

32-bit Domino Logic Adder

- Implement a 32-bit CLA (carry lookahead) adder in Domino logic
- Use Technology: tsmc018.model, $V_{dd} = 2.5\text{ V}$
- Will provide a testbench on Monday

Adder Equations

$$\text{sum}(i) = a(i) \text{ xor } b(i) \text{ xor } \text{cin}(i)$$

$$\text{c}(i+1) = (\text{cin}(i) (a(i) \text{ or } b(i))) \text{ or } (a(i) b(i))$$

Define:

$$\text{Generate } g(0) = a(0) b(0)$$

$$\text{Propagate } p(0) = a(0) + b(0)$$

$$\text{c}(1) = g(0) + p(0)\text{c}(0)$$

$$\text{c}(i+1) = g(i) + p(i)\text{c}(i)$$

Note that $p(0) = a(0) \text{ xor } b(0)$ works also since the generate case will take care of the $a(0)=1, b(0)=1$ case.

Adder Equations (cont)

Propagate/Generate across multiple bits

$$P(0,1) = p(0) p(1)$$

$$G(0,1) = g(1) \text{ or } p(1)g(0)$$

In general, any j with $i < j, j+1 < k$

$$c(k+1) = G(i,k) + P(i,k)c(i)$$

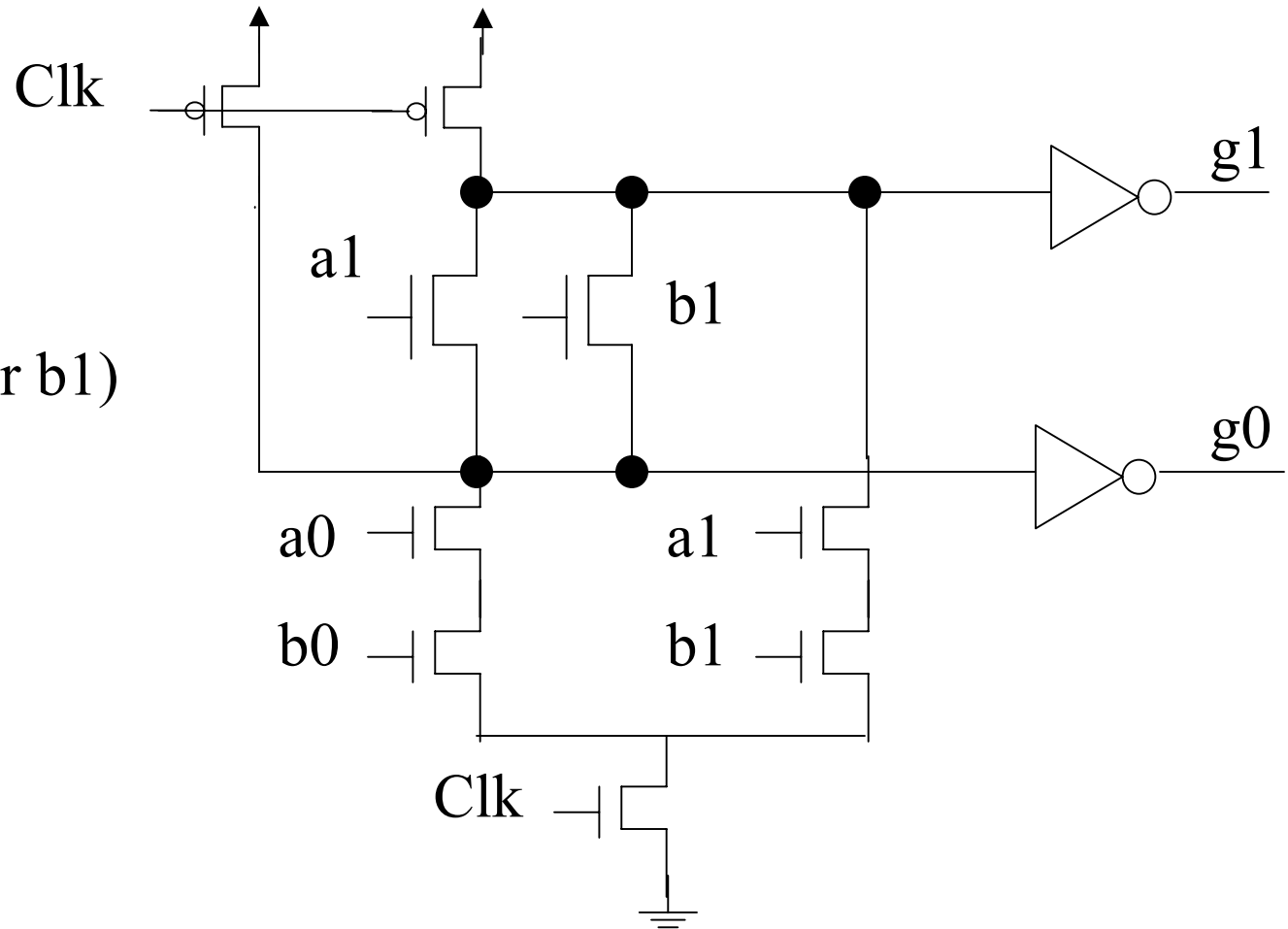
