Lecture 0_7.2

RT-level Combinational Blocks

Paolo PRINETTO
Politecnico di Torino (Italy)
University of Illinois at Chicago, IL (USA)
Paolo.Prinetto@polito.it
prinetto@uic.edu
www.testgroup.polito.it
www.comitato-girotondo.org

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Goal

– This lecture aims at presenting the set of combinational functional blocks that are usually considered to be “elementary” or “basic” at the RT level.

Prerequisites

– Track 0_4 - System & Device Modeling

Homework

Have a look at:

- Xilinx Virtex-5 Libraries Guide for HDL Designs - UG621 (v 11.3) September 16, 2009

Further readings

- **Students interested in making a reference to a text book on the arguments covered in this lecture can refer, for instance, to:**
  - ***G. Conte, A. Mazzeo, N. Mazzocca, P. Prinetto*: "Architettura dei calcolatori",**

    Città Studi, 2015
    (App.D - Componenti funzionali a livello RT)
    (In Italian)

Further readings

– M. Morris Mano, C.R.Kime:
“*Logic and Computer Design Fundamentals*,”
2nd edition updated
Prentice Hall, Upple Saddle River, NJ (USA),
2001, (chapter 3, pp. 111-148)

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Multiplier
- Adder-Subtractor
- Encoder
- Decoder
- ROM
- Comparator
- ALU
- 1's counter
- I/O interfaces

Basic combinational blocks

Multiplexer

Demultiplexer

Adder-Subtractor

Encoder

Decoder

Multiplier

Comparator

ALU

ROM

I/O interfaces

1's counter

Adders

We shall consider:

- ***half adder***
- ***full adder***
- ***N-bit adder***
- ***N-bit adder-subtractor***

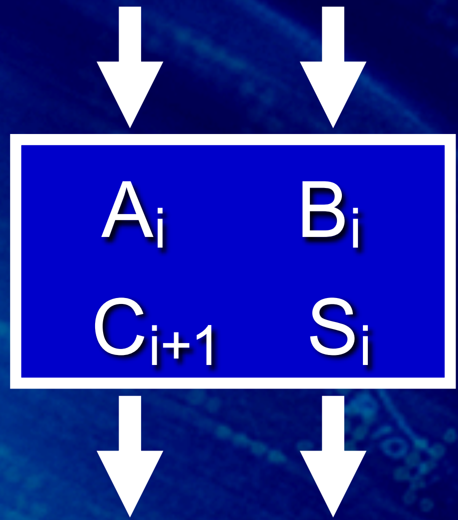
Half-Adder

A *half-adder* has:

- *2* inputs A_i, B_i
- *2* outputs S_i and C_{i+1}

and computes the binary sum of the 2 input bits, providing:

- the *sum* S_i
- the *carry* C_{i+1}



$$S_i = A_i \oplus B_i$$

$$C_{i+1} = A_i \cdot B_i$$

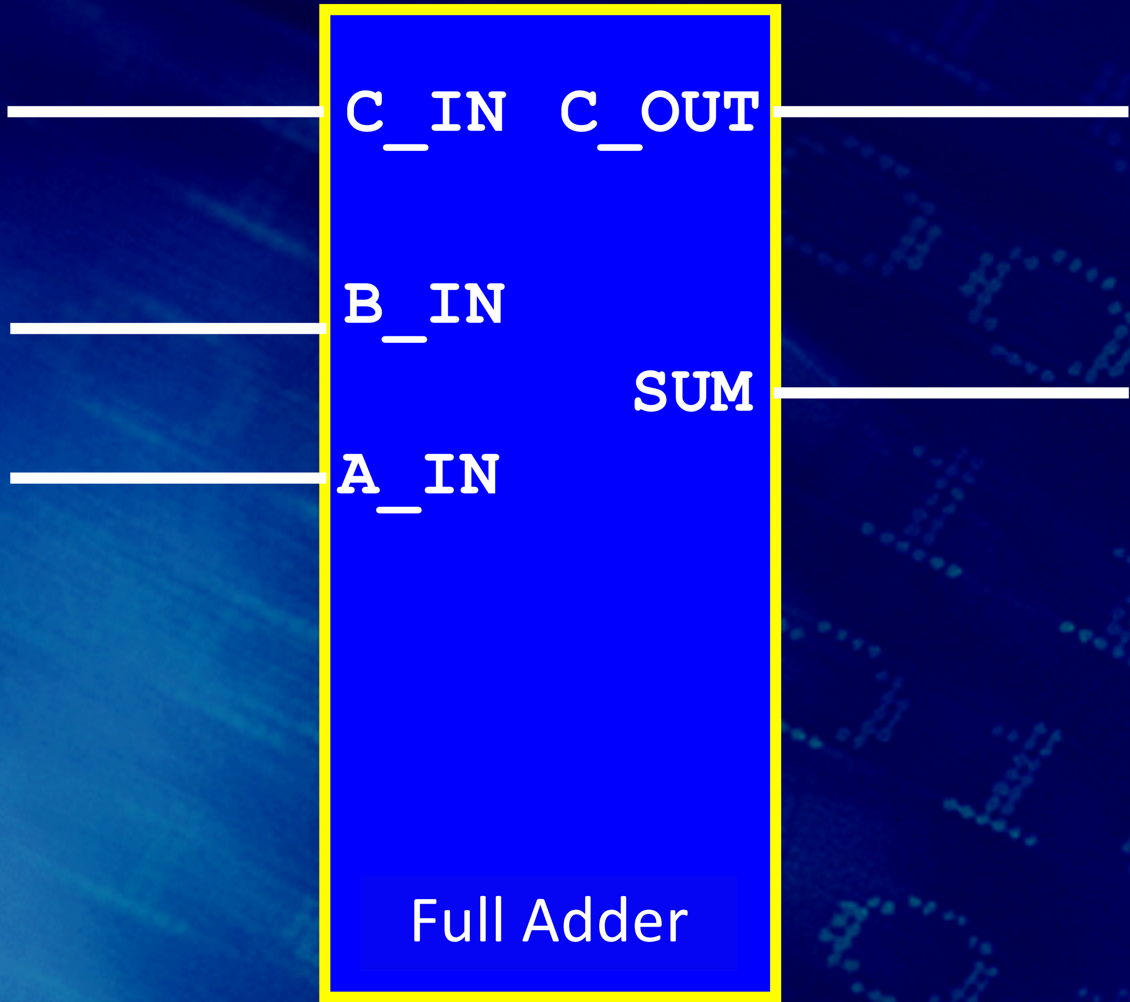
Full-Adder

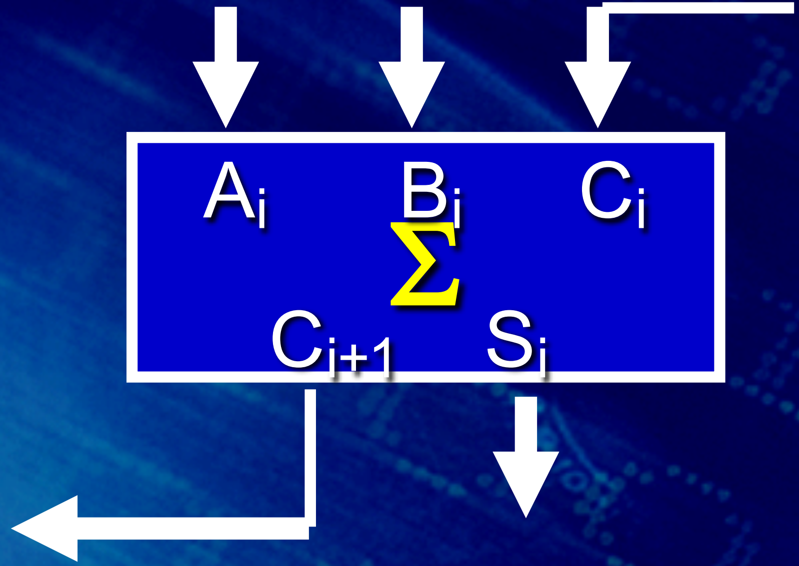
A *full-adder* has:

- *3* inputs A_i, B_i and C_i
- *2* outputs S_i and C_{i+1}

and computes the binary sum of the 3 input bits, providing:

- the *sum* S_i
- the *carry* C_{i+1}


C_IN
C_OUT
B_IN
SUM
A_IN
Full Adder



$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i =$$

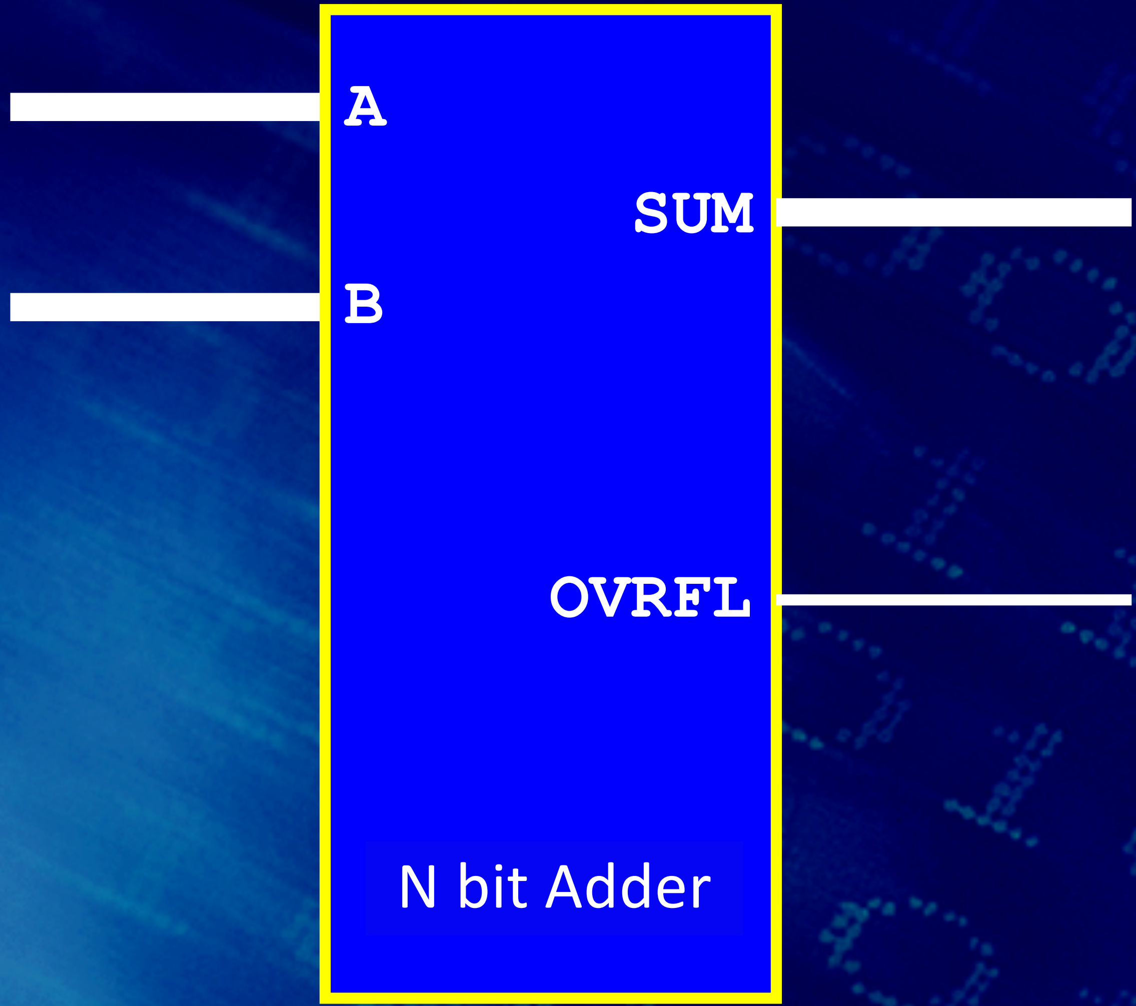
$$= A_i B_i + (A_i \oplus B_i) C_i$$

N-bit Adder

A n-bit adder is a combinational block capable of adding two n-bits inputs operands, detecting overflow conditions. It has

