OF THE
DEPARTMENT OF
INFORMATION ENGINEERING

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Digital Systems

Exercises on Combinational Circuits

Marta Bagatin, marta.bagatin@unipd.it

Degree Course in Information Engineering

Academic Year 2023-2024

Example 3-17: Mux Implementation of 4-Variable Function

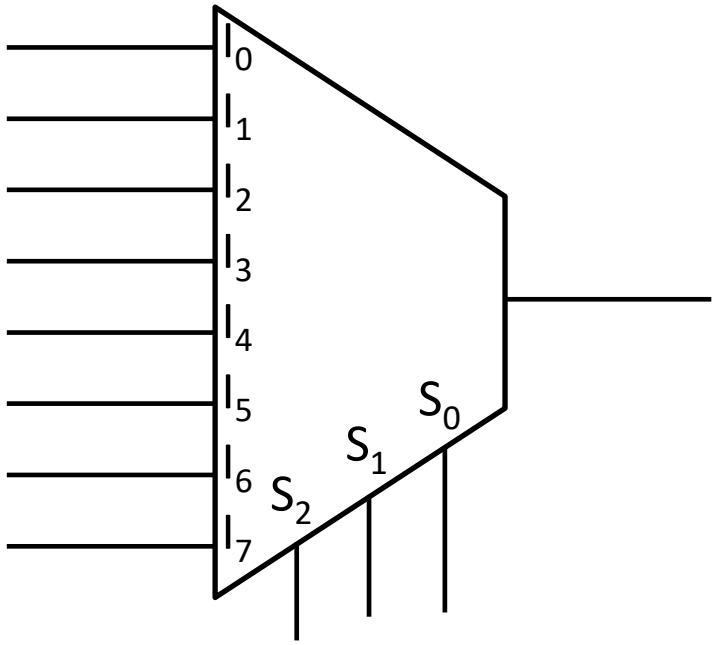
- Implement the following Boolean function through a mux with a 3-bit selection signal

$$\mathrm{F}\,(\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}) = \sum \mathrm{m}\,(1,3,4,11,12,13,14,15)$$

Example 3-17: Mux Implementation of 4-Variable Function

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

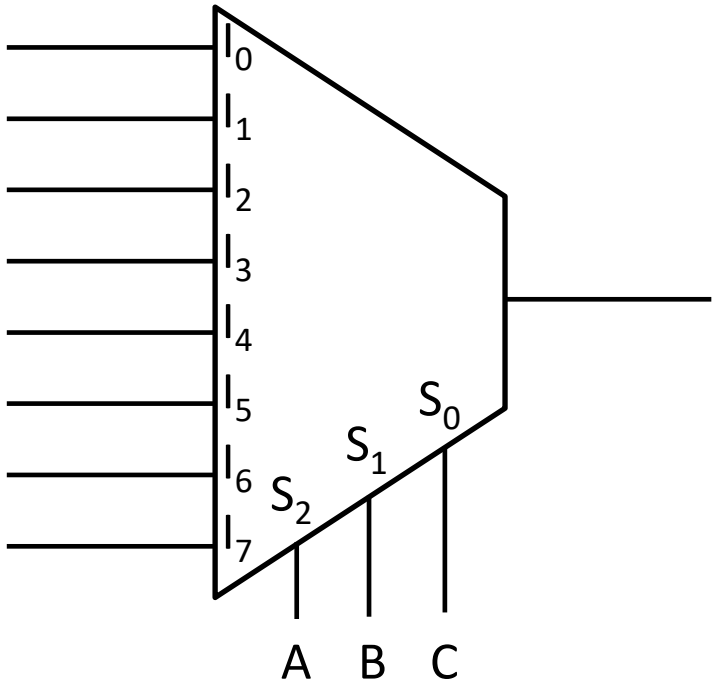
a) We find the truth table of the function
b) We use a mux with 3 selection inputs (8-to-1 line mux)



