II.4. Minimization of Boole Functions Using Karnaugh Diagrams

The Veitch-Karnaugh Method

- provides a visual way of putting together the conjunction terms in DNF for which unification can be applied
- unification is possible if two terms differ on one variable only
  - which is negated for one term and not negated for the other one
- such terms become neighbors in a Karnaugh diagram

Structure of a Karnaugh Diagram

2-dimensional table

- variable names
  - for rows and columns, respectively
- label area
  - label - bit string of length n
  - each bit corresponds to a variable (input)
  - all possible input combinations are present
- function value (output) area

Examples of Diagrams

2 variables

A \ B	0	1
0	1	1
1		

3 variables

A \ BC	00	01	11	10
0		1		1
1		1	1	

4 variables

AB \ CD	00	01	11	10
00	1	1		
01				
11		1		1
10	1			1

Grey Code

- labels are not written in increasing order, but in Grey order
- any two consecutive labels, including the first and the last, differ by one bit
  - 2 bits: 00, 01, 11, 10
  - 3 bits: 000, 001, 011, 010, 110, 111, 101, 100
  - 4 bits: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Diagram Adjacencies (1)

- two positions are adjacent if their corresponding labels differ by a single bit
  - Grey code: adjacency $\rightarrow$ neighborhood
- for an n-variable function, a location has n adjacent locations
  - $n < 5$: adjacent locations are found visually (up, down, left, right)
  - $n \geq 5$: there are also other adjacencies, not directly visible

Diagram Adjacencies (2)

- there may be more than 2 adjacent locations
  - extend the unification to more than 2 variables
- in Karnaugh diagrams, they correspond to blocks with 2^k locations
  - power of 2 both for rows and for columns
    - including power 0
    - rectangle-shaped
  - for each location, the block must contain precisely *k* adjacent locations

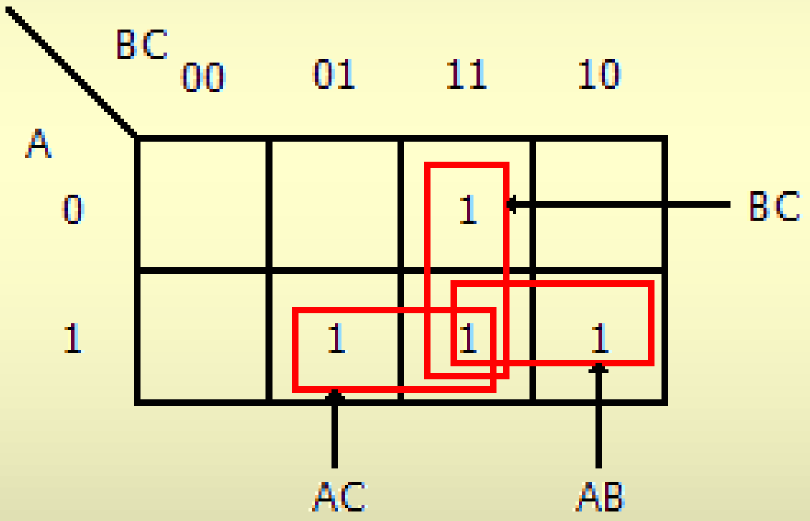
Karnaugh Minimization

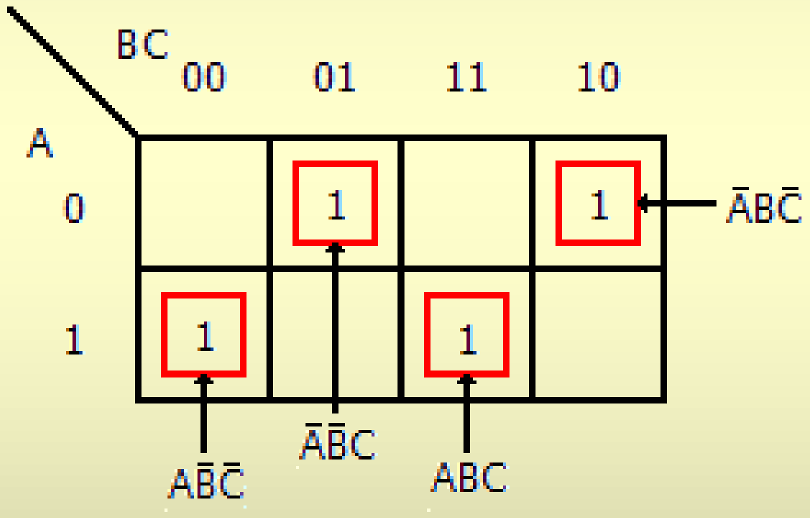
- look for blocks containing only 1s
  - corresponding to an adjacency (see before)
  - the blocks - as large and as few as possible
- for each block with 2^k locations (all 1s)
  - we have a conjunction term of n-k variables
  - contains the variables whose values are constant for all locations in the block
    - 0: variable is negated; 1: variable is not negated
  - all these terms are connected by disjunction

