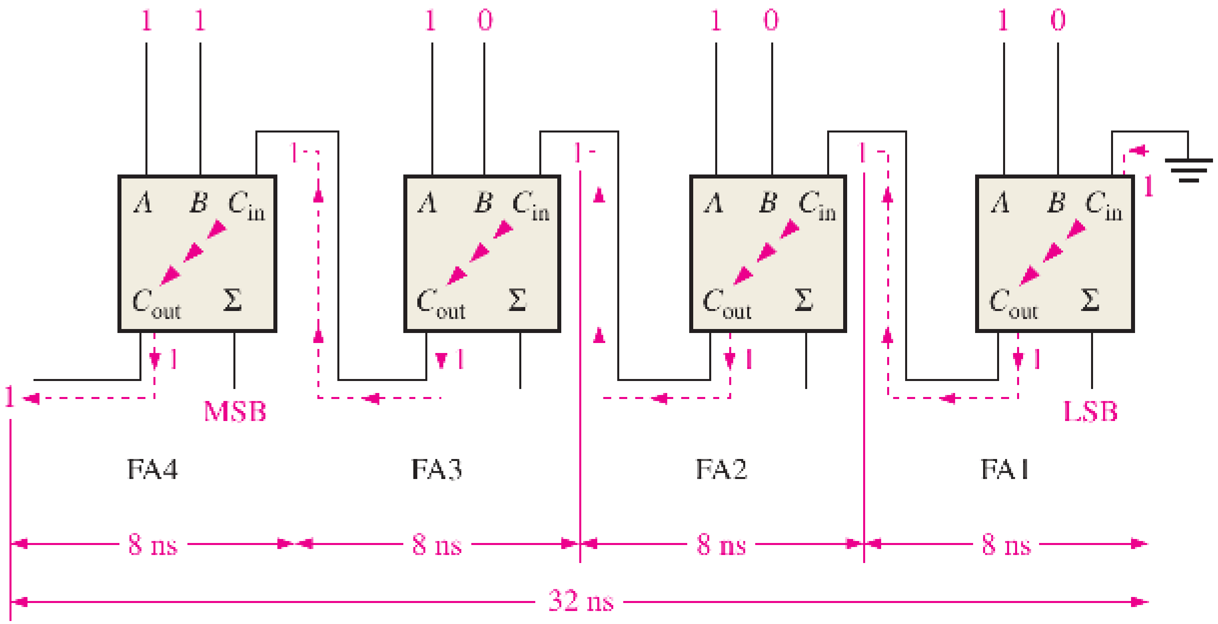
Lecture 14 – Combinational circuits 3

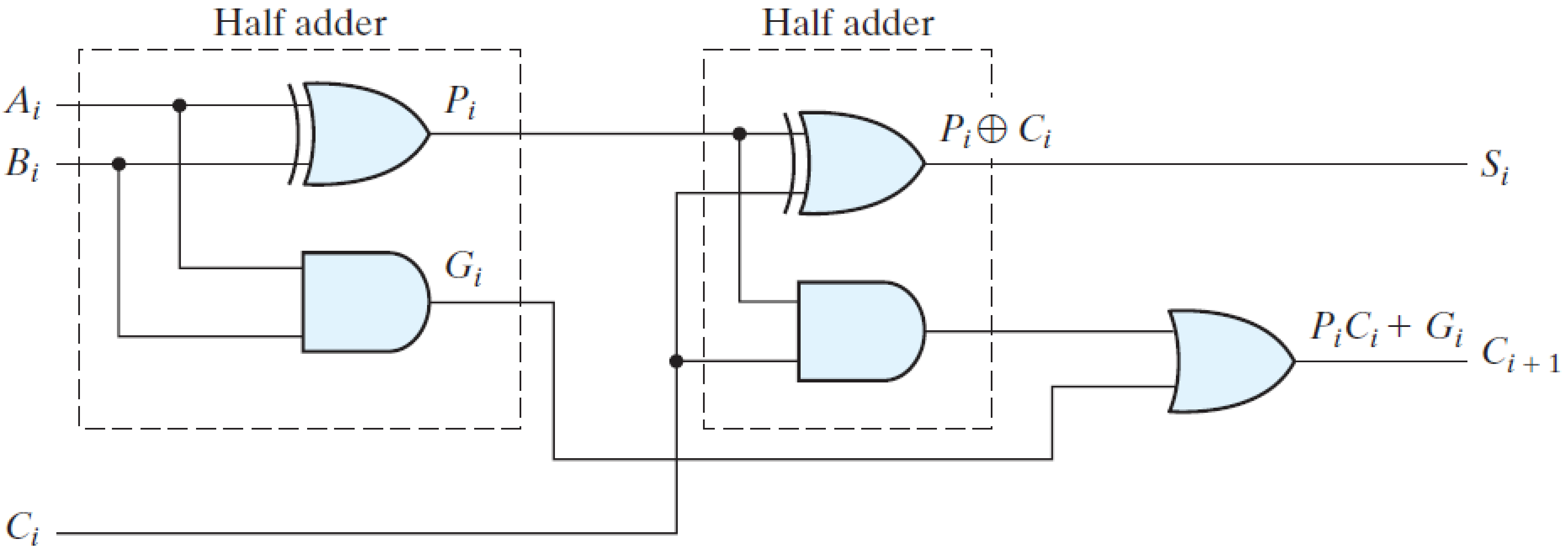
Carry propagation problem

- The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time
- As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals
- The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit



Carry propagation problem

- There are several techniques for reducing the carry propagation time in a parallel adder
- An obvious solution to this problem is to actually make the 2^n truth-table, K-map and get a two level implementation (either SoP or PoS)
- The most widely used technique employs the principle of *carry lookahead logic*

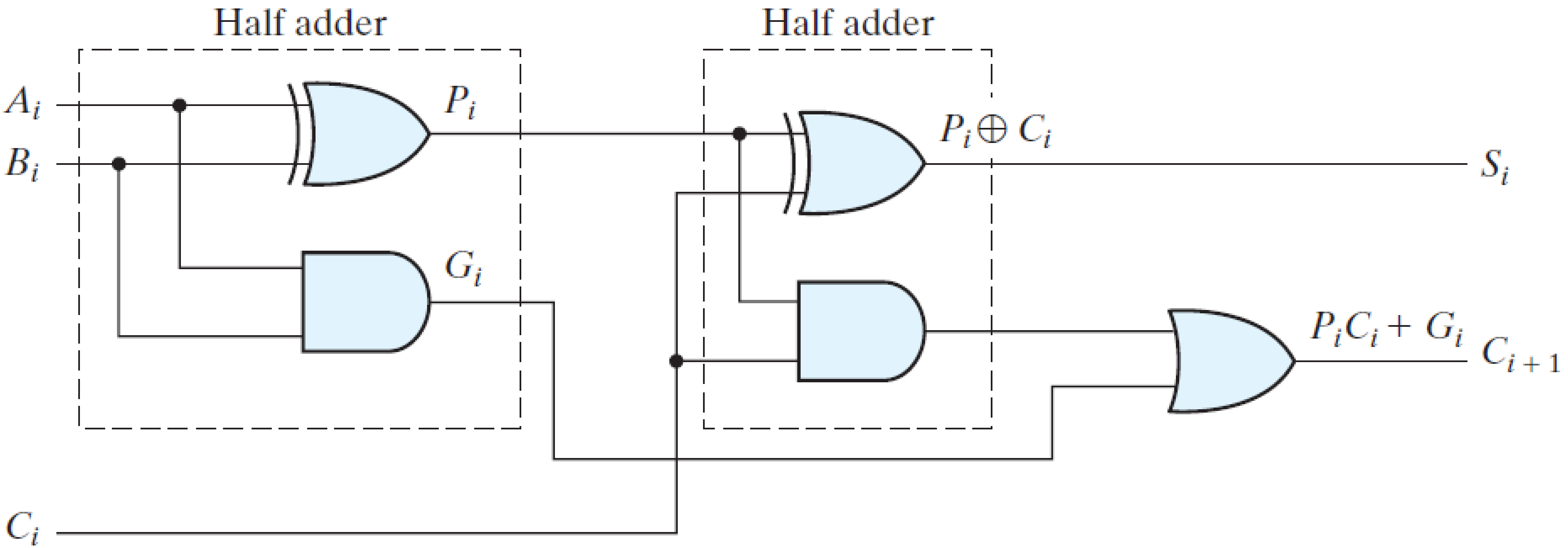


Carry propagation problem

- With the definition of P and G, we can write:

$$S_i = P_i + C_i \text{ and } C_{i+1} = G_i + P_i C_i$$

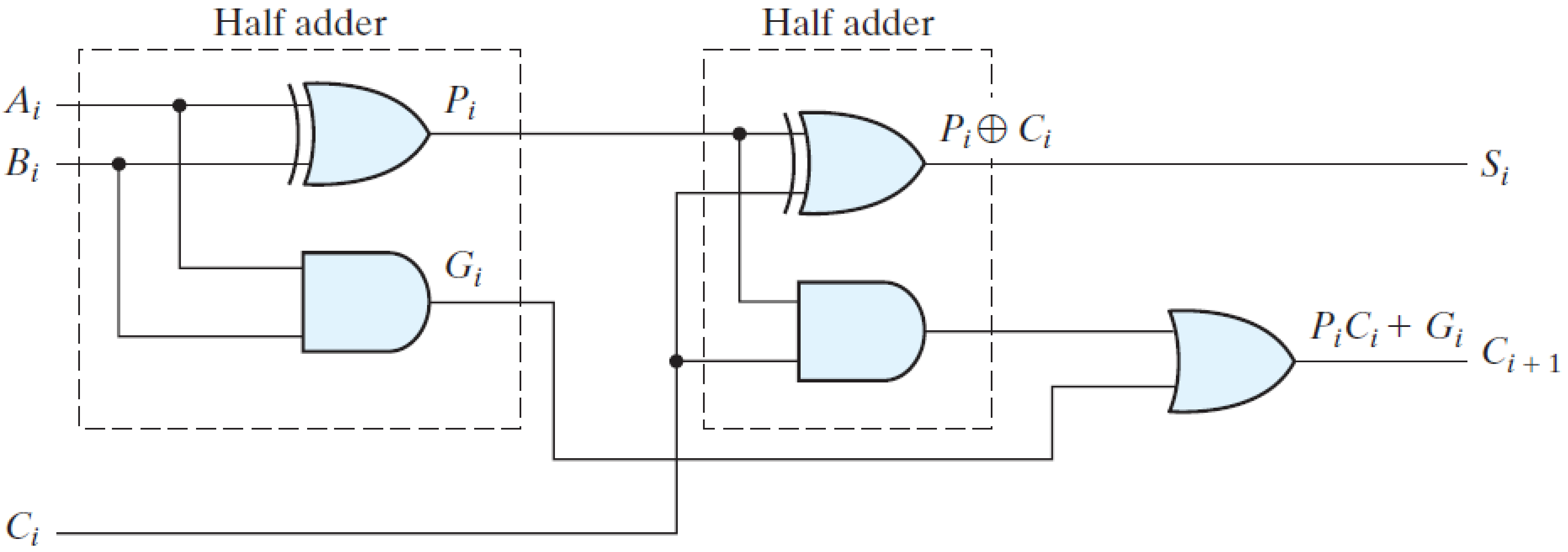
- G_i is called a *carry generate*, and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i
- P_i is called a *carry propagate*, because it determines whether a carry into stage *i* will propagate into stage *i* + 1



