c_{out} = (\sim a \cdot b \cdot c_{in}) + (a \cdot \sim b \cdot c_{in}) + (a \cdot b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$

Optimized

$$t = a \oplus b \qquad s = t \oplus c_{in} \qquad c_{out} = t \cdot c_{in} + a \cdot b$$

Full Adder: Optimization steps

$$s = (\sim a \cdot \sim b \cdot c_{in}) + (\sim a \cdot b \cdot \sim c_{in}) + (a \cdot \sim b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$
$$= (\sim a \cdot \sim b + a.b) \cdot c_{in} + (\sim a \cdot b + a \cdot \sim b) \cdot \sim c_{in}$$
$$= \sim(a \oplus b) \cdot c_{in} + (a \oplus b) \cdot \sim c_{in}$$
$$= a \oplus b \oplus c_{in}$$

$$c_{out} = (\sim a \cdot b \cdot c_{in}) + (a \cdot \sim b \cdot c_{in}) + (a \cdot b \cdot \sim c_{in}) + (a \cdot b \cdot c_{in})$$
$$= (a \oplus b) \cdot c_{in} + a \cdot b$$

$$c_{out} = (a+b) \cdot c_{in} + a \cdot b$$

This is also correct

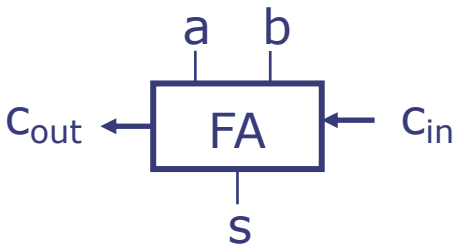
Sharing common sub-exressions

$$t = a \oplus b$$
$$s = t \oplus c_{in}$$
$$c_{out} = t \cdot c_{in} + a \cdot b$$

Carry Generation and Propagation

$c_{out} = a \cdot b + a \oplus b \cdot c_{in}$

generates a carry — propagates a carry



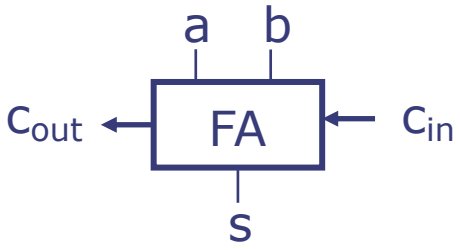
$$c_{out} = g + p \cdot c_{in}$$

where $g = a \cdot b$ and $p = a \oplus b$

- g=1 → $c_{out} = 1$ (FA generates a carry)
- p=1 (and g=0) → $c_{out} = c_{in}$ (FA propagates carry)

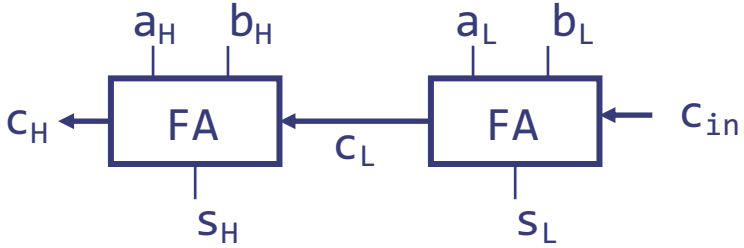
Notice p and g don't depend upon c_{in}

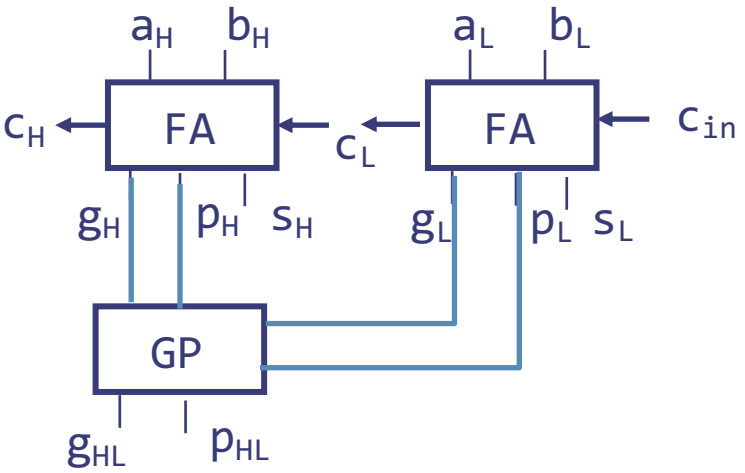
Generate and Propagate compose hierarchically