Examples

majority of 3

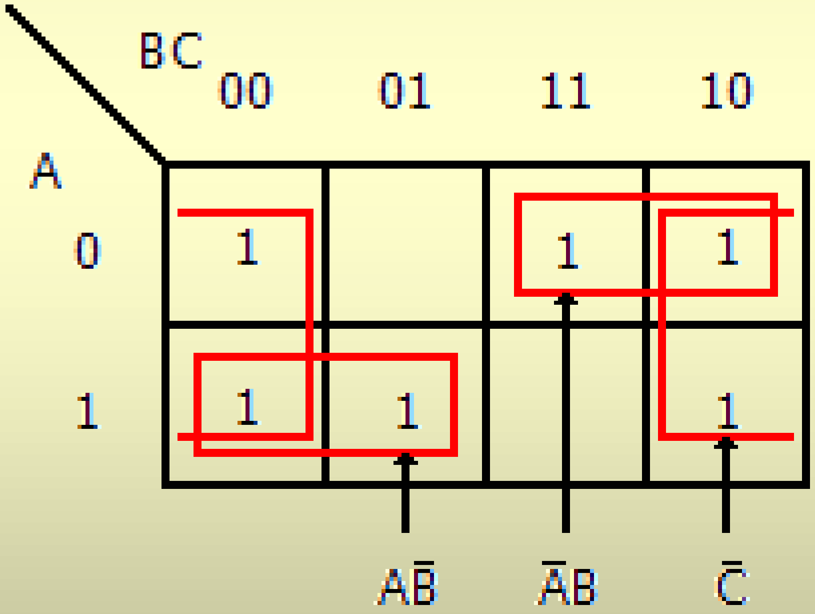
odd



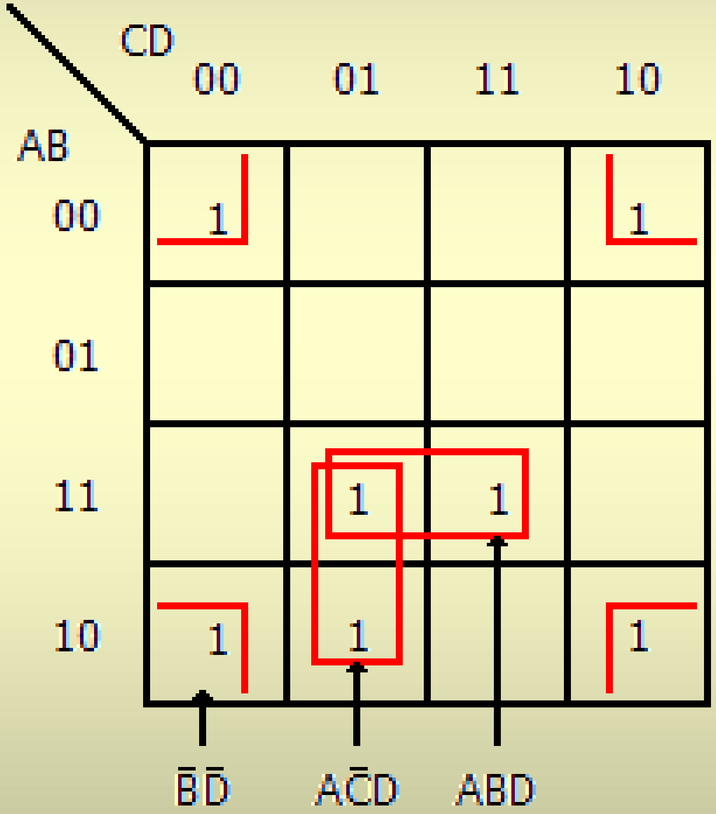


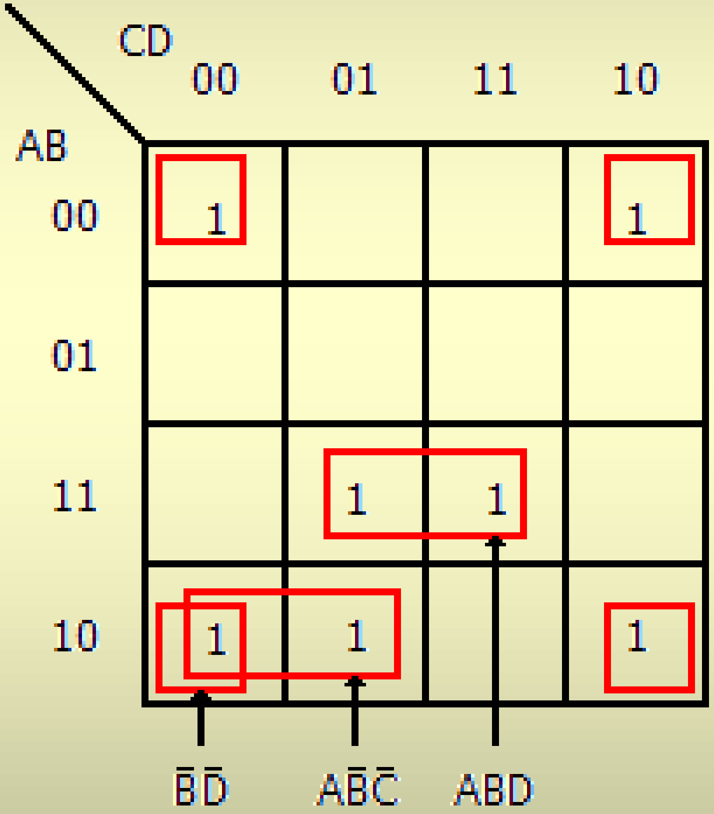
Adjacency of Extreme Rows/Columns

$$f = \Sigma(0,2,3,4,5,6)$$



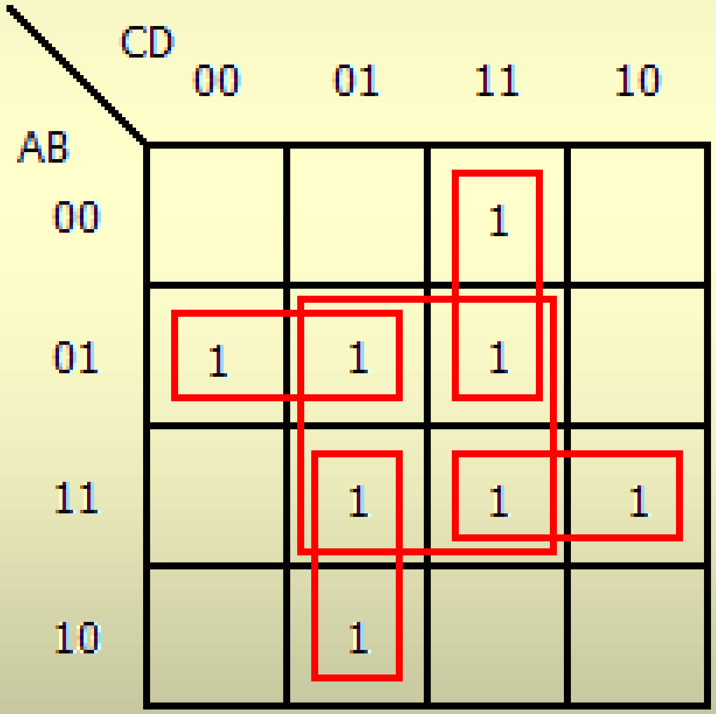
Expression Depends on Groups



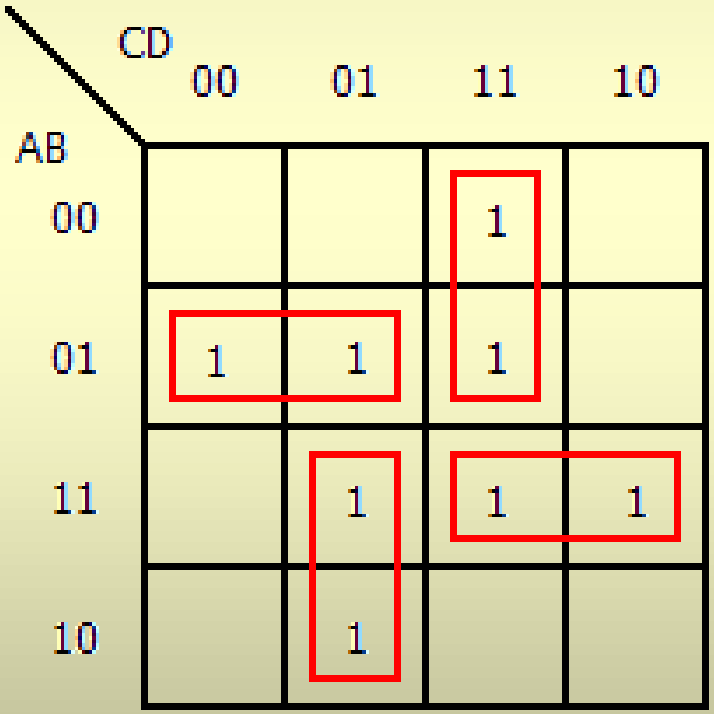


Avoiding Redundancies

non-minimal simplification



minimal simplification

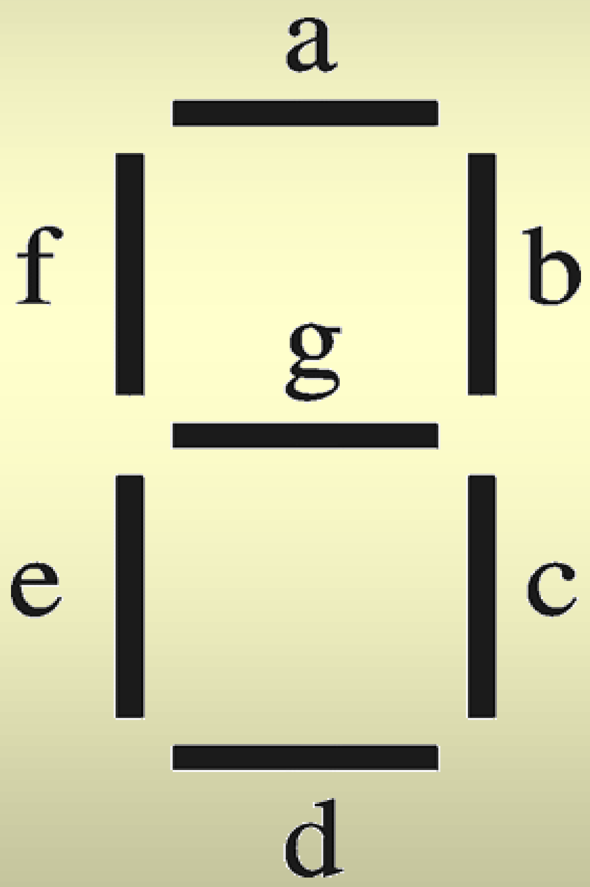


Impossible Value Combinations

- certain value combinations will never show up on input
  - according to the behavior we seek
  - but the diagram must be filled for all value combinations of the variables
- in the locations corresponding to these combinations we can write either 0 or 1
  - in order to get a simpler expression

Example - Displaying Decimal Digits

- 7 segment display
- selecting the segments for each digit