$$G(i,k) = G(j+1,k) + P(j+1),G(i,j)$$

$$P(i,k) = P(i,j) P(j+1,k)$$

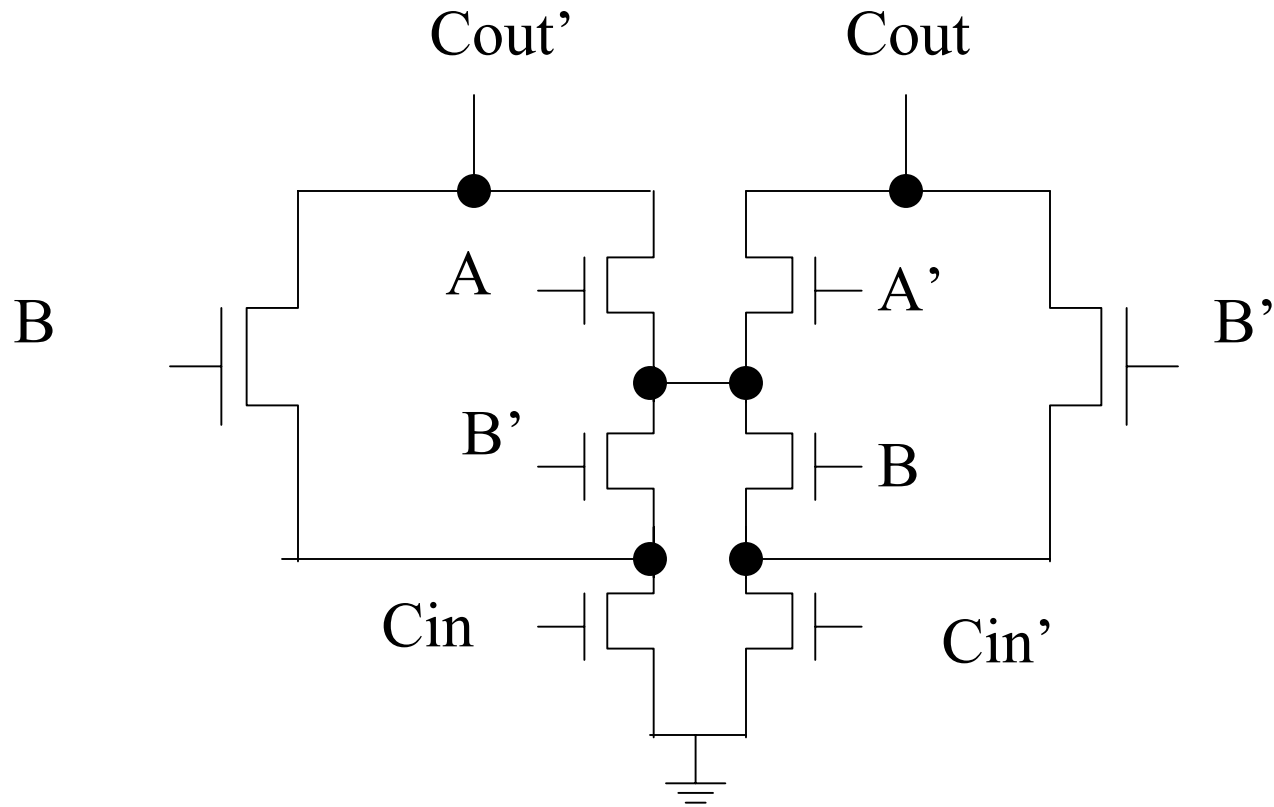
Example Carry Lookahead Generate Logic

$$g_0 = a_0 b_0$$

$$g_1 = a_1 b_1 + g_0 (a_1 \text{ or } b_1)$$



Full Adder, Cout logic



Could be used for ripple-carry
Carry Generation

8-Bit CLA

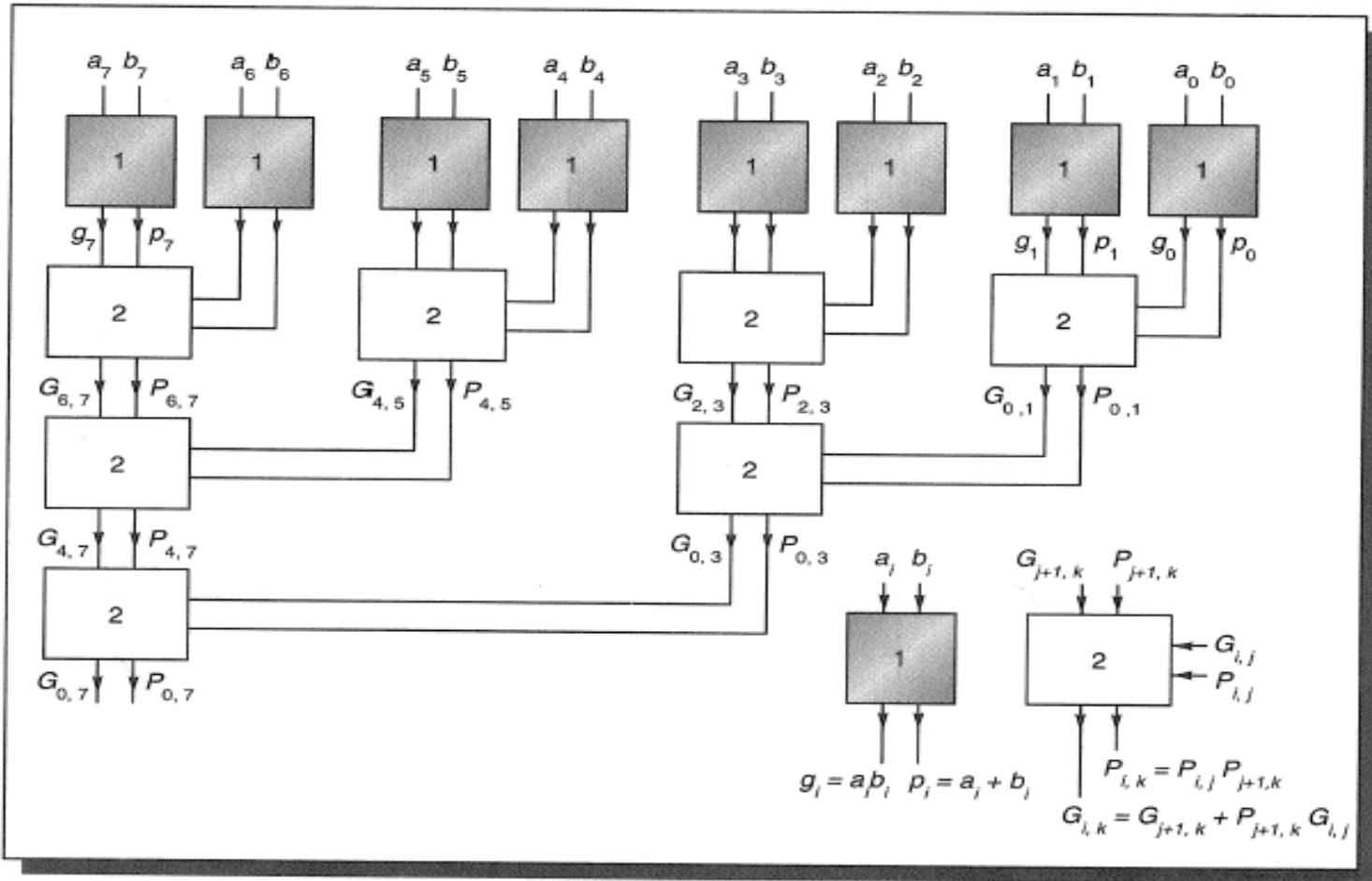


FIGURE A.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of P and G are computed.

