Interactive adder exercises

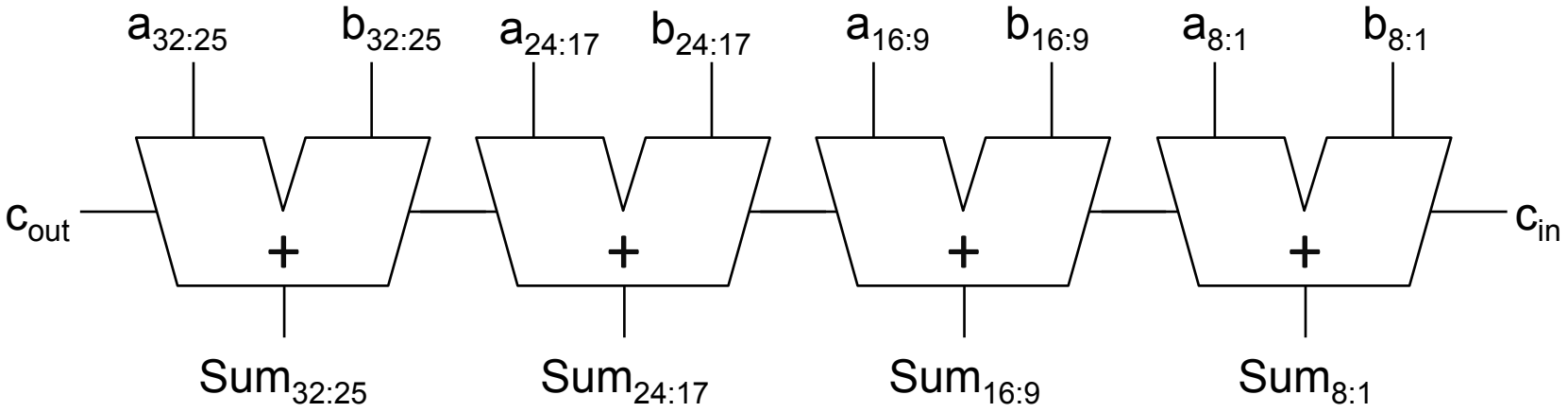
Studion

Exercise 2

2018

Purpose of today's lecture

- Learn how to design a 32-bit adder/subtractor where all sum cells have received their incoming carry signals after only **5 AND-OR** delay units!
  - Will be synthesized in Methods course.
  - Compare this delay to the delay of a ripple-carry adder: **31 AND-OR** unit delays, see figure below.
  - Or a ripple-carry lookahead adder with **17 AND-OR** unit delays.



From exercise 1
Ripple-carry adder

We assume that you have completed the first ripple-carry adder in the excel template.

	ADD/SUBTRACT			ADD=0			A=	-100	<<<<<	**ENTER TWO NUMBERS**							
	CONTROL SIGNAL	1		SUB=1			B=	-120	<<<<<	-128<NUMBER<128							
							SUM=	20										
	SUM result converted back to decimal:				20		Both sums are equal?		YES	OVERFLOW?			NO					
	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF		
	a8	b8	a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1		
	1	0	0	1	0	1	1	1	1	0	1	1	0	1	0	1		
←	1		1		1		1		1		1		1		1		1	←CIN
	0		0		0		1		0		1		0		0		SUM LOGIC	
	SUM8		SUM7		SUM6		SUM5		SUM4		SUM3		SUM2		SUM1			
	FF		FF		FF		FF		FF		FF		FF		FF			

From exercise 1
Ripple-carry timing

And that you are familiar with the timing of such a ripple-carry adder.

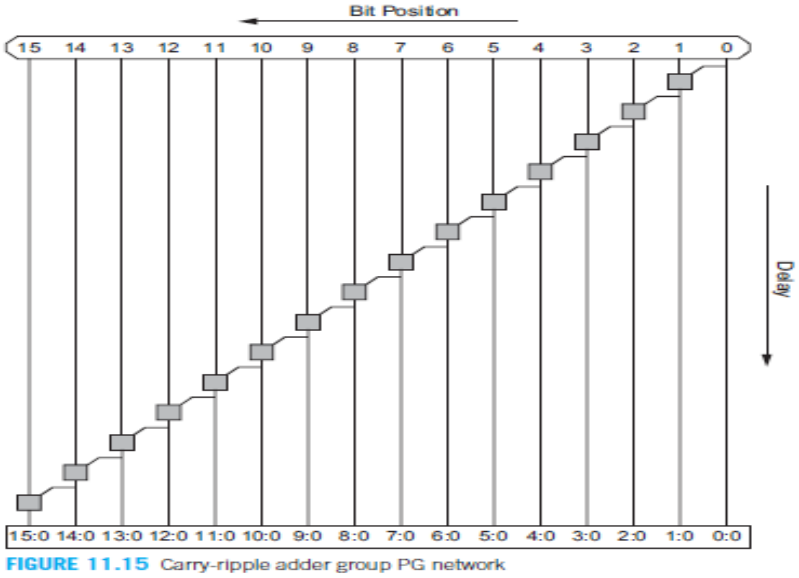
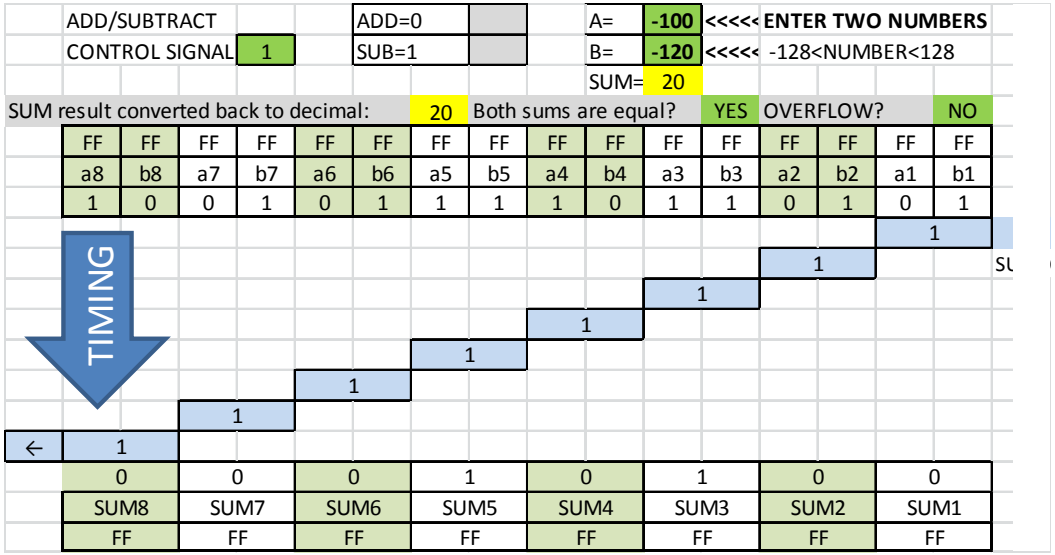