  - 0 - switched off
  - 1 - switched on
- input (command) - 4 variables
  - a decimal digit can be written on 4 bits

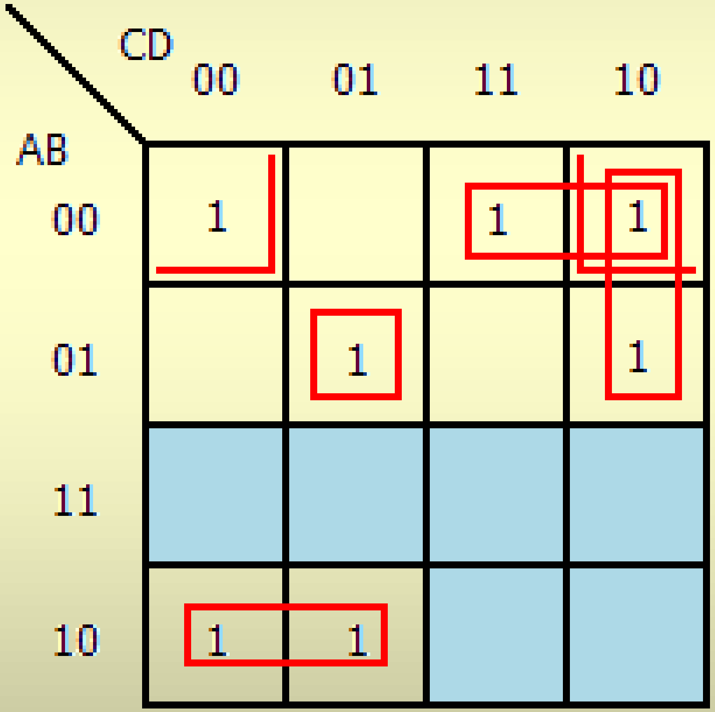


Segment *d* - Truth Table

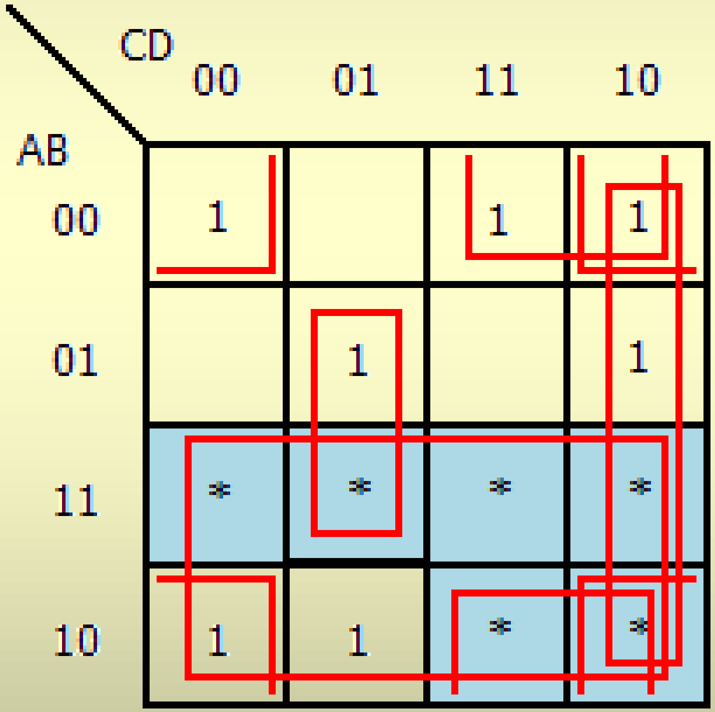
No	A	B	C	D	*d*
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	*
11	1	0	1	1	*
12	1	1	0	0	*
13	1	1	0	1	*
14	1	1	1	0	*
15	1	1	1	1	*

Simpler Expressions

"safe" simplification



exploiting impossible combinations



Homework: 2-bit Comparison

- 4 variables: A, B, C, D
- make 2 numbers
  - $N_1 = AB$
  - $N_2 = CD$
- 3 outputs - correspond to the truth values
  - $LT = (N_1 < N_2)$
  - $EQ = (N_1 = N_2)$
  - $GT = (N_1 > N_2)$

A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

Homework: 2-bit Multiplication

- 4 variables: A, B, C, D
- make 2 numbers
  - $N_1 = AB$
  - $N_2 = CD$
- 4 outputs - the product $N_1 \cdot N_2$