8-bit CLA (cont)

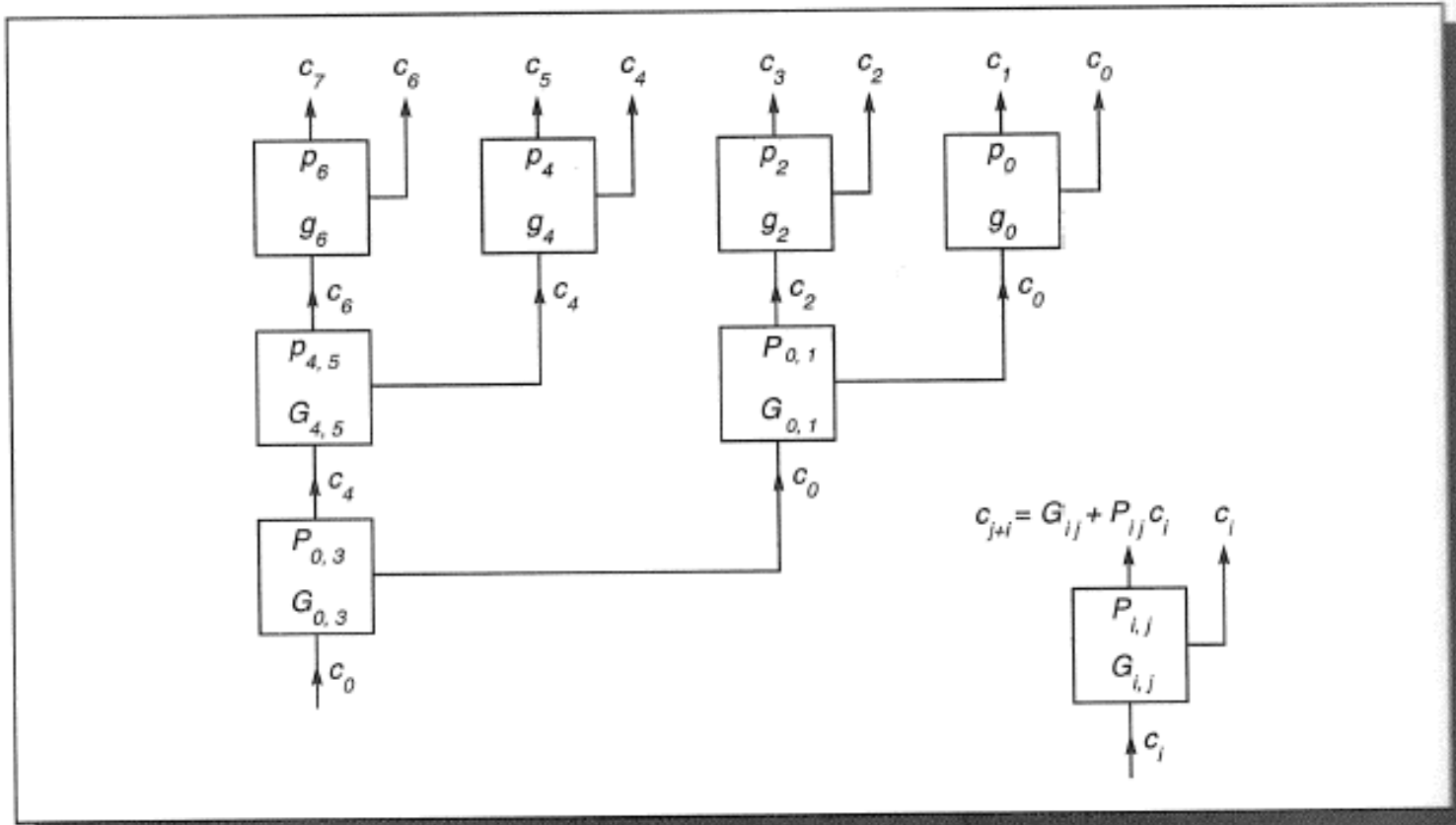


FIGURE A.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with P and G to form the carries.