FIGURE 11.15 Carry-ripple adder group PG network



From exercise 1
Propagate/generate setup

- Redesigned adder with G and P setup to get less complex carry cell.
  - Also task in hand-in problem set 2.
- Note: delay for setting up bit-G and bit-P signals is denoted t_{pg} in Weste & Harris.
  - Other books use other notation.

	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	
	a8	b8	a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	
	1	1	1	0	1	0	1	1	1	1	1	1	1	0	0	0	
	G8	P8	G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	
	1	0	0	1	0	1	1	0	1	0	1	0	0	1	0	0	
	to be left blank until exercise #3!																
←	1		1		1		1		1		1		0		0		0 ←CIN
	1		0		0		1		1		0		1		0		SUM LOGIC
	SUM8		SUM7		SUM6		SUM5		SUM4		SUM3		SUM2		SUM1		
	FF		FF		FF		FF		FF		FF		FF		FF		

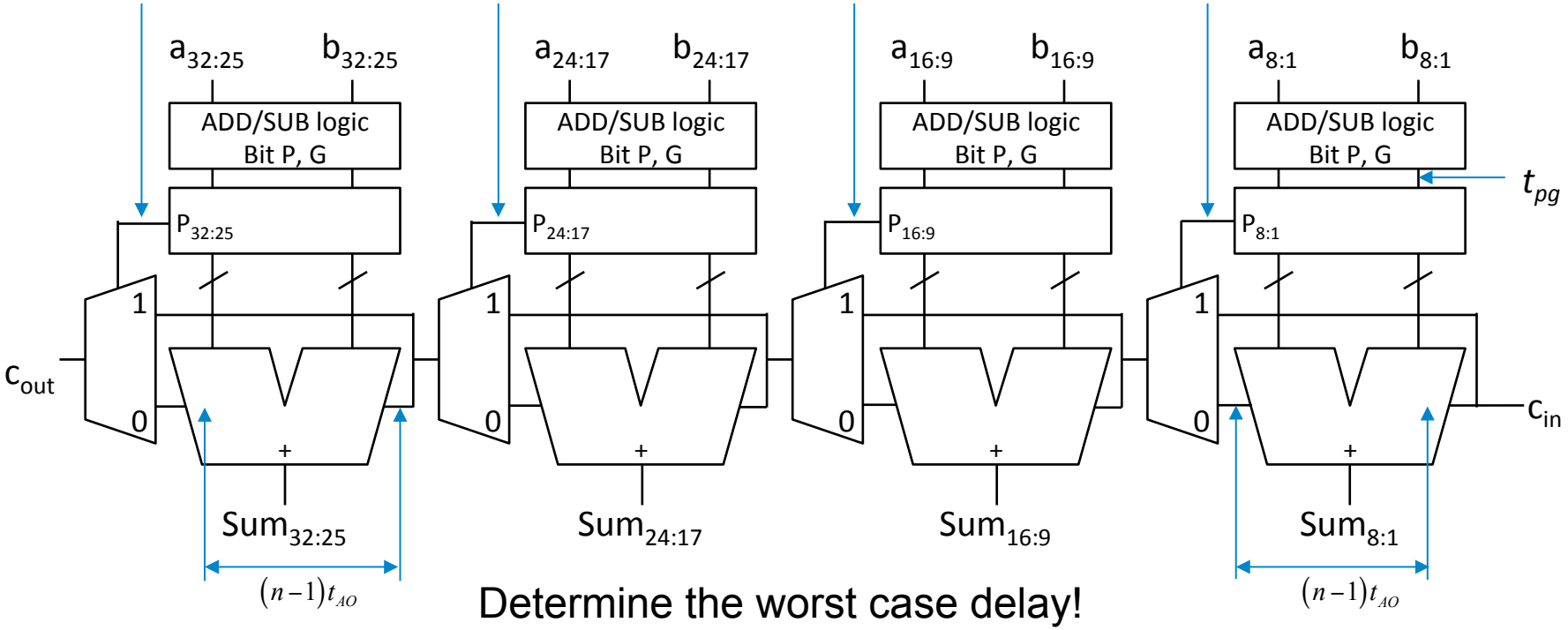
From exercise 1
Propagate/generate setup

- Redesigned the 8-bit adder block with G and P setup logic to get a less complex carry cell! => Use AO21 gate.
  - That is: your simplified carry cell from hand-in problem set 2.
- Added a block-propagate output. (n=8)
- Block propagate is available after delay of $(n\text{-}1)t_{AND}$.

	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF		
	a8	b8	a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1		
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		
	G8	P8	G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1		
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		1
		1		1		1		1		1		1		1		1		
←		0		0		0		0		0		0		0		0	0	←CIN
		1		1		1		1		1		1		1		1	SUM LOGIC	
		SUM8		SUM7		SUM6		SUM5		SUM4		SUM3		SUM2		SUM1		
		FF		FF		FF		FF		FF		FF		FF		FF		

From exercise 1
32-bit carry-skip adder

- Determine worst-case propagation delay for 32-bit adder.
  - For N-bit adder built with *k* *n*-bit blocks!
  - Our case: N=32, k=4, n=8



Determine the worst case delay!