Carry propagation problem

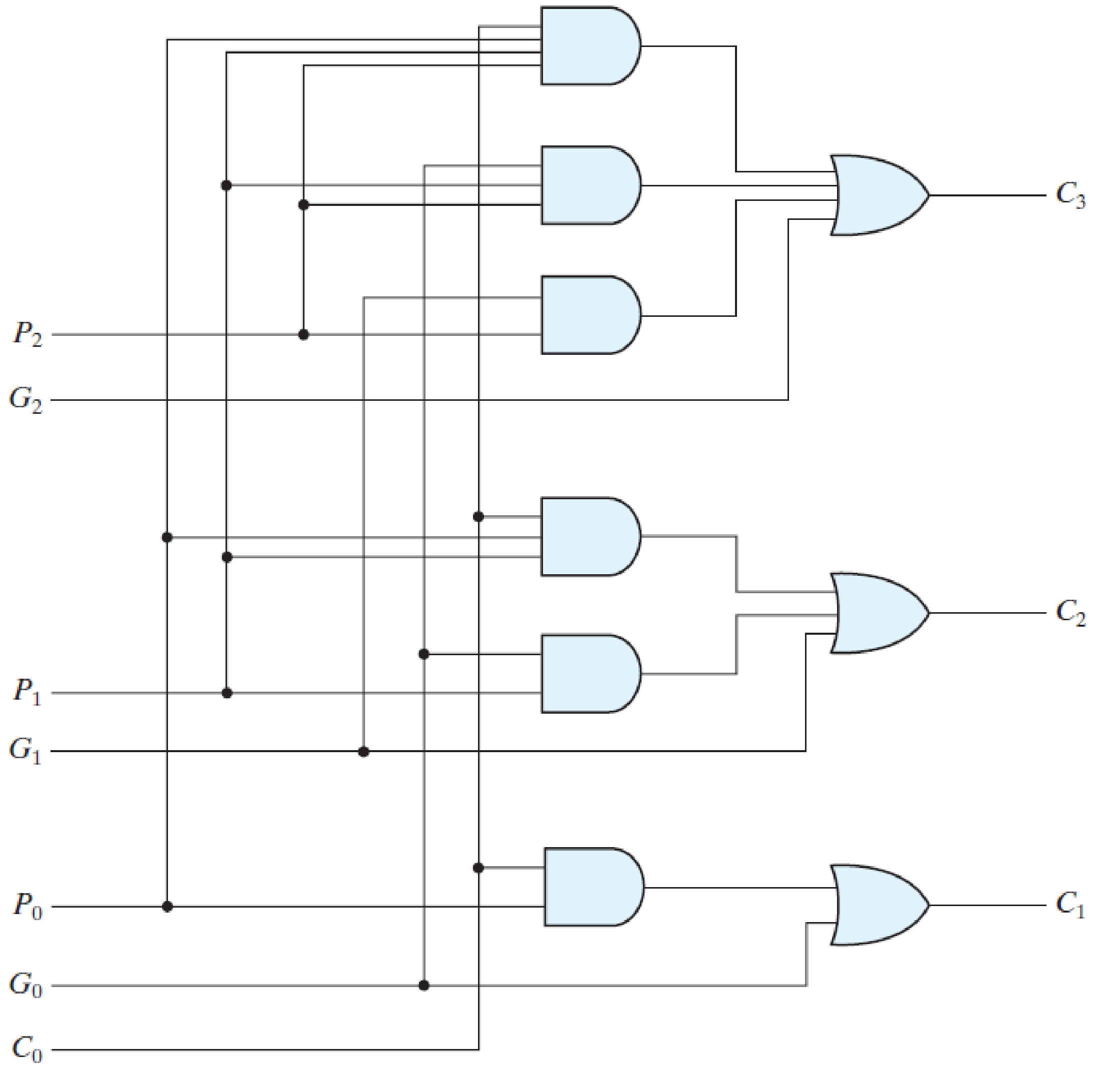
- We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

