Combinational Logic (CLN)

CO 2206 Computer Organization

Topics

- Design process refined
- Design example
- Combinational Functional Blocks
  - Decoders / Encoders
  - Multiplexers / Demultiplexers
  - Arithmetic

Design Procedure - 1

1. Specification
   - Write a specification for the circuit
   - Text or HDL description
     - Hardware Description Language (HDL) will be covered in future lectures
2. Formulation
   - Derive truth table or Boolean expressions that defines the relations between inputs and outputs
3. Optimization
   - Provide a netlist (connection information) for the resulting circuit

Design Procedure - 2

- Simplification or optimization based on specific criteria
    - e.g. gate cost, gate delay, fan-out limits, etc

4. Technology mapping
    - Transform the logic diagram or netlist to a new diagram or netlist that implementation technology supports
    - Optimization and mapping may repeat multiple times to meet specification
5. Verification
    - Verify the correctness of final design

BCD-to-Excess-3 Code Converter

- Specification
  - Excess-3 code: binary combination corresponding to the decimal digit plus 3
  - E.g. excess-3 code for 5 is 1000 (i.e. 5+3=8)
- Formulation
  - Can be obtained from BCD code word by adding 0011(3) to it
  - 1010 through 1111 are not listed since they have no meaning in BCD code

BCD-to-Excess-3 Code Converter

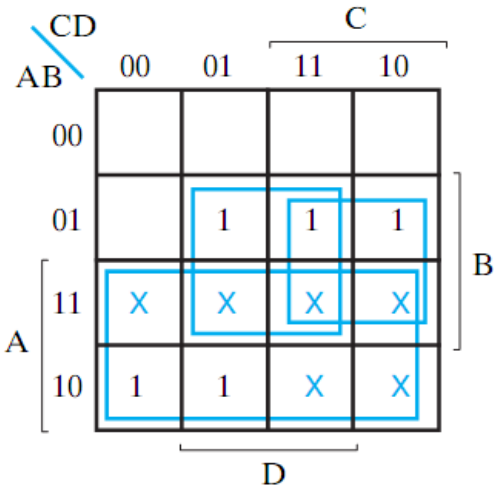
Truth Table for Code Converter Example

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

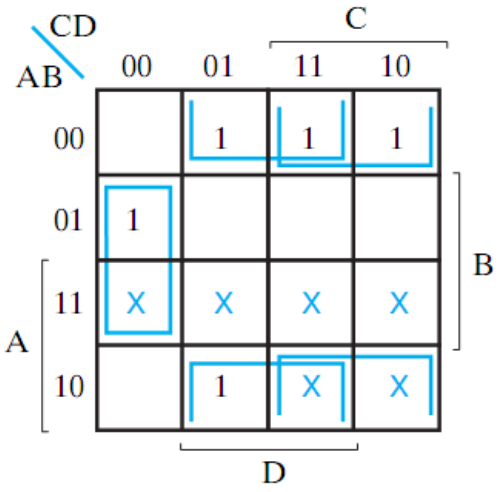
undefined outputs will be marked don't care

BCD-to-Excess-3 Code Converter

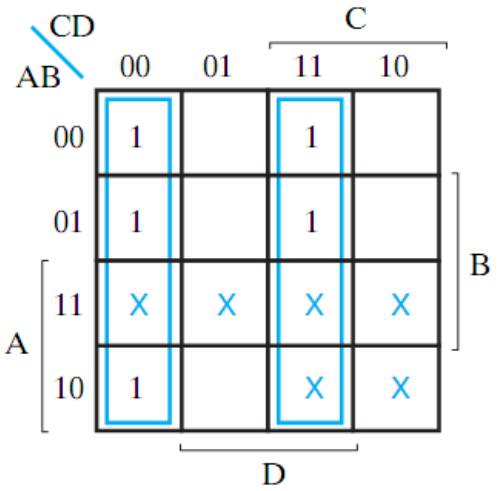
- Optimization



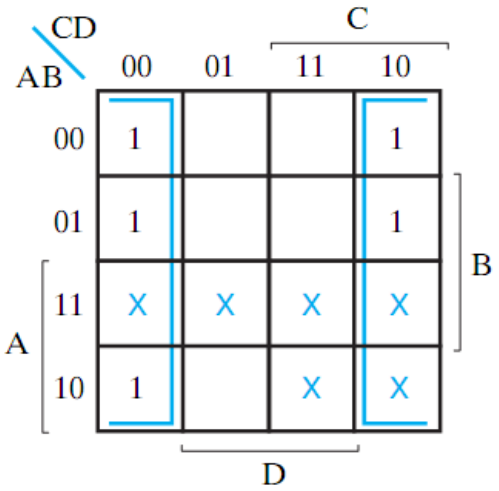
$W = A + BC + BD$



$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D}$



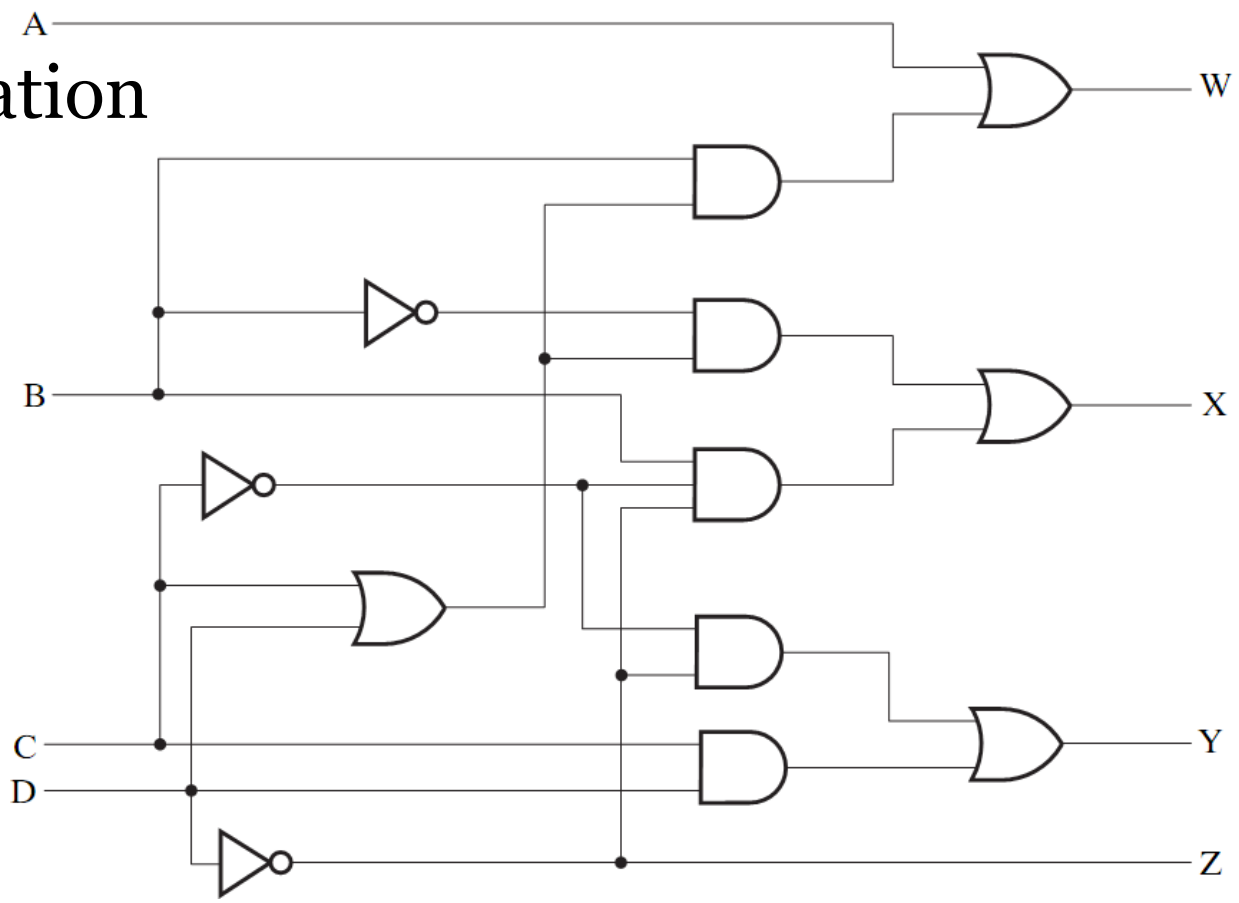
$Y = CD + \overline{C}\overline{D}$



$Z = \overline{D}$

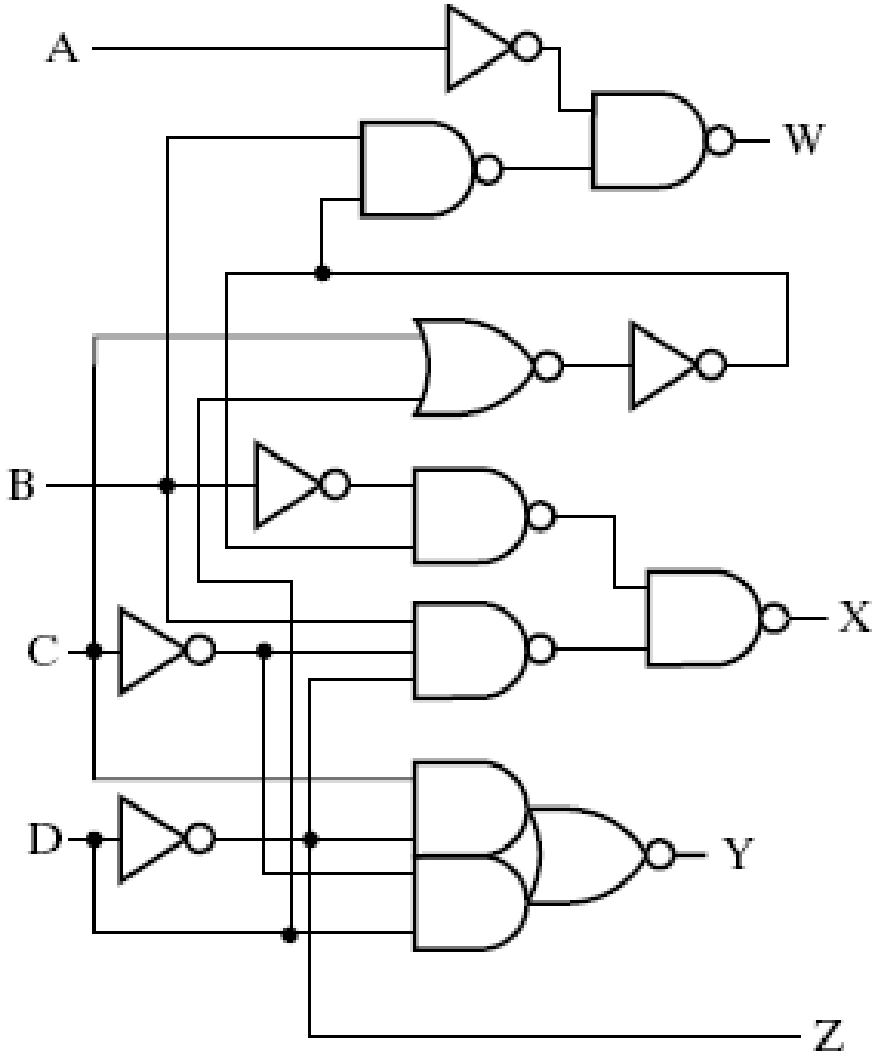
BCD-to-Excess-3 Code Converter

- Direct implementation



BCD-to-Excess-3 Code Converter

- Technology mapping (NAND, NOR)



Verification - 1

- Verification
  - Determination of whether or not a given circuit implements its specified function
    - if the circuit does not meet its spec, then it is incorrect
  - It is essential that the spec is unambiguous and correct
    - Spec such as truth table, Boolean equations and HDL code are most useful

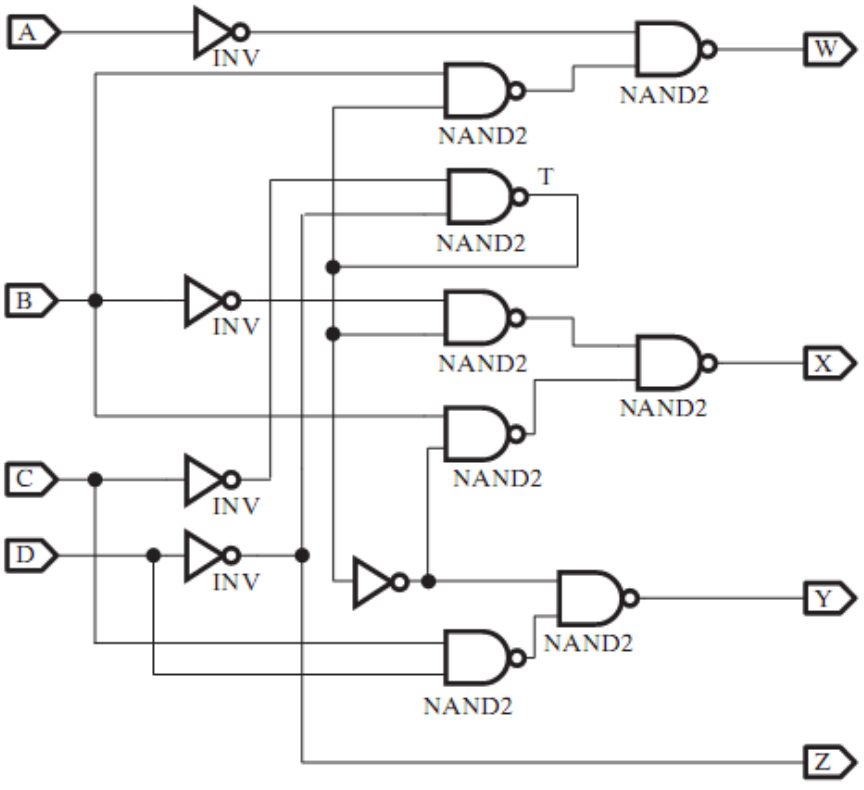
Verification - 2

- Manual logic analysis
  - Finding the equations and then using them to find the truth table, if necessary
  - If new truth table matches the original one, the circuit is correct
  - E.g. manual verification of BCD-to-Excess-3 code converter

Verification - 3

Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

(a)



(b)