8-bit CLA (cont)

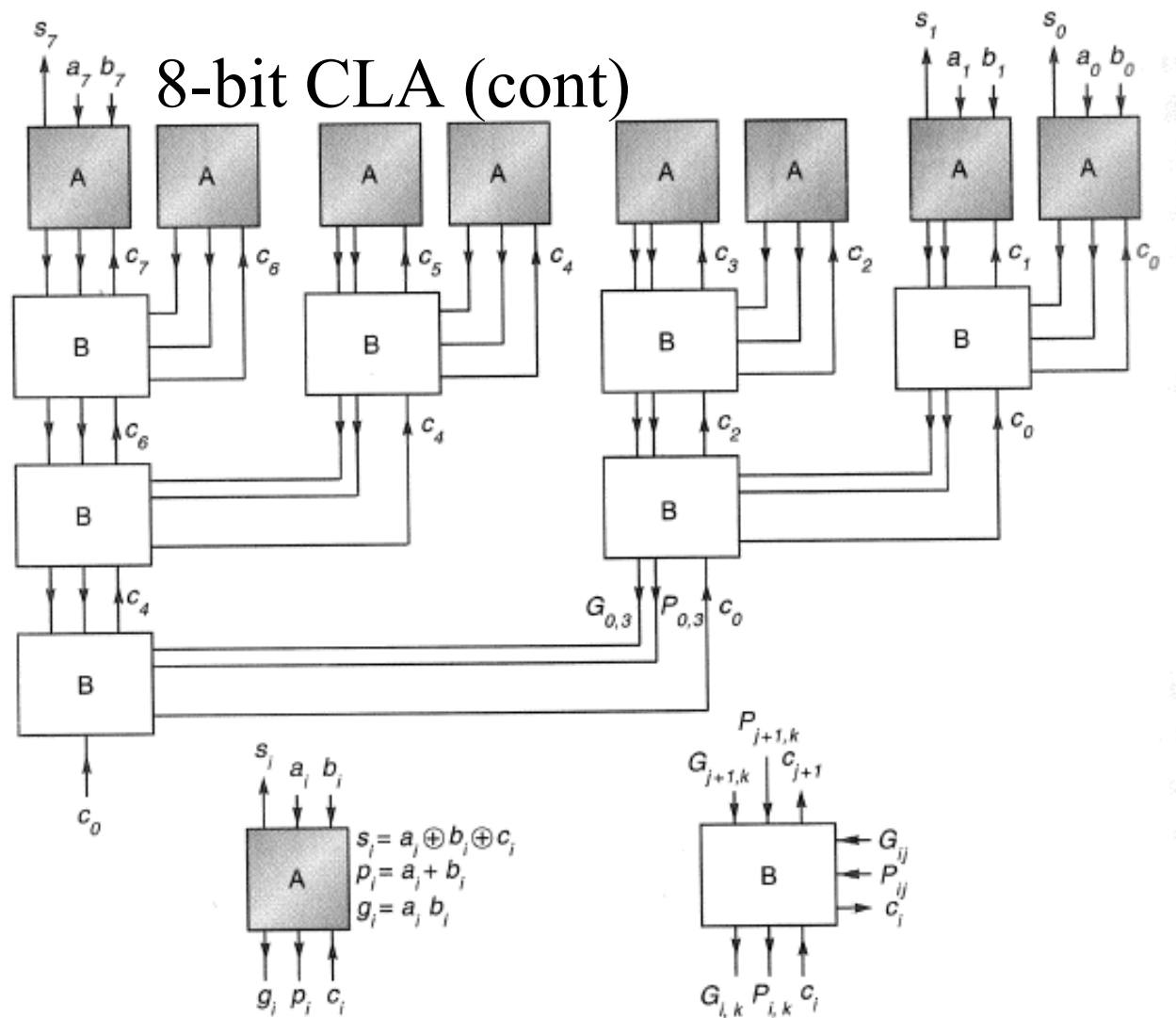