$$c_{out} = g + p \cdot c_{in}$$

where $g = a \cdot b$ and $p = a \oplus b$





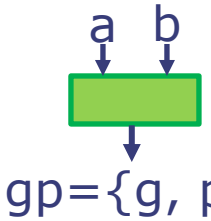
$$c_H = g_H + p_H \cdot c_L$$
$$= g_H + p_H \cdot (g_L + p_L \cdot c_{in})$$
$$= \underbrace{g_H + p_H \cdot g_L}_{g_{HL}} + \underbrace{p_H \cdot p_L}_{p_{HL}} \cdot c_{in}$$

$$g_{HL} = g_H + p_H \cdot g_L$$

$$p_{HL} = p_H \cdot p_L$$

CLA Building Blocks

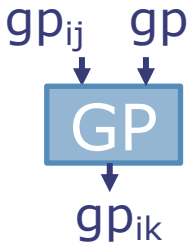
- Step 1: Generate individual g & p signals



$g = a \cdot b$

$p = a \oplus b$

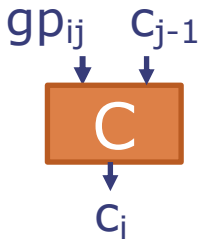
- Step 2: Combine adjacent g & p signals



$g_{ik} = g_{ij} + p_{ij} \cdot g_{(j-1)k}$

$p_{ik} = p_{ij} \cdot p_{(j-1)k}$ $\quad (i \geq j > k)$

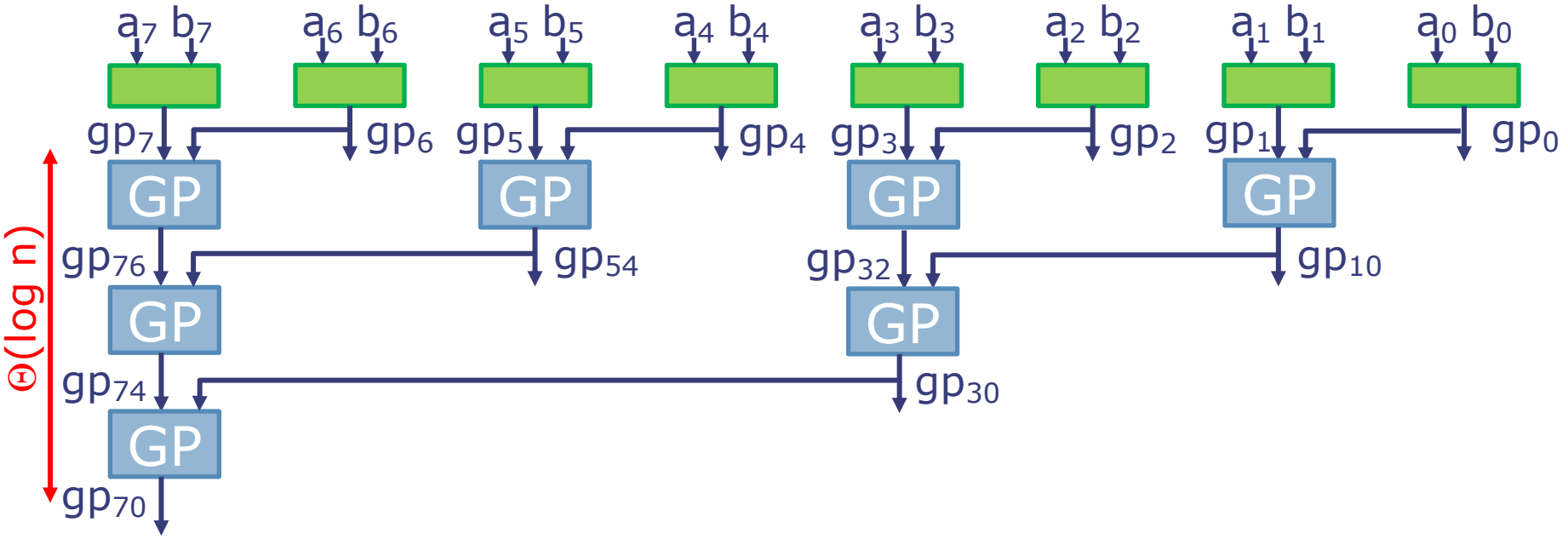
- Step 3: Generate individual carries



$c_i = g_{ij} + p_{ij} \cdot c_{j-1}$

There are many CLA variants. Let's derive the Brent-Kung CLA.