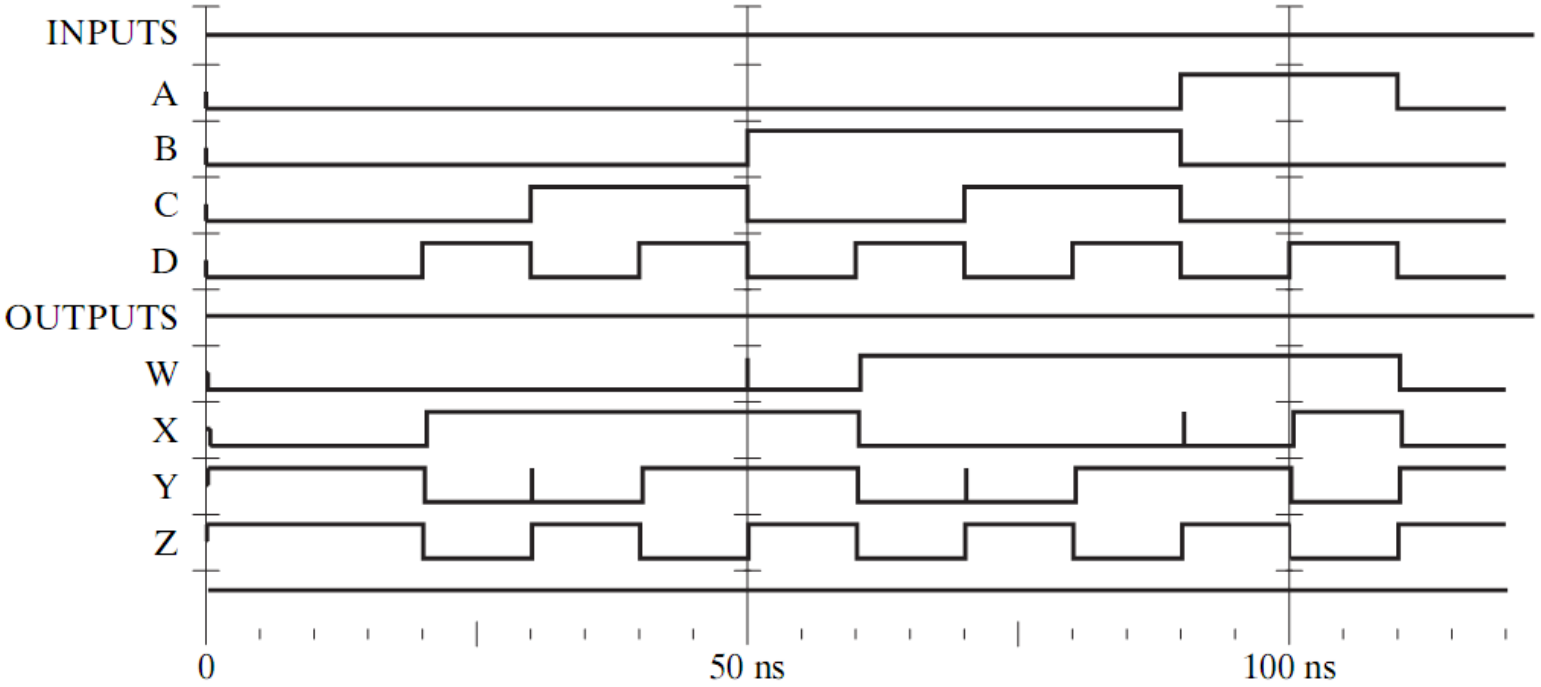
Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0				1
0	0	0	1				
0	0	1	0		1		1
0	0	1	1		1	1	
0	1	0	0				1
0	1	0	1	1			
0	1	1	0	1			1
0	1	1	1	1		1	
1	0	0	0	1			1
1	0	0	1	1			

(c)

W, Z correct
X, Y incorrect

Verification - 4

- Using computer simulation
  - Useful for large number of variables
  - Greatly reduces the tedious analysis effort required

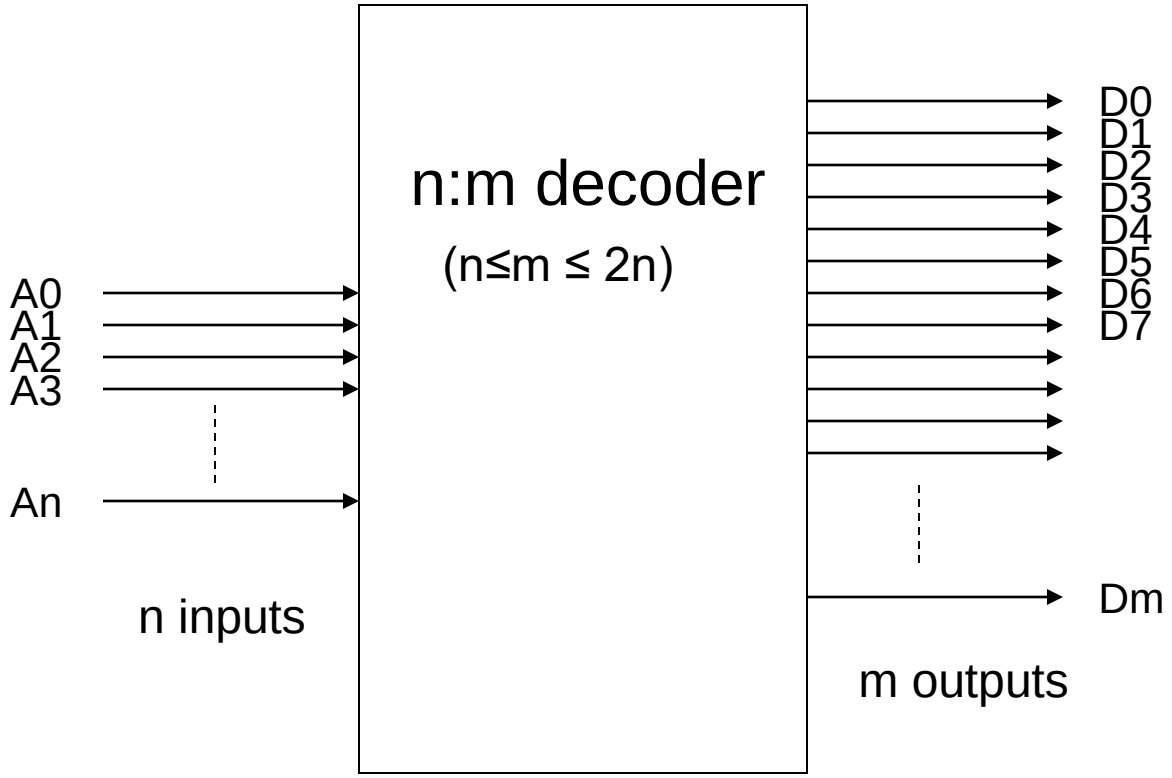


Combinational Functional Blocks

n-to-m Decoder

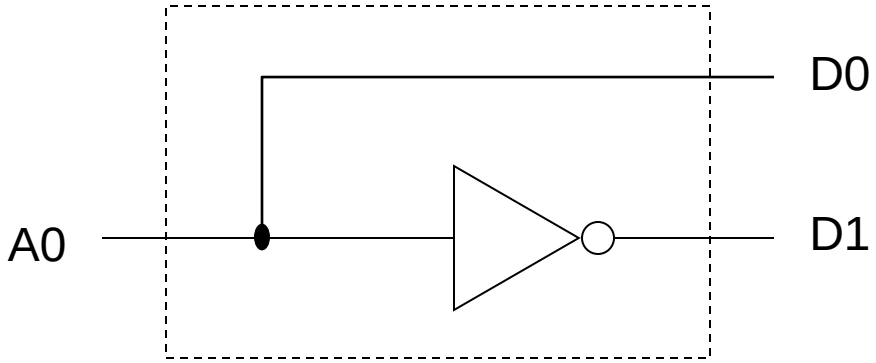
($n \leq m \leq 2^n$)

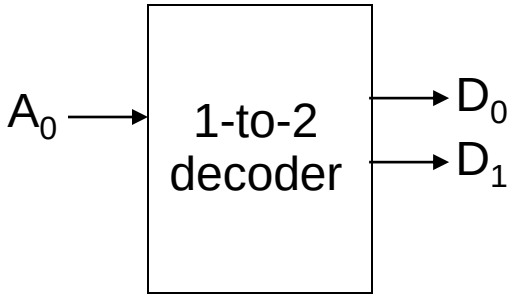
- A decoder converts binary information from n input lines to a maximum of 2^n unique output
  - If there are unused or don't care input combinations, the decoder output will have fewer than 2^n outputs
  - Each output represents one minterm
  - Only one output is active at any one time
  - Equivalent to binary-to-decimal
    - Decode: binary is the code, decimal is the meaning
  - Usages:
    - selecting boards or devices connecting to same bus
    - decode instructions to determine the operations to be performed in the processor



D0=1 when An…A3A2A1A0 = 0 D6=1 when An…A3A2A1A0 = 6

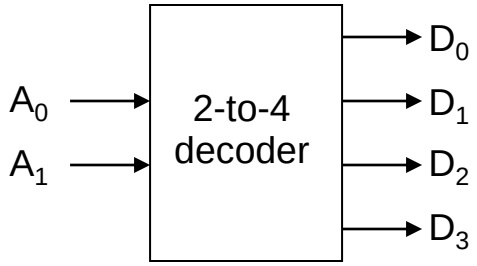
1-to-2 Decoder





A0	D1	D0
0	0	1
1	1	0

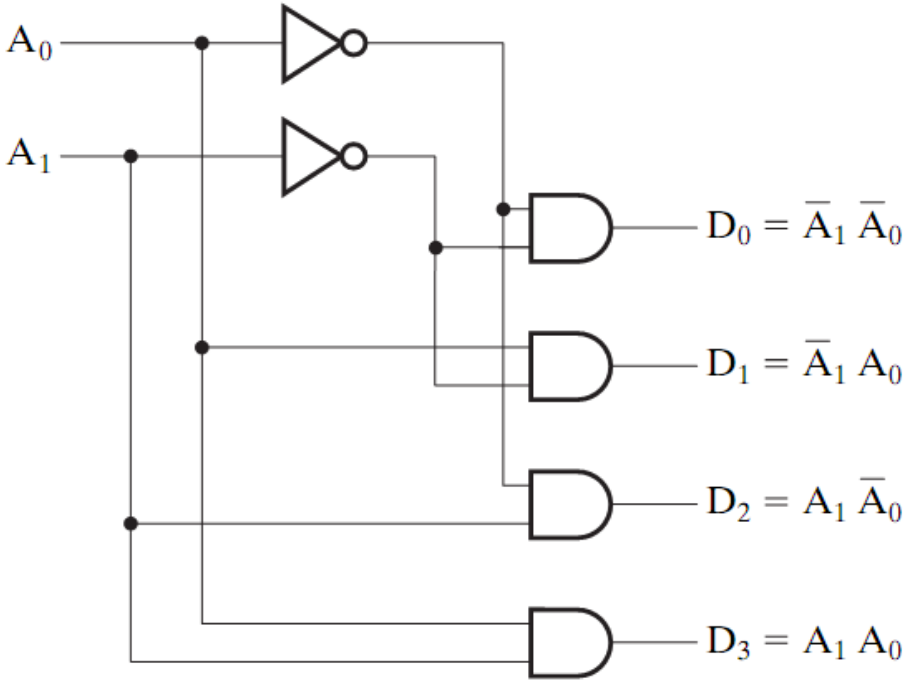
2-to-4 Decoder



- 2-to-4 line decoder
  - Only one output can be equal to 1 at any one time

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



(b)

Each minterm is implemented by an AND

Decoder: Minterm Implementation

- Large decoders can be constructed by
  - Implementing each minterm using a AND gate with more inputs
  - Unfortunately, this approach gives high gate-input costs
    - number of inputs