$$C_0 = input\ carry$$
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$



Carry propagation problem

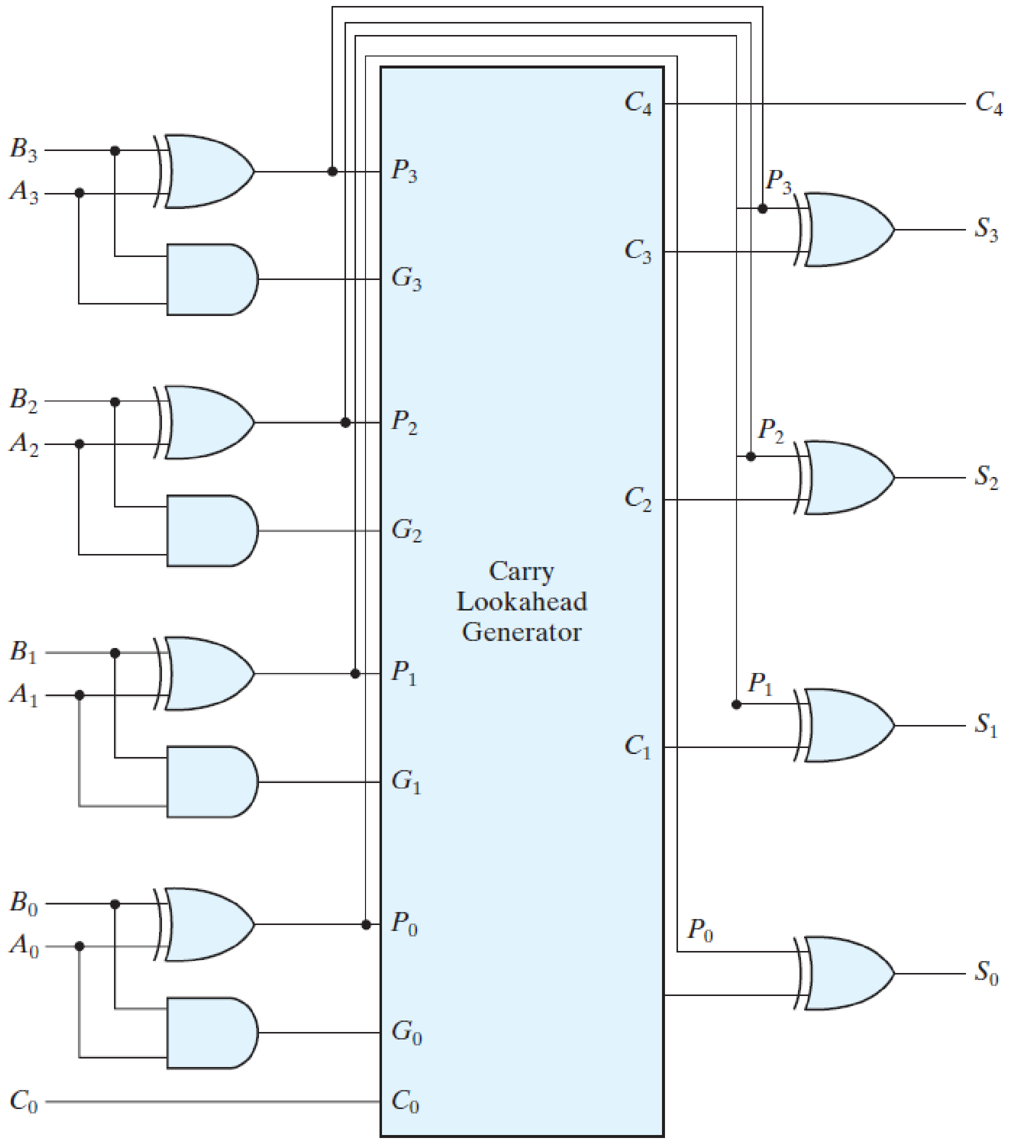
- Each output carry is expressed in sum-of-products form only dependent on P and G, each function can be implemented with one level of AND gates followed by an OR gate
- This circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2
- This gain in speed of operation is achieved at the expense of additional complexity (hardware)



Carry lookahead generator

Carry propagation problem

- Each sum output requires two XOR gates
- The output of the first XOR gate generates the P_i variable, and the AND gate generates the G_i variable
- The carries are propagated through the carry lookahead generator and applied as inputs to the second XOR gate
- *All output carries are generated after a delay through only two levels of gates*
- Thus, outputs S_1 through S_3 have equal propagation delay times



The Binary subtractor

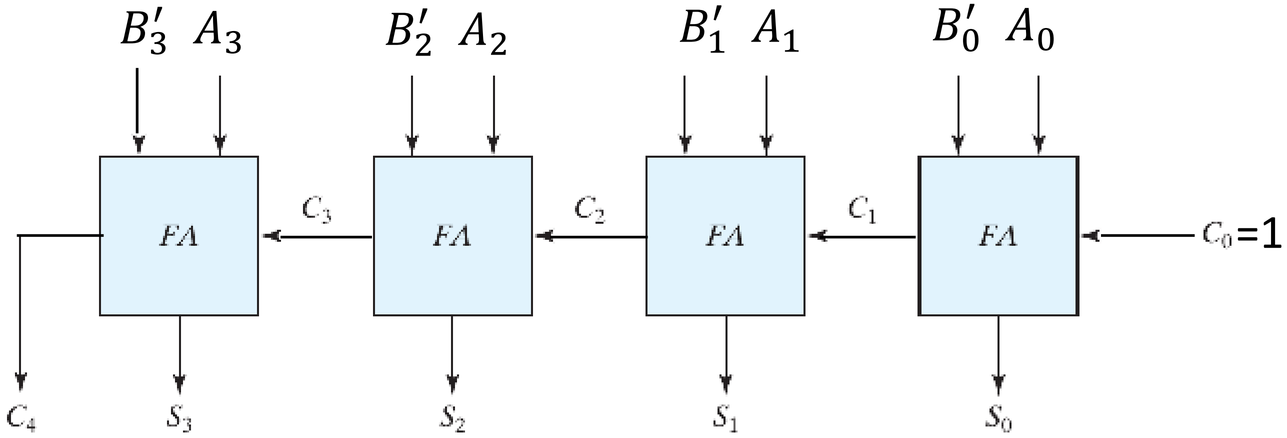
Revisit : Subtraction with Radix complements