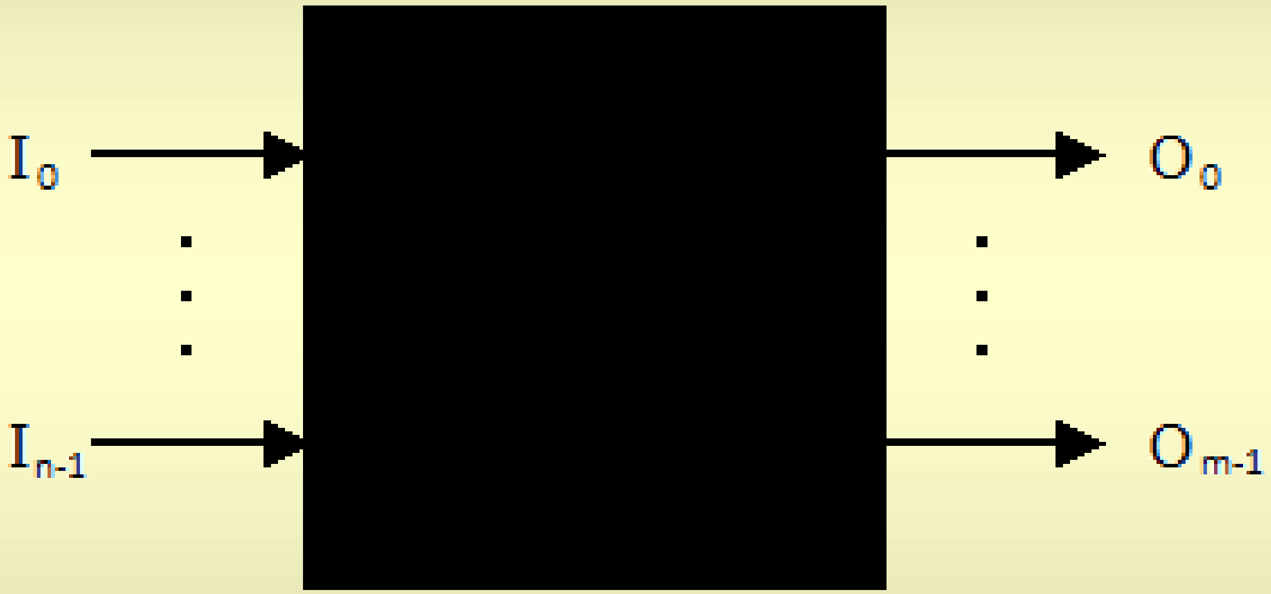
A	B	C	D	P8	P4	P2	P1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Homework: BCD Increment by 1

- 4 variables
  - make a BCD numbers
  - between 0 and 9
- 4 outputs - the input number, incremented
  - the result is also a BCD number

I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	*	*	*	*
1	0	1	1	*	*	*	*
1	1	0	0	*	*	*	*
1	1	0	1	*	*	*	*
1	1	1	0	*	*	*	*
1	1	1	1	*	*	*	*

II.5. Combinational Circuits

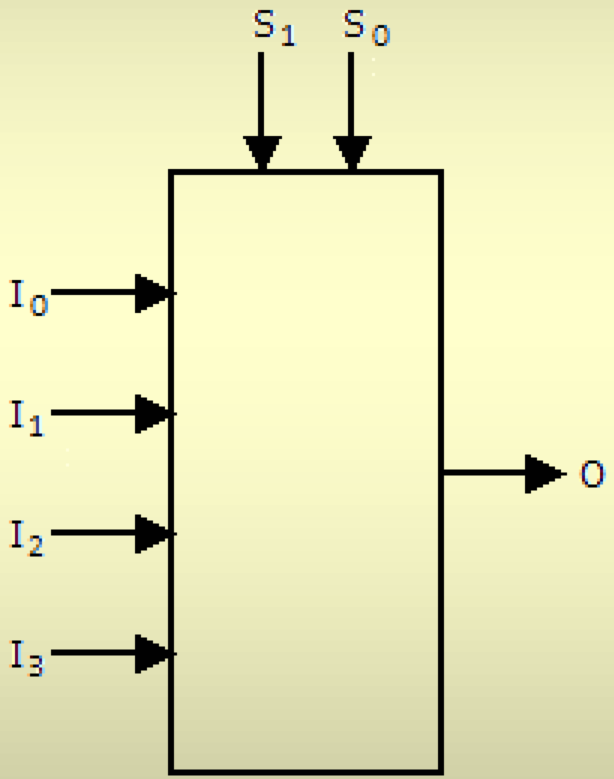


- output values depend only on the input values at the current moment

The Multiplexer

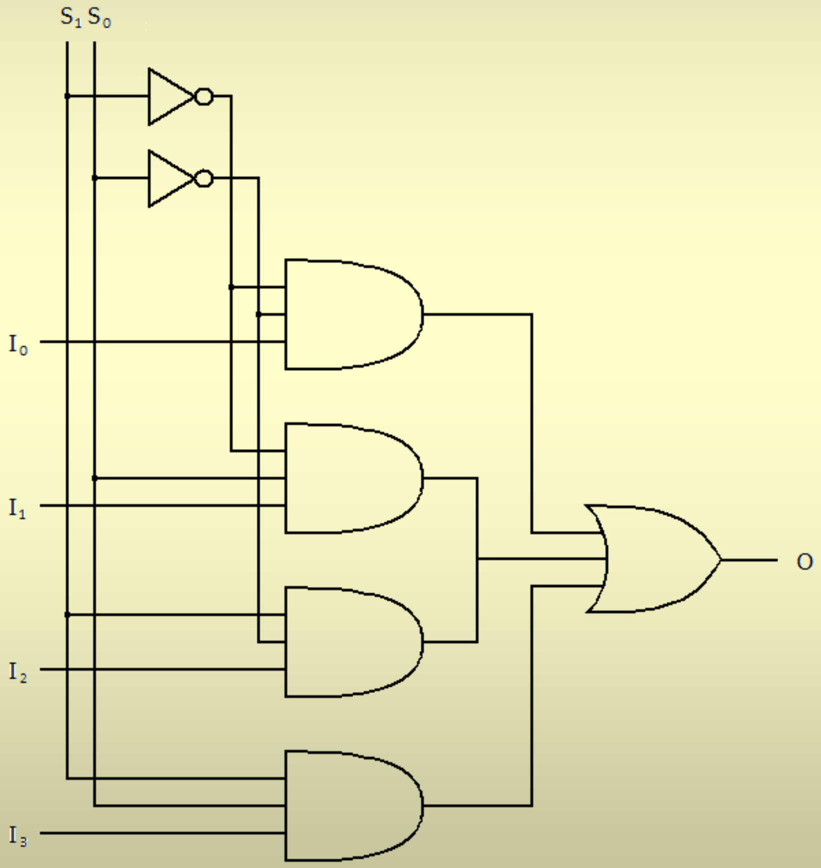
- 2^n inputs (data)
- *n* selection inputs (control variables)
  - control (address) bits
- a single output
- each data input corresponds to a DNF term
- one of the inputs (data bit) is selected - copied to the output

Multiplexer 4→1 (*n*=2)



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

Logic Diagram (4→1)