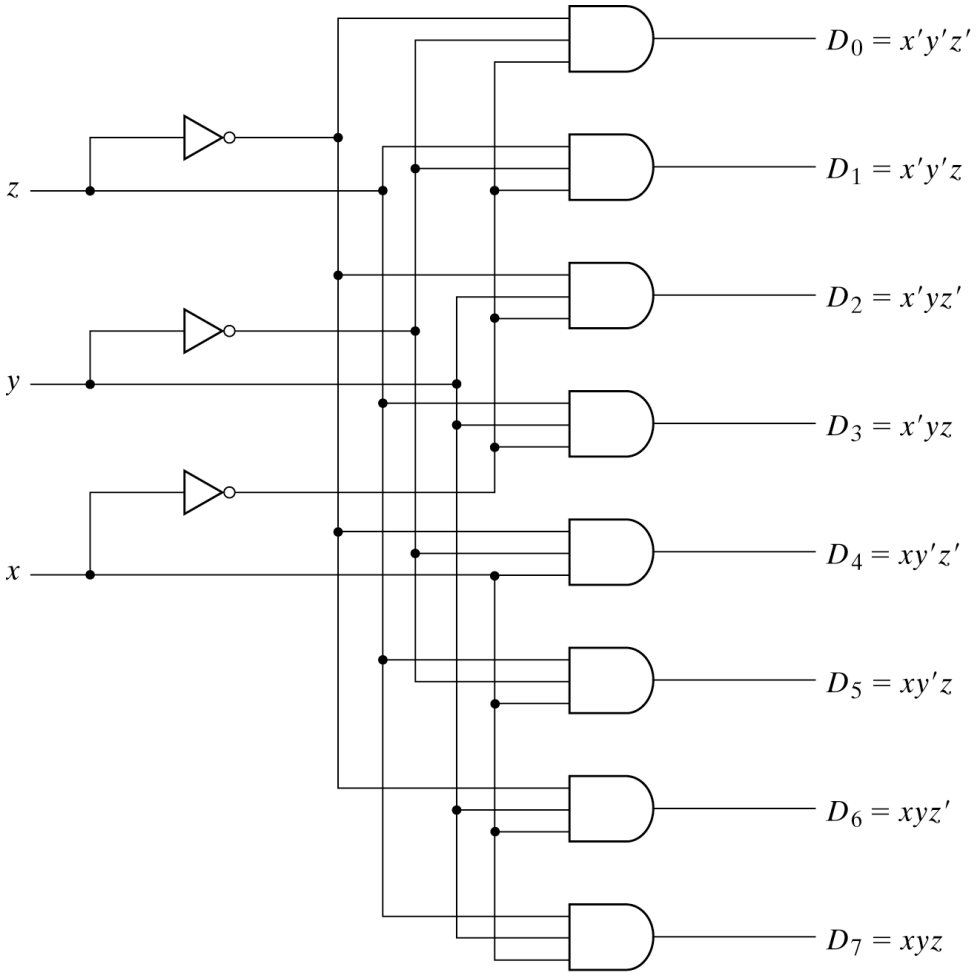


Fig. 4-18 3-to-8-Line Decoder

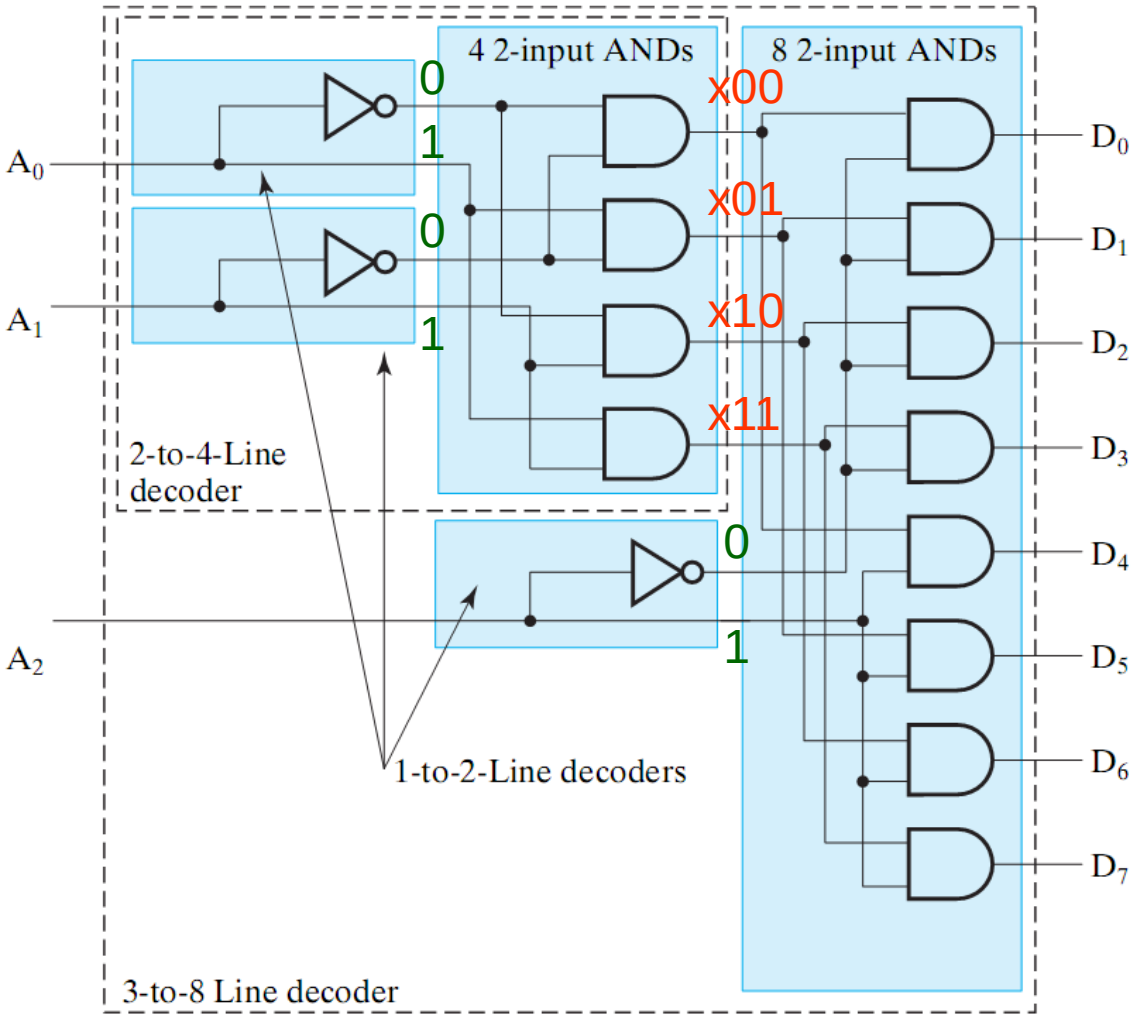
Decoders: Hierarchy Implementation

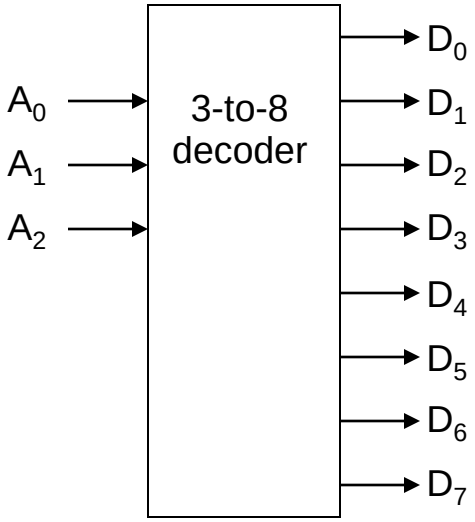
- Large decoders can be constructed using smaller decoders
- General procedure
  - If n is even
    - Use 2^n AND gates driven by
      - 2 decoders of output size $2^{n/2}$
  - If n is odd
    - Use 2^n AND gates driven by
      - 1 decoder of output size $2^{(n+1)/2}$
      - 1 decoder of output size $2^{(n-1)/2}$
  - Continue to divide n by 2 until n=1
    - For n=1, use a 1-to-2 decoder

3-to-8 Decoder - 1

- 3-to-8-line decoders, i.e. n=3
  - k_1=n=3
    - 2^{k1}=8 2-input AND gates
    - Driven by
      - 1 decoder of output size $2^{k1-1/2}$=2 (no further reduction) and
      - 1 decoder of output size $2^{k1+1/2}$=4 (k2=k1+1/2=2)
  - k_2=2
    - 2^{k2}=4 2-input AND gates
    - Driven by 2 decoders of output size $2^{k2/2}$=2

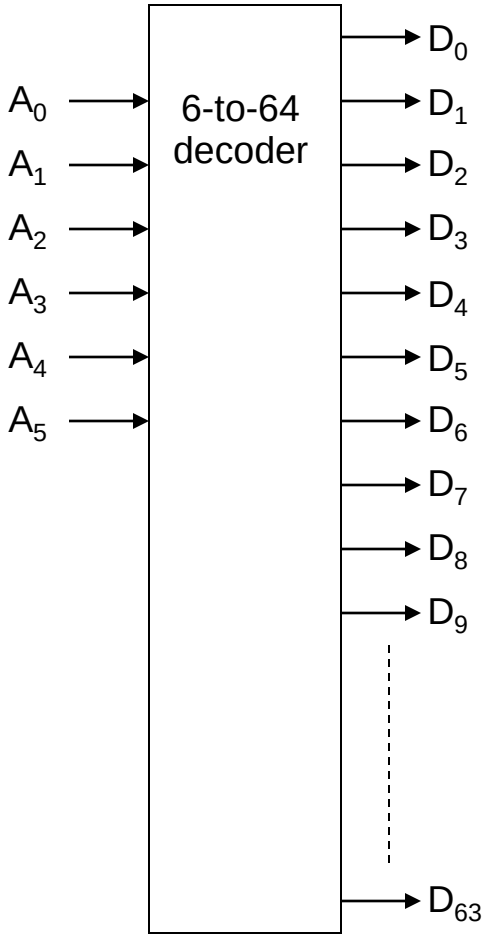
3-to-8 Decoder - 2



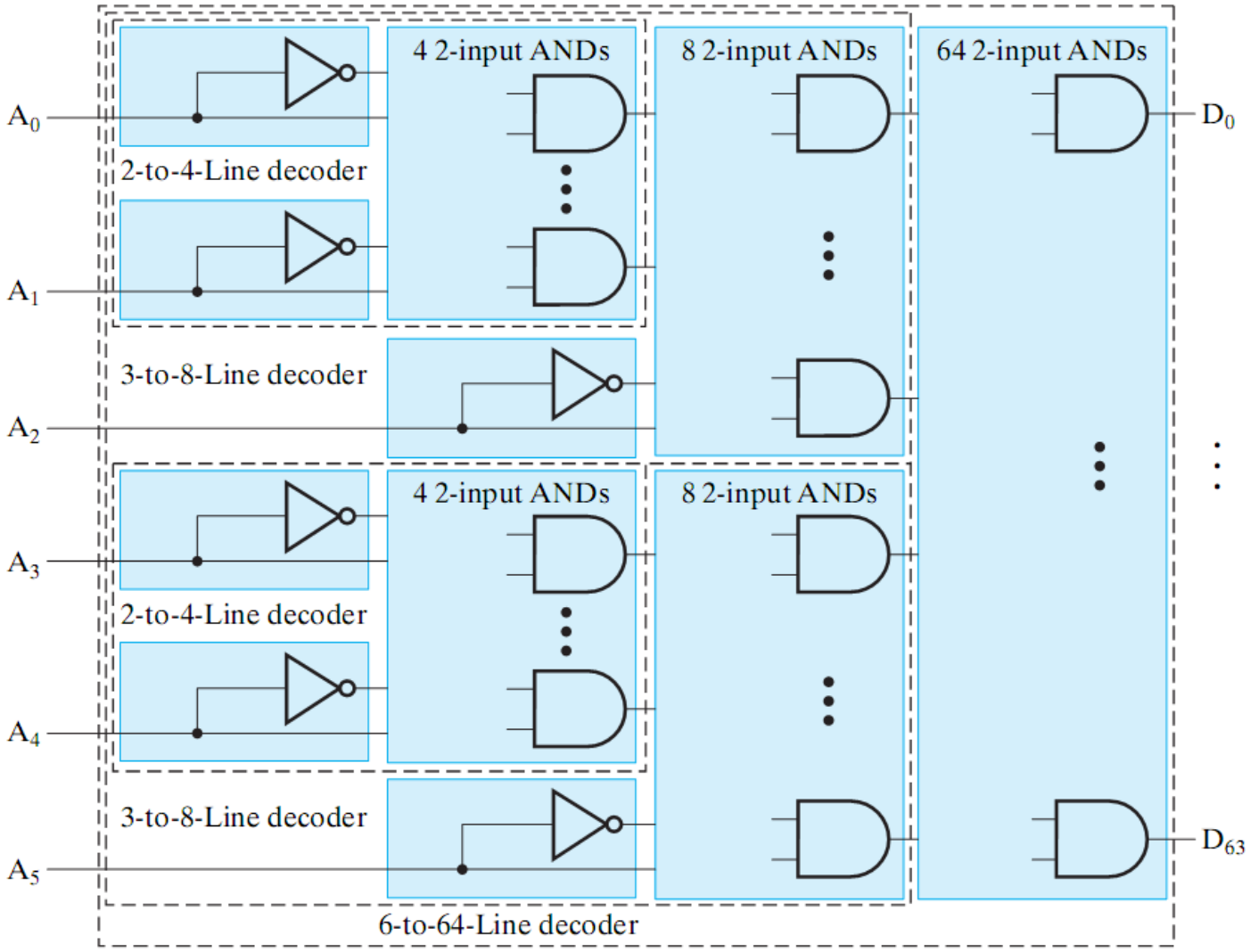


6-to-64 Decoder - 1

- 6-to-64-line decoders, i.e. n=6
  - $k_1=n=6$
    - $2^{k1}=64$ 2-input AND gates
    - Driven by 2 decoders of output size $2^{k1/2}=8$ ($k_2=k_1/2=3$)
  - $k_2=3$
    - $2^{k2}=8$ 2-input AND gates
    - Driven by
      - 1 decoder of output size $2^{k2-1/2}=2$ (no further reduction) and
      - 1 decoder of output size $2^{k2+1/2}=4$ (k3=k2+1/2=2)
  - $k_3=2$
    - $2^{k3}=4$ 2-input AND gates
    - Driven by 2 decoders of output size 2


A0
A1
A2
A3
A4
A5
6-to-64
decoder
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D63

6-to-64 Decoder - 2

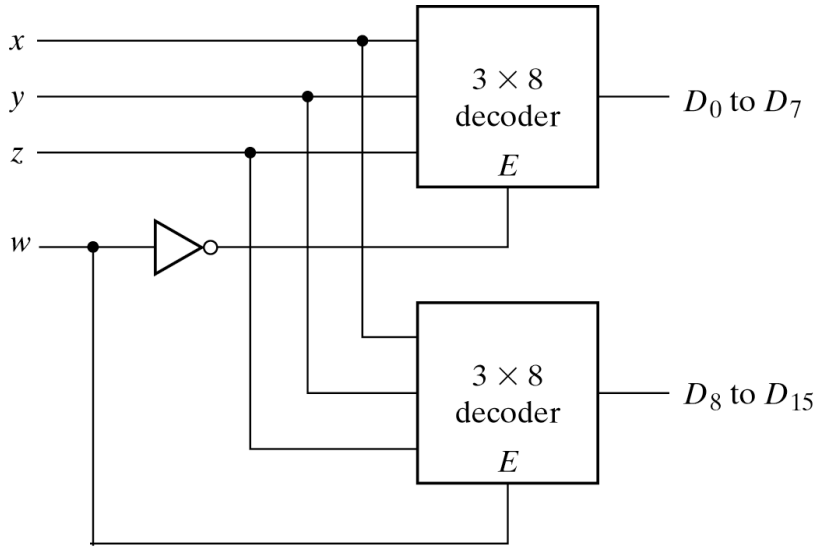


Decoders – the Cost

- *Gate input costs* - the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations
  - G - inverters not counted, GN - inverters counted
- For 6-to-64 decoder, if a single AND gate for each *minterm* were used
  - Gate-input cost
    - GN = 3 + (6x64) = 387
- Smaller decoders used
  - Gate-input cost
    - GN = 6 + 2(2x4) + 2(2x8) + 2x64 = 185

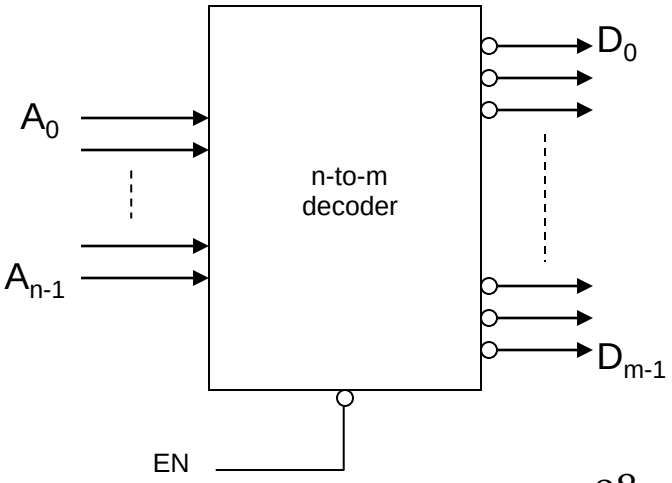
Decoder with Enable

- Large decoders can be constructed using smaller (one level, i.e. m/2) decoders with enabling
  - For example, 3x8 decoders with enable inputs connected to form a 4x16 decoder
    - When w=0 (top decoder enabled)
      - Top outputs generate *minterms* 0000 to 0111
    - When w=1 (bottom decoder enabled)
      - Bottom outputs generate *minterms* 1000 to 1111



Active-Hi vs Active-Lo

- To implement with NAND gates, it becomes more economical to generate the decoder minterms in their complemented form
- Small circles on the output lines indicate this
  - decoders designed to produce active-LOW outputs, where only the selected output is LOW while all others are HIGH
- Input can also be active-lo
  - EN=1, decoder disabled (all output inactive)
  - EN=0, decoder enabled

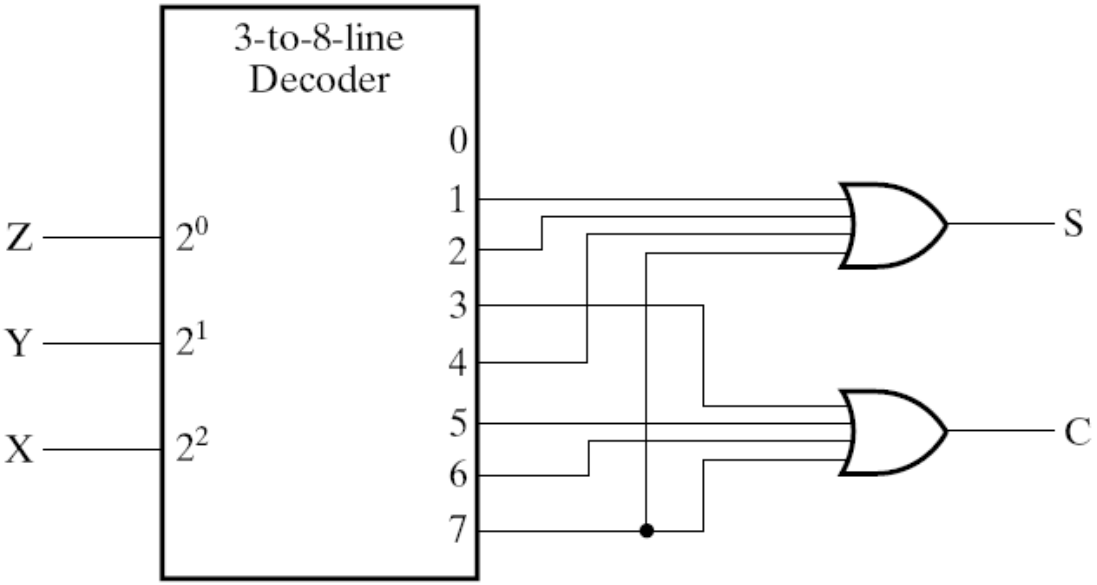


Implement CLN using Decoders

- Any combinational circuit with *n* inputs and *m* outputs, expressed as sum of minterms can be implemented with
  - an *n*-to-2^n decoder and
  - *m* OR gates
- We can practically implement any *CLN* by expressing the output functions in sum of *minterms*