- Subtraction using method of borrowing is less efficient when implemented with digital hardware
- Consider subtraction $M - N$ in base *r*

- Here is the algorithm using Radix complement:
  1. Take radix complement of subtrahend *N*: $r^n - N$
  2. **Add** this to *M*: $(r^n - N) + M = r^n + (M - N) = r^n - (N - M)$
  3. If you get a carry in the (n+1)th digit, then the result is positive, discard the carry and you are done
  4. If you ***do not*** get a carry in the (n+1)th digit, then the result is **negative**. Take the radix complement of the number to get the answer, then put a negative sign

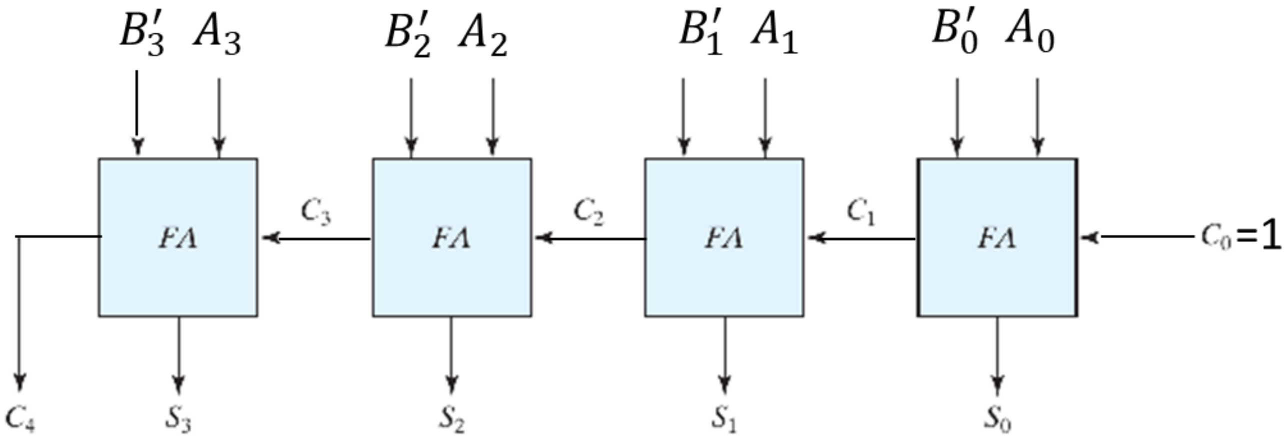
Binary subtractor

- The subtraction of binary numbers can be done most conveniently by means of complements
- Remember that the subtraction *A* - *B* can be done by taking the 2’s complement of *B* and **adding** it to *A*
- The 2’s complement can be obtained by taking the 1’s complement and adding 1 to the least significant pair of bits
- The 1’s complement can be implemented with inverters, and a 1 can be added to the sum through the input carry