Example 3-17: Mux Implementation of 4-Variable Function

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

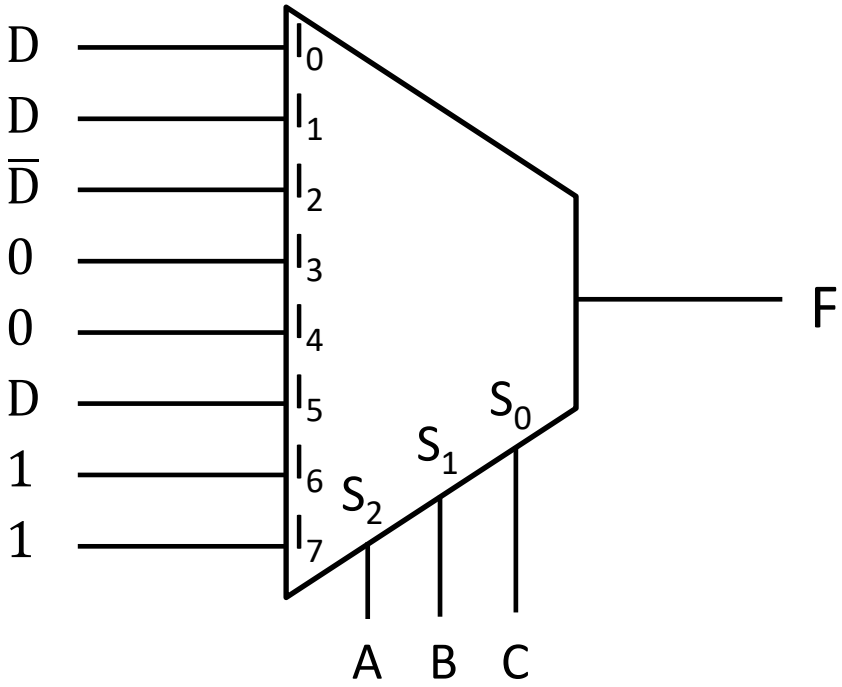
a) We find the truth table of the function
b) We use a mux with 3 selection inputs (8-to-1 line mux)
c) We connect the 3 most significant among the 4 inputs A, B, C, D to the mux selection inputs (pay attention to the order!)



Example 3-17: Mux Implementation of 4-Variable Function

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	$F = \overline{D}$
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	

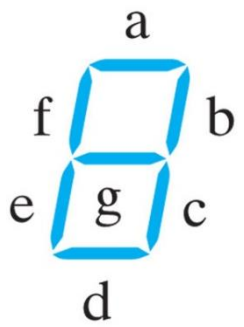
a) We find the truth table of the function
b) We use a mux with 3 selection inputs (8-to-1 line mux)
c) We connect the 3 most significant among the 4 inputs A, B, C, D to the mux selection inputs (pay attention to the order!)
d) We appropriately set the value of the mux data inputs (value fixing) based on the truth table



Example 3-18: BCD-to-Seven Segments Decoder

- **Specifications**: Design a combinational circuit that accepts a decimal digit in BCD as an input and produces the appropriate outputs for the LEDs (Light Emitting Diodes) segments of the display for that decimal digit.

  The circuit needs to control the correct illumination of the 7 segments to represent the input digit. For the unused input combinations (1010-1111) the display needs to be off



Names of the 7 segments

Representation of the 9 BCD digits on the display

Example 3-18: BCD-to-Seven Segments Decoder

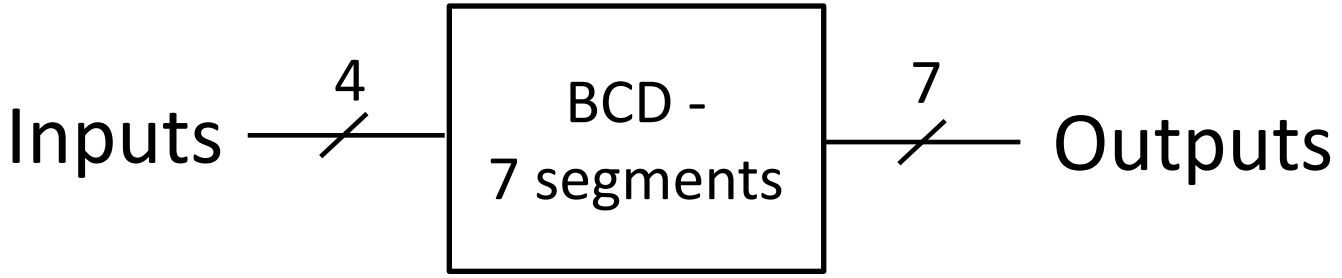
- **Specifications**: Design a combinational circuit that accepts a decimal digit in BCD as an input and produces the appropriate outputs for the LEDs (Light Emitting Diodes) segments of the display for that decimal digit.