Decoder Application Example

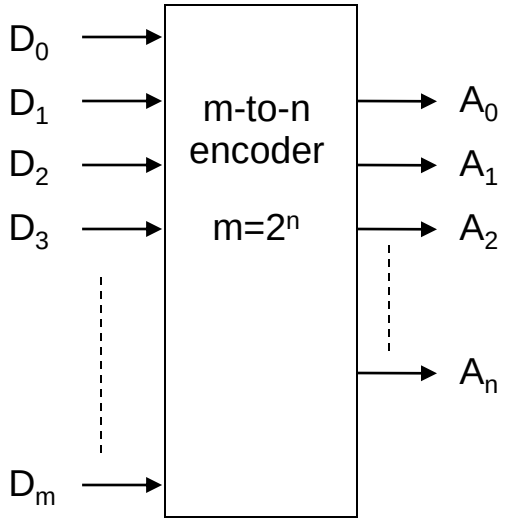
- Binary adder (one-bit with carry-in)
  - S(X,Y,Z) = Σm(1,2,4,7)
  - C(X,Y,Z) = Σm(3,5,6,7)



X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Encoders

- Performs inverse operation of a decoder
- Has 2^n (or fewer) input and *n* output lines
- Only one input has 1 at any time
  - simplify output expressions
- Usages:
  - converting "anything" to binary
  - encoding inputs, e.g. keyboards



When D0=1, An…A2A1A0 =0
When D3=1, An…A2A1A0 =3

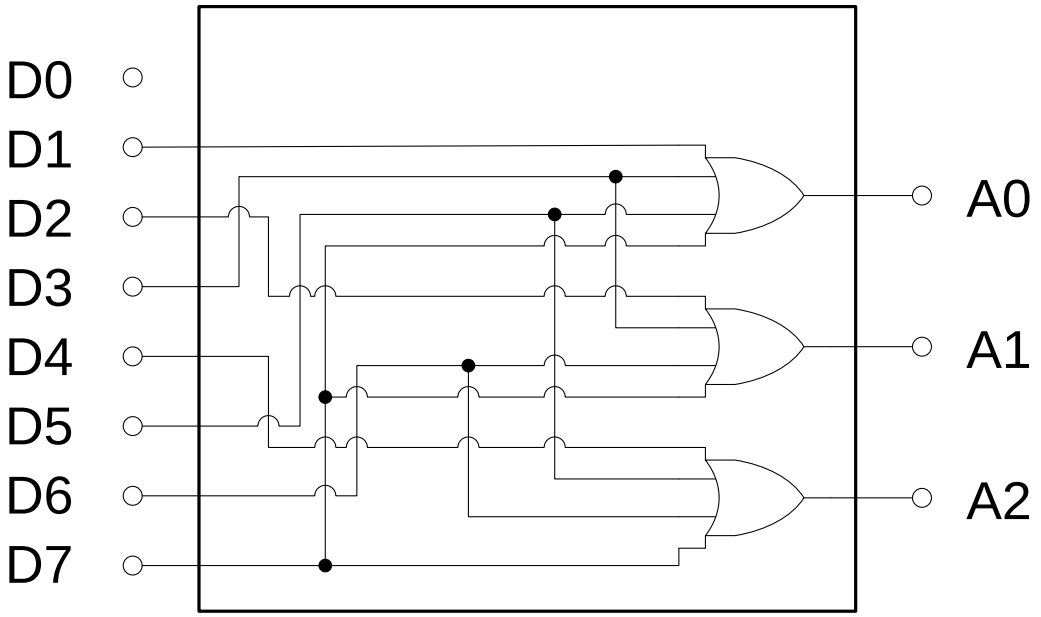
Encoder Example: Octal-to-binary

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

- Octal-to-binary encoder
  - 8 inputs, one for each octal digits
  - 3 outputs that generate the corresponding binary number
  - Truth table has only 8 rows
    - For the remaining 56 rows, all outputs are don't cares

A0 = D1 + D3 + D5 + D7
A1 = D2 + D3 + D6 + D7
A2 = D4 + D5 + D6 + D7

Draw the circuit: Octal to binary



Priority encoder

- 2 inputs cannot be active simultaneously
  - If D_3 and D_6 are 1 simultaneously, encoder output is 111 (7), not 3 or 6
    - output is incorrect
    - to resolve, establish input priority
- Priority encoder ensures if 2 or more inputs are active simultaneously
  - Highest priority input will take precedence
- Another ambiguity
  - Output 000 generated when all inputs 0
    - but 000 output when D_0=1
    - to resolve, assign another valid-output indicator to indicate at least one input is 1

Priority Encoder Illustration - 1

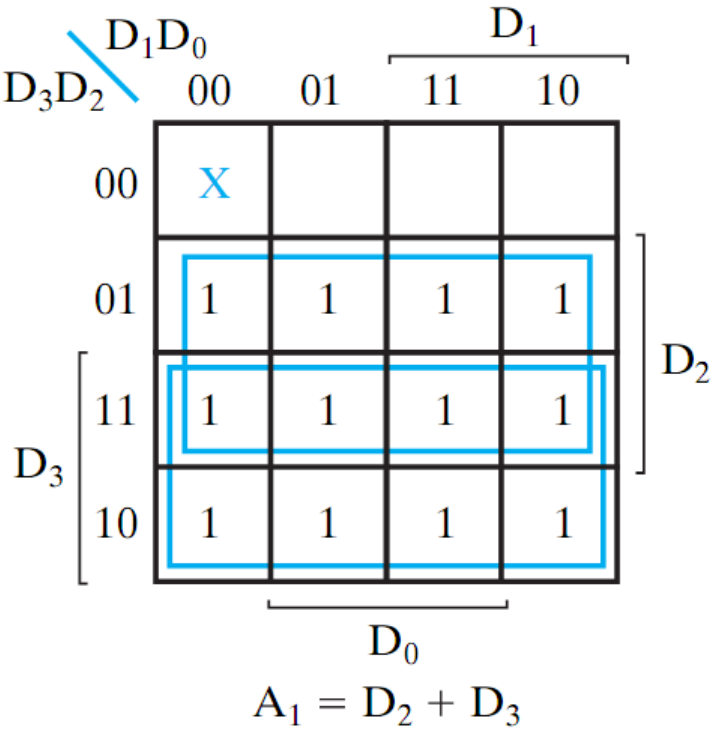
- D_3 has the highest priority
  - When $D_3 = 1$
    - Output for A_1A_0 is 11
  - When $D_2 = 1$
    - Output is 10, provided $D_3 = 0$
      - Otherwise output is 11

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

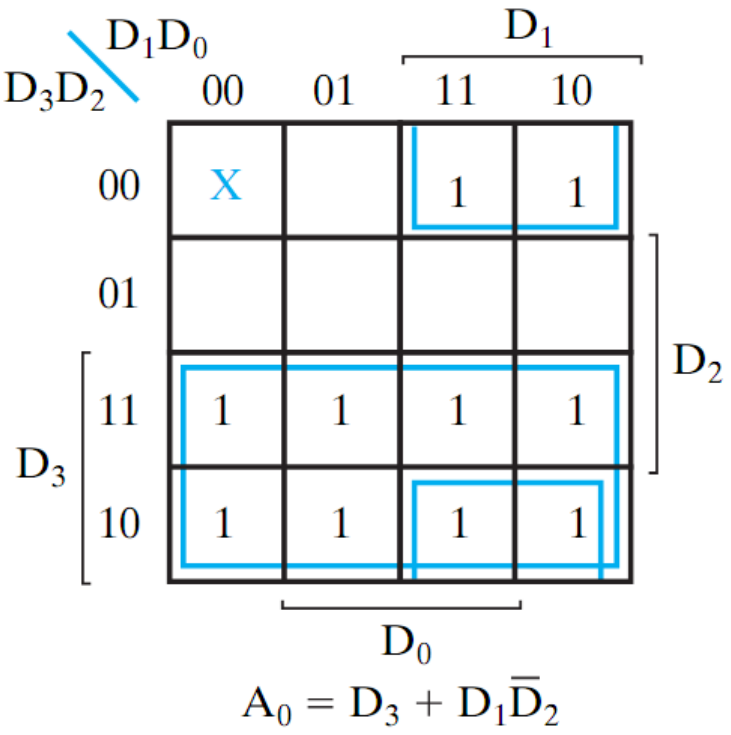
Priority Encoder Illustration - 2

  - Output for D_1 generated only if higher-priority inputs are 0
- Valid-output indicator V is set to 1
  - only when one or more inputs equal 1
  - if all inputs 0, V=0, and A_1A_0 not used
- Boolean functions
  - $A_0 = D_3 + D_1D_2'$
  - $A_1 = D_2 + D_3$
  - $V = D_0 + D_1 + D_2 + D_3$

Priority Encoder Illustration - 3

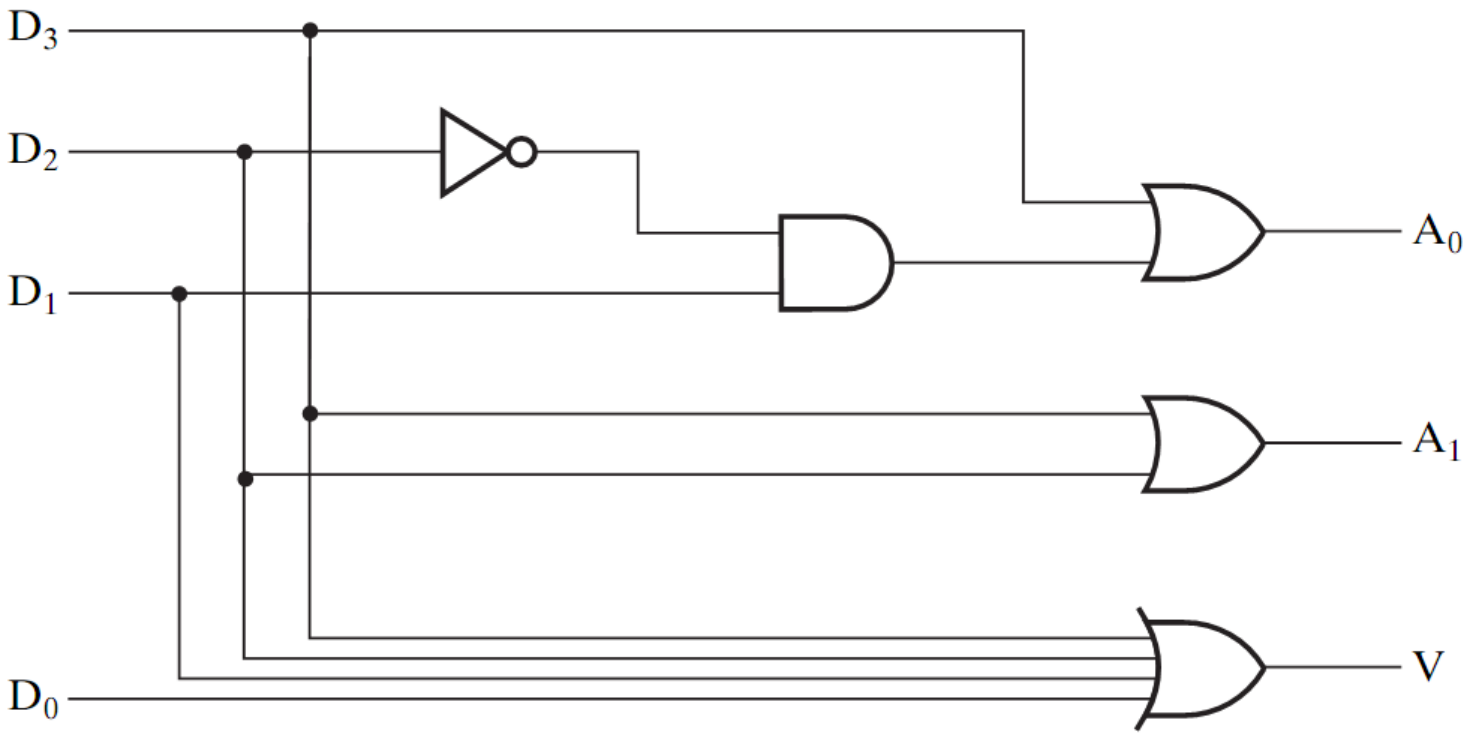


$A_1 = D_2 + D_3$



$A_0 = D_3 + D_1\overline{D}_2$

Priority Encoder Illustration - 4



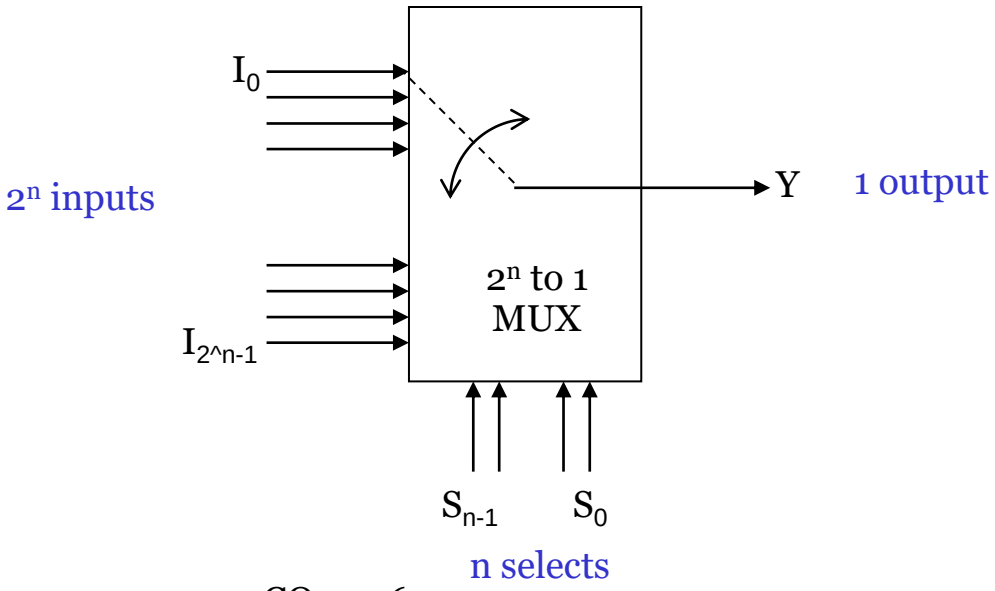
Selecting

- Selection of info is a very important function
- Circuits that perform selection typically have
  - A set of inputs from which selection are made
  - A single output
  - A set of control lines for making the selection

Multiplexers

2^n-to-1 MUX

- ***Multiplexer*** selects binary info from one of many input lines and directs it to a single output line
  - Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected



Example: 2-to-1 Multiplexer - 1

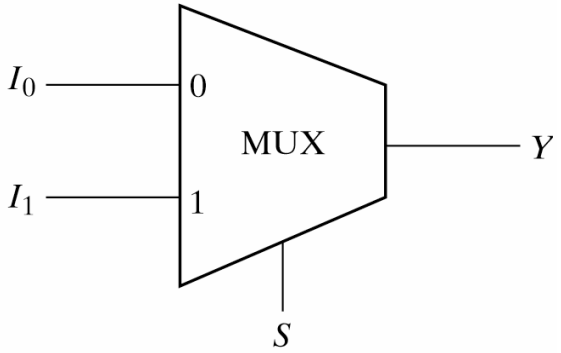
- 2-to-1 mux has
  - 2 inputs I_0 and I_1, and
  - Selection line S
  - When S = 0,
    - Output Y = I_0
  - When S = 1,
    - Output Y = I_1
  - Thus S selects which input to appear at Y
  - $Y = S' I_0 + S I_1$