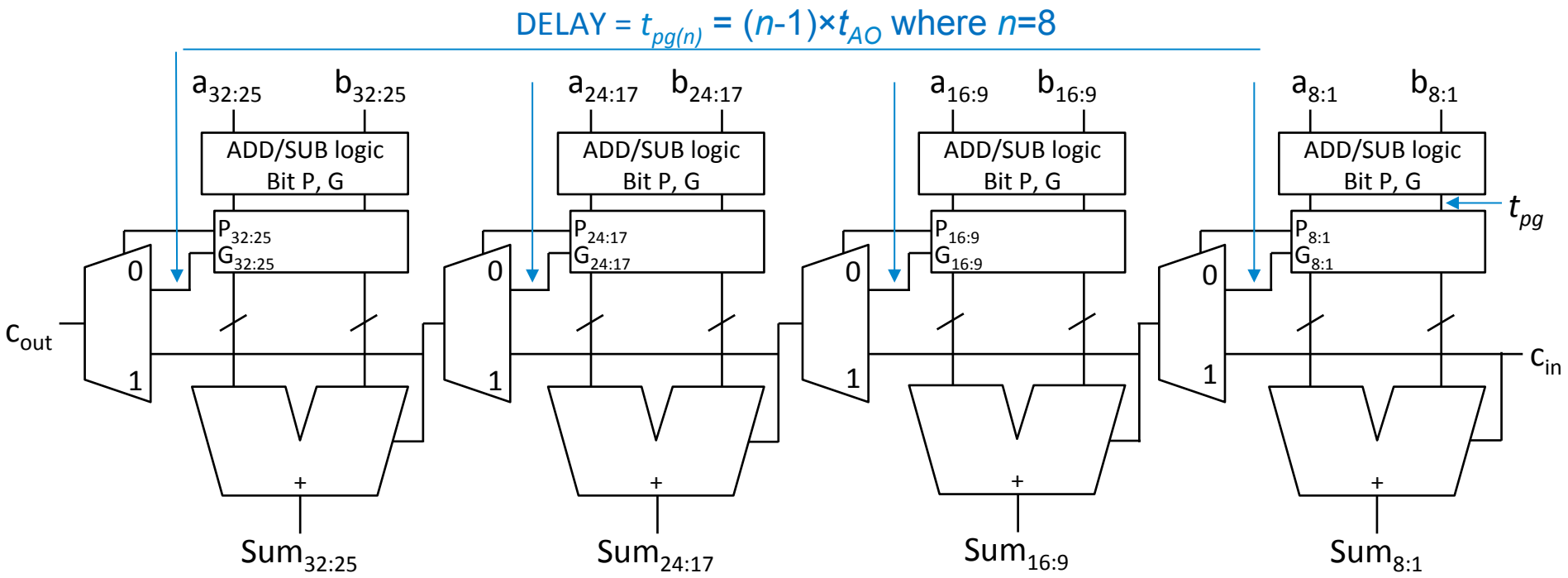
$$t_{skip} = t_{pg} + 2(n-1)t_{AO} + (k-1)t_{mux} + t_{XOR}$$

Carry lookahead adders

- Next step: add a block generate output!
  - Will not do this part in class, but you have it in the excel file from execise 1.
- Block-G is available $t_{pg(n)}=(n\text{-}1)t_{AO}$ after bit-P and bit-G setup.
- Again, build a 32-bit adder with 4 instances of this 8-bit block

	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF		
	a8	b8	a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1		
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1		
	G8	P8	G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1		
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0		
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		
←	1		1		1		1		1		1		1		1		0	←CIN
	0		0		0		0		0		0		0		0		SUM LOGIC	
	SUM8		SUM7		SUM6		SUM5		SUM4		SUM3		SUM2		SUM1			
	FF		FF		FF		FF		FF		FF		FF		FF			

32-bit carry-lookahead adder



DELAY through k MUXES = (k-1)×t_{mux} where k=4

Determine the worst case delay!