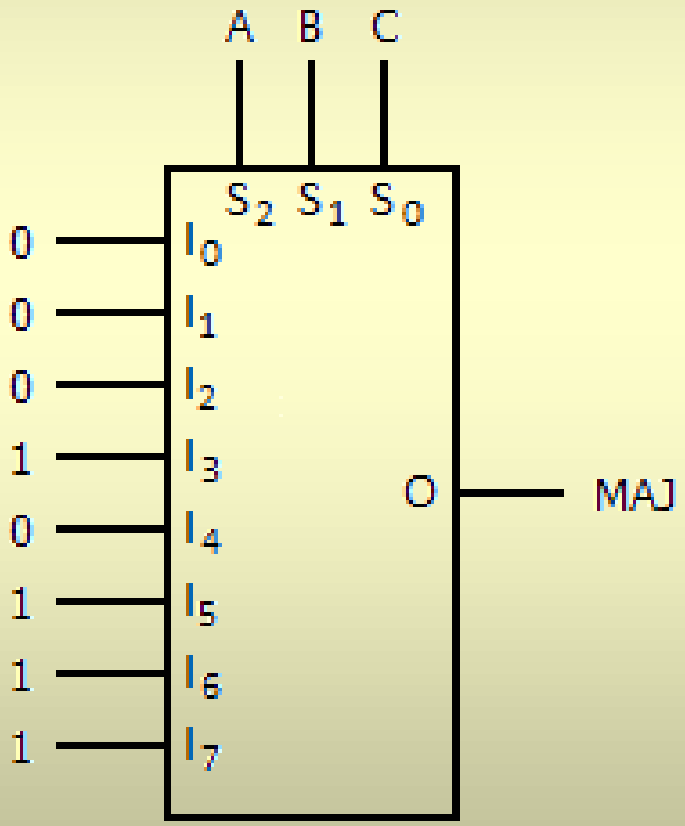


Implementation of Boole functions

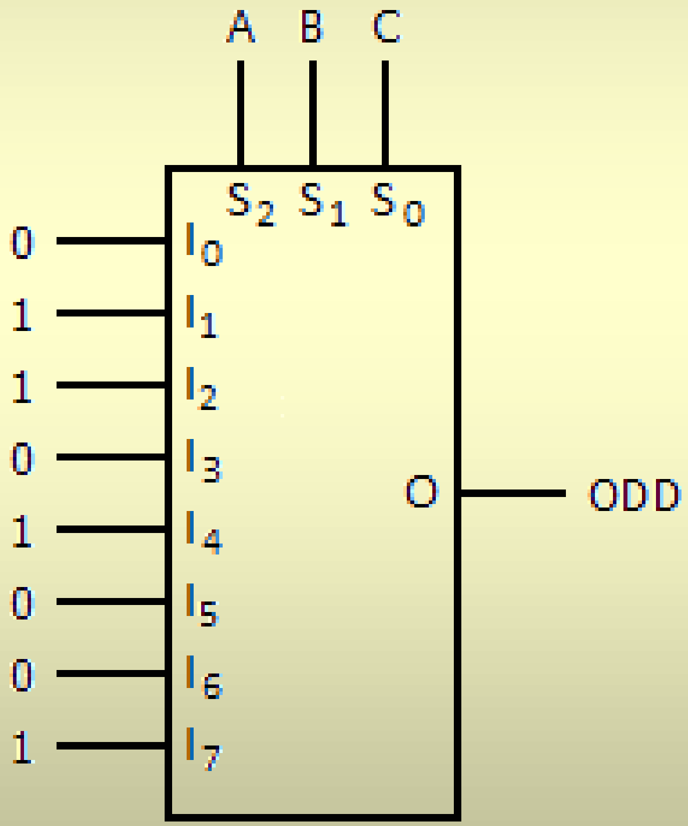
- selection inputs make a number
- which is the index of the data input to be selected for the output
- we can thus implement Boole functions through the multiplexer
  - data inputs - the outputs corresponding to the rows in the truth table
  - selection inputs - inputs of the Boole function

Examples

majority of 3



odd

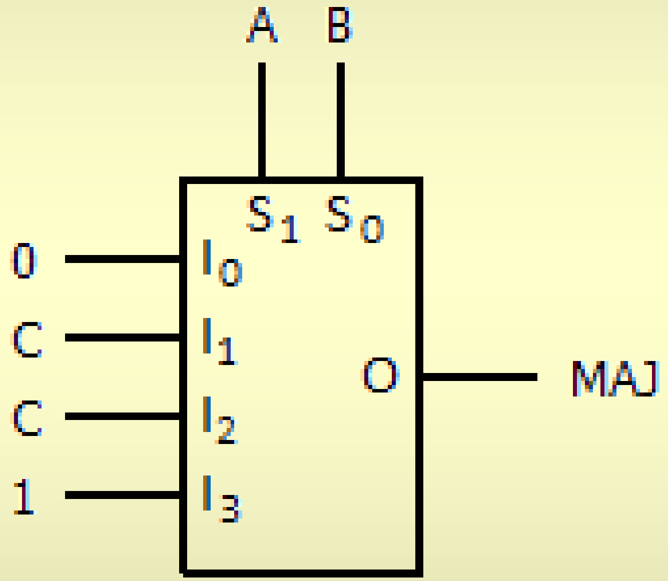


Efficient Implementation - folding

majority of 3

A	B	C	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A	B	
0	0	0
0	1	C
1	0	C
1	1	1

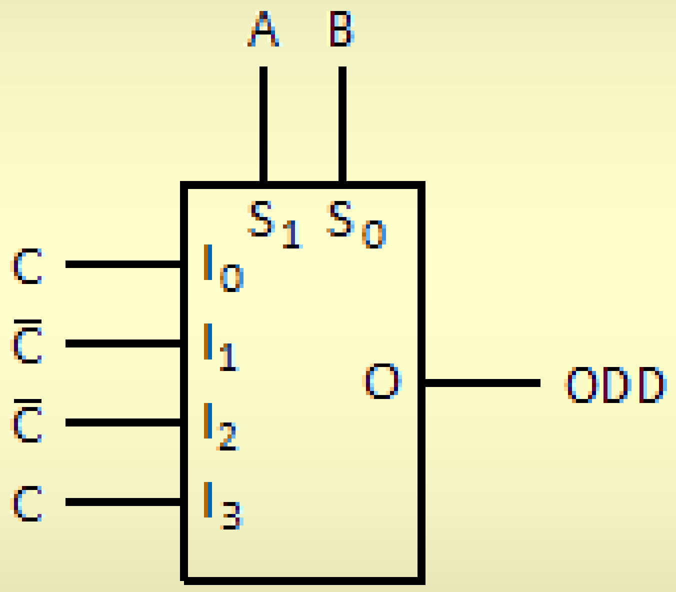


Efficient Implementation - folding

odd

A	B	C	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

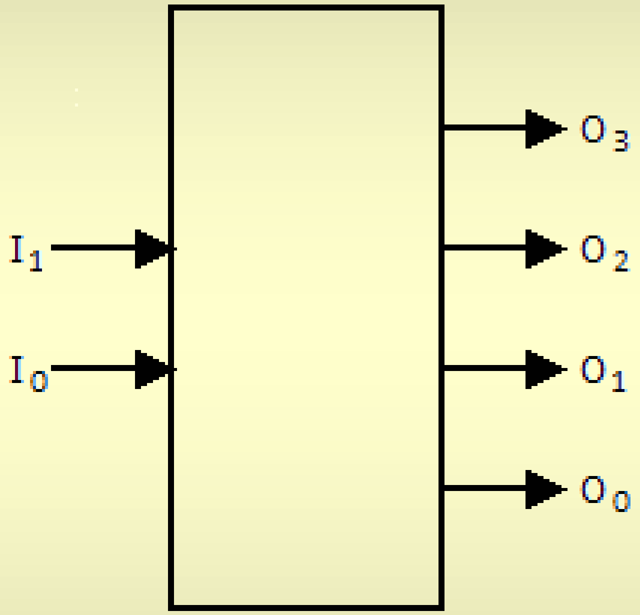
A	B	
0	0	C
0	1	$\overline{C}$
1	0	$\overline{C}$
1	1	C