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

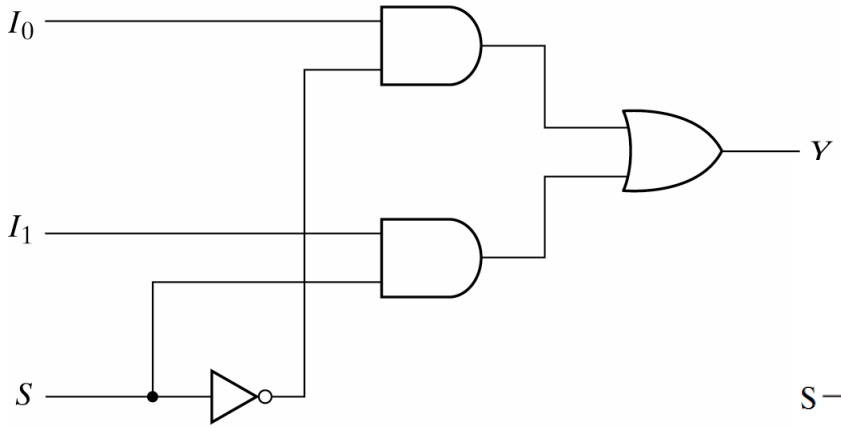
$S I_0 I_1$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

Example: 2-to-1 Multiplexer - 2

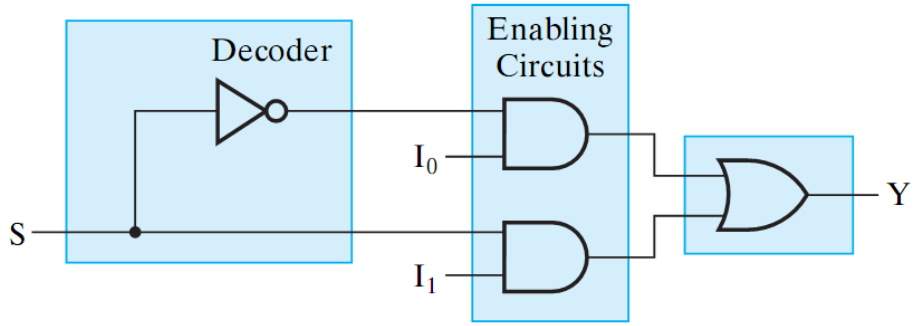
- *Multiplexer* can be constructed from:
  - an n-to-2^n decoder
  - n AND gates (enabling circuit); one at each decoder output
  - an OR gate at the output



(b) Block diagram

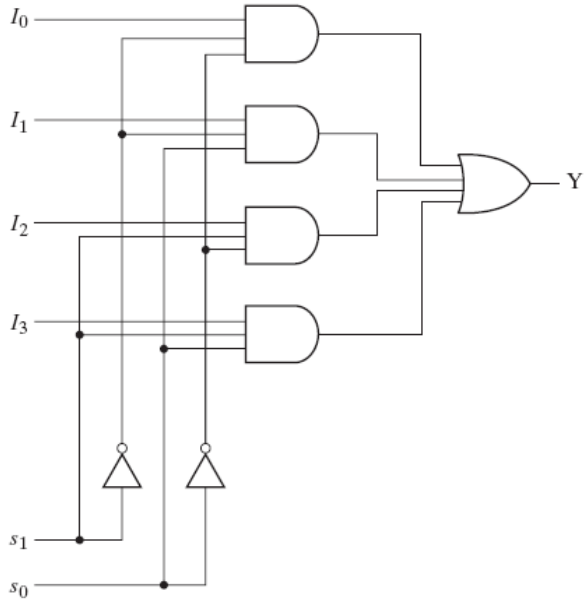


(a) Logic diagram

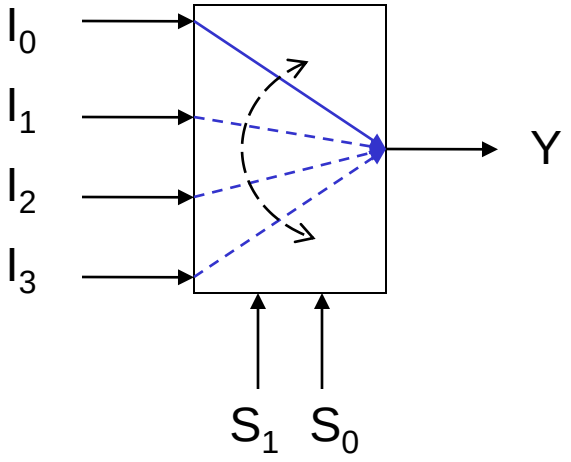


Example: 4-to-1 Multiplexer - 1

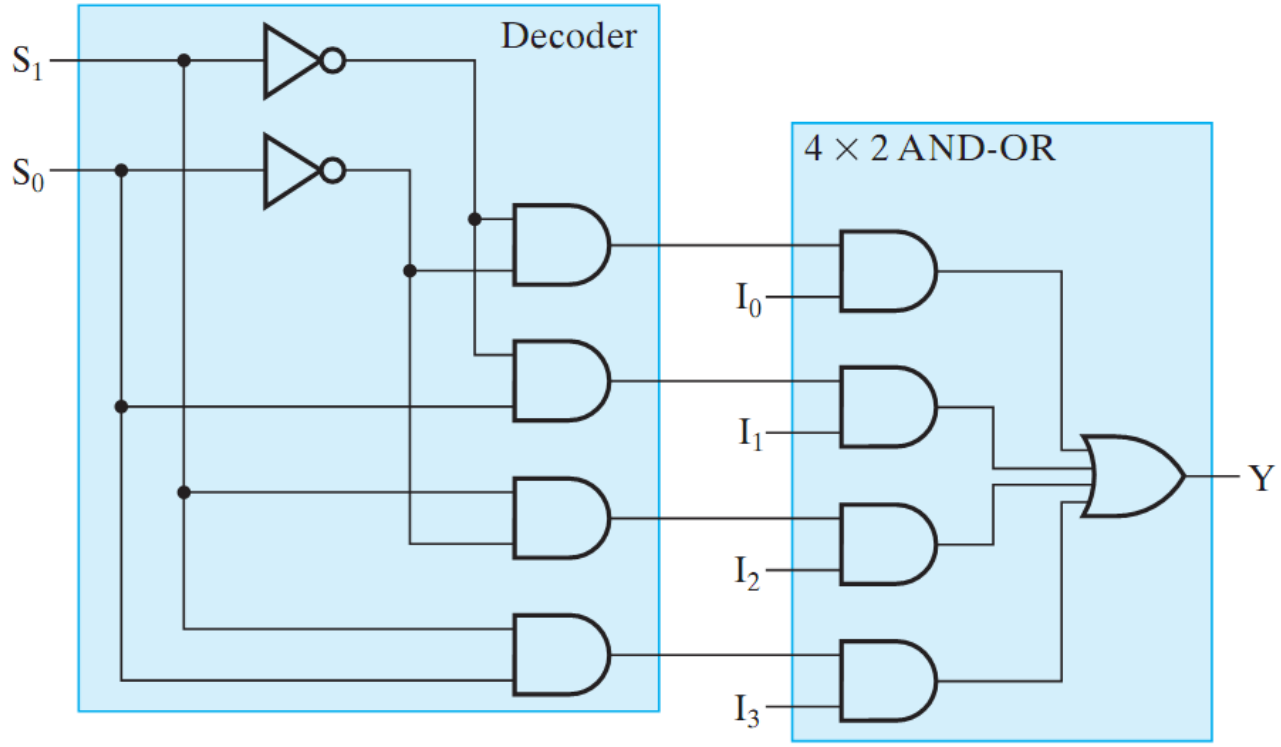
- 4-to-1-line multiplexer
  - When s_1s_0=10
    - AND gate associated with I_2 has 2 inputs equal 1;
    - The other 3 AND gates have at least one input 0



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Example: 4-to-1 Multiplexer - 2



Implement CLN using MUXs - 1

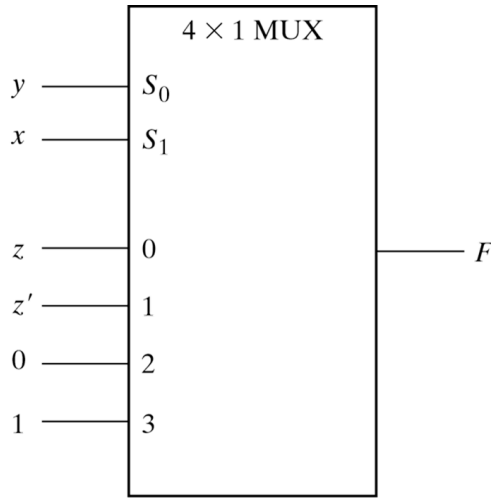
- MUXs consist of decoders and an OR gate, which makes it possible to implement CLN using MUXs without any other gate
- Procedure for implementing function of *n* variables with a 2^{n-1}-to-1 multiplexer
  1. Express function in sum of minterms
  2. Assume ordered sequence of variables is ABCD… where A is the leftmost variable
  3. Choose one variable as input, usually the right most
  4. Connect remaining n-1 variables to selection lines, with the rightmost variable connected to lowest-order selection line (S_0)

Implement CLN using MUXs - 2

5. Construct truth table and divide into sections with identical values for the remaining n-1 variables
6. Associate function output with the chosen input variable
   - This function will have value of either 0, 1 or the literal of the chosen input

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

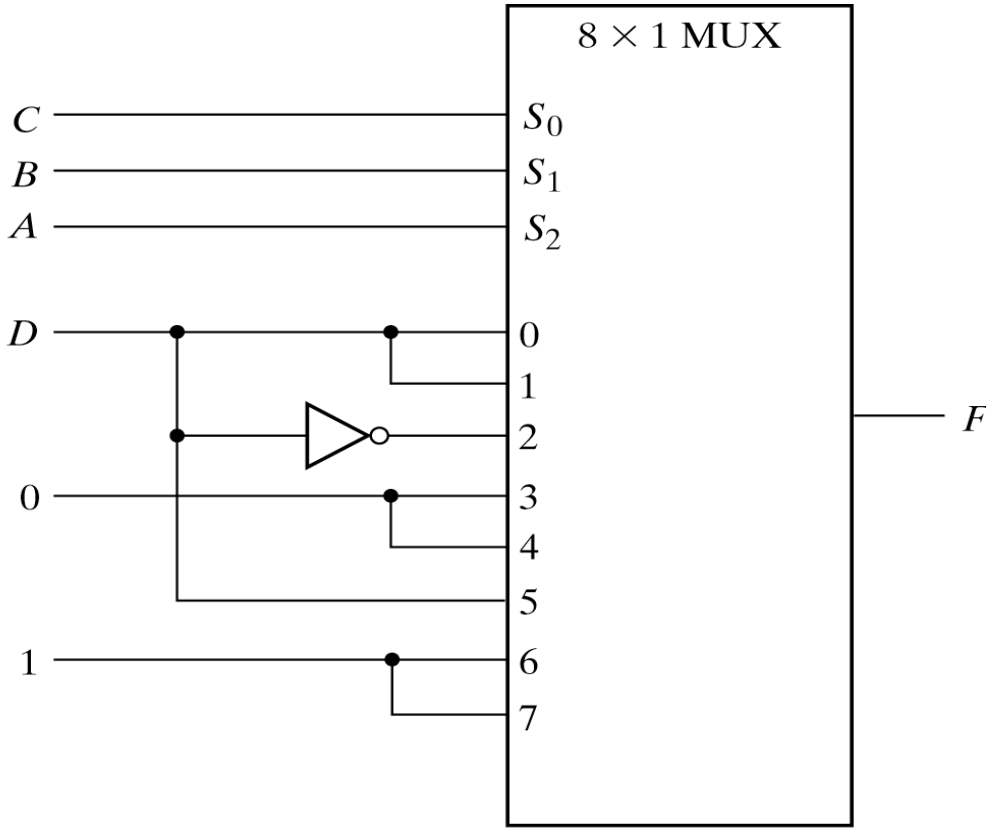
(a) Truth table



(b) Multiplexer implementation

MUX Implementation Example

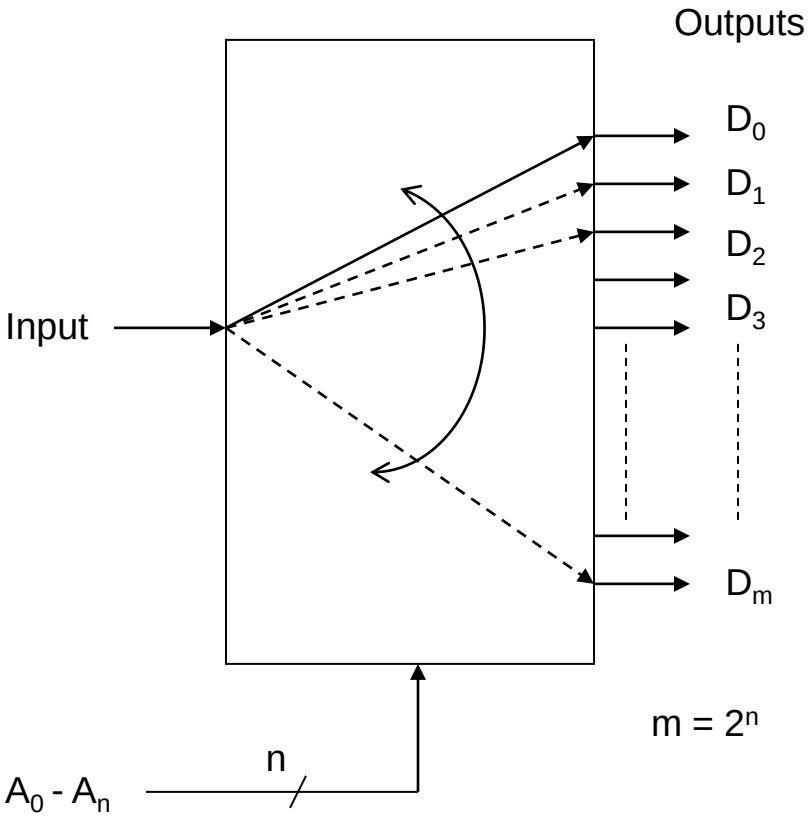
A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

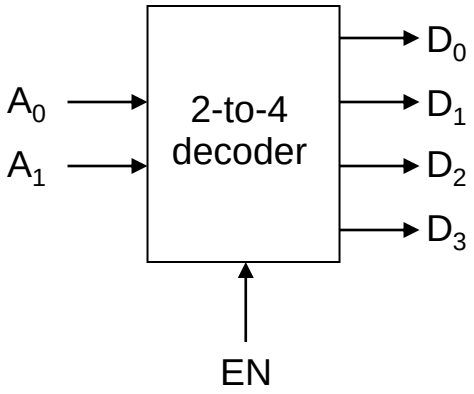


Demultiplexers

- A decoder with an enable input can function as a ***demultiplexer***
  - the *Enable* (*EN*) line is taken as data input line while the *Binary* (A_1A_0) inputs as selection lines
- A *demultiplexer* receives info on a single line and transmits this info to one of 2^n possible output
  - *EN* has a path to all 4 outputs

Demultiplexers





1:4 Demux using 2-4 Decoder

if $A_{0\text{-}n}$ = Y, D_Y = Input, else D_Y = 0

Arithmetic Circuits

Adders

- Digital computers perform a variety of information-processing of information-processing tasks
  - The most basic arithmetic operation is the addition of two binary digits
- Addition consists of 4 possible elementary operations:
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10
- When augend and addend bits are 1, the binary sum consists of 2 digits, *carry* and *sum*
  - carry is the higher significant bit

Adders

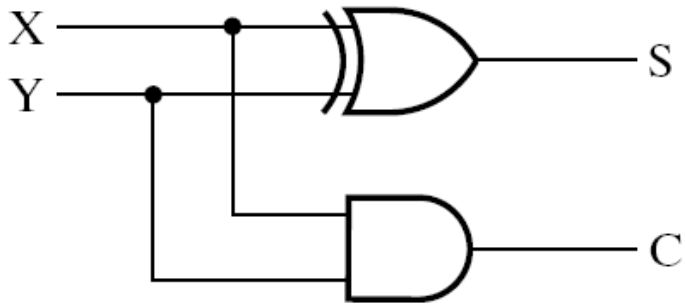
- *Half adder*
  - addition of 2 bits (two 1-bit numbers)
- *Full adder*
  - addition of 3 bits
  - i.e. two 1-bit numbers and 1 carry
- 2 half adders can be employed to implement a full-adder