$$t_{cla} = t_{pg} + t_{pg(n)} + (n-1)t_{AO} + (k-1)t_{mux} + t_{XOR}$$

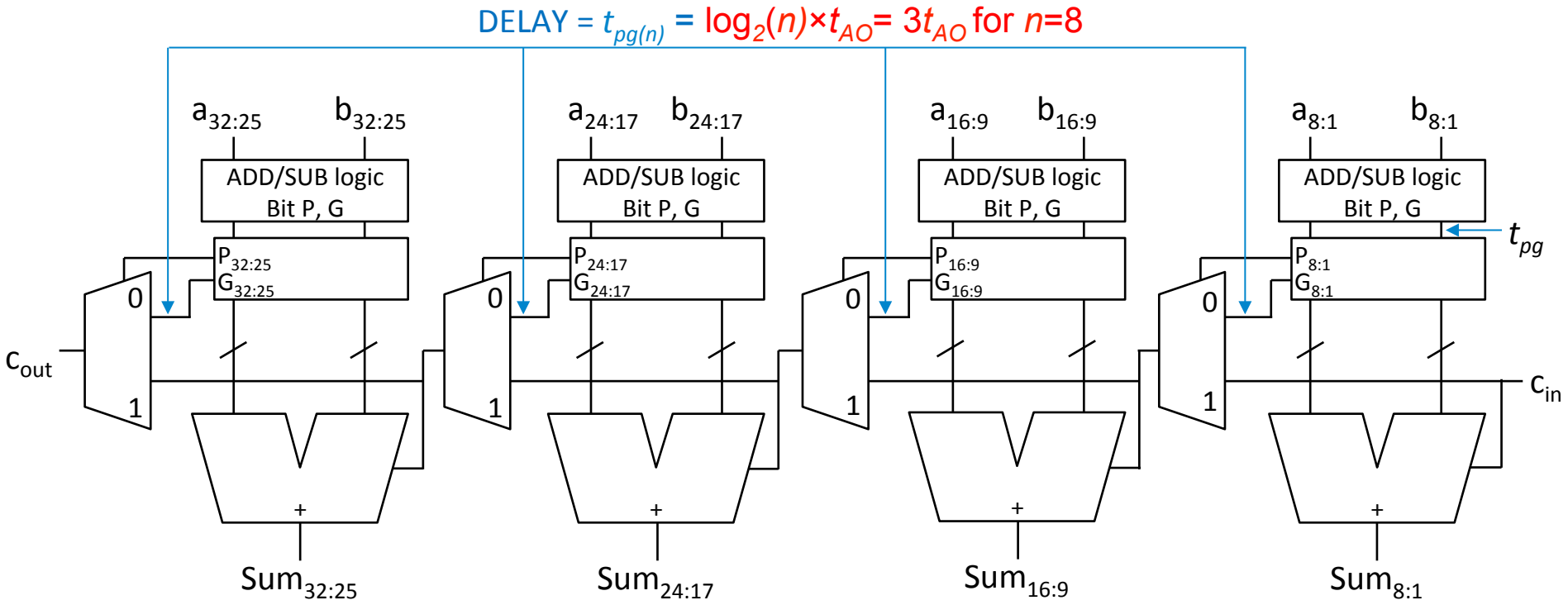
Carry lookahead adders

- Today's goal: reduce the time for obtaining block-P and block-G by obtaining them from a binary tree.
- Then delay will increase as $\log_2(N)$ for an N-bit adder

	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	
	a8	b8	a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	
	1	0	1	0	0	1	1	0	1	0	1	0	1	0	1	1	
	G8	P8	G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	
	0	1			0	1			0	1			1	0			
	0	1							1	0							
←	1	0															
	1		1		1		1		1		1		1		1		0 ←CIN
	0		0		0		0		0		0		0		0		SUM LOGIC
	SUM8		SUM7		SUM6		SUM5		SUM4		SUM3		SUM2		SUM1		
	FF		FF		FF		FF		FF		FF		FF		FF		

Carry-lookahead adder

Assume we can use binary tree instead of ripple computation for block P and block G

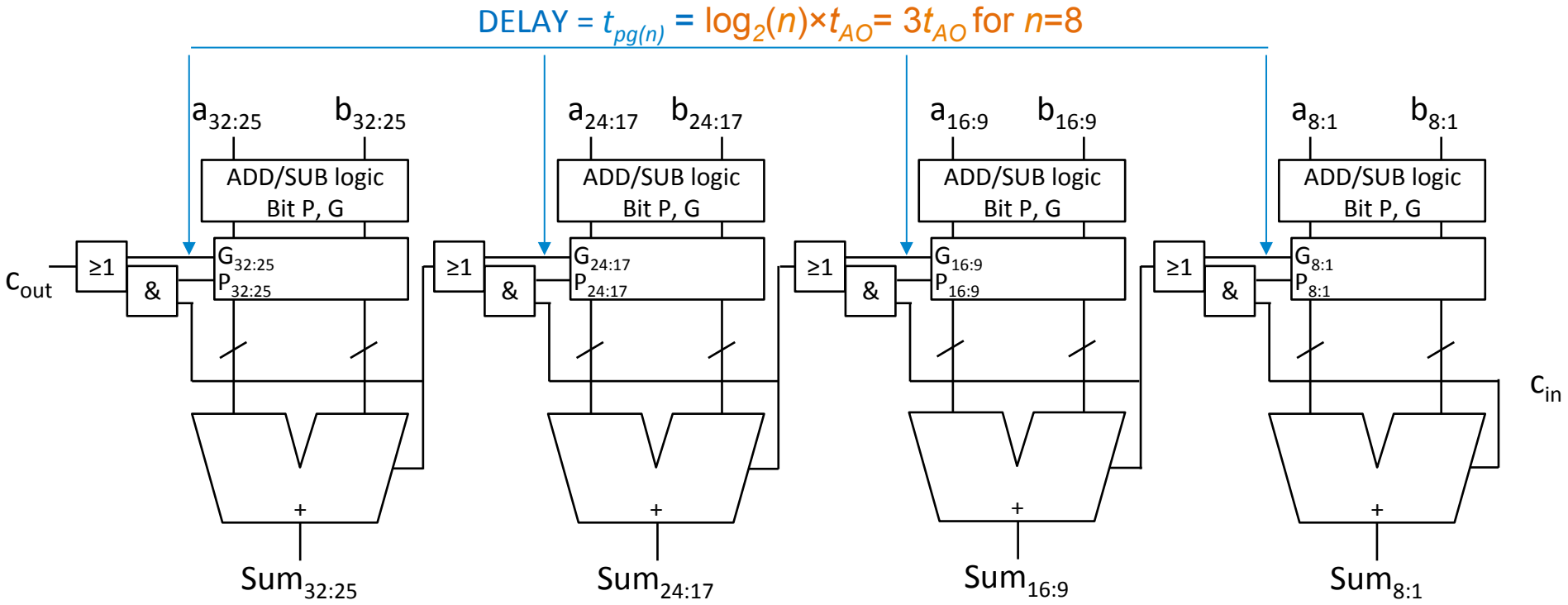


Determine the worst case delay!

$$t_{cla} = t_{pg} + t_{pg(n)} + (n-1)t_{AO} + (k-1)t_{mux} + t_{XOR}$$

Carry-lookahead adder

Use AO gate rather than mux



DELAY through k "MUXES" = $(k\text{-}1)\times t_{AO}$ where k=4

Use of AO gates possible because block P and block G are never true at the same time!