The Decoder

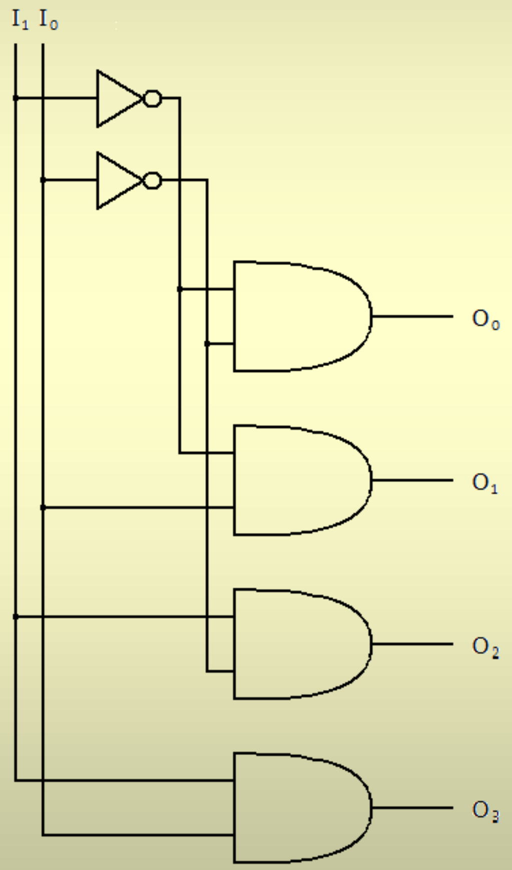
- *n* inputs
- 2^n outputs
- at each moment, exactly one of the outputs is activated
  - the one whose index is the number made by the inputs
  - each output corresponds to a DNF term written with the input variables

Decoder *n*=2



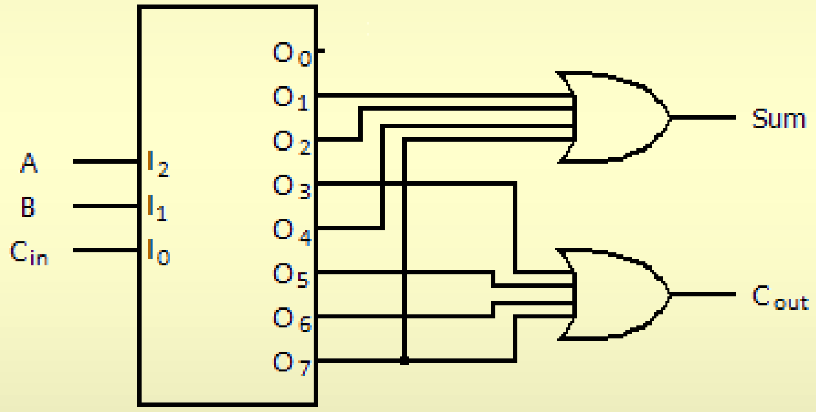
I_1	I_0	O_3	O_2	O_1	O_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Logic Diagram (*n*=2)



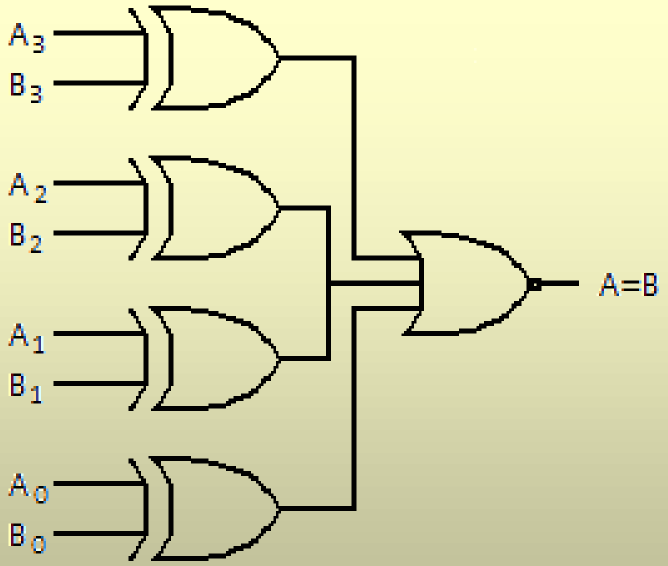
Addition - Decoder Implementation

A	B	C_{in}	Sum	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



The Comparison Circuit

- comparison operators: = , > , < , ≥, ≤
  - example of implementation: 4-bit equality
  - homework: complete comparison (<, =, >)

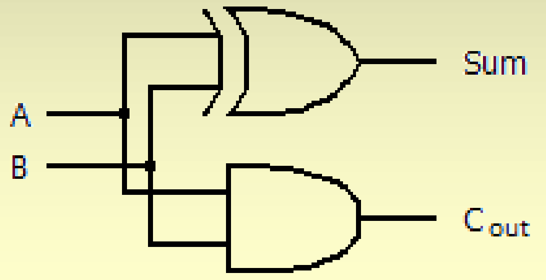


Adders

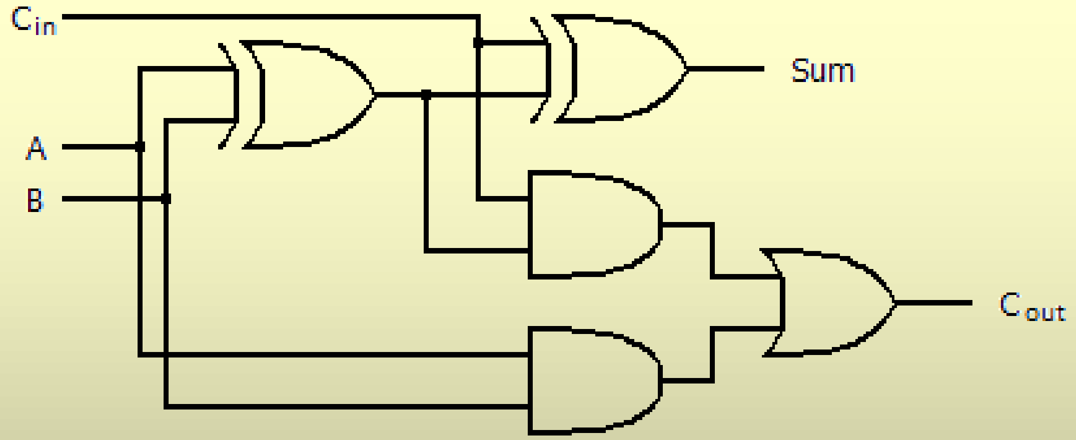
- half-adder
  - Adds the two input bits
  - outputs: a sum bit and a carry bit
  - cannot be extended for adding longer numbers
- full adder
  - Adds the three input bits (including carry in)
  - same outputs: a sum bit and a carry (out) bit
  - can be extended for multiple digits (bits)