Half Adder: 1-bit

- Half-adder adds 2 bits (1 bit to 1 bit) and produces a sum and carry output
- S = x'y + xy'
  = x ⊕ y
- C = xy

		X
+		Y
	C	S

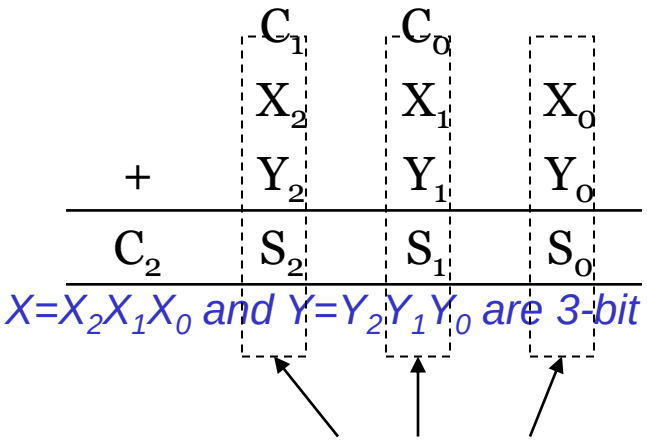
X and Y are 1-bit



Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

More than 1-bit

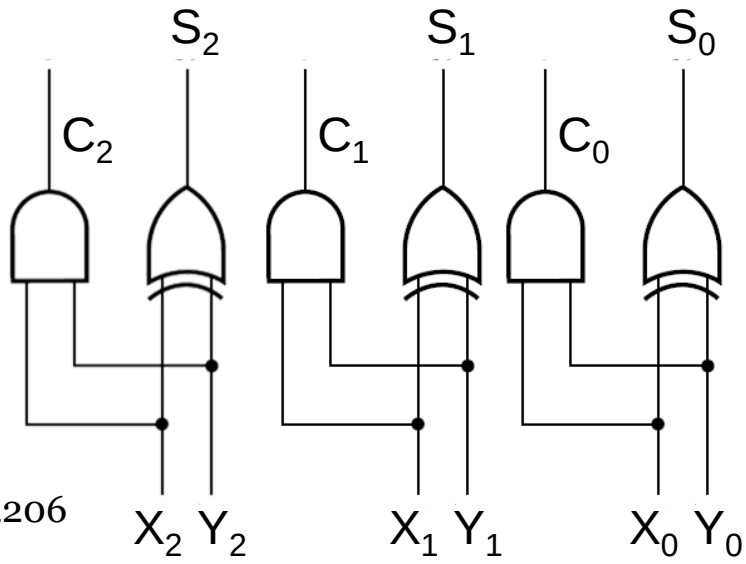
- Each half adder add each bit (position)



	1	1		
	1	0	1	
+	0	1	1	
1	0	0	0	

e.g. X=101 and Y=11

Will the circuit below give correct sum?

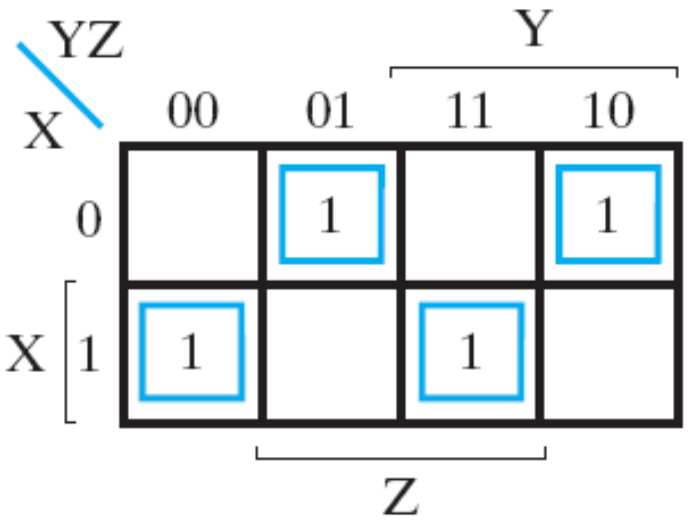


Full Adder

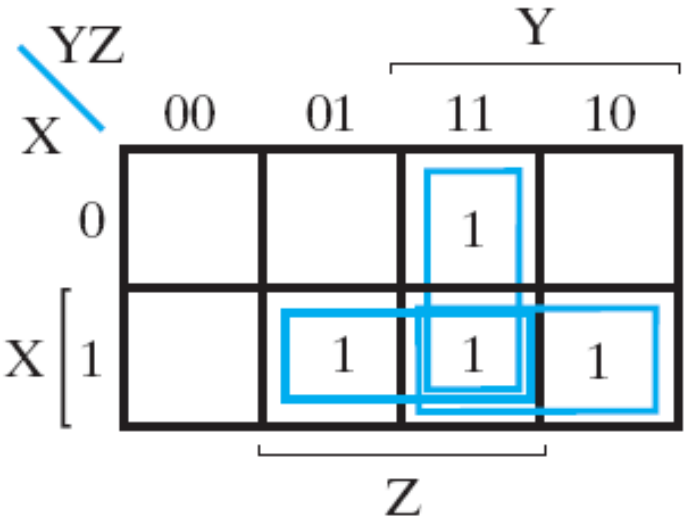
- Problem with half adder is that we cannot use it to build adders that can add more than two 1-bit no.
- A full adder takes 3 inputs and generates 2 outputs
- z represents the carry from the previous lower significant position

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: K-Map



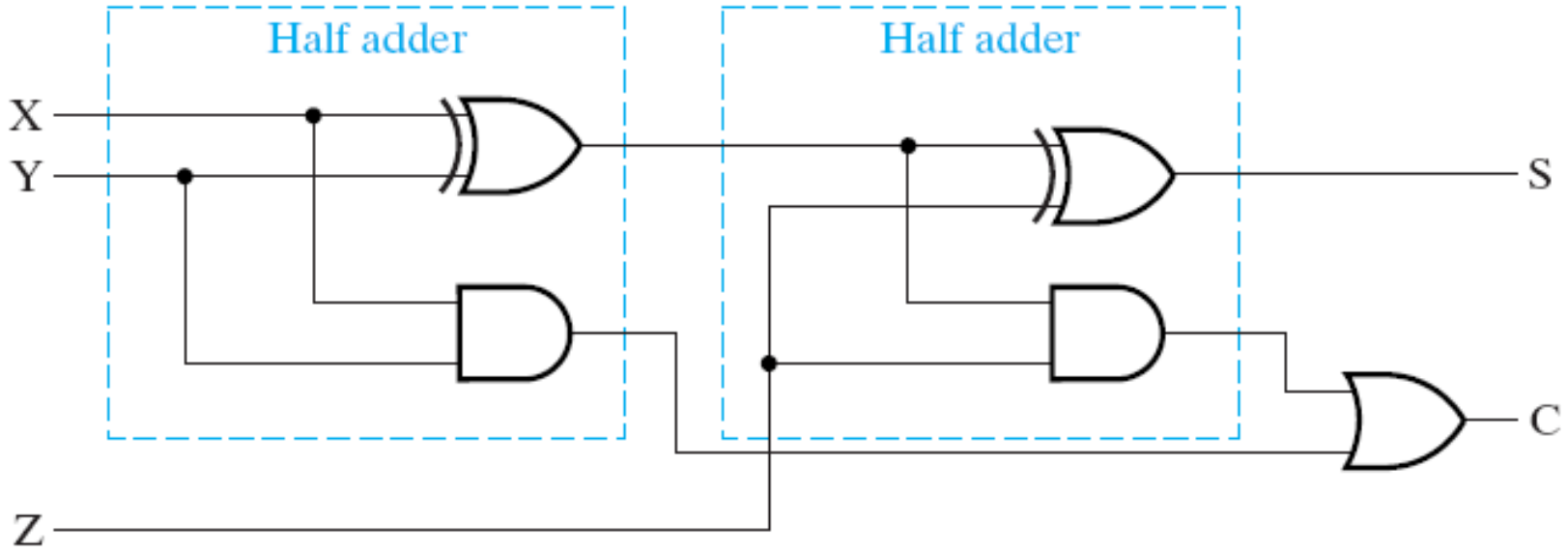
$$S = \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$



$$C = XY + XZ + \overline{YZ}$$
$$= XY + Z(X\overline{Y} + \overline{X}Y)$$
$$= XY + Z(X \oplus Y)$$

Full Adder: Circuit

- Using 2 half adders and an OR gate
  - $S = z \oplus (x \oplus y)$
  - $C = z(x \oplus y) + xy$



Binary Adder - 1

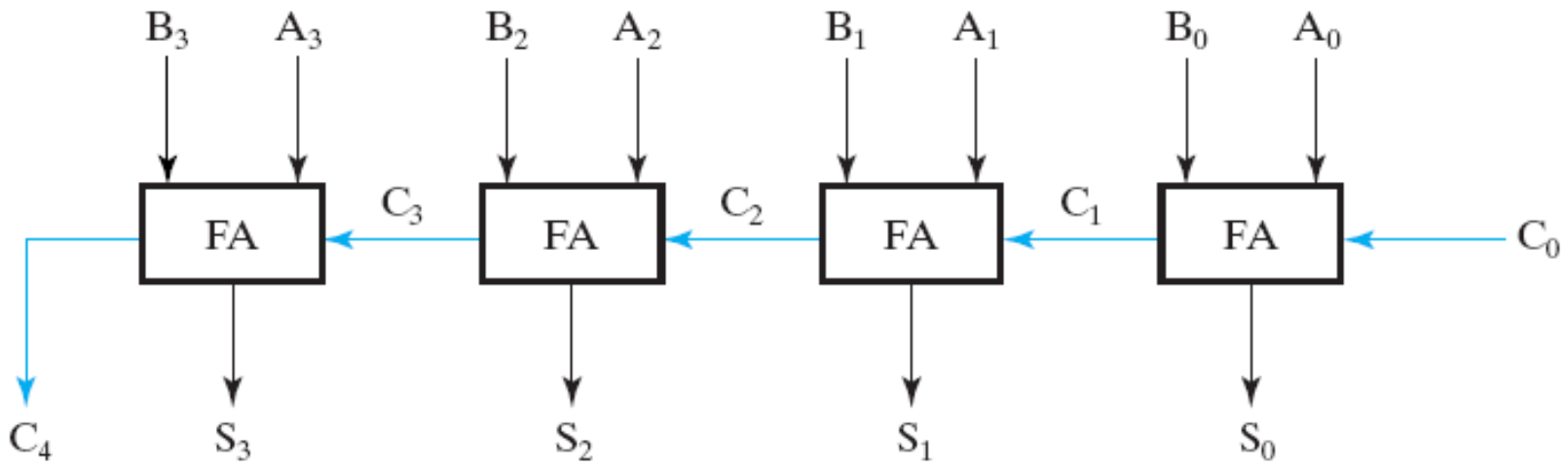
- Adding 2 binary numbers of *n* bits each
- Bits added with full adders, starting from least significant position (i.e. subscript 0)
- The input carry C (Z is the input carry) in the least significant position (i.e. C_0) must be 0
- The value of input carry C_{i+1} is the output carry C_i of the full-adder to the right

Binary Adder - 2

- Sum bits generated as soon as the previous carry bit is generated
- Sum can be generated in serial or parallel fashion
  - Serial method uses only one full adder and a storage device to hold the generated carry
  - Parallel method uses *n* full adder, and all bits of augend and addend are applied simultaneously

Binary Parallel Adder

- Consists of full adders connected in chain, with the output carry from each full-adder connected to the input carry of the next full-adder
- *n*-bit parallel adder requires *n* FA



Binary Parallel Adder: Cons - 1

- Parallel adder has a long delay due to many gates in the carry path from the least significant bit to the most significant bit
  - Each bit of output sum depends on input carry
- Outputs not correct unless signals given enough time to propagate
- *Total propagation time = typical gate delay x number of gate levels*

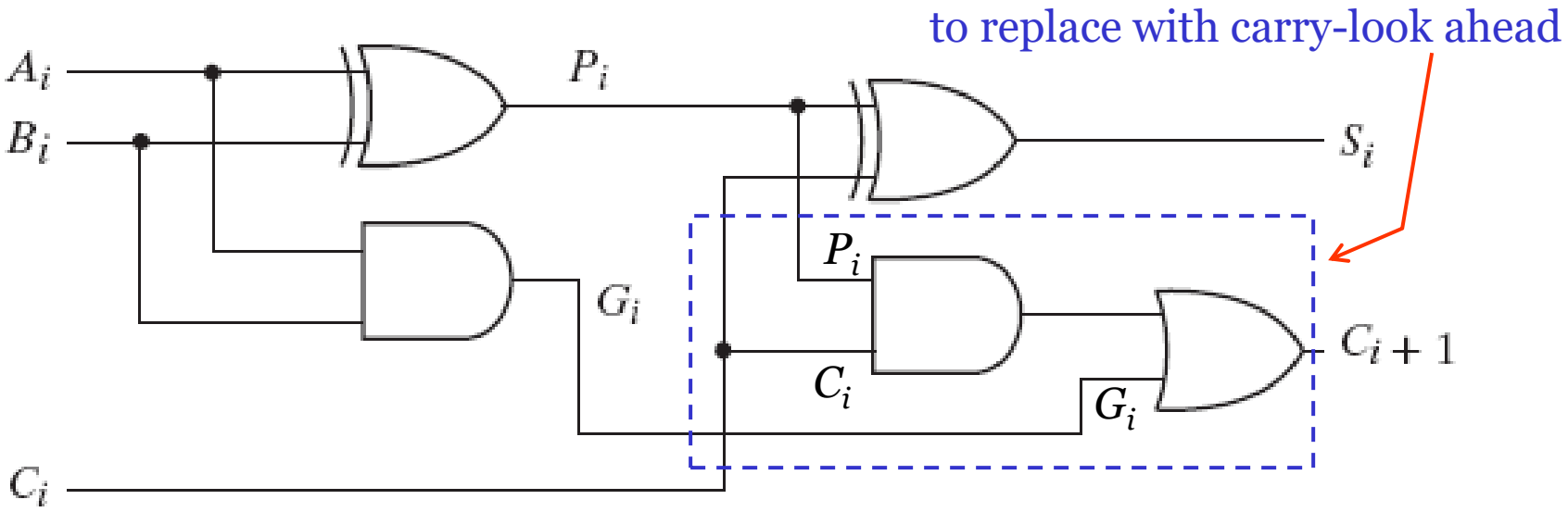
Binary Parallel Adder: Cons - 2

- Signal from input carry C_i to output carry C_{i+1} propagates through an AND and OR gate (i.e. 2 gate-levels)
- *n*-bit parallel adder ≈ 2*n* gate level carry propagation

Carry Look-ahead Adder - 1

- To reduce carry propagation delay in parallel adder
  - Employ faster gates
  - Several reduction techniques, but carry look-ahead most widely used
- Two conditions for a carry:
  - G_i carry generate
    - produces an output carry when A_i and B_i are available, regardless of input carry
  - P_i carry propagate
    - associated with propagation of carry from C_i to C_{i+1}

Carry Look-ahead Adder - 2



- Define $P_i = A_i \oplus B_i$ (equally valid using Pi = A_i + B_i for carry look-ahead)