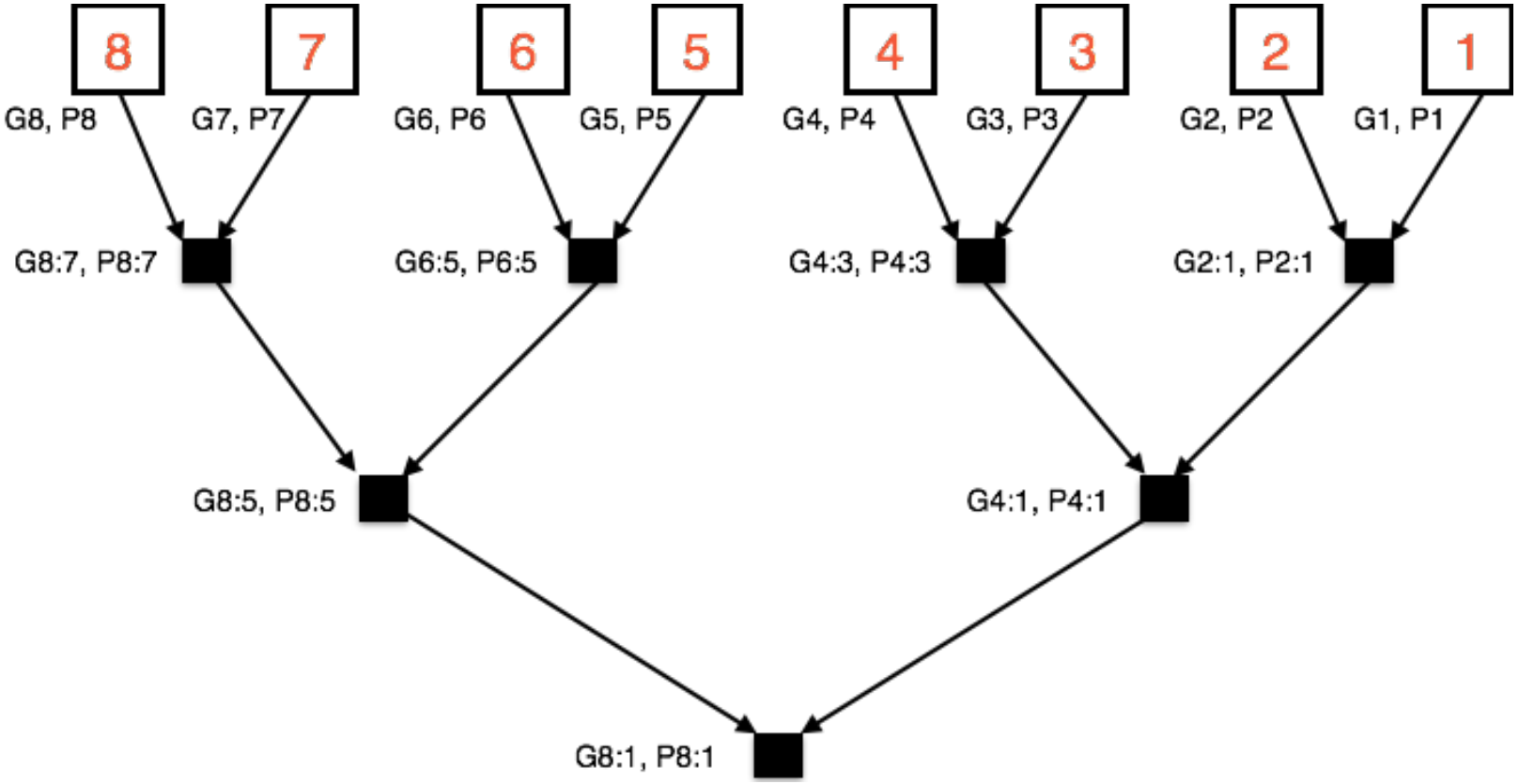
Determine the worst case delay!

$$t_{cla} = t_{pg} + t_{pg(n)} + \left[\left(n-1\right)+\left(k-1\right)\right]t_{AO} + t_{XOR}$$