  The circuit needs to control the correct illumination of the 7 segments to represent the input digit

  Inputs and Outputs:

  - 4 inputs: A B C D (represent the ten BCD digits)
  - 7 outputs: a b c d e f g (control of the 7 segments)

=> The outputs of the circuit select the corresponding segment in the display, to correctly represent the input digit (output = '0': LED turned off, output= '1': LED turned on)



Example 3-18: BCD-to-Seven Segments Decoder

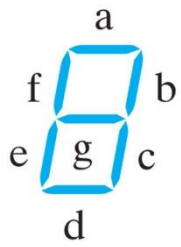
- **Truth table:**

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

Input:
BCD code of
digits 0 to 9

Output = ‘0’: LED segment off
Output = ‘1’: LED segment on

Combinations 1010-1111: display off



Example 3-18: BCD-to-Seven Segments Decoder

- **Boolean function determination:**

  1st implementation: with logic gates

  - Optimization: 7 Karnaugh-maps, one for each output (a, b, c, d, e, f, g)
  - Independent implementation of these 7 functions requires 27 AND gates and 7 OR gates

$$a = \overline{A}C + \overline{A}BD + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}$$

$$b = \overline{A}\,\overline{B} + \overline{A}\,\overline{C}\,\overline{D} + \overline{A}CD + A\overline{B}\,\overline{C}$$

$$c = \overline{A}B + \overline{A}D + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}$$

$$d = \overline{A}C\overline{D} + \overline{A}\,\overline{B}C + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + \overline{A}B\overline{C}D$$

$$e = \overline{A}C\overline{D} + \overline{B}\,\overline{C}\,\overline{D}$$

$$f = \overline{A}B\overline{C} + \overline{A}\,\overline{C}\,\overline{D} + \overline{A}B\overline{D} + A\overline{B}\,\overline{C}$$

$$g = \overline{A}C\overline{D} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C}$$

Example 3-18: BCD-to-Seven Segments Decoder

- **Boolean function determination:**

  1st implementation: with logic gates

  - Optimization: 7 Karnaugh-maps, one for each output (a, b, c, d, e, f, g)
  - Independent implementation of these 7 functions requires 27 AND gates and 7 OR gates
  - By sharing the common product terms among the different outputs, the number of AND gates can be reduced to 15

$$a = \overline{A}C + \overline{A}BD + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}$$

$$b = \overline{A}\,\overline{B} + \overline{A}\,\overline{C}\,\overline{D} + \overline{A}CD + A\overline{B}\,\overline{C}$$

$$c = \overline{A}B + \overline{A}D + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}$$

$$d = \overline{A}C\overline{D} + \overline{A}\,\overline{B}C + \overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + \overline{A}B\overline{C}D$$

$$e = \overline{A}C\overline{D} + \overline{B}\,\overline{C}\,\overline{D}$$

$$f = \overline{A}B\overline{C} + \overline{A}\,\overline{C}\,\overline{D} + \overline{A}B\overline{D} + A\overline{B}\,\overline{C}$$

$$g = \overline{A}C\overline{D} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C}$$

Example 3-18: BCD-to-Seven Segments Decoder

2nd implementation: with decoder

- We use a 4-to-16 decoder to generate all the minterms corresponding to the 4 inputs of the circuit
- Then we connect the minterms of the functions to 7 OR gates (one for each output)

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

$$a(A, B, C, D) = \Sigma m(0, 2, 3, 5, 6, 7, 8, 9)$$

$$b(A, B, C, D) = \Sigma m(0, 1, 2, 3, 4, 7, 8, 9)$$

$$c(A, B, C, D) = \Sigma m(0, 1, 3, 4, 5, 6, 7, 8, 9)$$

$$d(A, B, C, D) = \Sigma m(0, 2, 3, 5, 6, 8, 9)$$

$$e(A, B, C, D) = \Sigma m(0, 2, 6, 8)$$

$$f(A, B, C, D) = \Sigma m(0, 4, 5, 6, 8, 9)$$

$$g(A, B, C, D) = \Sigma m(2, 3, 4, 5, 6, 8, 9)$$

Example 3-18: BCD-to-Seven Segments Decoder

3rd implementation: with multiplexer

- Seven line mux 8-to-1 (one for each output)
  - Selection inputs $S_2\,S_1\,S_0$ connected to A, B, C (pay attention to the order!)
  - Data inputs I_0, I_1, I_2…, I_7 connected appropriately to $D, \overline{D}, 0, 1$