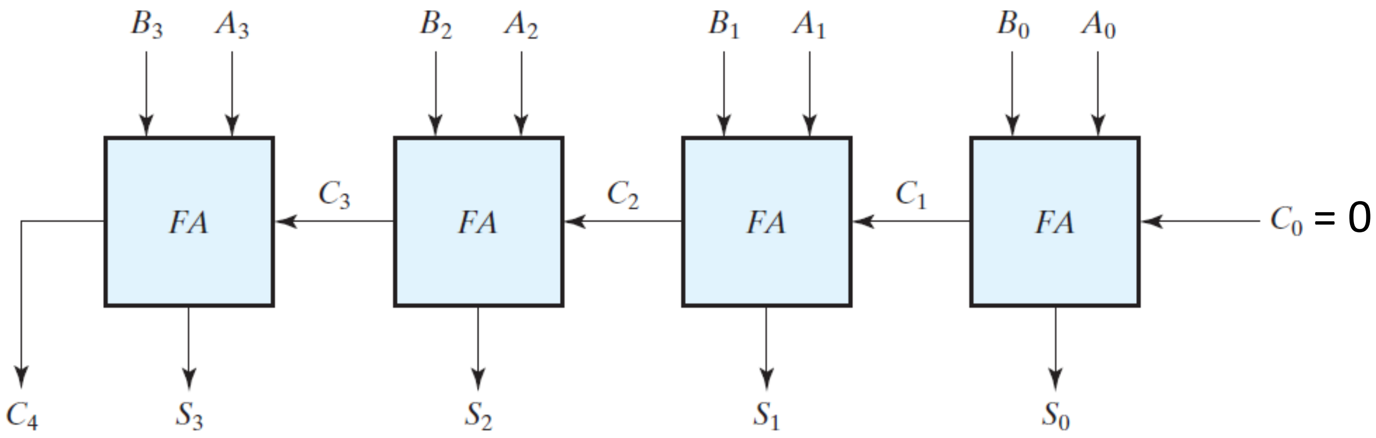
Binary subtractor

- The circuit for subtracting *A* - *B* consists of an adder with inverters placed between each data input *B* and the corresponding input of the full adder
- **The input carry C_0 must be equal to 1 when subtraction is performed**
- The operation thus performed becomes: *A* + 1's complement of *B* + 1.
  - This is equal to *A + 2's complement of B*
- That gives *A* - *B* if $A \geq B$ or the 2's complement of *B* - *A* if $A < B$

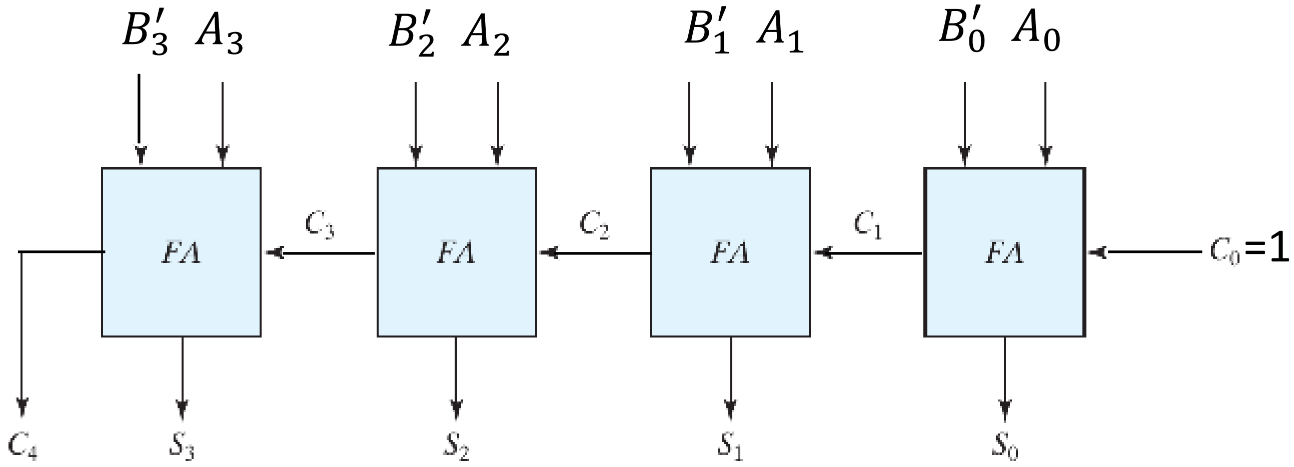


Can we combine the binary adder & subtractor

- Both use the 4-bit full adder. In one case, we use B and in another we use inverted B