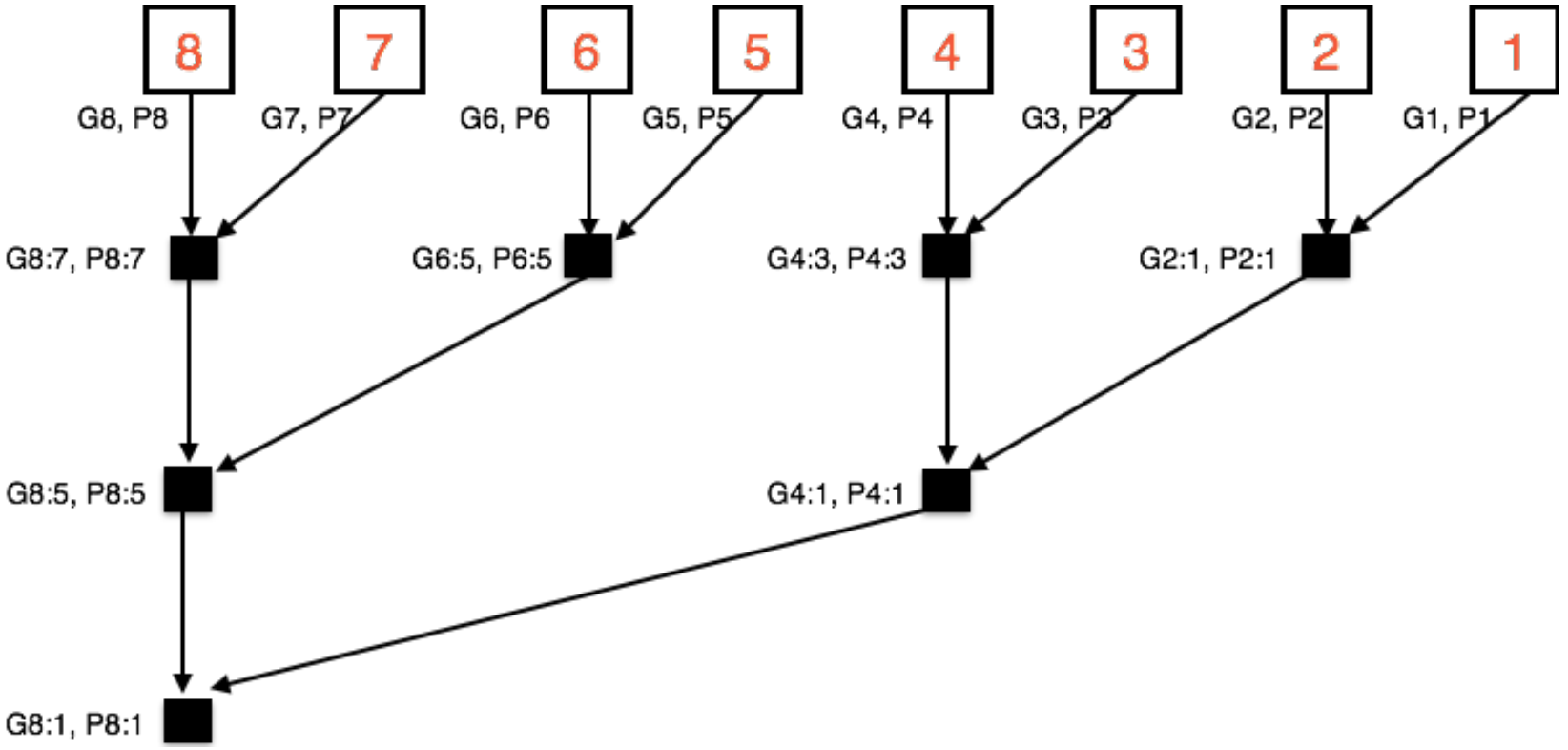
Tree adders

- Fundamental problem: to know C_i we need C_{i-1}
  - Solution: look ahead over multiple levels to figure out carry.
  - Called "Prefix computation" –
    - Time dependence from linear in N to logarithmic in N
- Notation:
  - $X_{i:j}$ means the signal "X" for the ith to jth position
  - P = Propagate ($A \oplus B$)
  - G = Generate (AB)
- Note about Numbering:
  - Weste & Harris use numbers N to 1 for the adder itself and position 0 for the C_{IN} signal to the adder.
  - Note! Many other books use N-1 to 0 for the adder.