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

Select Inputs	Multiplexer Data Inputs for Each Output Function						
$S_2S_1S_0$	a	b	c	d	e	f	g
000	$\overline{D}$	1	1	$\overline{D}$	$\overline{D}$	$\overline{D}$	0
001	1	1	D	1	$\overline{D}$	0	1
010	D	$\overline{D}$	1	D	0	1	1
011	1	D	1	$\overline{D}$	$\overline{D}$	$\overline{D}$	$\overline{D}$
100	1	1	1	1	$\overline{D}$	1	1
101	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0

Example 3-18: BCD-to-Seven Segments Decoder

3rd implementation: with multiplexer

- Seven line mux 8-to-1 (one for each output)
  - Selection inputs $S_2\,S_1\,S_0$ connected to A, B, C (pay attention to the order!)
  - Data inputs I_0, I_1, I_2…, I_7 connected appropriately to $D, \overline{D}, 0, 1$

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

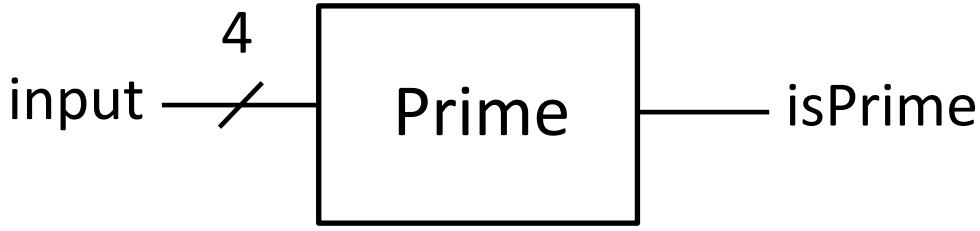
Select Inputs	Multiplexer Data Inputs for Each Output Function						
$S_2S_1S_0$	a	b	c	d	e	f	g
000	$\overline{D}$	1	1	$\overline{D}$	$\overline{D}$	$\overline{D}$	0
001	1	1	D	1	$\overline{D}$	0	1
010	D	$\overline{D}$	1	D	0	1	1
011	1	D	1	$\overline{D}$	$\overline{D}$	$\overline{D}$	$\overline{D}$
100	1	1	1	1	$\overline{D}$	1	1
101	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0

Prime Circuit

- Write the VHDL code describing the following circuit
  - Input (4 bit): input
  - Output (1 bit): isPrime, '1' if input is prime, '0' otherwise

  in the following ways:

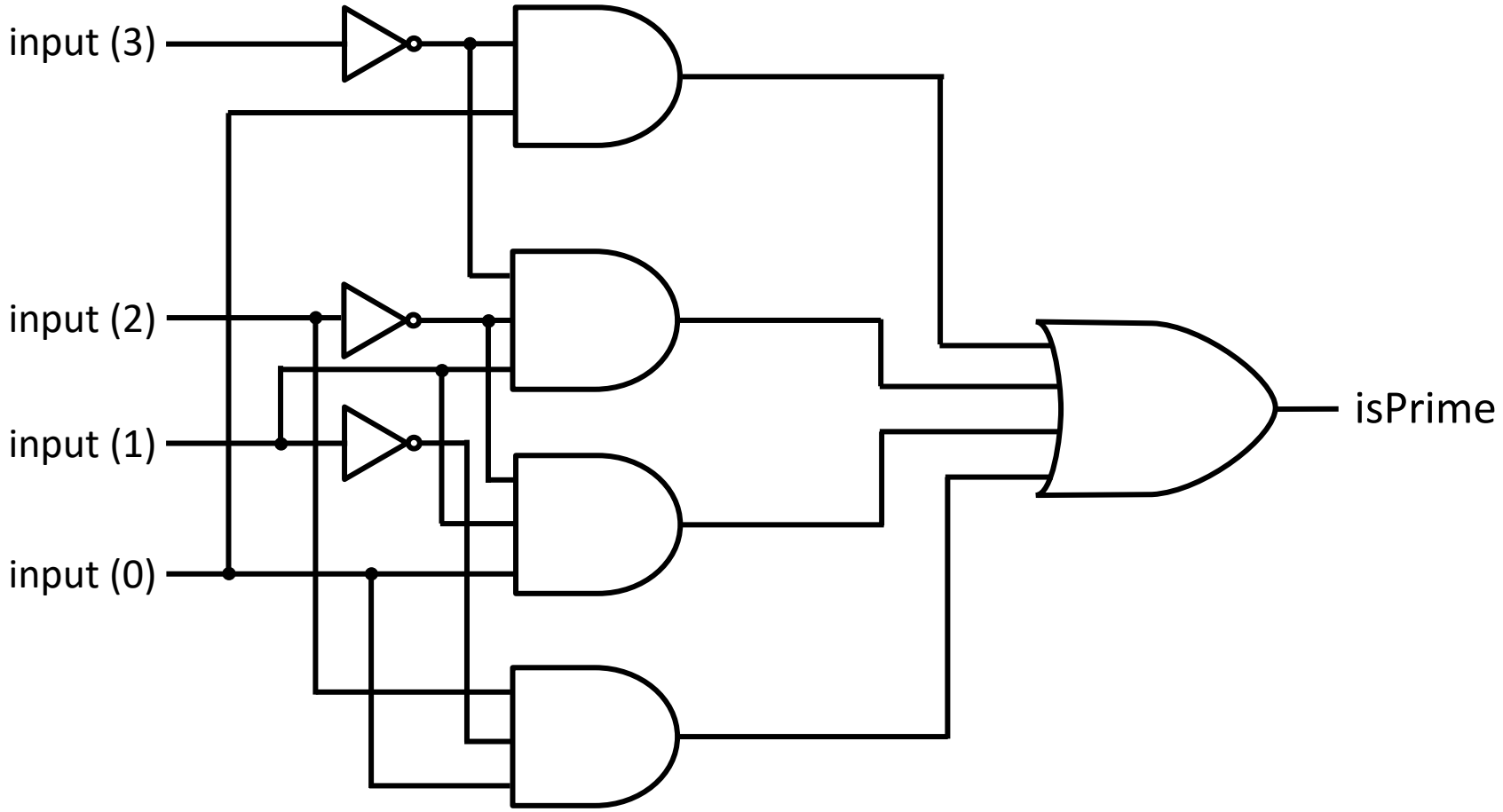
  1) Structural description
  2) Dataflow description
  3) Behavioral description



- Write a testbench to check the circuit functionality and simulate it with EDA Playground

Prime Circuit

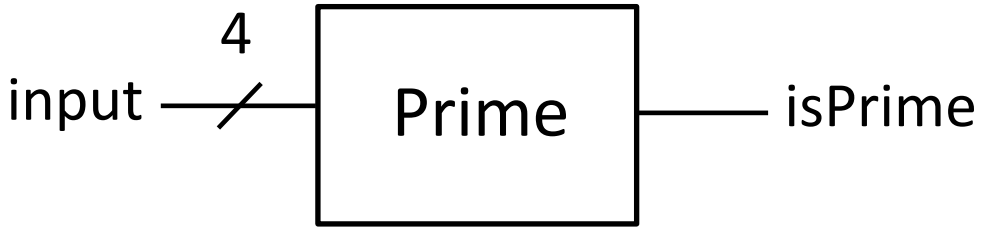
As a first step we find the logic circuit (optimized with K-maps):



Prime Circuit

1) **Structural-dataflow** description **(1/2)**

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Prime is
    port (input: in std_logic_vector (3 downto 0);
        isPrime: out std_logic);
end Prime;