  $G_i = A_i B_i$
- Hence $S_i = P_i \oplus C_i$

  $C_{i+1} = G_i + P_i C_i$

Carry Look-ahead Generator

For 2-bit adder,

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1$

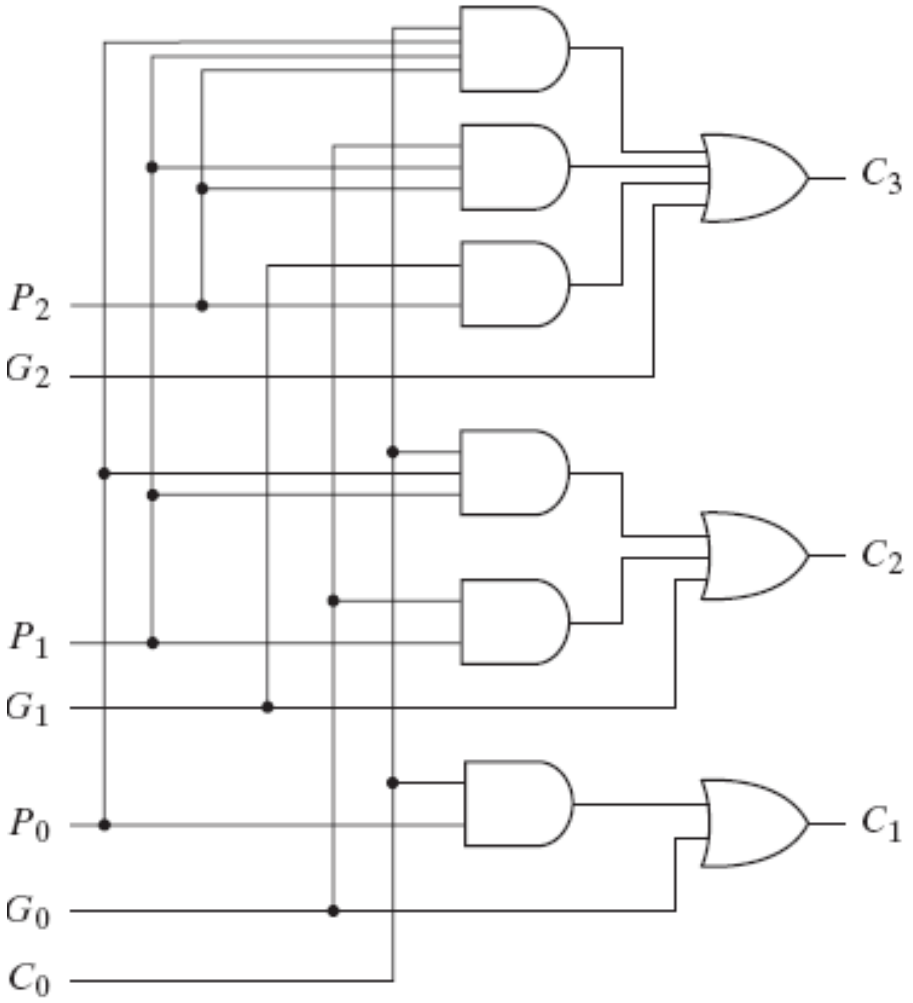
$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2$

$= G_2 + P_2G_1 + P_2P_1G_0 +$

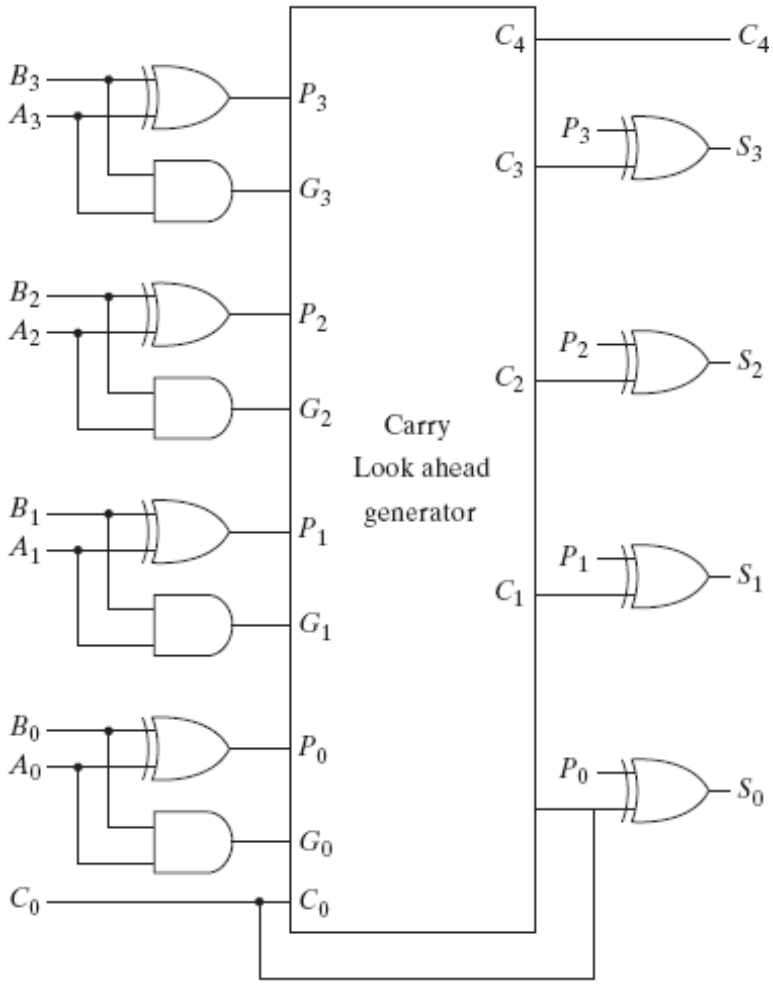
$P_2P_1P_0C_0$

Note that all Carries are generated directly from C_0 and the augend and addend bits, i.e. no carry propagation



Carry Look-ahead Adder - 3

- After P and G signals settle into their steady-state values, all outputs generated after a delay of 2 levels of gates.
- S_1, S_2, and S_3 have equal propagation delay



Signed Binary - 1

- If the binary number is signed, then the leftmost bit represents the sign and the rest represent the number
  - In convention, sign bit 0 for +ve and 1 for –ve
  - E.g. 11001 can be considered as
    - 25 (unsigned)
    - -9 (signed)

Signed Binary - 2

- In a *signed-complement* system, a –ve number is represent by its complement

Decimal	Signed 2's Complement	Signed 1's Complement	Signed Magnitude
+ 7	0111	0111	0111
+ 6	0110	0110	0110
+ 5	0101	0101	0101
+ 4	0100	0100	0100
+ 3	0011	0011	0011
+ 2	0010	0010	0010
+ 1	0001	0001	0001
+ 0	0000	0000	0000
– 0	—	1111	1000
– 1	1111	1110	1001
– 2	1110	1101	1010
– 3	1101	1100	1011
– 4	1100	1011	1100
– 5	1011	1010	1101
– 6	1010	1001	1110
– 7	1001	1000	1111
– 8	1000	—	—

Signed Binary - 3

- Signed-magnitude system is awkward when employed in computer arithmetic
  - Separate handling of the sign
  - Correction step required for subtraction
- 1's complement imposes difficulty
  - +0 and -0 seldom used for arithmetic operations
- Signed-2's complement most prevalent in modern system

Signed Binary - 4

- 1's complement of a binary number is formed by complementing each of the bits
  - E.g. 1's complement of 0001111 is 1110000
- 2's complement can be formed by
  - Adding 1 to the 1's complement value, or
  - Leaving all least significant 0's and the first 1 unchanged and then complementing all higher significant bits
  - E.g. 2's complement of 1101100 is 0010100
- Note that the complement of the complement restores the number to its original value

Binary Subtraction - 1

- Subtraction can be done by negate then add:
  - A – B can be done by finding the negative of B and then add to A
  - A – B = A + (–B)
- Using either 1's or 2's complement, subtraction can be performed using
  - Complementer and
  - Adder

  (complement then add)

Binary Subtraction - 2

- Signed addition using 2's complement
  - Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form
- Signed subtraction using 2's complement
  - 2's complement the subtrahend and add
- One common hardware can be used to handle both signed and unsigned binary addition and subtraction
  - But the results must be interpreted differently depending on whether the numbers are signed or unsigned
  - The same circuit in next slide can be used with no correction step required for signed-2's complement

Signed Binary Subtraction

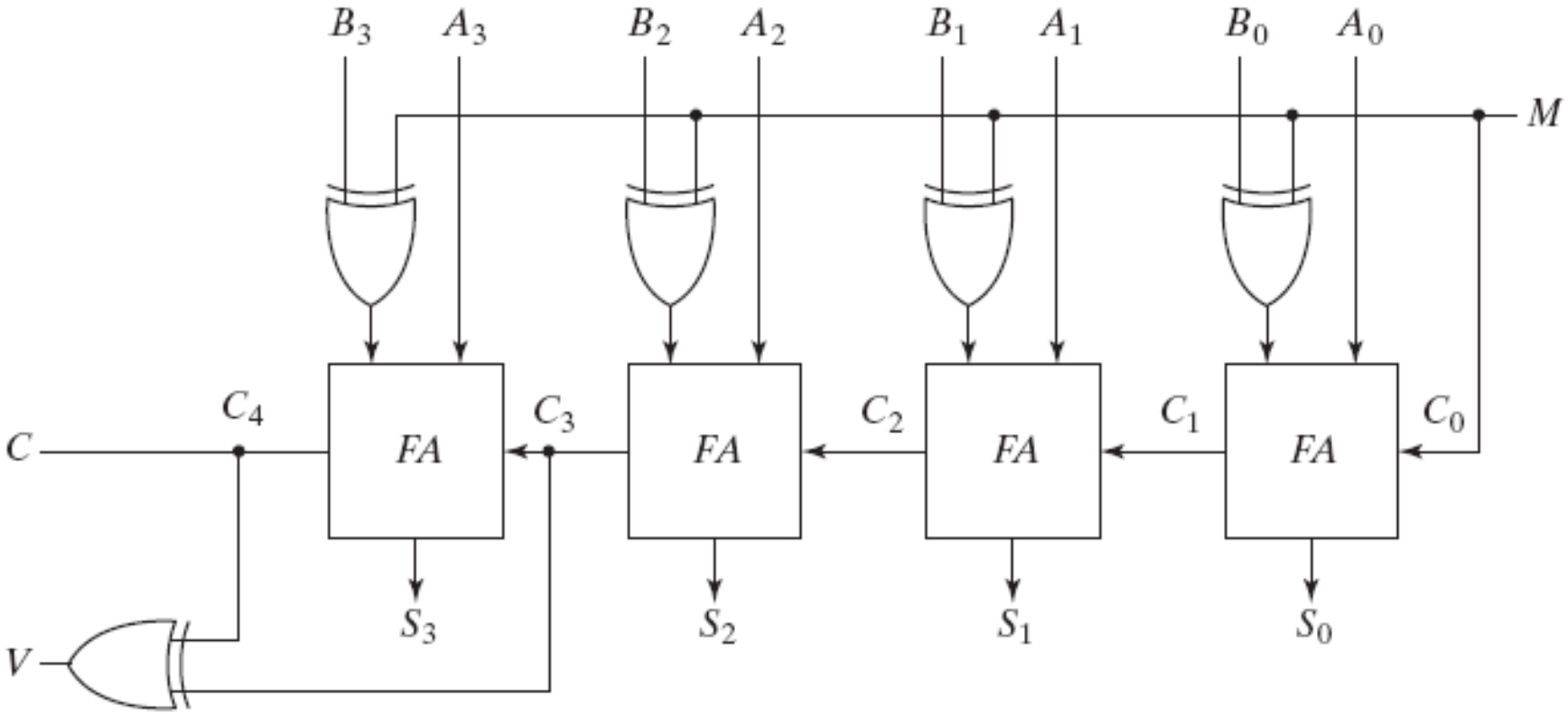
- 2's complement

  x = 0101100 (44), y = 0111101 (61)

  x – y = x + (-y)

  x = 0101100
  -y = 1000011 (2's complement of y)
  sum = 1101111 (-17)
  x - y = 0010001 (-2's complement of sum)

Adder-subtractor - 1



Adder-subtractor - 2

- A – B = A + 2's complement of B
  = A + 1's complement of B + 1
- 1 can be added to sum through input carry
  - therefore C_0 must be equal to 1 when subtract
- When M = 0, adder
  - $B \oplus 0 = B$, and $C_0 = 0$
- When M = 1, subtractor
  - $B \oplus 1 = B'$, and $C_0 = 1$

Overflow

- If we start with two n-bit numbers, but the result occupies n+1 bits , an *overflow* occurs
  - For unsigned numbers, an overflow is detected from the end carry out of the most significant position
  - For signed numbers, if the carry into and carry out of the sign bit position are not equal, an overflow has occurred
    - If V = 0, no overflow
    - If V = 1, overflow

Overflow

Carries:	0 1		1 0
+ 70	01000110	- 70	10111010
+ 80	01010000	- 80	10110000
+ 150	10010110	- 150	01101010

range of 8-bit signed: -128 to +127

BCD Adder - 1

- A decimal adder (4-bit) requires a minimum of 9 inputs and 5 outputs
- Adder produce sum in binary and range from 0 to 19
- Find rule to convert invalid binary sum to correct BCD representation
  - Binary sum ≤ 1001 (9), no conversion needed
  - Binary sum > 1001, invalid BCD ∴ add 0110 (6)

Binary vs BCD

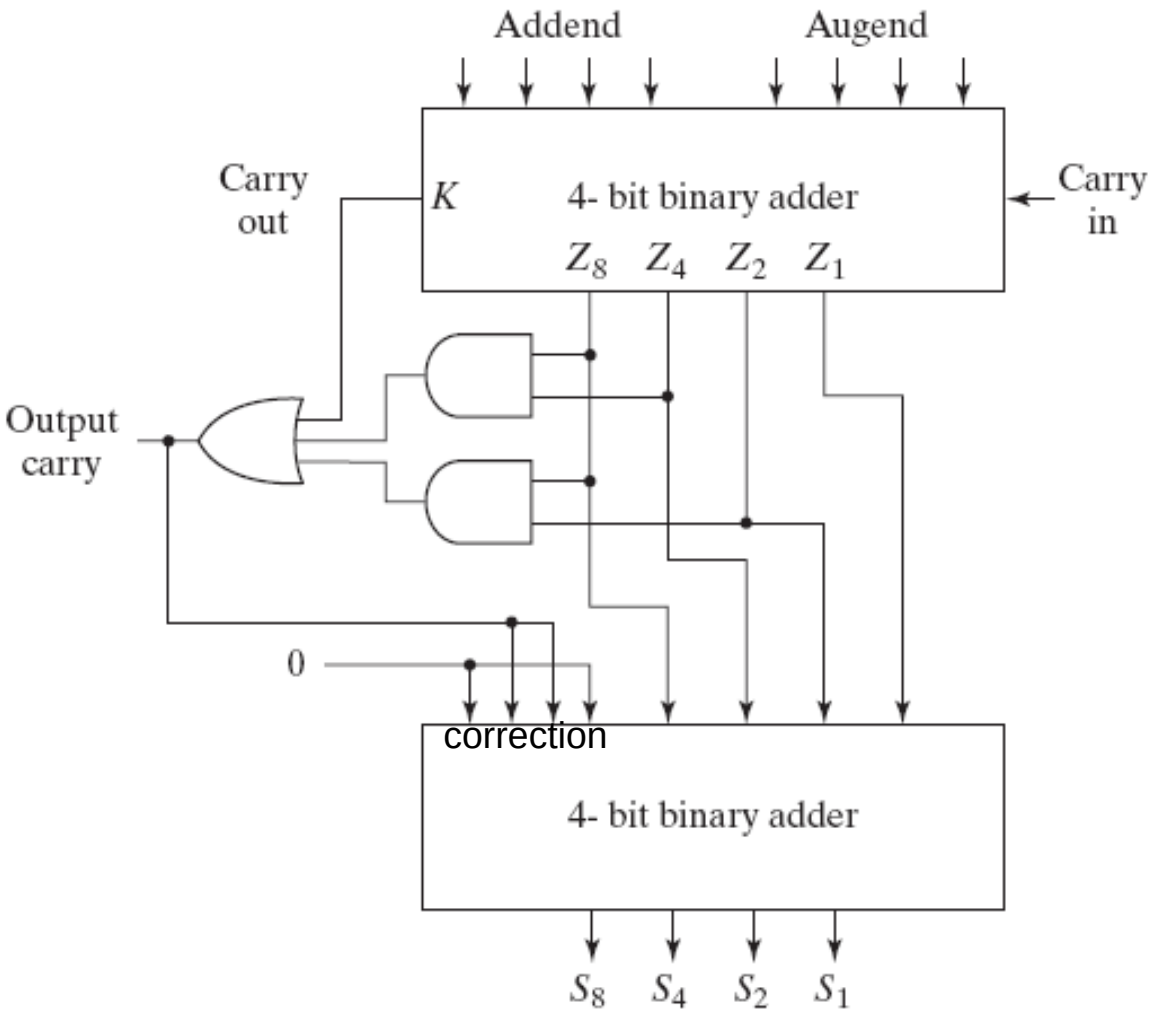
Binary Sum						BCD Sum					
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0		0	0	0	0	0	0
0	0	0	0	1		0	0	0	0	1	1
0	0	0	1	0		0	0	0	1	0	2
0	0	0	1	1		0	0	0	1	1	3
0	0	1	0	0		0	0	1	0	0	4
0	0	1	0	1		0	0	1	0	1	5
0	0	1	1	0		0	0	1	1	0	6
0	0	1	1	1		0	0	1	1	1	7
0	1	0	0	0		0	1	0	0	0	8
0	1	0	0	1		0	1	0	0	1	9
0	1	0	1	0	→ +6	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 - 2