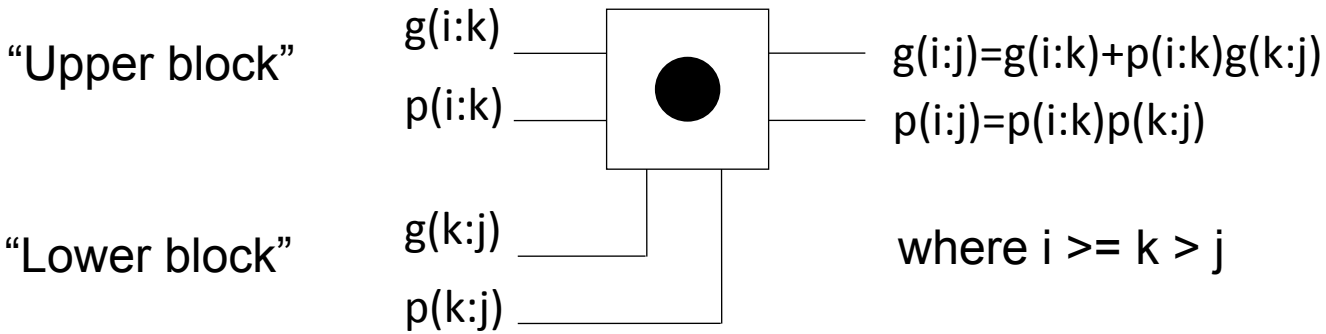
8-bit binary tree for calculating G,P



8-bit binary tree for calculating G,P

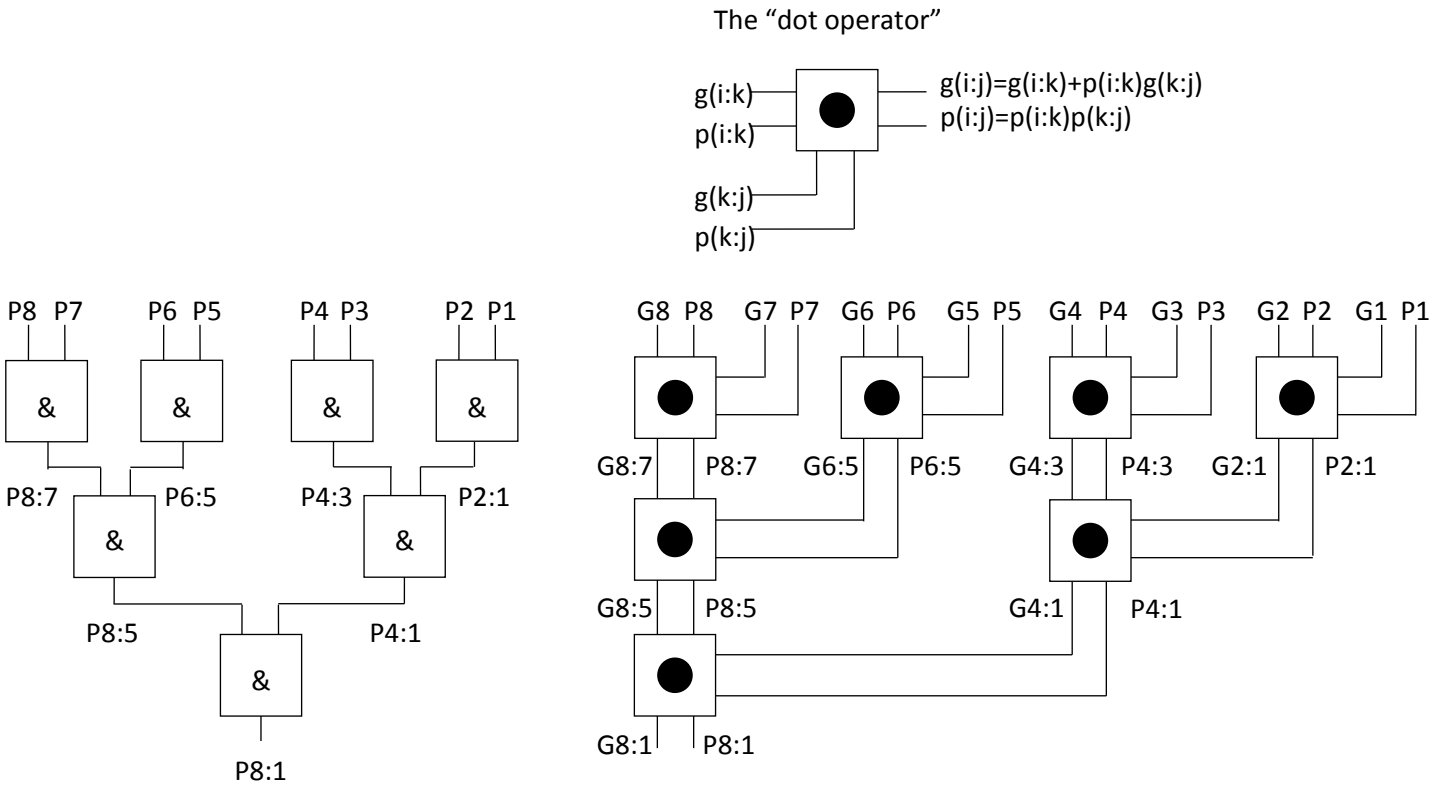


The “dot operator” / PG cell



Generate for combined block =
The upper block has generate
OR
The upper block has propagate AND the lower block has generate

Binary trees



Show that $\left(G_{4:3}, P_{4:3}\right) \bullet \left(G_{2:1}, P_{2:1}\right) = G_4 + P_4\left(G_3 + P_3\left(G_2 + P_2 G_1\right)\right), P_4 P_3 P_2 P_1$

Sklansky adder

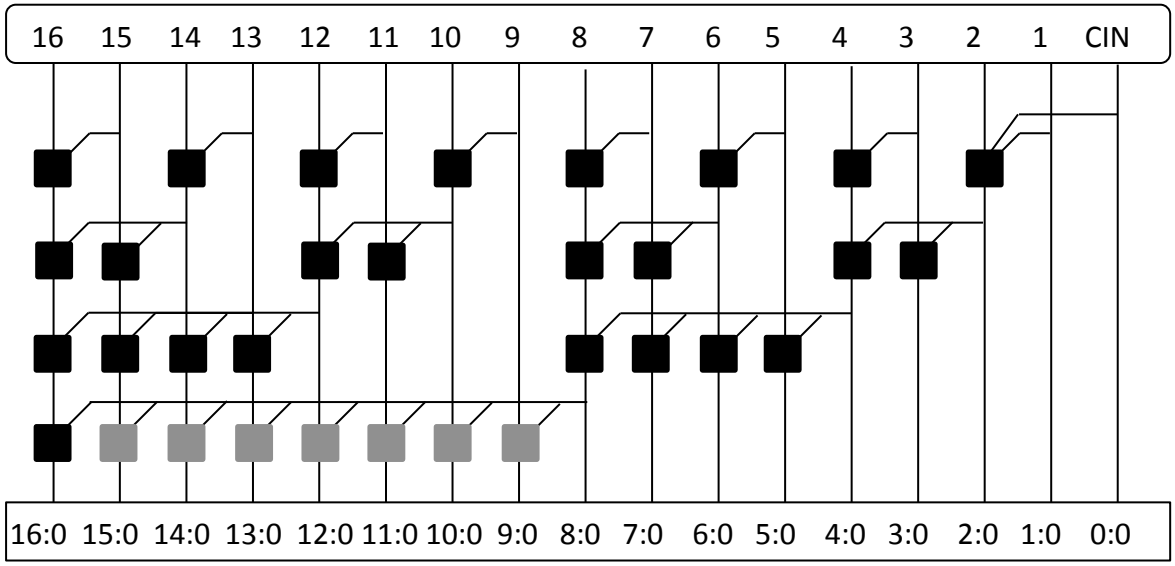


FIGURE 11.29 Tree adder PG networks

Do you see the tree?

Sklansky adder

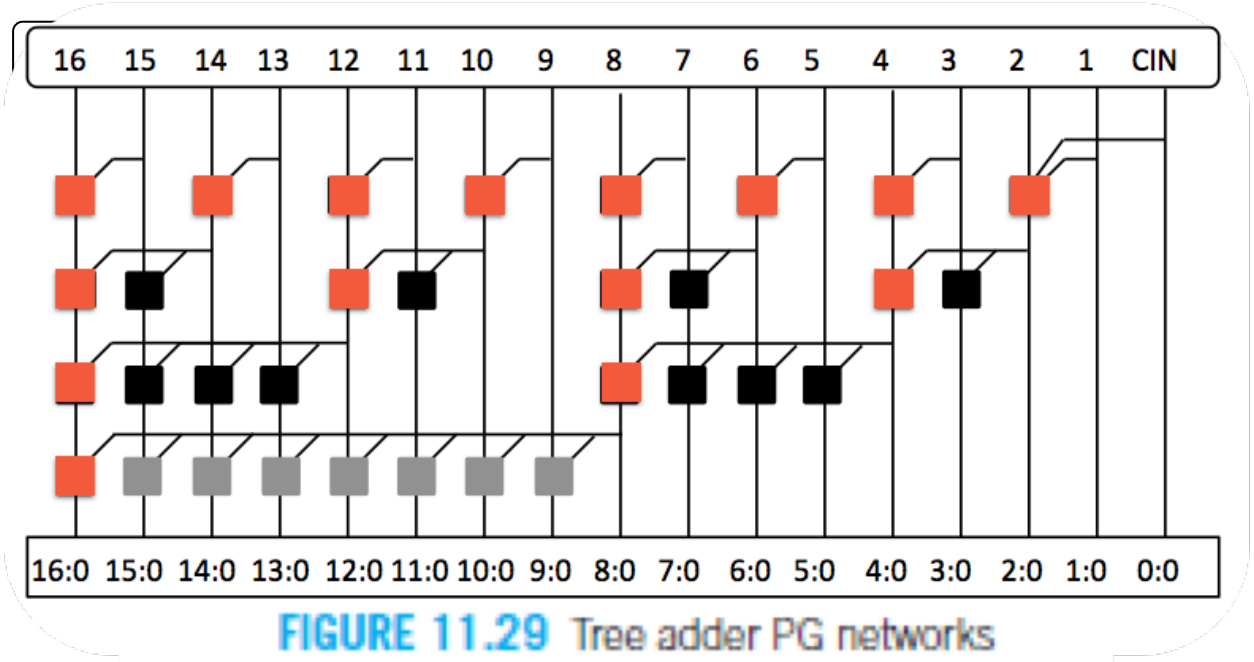


FIGURE 11.29 Tree adder PG networks

Here the forward tree is marked in red.