architecture implStruct of Prime is
signal a1, a2, a3, a4, n1, n2, n3: std_logic;
    begin
    AND1: entity work.andGate(andImpl) port map(a1, n3, input(0));
    AND2: entity work.andGate(andImpl) port map(a2, input(2), n1, input(0));
    AND3: entity work.andGate(andImpl) port map(a3, n2, input(1), input(0));
    AND4: entity work.andGate(andImpl) port map(a4, n3, n2, input(1));
    n3 <= NOT input(3);
    n2 <= NOT input(2);
    n1 <= NOT input(1);
    isPrime <= a1 OR a2 OR a3 OR a4;
end implStruct;
```



AND component with 3 inputs, set to '1' by default (defined in the next slide)

Prime Circuit

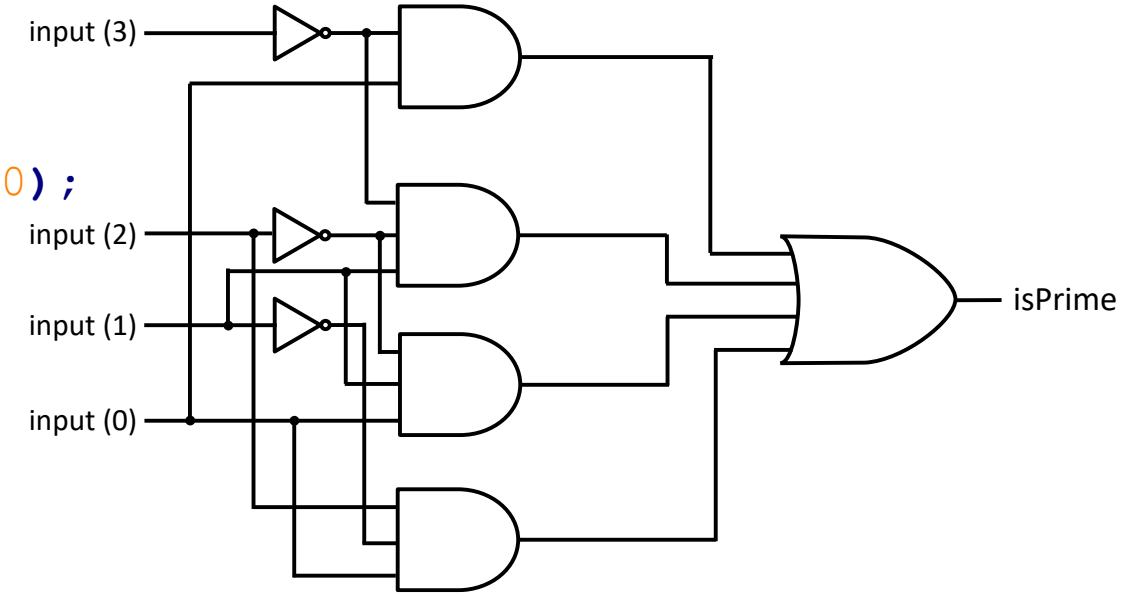
1) **Structural-dataflow** description **(2/2)**

```
library IEEE;
use IEEE.std_logic_1164.all;

entity andGate is
    port (y: out std_logic;
    a, b, c: in std_logic: = '1');
end andGate;

architecture andImpl of andGate is
    begin
    y <= a and b and c;
end andImpl;
```

Prime Circuit

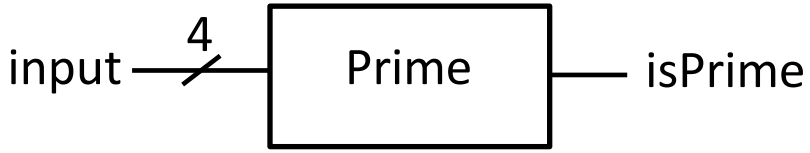
2) **Dataflow** description

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Prime is
    port (input: in std_logic_vector (3 downto 0);
          isPrime: out std_logic);
end Prime;

architecture implLogic of Prime is
    begin
    isPrime <= (input(0) AND NOT(input(3))) OR
               (input(1) AND NOT (input(2)) AND NOT(input(3))) OR
               (input(0) AND NOT (input(1)) AND input(2)) OR
               (input(0) AND input(1) AND NOT (input(2)));
end implLogic;
```



Prime Circuit

3.a) Behavioral description with **`case`** statement




```
library IEEE;
use IEEE.std_logic_1164.all;

entity Prime is
    port (input: in std_logic_vector (3 downto 0);
    isPrime: out std_logic);
end Prime;

architecture implCase of Prime is
    begin
    process(input) begin
    case input is
       when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d" => isPrime <= '1';
       when others => isPrime <= '0';
    end case;
 end process;
end implCase;
```

x to indicate hexadecimal numbers

Case Statement: Parallel Logic

Syntax of case statement:

```
case expression is
    when choice1 => {statements}
    when choice2 => {statements}
    when others => {statements}
end case;
```