Adder

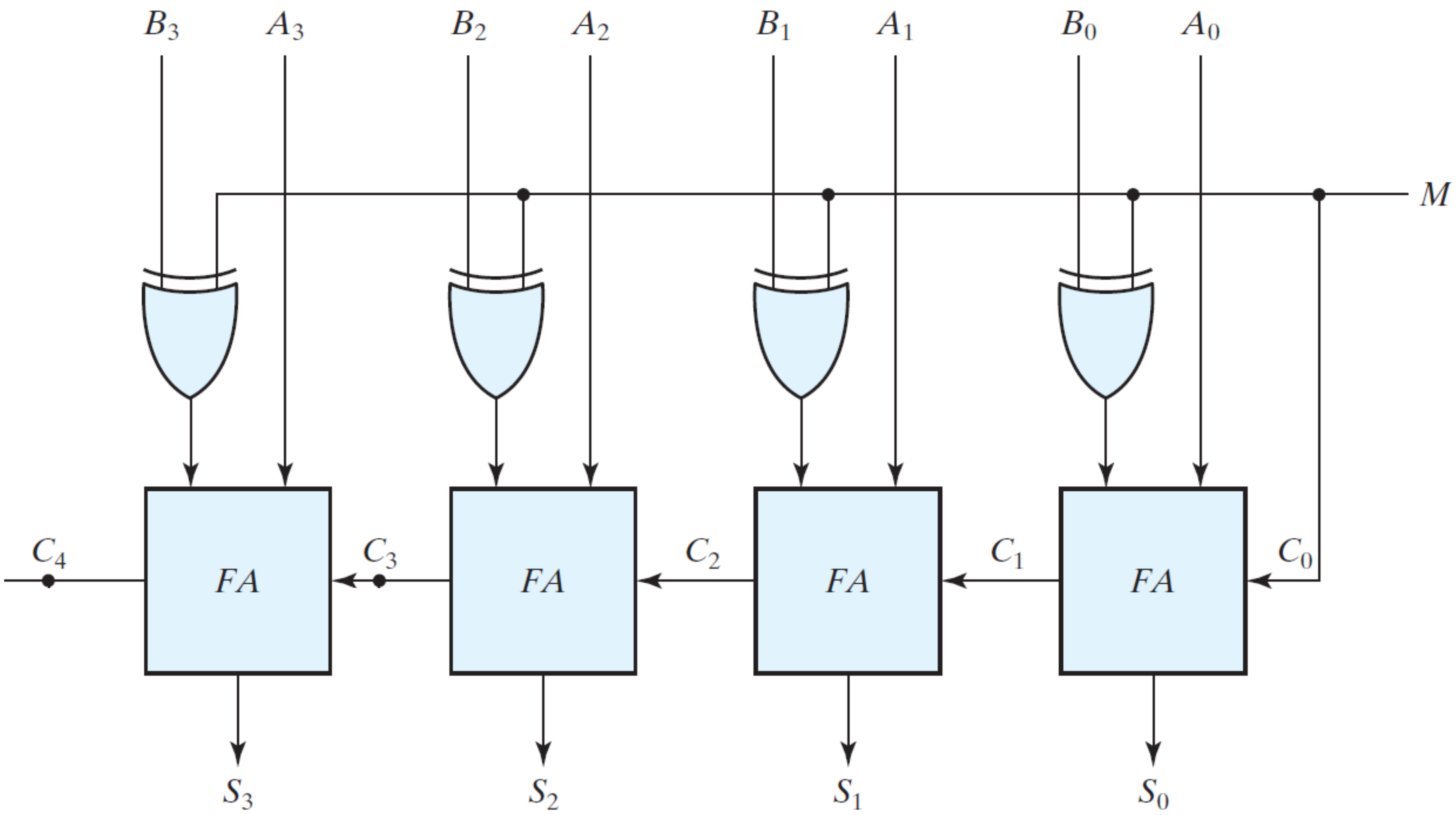


Subtractor

Binary adder-subtractor

- Here is some magic: The addition and subtraction operations can be combined into one circuit
- The mode input *M* controls the operation
- When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor
- When $M = 0$, the full adders receive the value of *B* , the input carry is 0, and the circuit performs $A + B$
- When $M = 1$, the full adders receive B' and $C_0 = 1$
- Thus, the *B* inputs are all complemented and a 1 is added through the input carry
- The circuit performs the operation *A* plus the 2’s complement of *B*

Binary adder-subtractor



The BCD adder

BCD adder

- Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage
- Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry
- Suppose we apply two BCD digits to a four-bit binary adder
- The adder will form the sum in *binary* and produce a result that ranges from 0 through 19

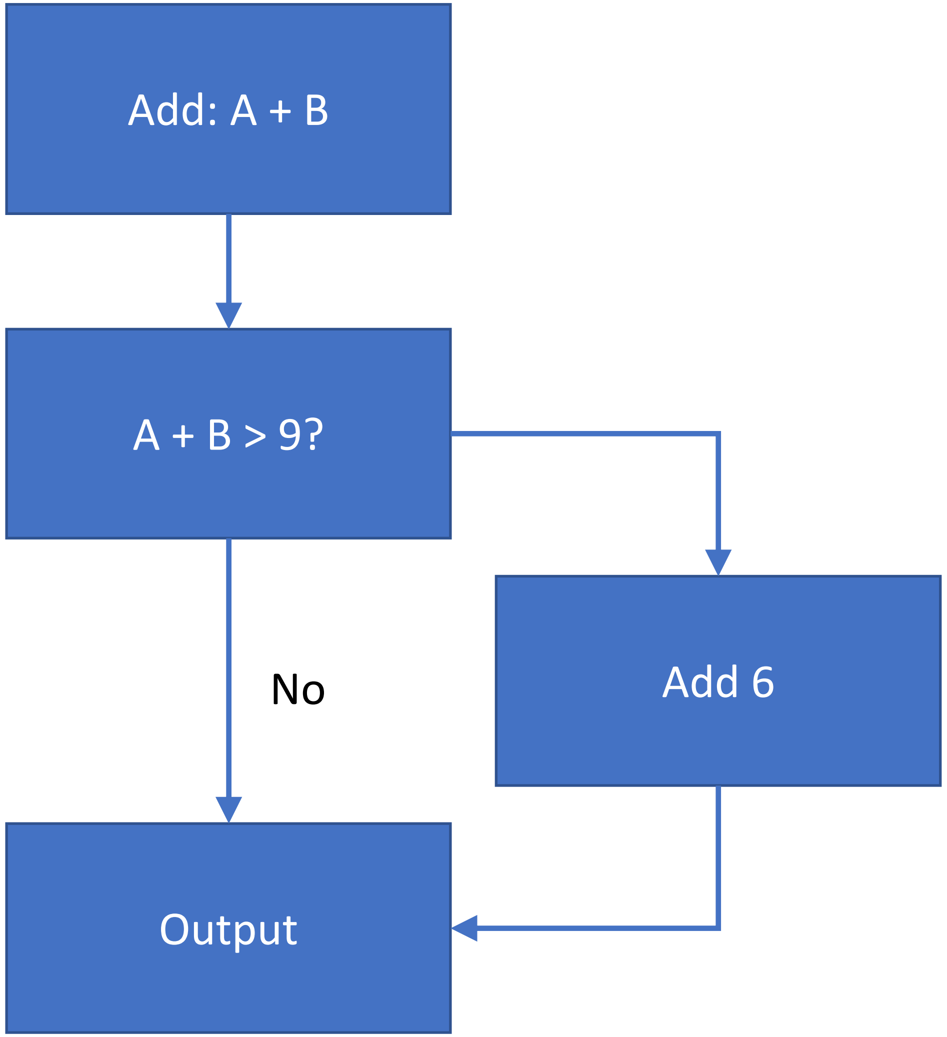
Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