Logic Diagrams

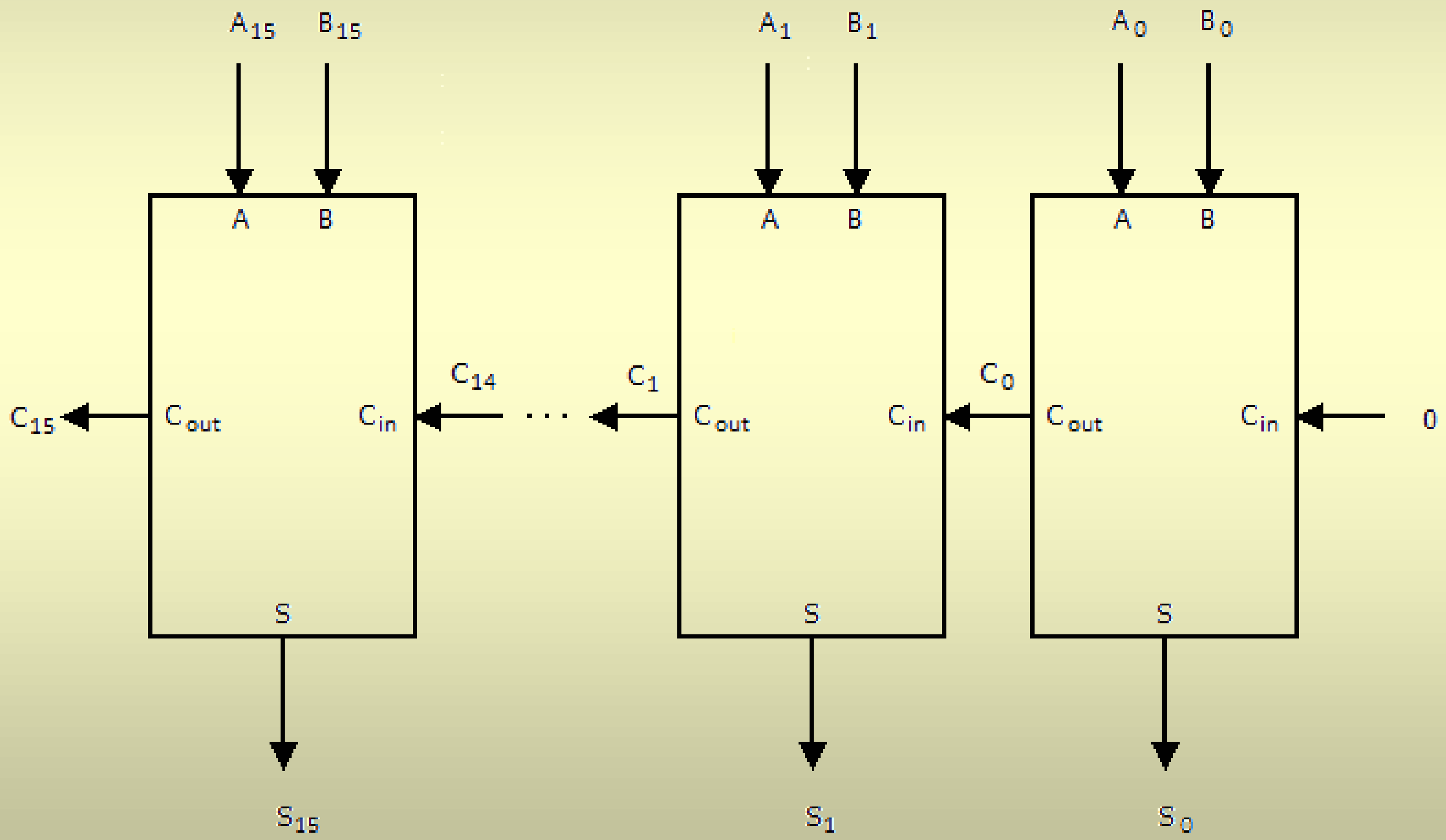
A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



A	B	C_{in}	Sum	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



Ripple-carry Adder (16 bits)



Ripple-carry Adders

- this kind of addres uses carry propagation
- advantage: the same (simple) circuit, repeated
- drawback: low speed
  - for each rank, one must wait for the result from the previous rank
  - so the delay is proportional to the number of bits

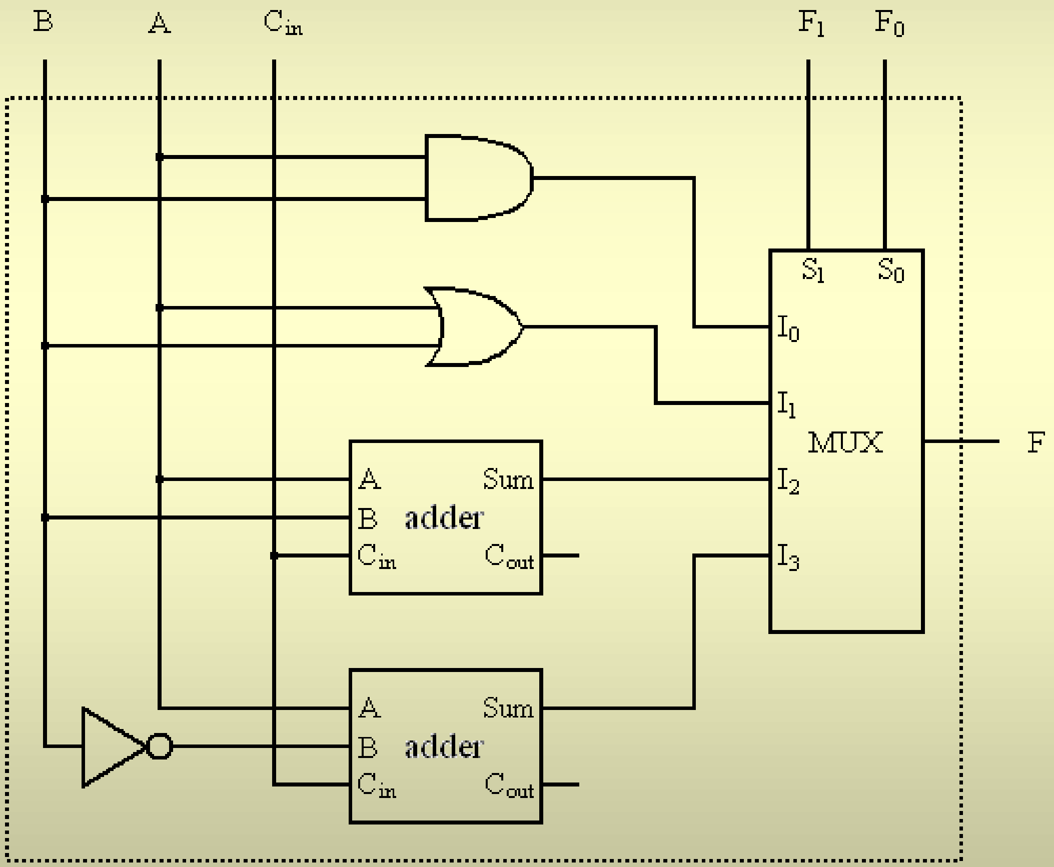
Speed-up (1)

- carry lookahead adder
  - carry in - generated independently for each rank
    - $C_0 = A_0 B_0$
    - $C_1 = A_0 B_0 A_1 + A_0 B_0 B_1 + A_1 B_1$
    - …
    - $C_i = G_i + P_i C_{i-1} = A_i B_i + (A_i + B_i) \cdot C_{i-1}$
    - …