- The case statement is a sequential statement similar to combinational statement "with-select“ and it **needs to appear inside a process**
- Allows us to specify the value of a logic function for all the combinations of the inputs
- **All possible choices need to be covered** (as “with select”) and the various **choices cannot overlap**
- The statement "null” can be used to perform no action under specific conditions
- **All inputs must appear in the sensitivity list of a process to generate combinational logic**

Prime Circuit

3.b) **Behavioral** description with **`if`** statement

```
entity Prime is
 port (input: in std_logic_vector (3 downto 0);
 isPrime: out std_logic);
end Prime;

architecture implIf of Prime is
 begin
 process (input) begin
  if input = 4d"1" then isPrime <= '1';
 elsif input = 4d"2" then isPrime <= '1';
 elsif input = 4d"3" then isPrime <= '1';
 elsif input = 4d"5" then isPrime <= '1';
 elsif input = 4d"7" then isPrime <= '1';
 elsif input = 4d"11" then isPrime <= '1';
 elsif input = 4d"13" then isPrime <= '1';
 else isPrime <= '0';
  end if;
 end process;
end implIf;
```

4d: 4-bit decimal

If Statement: Priority Logic

Syntax for if statement:

```
if condition1 then
    {sequential_statement1}
elsif condition2 then
    {sequential_statement2}
else
    {sequential_statement3}
end if;
```

- The if statement is a sequential statement similar to combinational statement "when-else“ and it must appear **inside a process**
- Evaluates each condition in order, i.e. it generates a priority structure (similar to concurrent statement "when-else ")
- Caution: All choices need to be covered to implement combinational logic using if-else statement
- **All inputs must appear in the sensitivity list of a process to generate combinational logic**

Exercise 3-55: Arithmetic Operations

- **Perform the following arithmetic operations between the following signed numbers in 2s complement representation**. Then verify the results and indicate whether overflow occurs for each computation

  a) 110001 + 011101
  b) 0110111 + 0101111
  c) 00000111 - 11110100
  d) 0110111 - 0101111

Exercise 3-55: Arithmetic Operations

a) 110001 + 011101

Carry: 110001

```
  110001 +
  011101
--------
  001110
```

Verification:
-15 + 29 = + 14

NOTE:

- There is no overflow since the carry bits in the n and n-1 positions are equal
- Carry out in the sign bit position is 1 (and it must be discarded)

Exercise 3-55: Arithmetic Operations

b) 0110111 + 0101111

Carry: 0111111

```
  0110111 +
  0101111
 --------
 01100110
```

Verification:
+55 + 47 = + 102

NOTE:

- **There is overflow** since the carry bits in the n and n-1 positions are different: we need to consider the carry-out in the sign bit position as the sign bit of the result
- The carry out in the sign bit position is zero

Exercise 3-55: Arithmetic Operations

c) 0000 0111 - 1111 0100

For subtraction with signed numbers, we sum the 2's complement of the sutraend to the minuend

Carry: 0001 1000

```
  0000 0111 +
  0000 1100
-------------
  0001 0011
```

Verification:
+7 - (- 12) = + 19

NOTE:

- There is no overflow

Exercise 3-55: Arithmetic Operations

d) 0110111 - 0101111

Carry: 1110111

```
  0110111 +
  1010001
-------------
  0001000
```

Verification:
+55 - (+ 47) = + 8

NOTE:

- There is no overflow
- There is a carry out in the sign bit position (and it must be discarded)

Exercize

- Write the VHDL code describing a 2-to-4 decoder with:
  - Dataflow architecture
  - Behavioral architecture using the ‘when else’ statement
  - Behavioral architecture using the ‘with select’ statement
- Realize a VHDL testbench to simulate the correct functionality of the circuit with all the input combinations using one of the architectures above

Exercize

- Write the VHDL code describing a 4-to-1 multiplexer with:
  - Dataflow architecture
  - Behavioral architecture using the ‘if’ statement
  - Behavioral architecture using the ‘case’ statement
- Realize a VHDL testbench to simulate the correct functionality of the circuit with all the input combinations using one of the architectures above

Disclaimer

Figures from *Logic and Computer Design Fundamentals*, Fifth Edition, GE Mano | Kime| Martin