BCD adder

- When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed
- When the binary sum is greater than 1001, we obtain an invalid BCD representation
- *The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required*

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

BCD adder - algorithm



Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

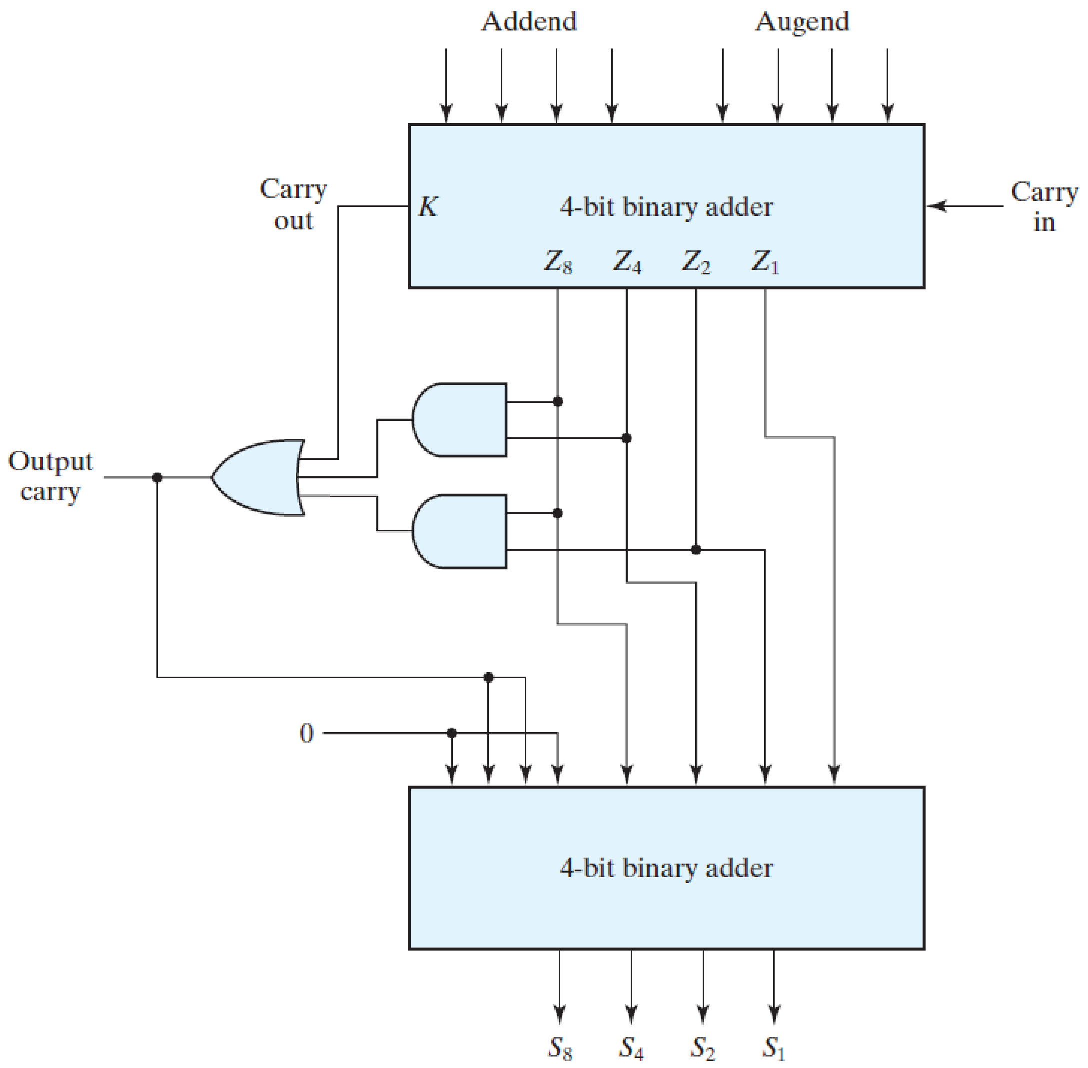
BCD adder

- It is obvious that a correction is needed when the binary sum has an output carry *K* = 1
- The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z_8
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1
- Thus, the condition for a correction and an output carry can be expressed by the Boolean function: $Cor = K + Z_8 Z_4 + Z_8 Z_2$
- When *Cor* = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

BCD adder

- The logic circuit that detects the necessary correction can be derived from the entries in the table
- It is obvious that a correction is needed when the binary sum has an output carry *K* = 1
- The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z_8
- To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1
- Thus, the condition for a correction and an output carry can be expressed by the Boolean function: $Cor = K + Z_8Z_4 + Z_8Z_2$
- When *Cor* = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage



The Binary multiplier

Binary multiplier

- Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers
- The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit
- Each such multiplication forms a partial product
- Successive partial products are shifted one position to the left
- The final product is obtained from the sum of the partial products
- Consider a 2-bit x 2-bit multiplier. Number of inputs/outputs?

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0

Binary multiplier

- The multiplicand bits are B_1 and B_0, the multiplier bits are A_1 and A_0, and the product is $C_3C_2C_1C_0$

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0