- Correction needed when:
  - Carry K = 1
  - Combinations that have 1 in position Z_8 and 1 either in Z_4 or Z_2 (to distinguish valid 1000 and 1001)
- BCD addition

	0110 (6)		0100	0111 (47)		0101	1001 (59)
	+ 0111 (7)	+	0011	0101 (35)	+	0011	1000 (38)
	1101 invalid		0111	1100 invalid		1000	0001 invalid
	+ 0110 (+6)	+	1	0110 (+6)	+	1	0110 (+6)
0001	0011 (13)		1000	0010 (82)		1001	0111 (97)

BCD Adder - 3


Addend
Augend
Carry out
K
4- bit binary adder
Carry in
Z_8 Z_4 Z_2 Z_1
Output carry
0
correction
4- bit binary adder
S_8 S_4 S_2 S_1

BCD Adder - 3

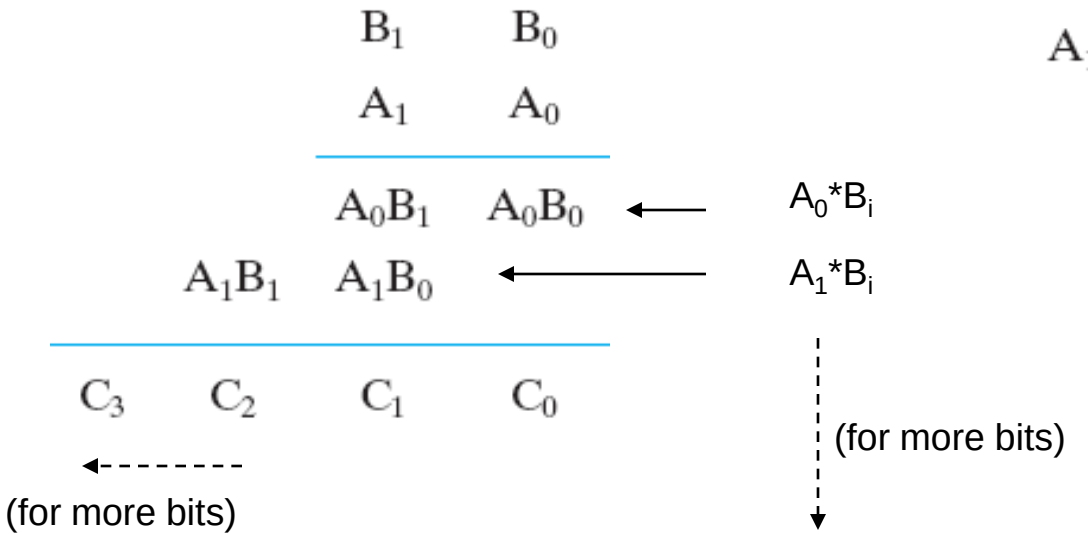
- Condition for correction and an output carry

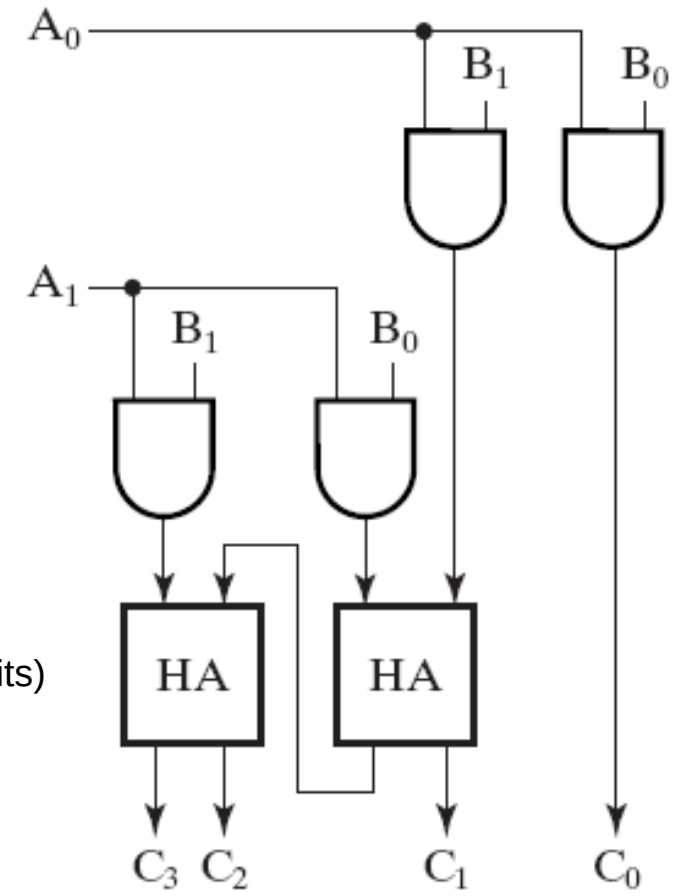
  $C = K + Z_8Z_4 + Z_8Z_2$

- When $C = 1$, add 0110 through bottom 4-bit binary adder
- When $C = 0$, add 0000
- Output carry generated from bottom adder ignored

Binary Multiplier - 1

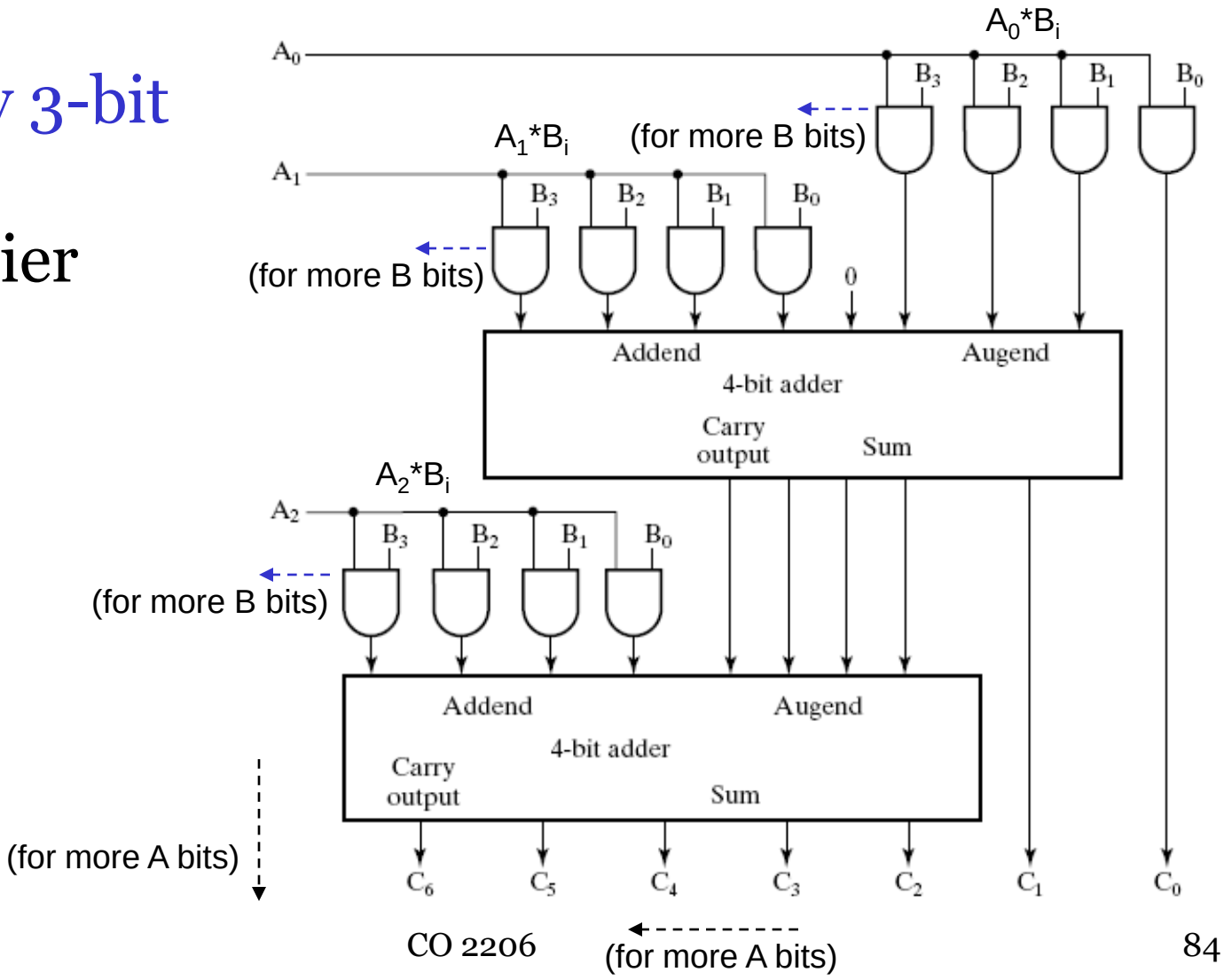
- 2-bit by 2-bit binary multiplier





Binary Multiplier - 2

- 4-bit by 3-bit binary multiplier



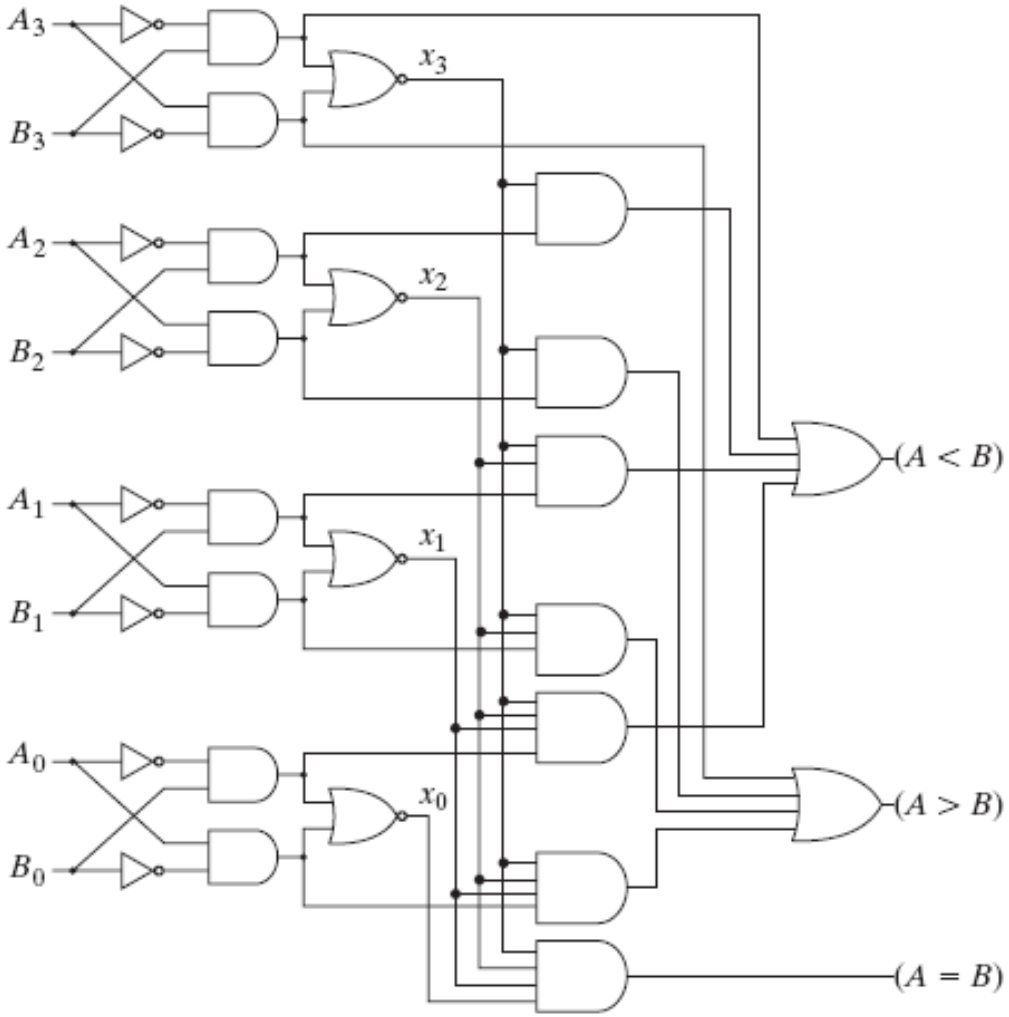
Magnitude Comparator - 1

- A circuit that compares two numbers and determine their relative magnitudes
- Consider A, B with 4 digits each
  - $A = A_3A_2A_1A_0$ $\quad\quad$ $B = B_3B_2B_1B_0$
- Equality relation of each pair of bits can be expressed as $x_i = A_iB_i + A_i'B_i'$
  - $x_i = 1$ only if the pair of bits in position i are equal
  - $(A=B) = x_3x_2x_1x_0$

Magnitude Comparator - 2

- $(A>B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$
  - if $A_3>B_3$ or $A_3=B_3$ but $A_2>B_2$ or $A_3=B_3$, $A_2=B_2$ but $A_1>B_1$ or $A_3=B_3$, $A_2=B_2$, $A_1=B_1$ but $A_0>B_0$
- $(A<B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$
- Comparison starts from MSB until a pair of unequal bits is reached
  - if A_i is 1 and B_i is 0, $A_i>B_i$
  - If A_i is 0 and B_i is 1, $A_i<B_i$

Magnitude Comparator



Summary

- Important combinational functional blocks were introduced
- Functional blocks are build from logic gates or smaller functional blocks
- Design of functional blocks based on truth-table, i.e. knowing its function
- Some designs simplified based on confined definition of the function, i.e. not all input combinations need to be considered
- These functional blocks will be used to build up larger system, eventually the computer