The other cells are needed to make sure all carries exist so that all sums can be computed.

Reverse tree: different prefix adders use different solutions.

Sklansky adder

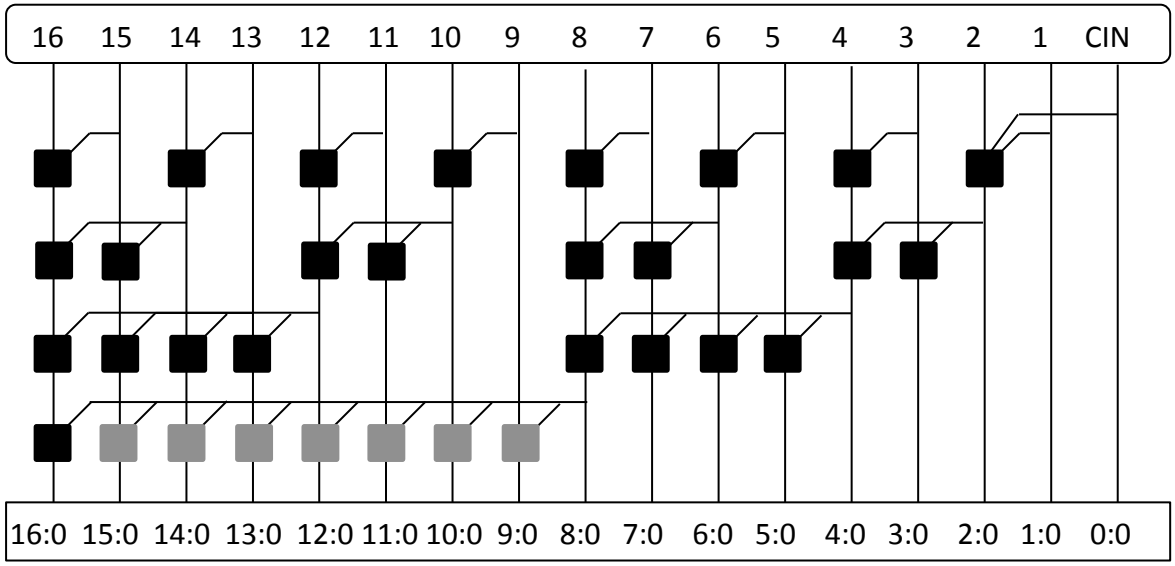


FIGURE 11.29 Tree adder PG networks

Determine the worst-case delay!

$$t_{tree} = t_{pg} + t_{pg(n)} + t_{XOR}$$

$$t_{pg(n)} = \lceil \log_2 N \rceil t_{AO}$$

Sklansky adder

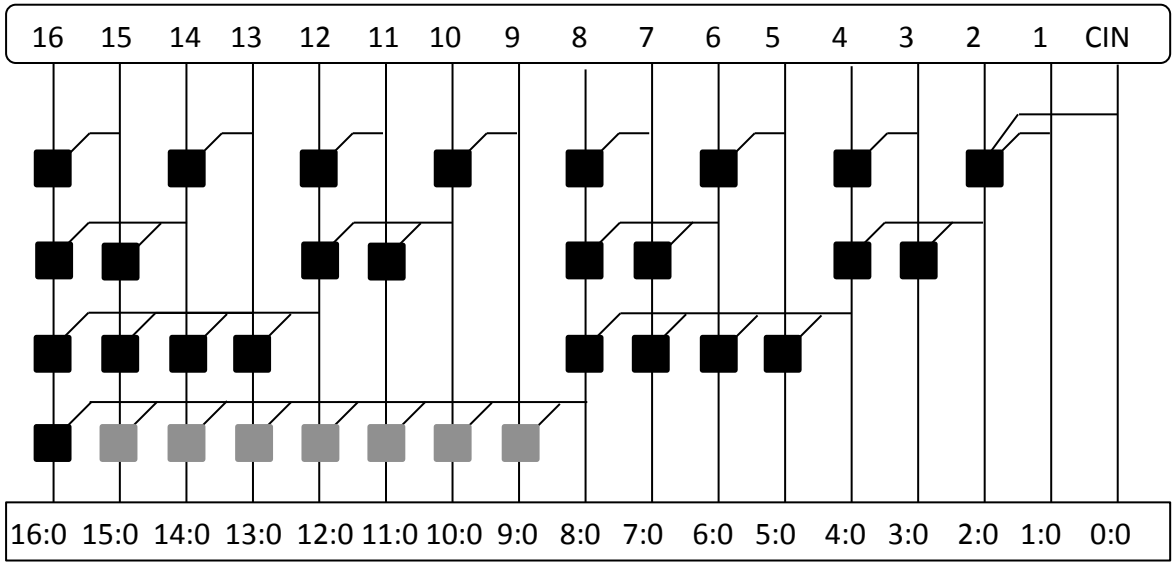


FIGURE 11.29 Tree adder PG networks

Determine the worst-case delay!

$$t_{tree} = t_{pg} + t_{pg(n)} + t_{XOR}$$

$$t_{pg(n)} = \lceil \log_2 N \rceil t_{AO}$$

Summary

After this interactive lecture you will know about

- Carry-skip adders, carry-lookahead adders and prefix-tree adders like Sklansky adders, etc.
- You can identify the worst-case propagation delay for *N*-bit carry-skip and carry-lookahead adders built from *k* *n*-bit blocks
- You can identify the worst-case delay for prefix-tree adders like Sklansky adders.
- Using the same principles, you should also be able to do so for the unknown prefix-tree adder in home assignment 3.
- Using the same principles, you should also be able to do so for some of the other prefix-tree adders like Brent-Kung and Ladner-Fischer.