Speed-up (2)

- carry lookahead adder (continued)
  - advantage - hish speed
    - eliminates the delay caused by carry propagation
  - drawback - requires complex additional circuits
  - usually - combination of the two methods
- carry selection adder
  - for each rank, compute the sum for both C_{in}=0 and C_{in}=1, the select the correct result

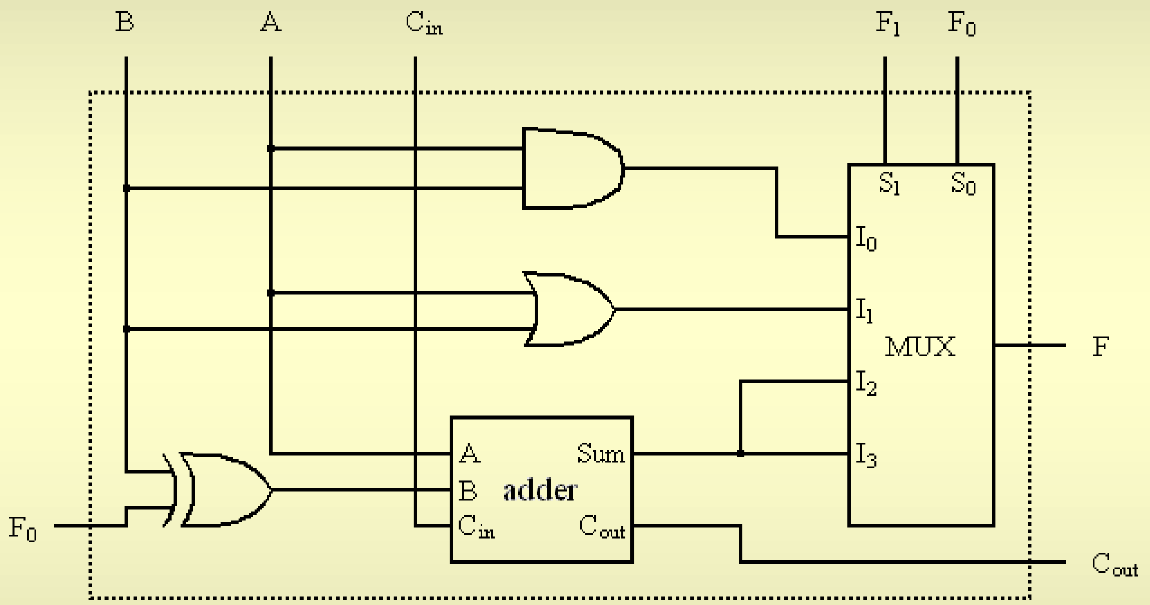
Elementary Arithmetic Logic Unit (1 Bit)



F_1	F_0	F
0	0	A and B
0	1	A or B
1	0	A+B
1	1	A-B

F_1,F_0 - control signals

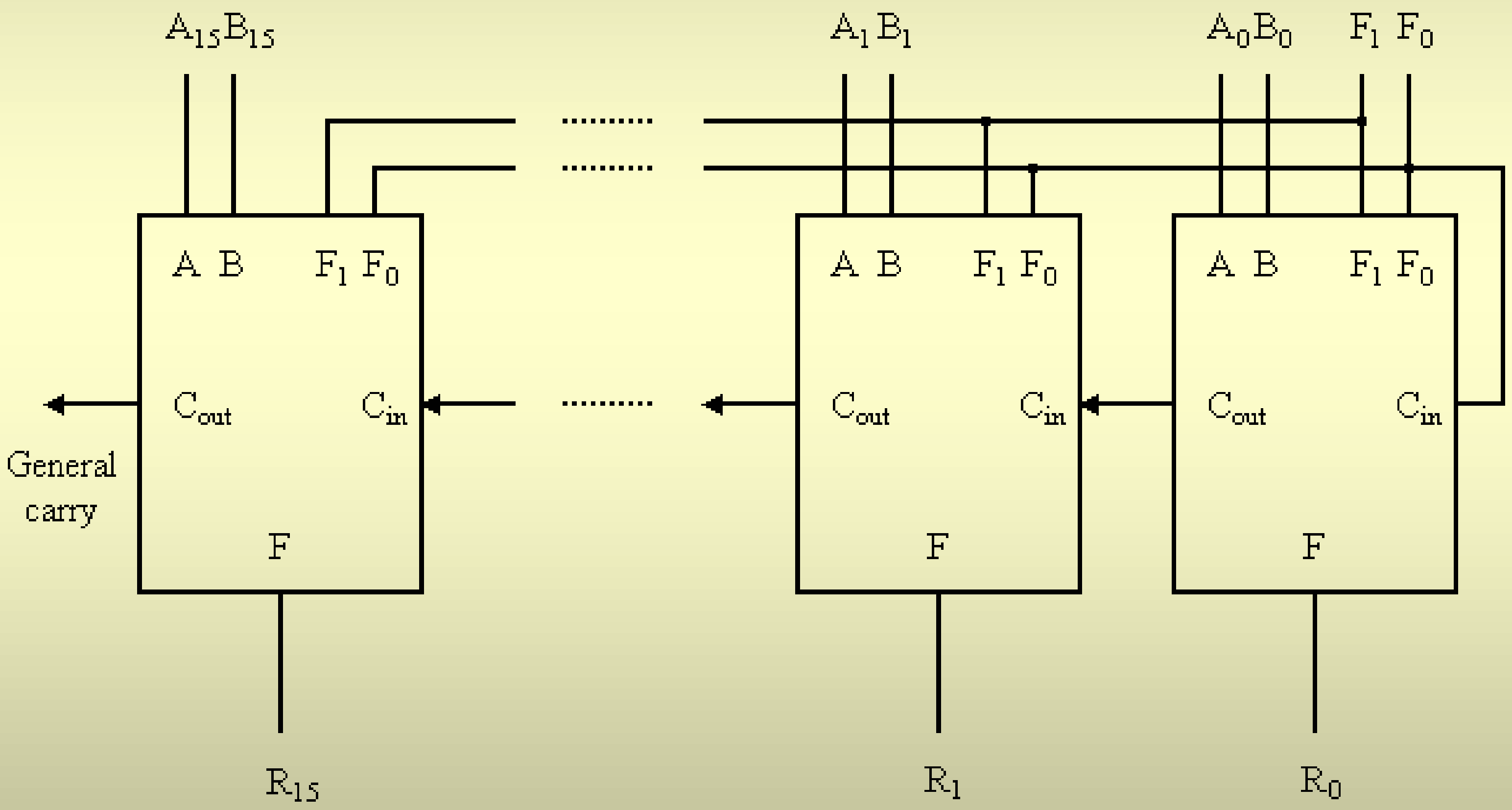
Improved Design



F_1	F_0	F
0	0	A and B
0	1	A or B
1	0	A+B
1	1	A-B

F_1,F_0 - control signals

Arithmetic Logic Unit (16 Bits)


$A_{15}B_{15}$
$A_1 B_1$
$A_0 B_0$
F_1 F_0
A B F_1 F_0
C_{out}
C_{in}
F
General carry
R_{15}
R_1
R_0