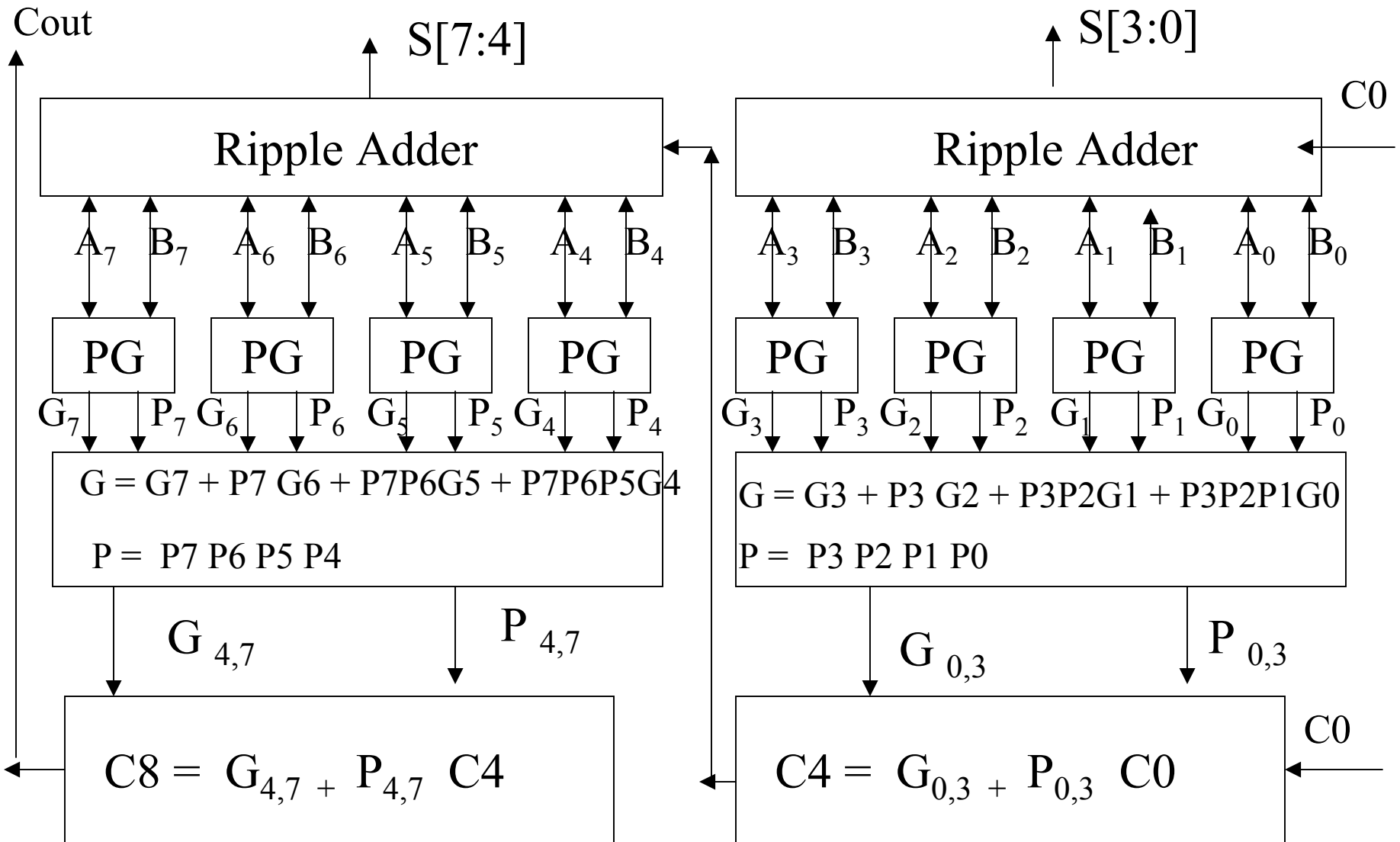
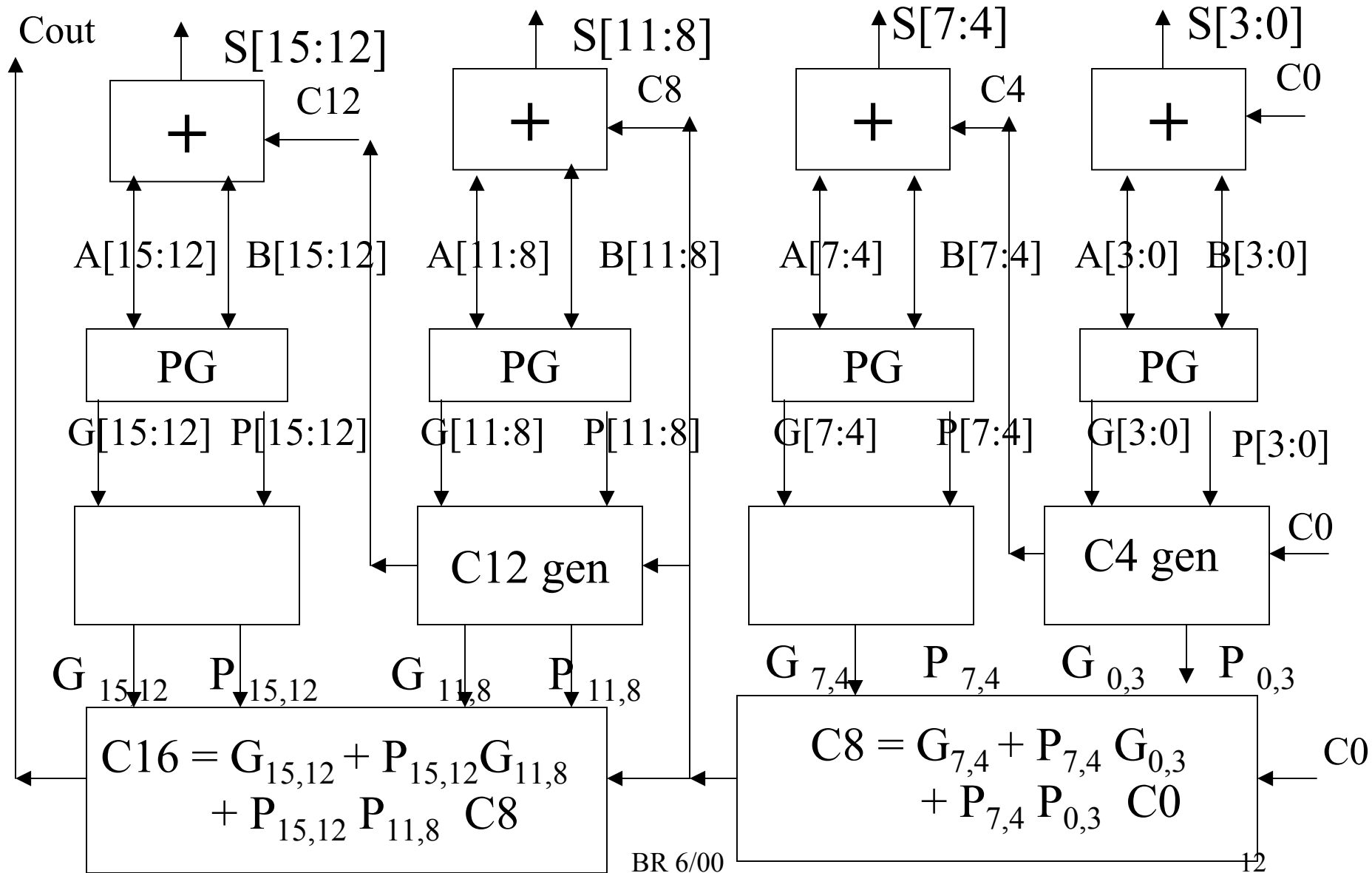