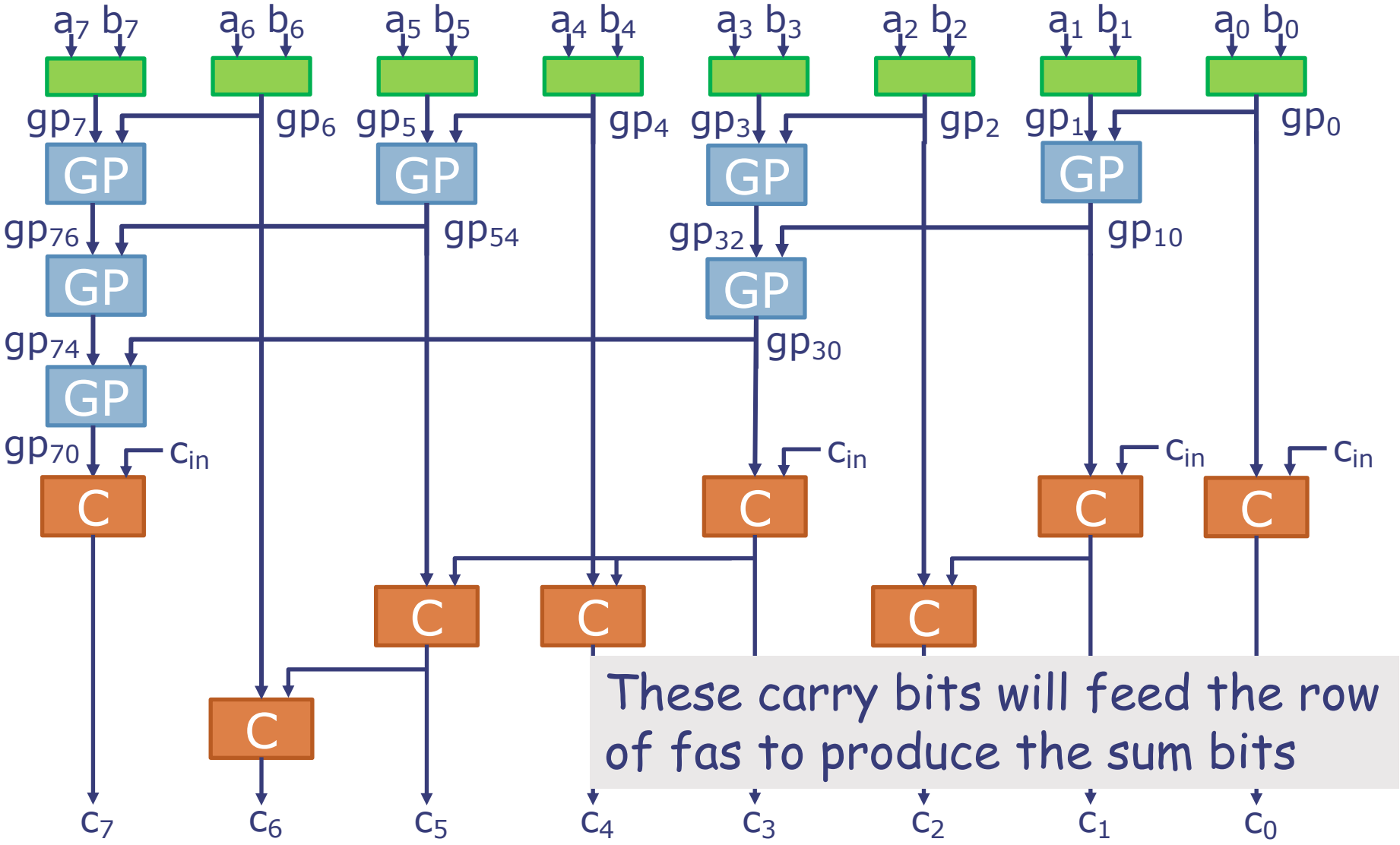
Generating and Combining gp's



How does delay grow with number of bits?

Θ(log n)

Generating the Carries



Carry-Lookahead Adder Takeaways

- There are many CLA designs
  - We've seen a Brent-Kung CLA
  - Several other types (e.g., Kogge-Stone)
  - Different variants for each type, e.g., using higher-radix trees to reduce depth

- This technique is useful beyond adders: computes any one-dimensional binary recurrence in $\Theta(\log n)$ delay
  - e.g., comparators, priority encoders, etc.

Some Synthesis Results
Brian Wheatman

	Time Opt Basic gates	Time Opt Ext gates	Space Opt Basic gates	Space Opt Ext gates
Ripple-Carry Adder (RCA)				
Gates	413	164	414	158
Area (um^2)	295	180	296	177
Delay (ps)	742	680	758	690
Recursive Carry-Select Adder				
Gates	970	663	840	564
Area (um^2)	727	604	630	519
Delay (ps)	226	233	340	302
Kogge-Stone Adder				
Gates	963	530	905	498
Area (um^2)	697	483	639	464
Delay (ps)	175	167	182	201

Summary

- Choosing the right algorithms is crucial to design good digital circuits—tools can only do so much!
- Carry-lookahead adders perform $\Theta(\log n)$ addition with some area cost. This technique can be used to optimize a broad class of circuits.