FIGURE A.17 Complete carry-lookahead tree adder. This is the combination of Figures A.15 and A.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0 , and then flow back up to compute the sum bits.

Using Carry Lookahead + Ripple in a 8-bit adder



Using Carry Lookahead + Ripple in a 16-bit adder



Which Approach do you use?

- Do you use full carry lookahead or ripple+carry-lookahead?
 - How complex should your generate/propagate blocks be? You decide.
- Use whatever approach you want.
- Work must be individual
- Will be graded on functionality and performance
- If functionality is 100%,
 - top 1/3 of class gets 25 extra points to test grade
 - middle 1/3 gets 12 extra points
 - bottom 1/3 gets not extra points.
- All logic must be domino logic.

Testbench

- I will provide a Spectre Test bench at a later date.
 - My testbench will test your design with several vectors
- All inputs, including the clock, will driven by 1X buffers
 - All inputs, including the clock, will driven by 1X buffers
 - All extra drive will have to provided by your spice netlist implementation
 - Maximum sized inverter you can use is 20x of a 2/1 inverter.
- Input names:
 - a0,a1,a2..a31; b0,b1,...b31, ci
 - low true versions will also be supplied (na0,na1..nb0,nb1..nci)
 - clk
- Output names:
 - s0,s1..s31; co

Testbench

- Inputs from testbench will change some propagation delay after rising clock edge to simulate arrival from static logic
 - You must provide a suitable static to dynamic interface
- Testbench will capture inputs on falling edge of clock to capture outputs after evaluation
 - Do not provide dynamic to static interface, testbench will do this.
- You must specify to me the high/low times of the clock (duty cycle does not have to 50%).
- Performance will be based on clock period.
- Due Monday, Sept 29th.