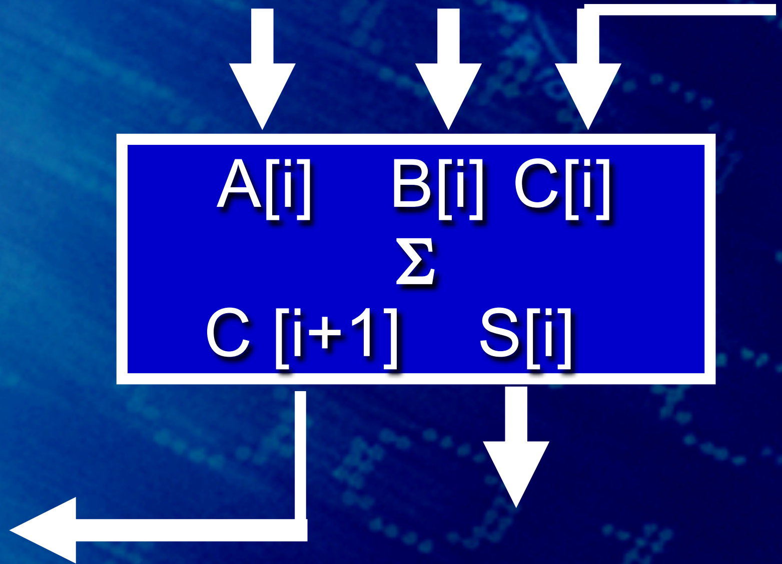
- ***2 n-bits*** **data inputs, both labeled from** ***n-1*** **to** ***0***
- ***1 n-bits*** **data output**
- ***1*** **control output asserted when an overflow condition is occurred.**

Rel. 14.06.2018


A
B
SUM
OVRFL
N bit Adder

N-bit adder implementation

- Design an ***N-bit adder*** resorting to full-adders:



Solution

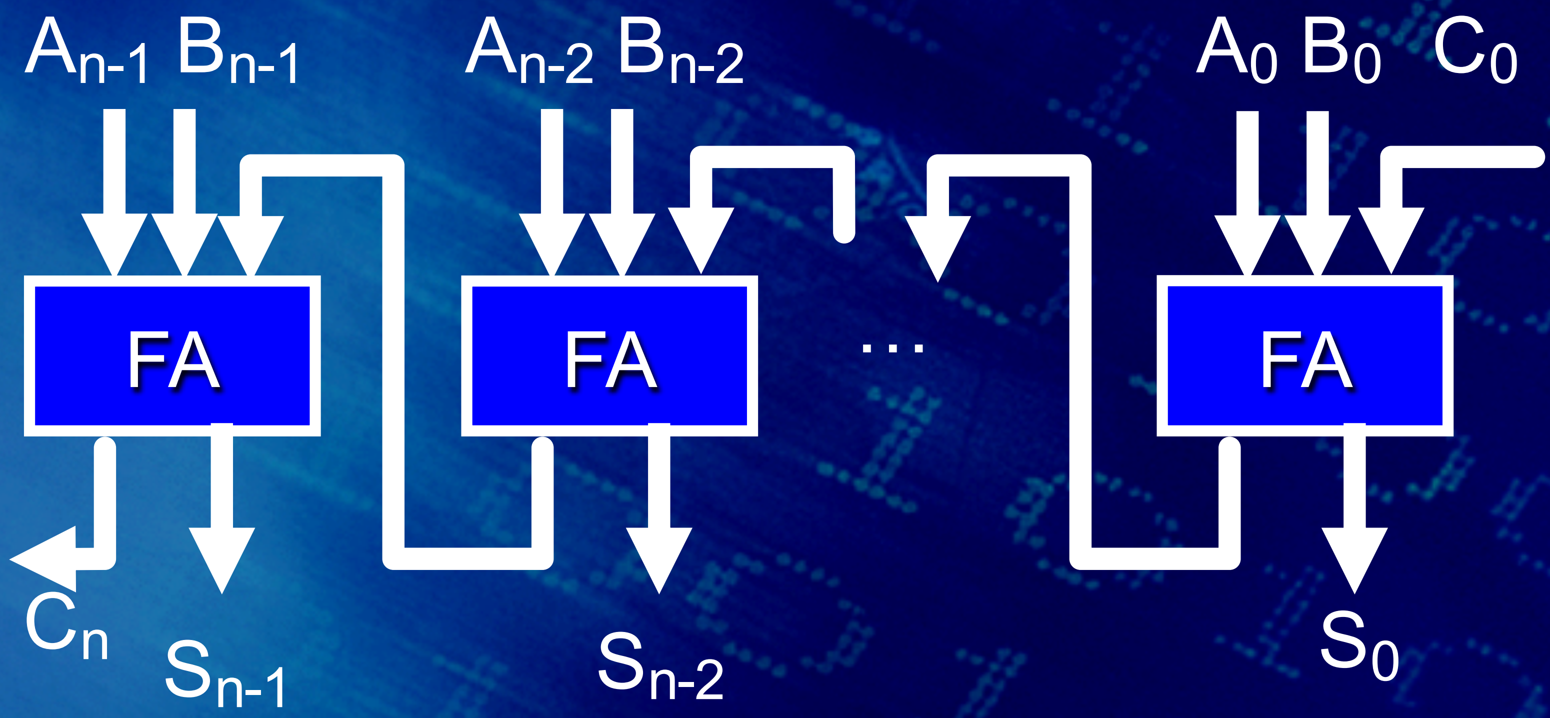
- The ***N*** full-adders can be connected according to two alternative architectures:
  - *ripple carry*
  - *look-ahead carry.*

Ripple carry adder

- **The implementation in based on the same algorithm used when the addition is performed manually by paper and pencil:**

  the carry input C_i of each cell
  is fed by the carry output C_{i-1}
  of the adjacent less significant cell.

Ripple carry adder



Delays

- The global delay of the adder is:

$$n \cdot \Delta_{FA}$$

being Δ_{FA} the delay of each full-adder cell.

Look-ahead carry adder

A ***look-ahead carry adder*** **is a faster, but more expensive, implementation.**

Each cell generates 2 outputs:

$P_i = A_i \oplus B_i$: ***propagation function***

$G_i = A_i \cdot B_i$: ***generation function***

Look-ahead carry adder *(cont'd)*

The output of the i-th cell can be expressed as:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i C_i$$

Look-ahead carry adder *(cont'd)*

The output of the i-th cell can be expressed as:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i C_i$$

Carry signals are evaluated as follows:

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0$

C_0

…

Lo

All carry signals depend on A and B signals, only, and can be easily generated by a combinational 2 level circuit (carry look-ahead generator)

The outp

S_i

$C_{i+1} = \ldots C_i$

Carry ...gnals are evaluated as follows:

$C_1 = G_0 + P_0 C_0$

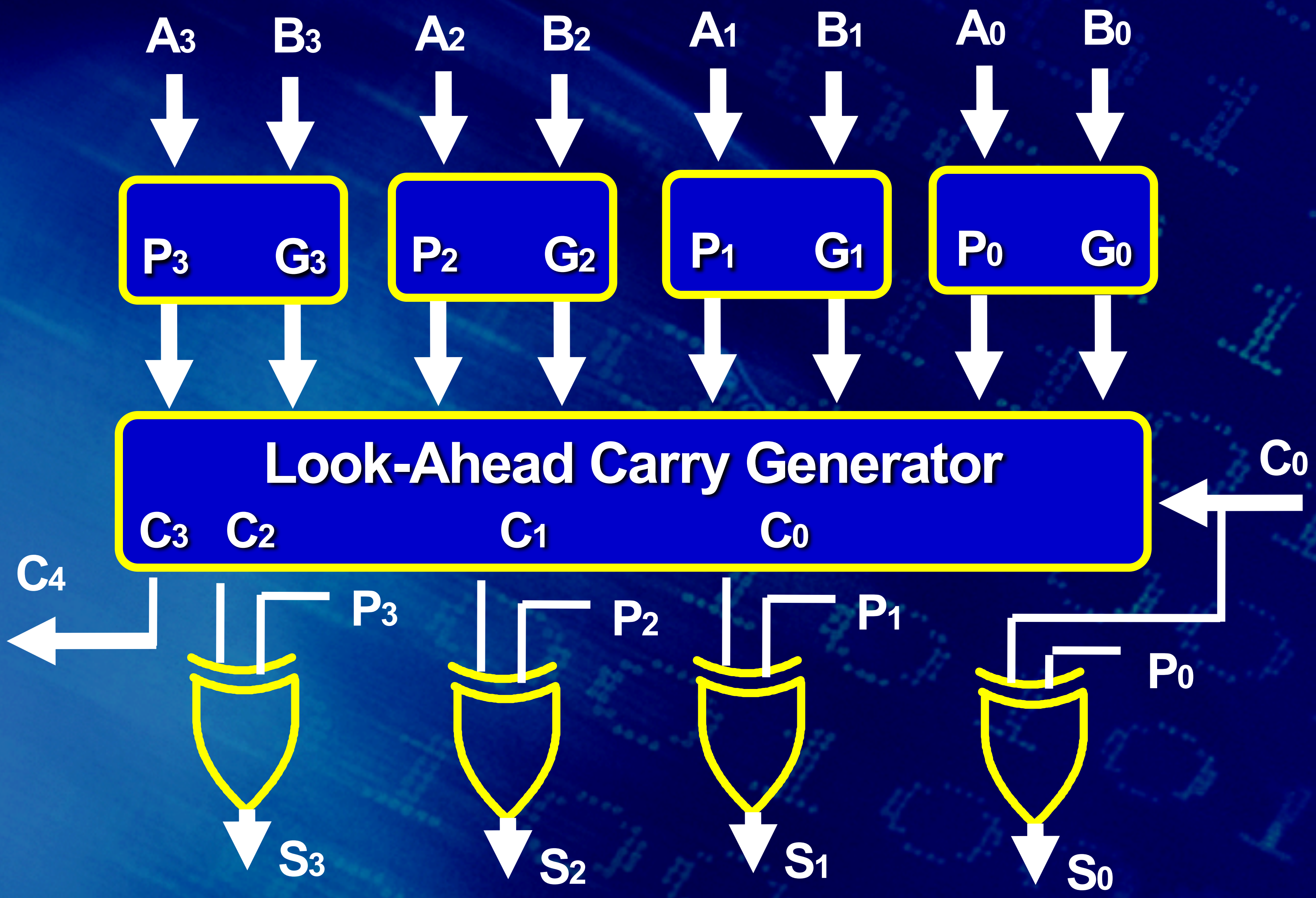
$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0$

C_0

…

4-bit look-ahead carry adder



Overflow signal generation

- The overflow signal OVFL is generated differently, according whether the internal carry signals are:
  - not accessible
  - accessible

Overflow signal generation

- The overflow signal OVFL is generated differently, according whether the internal carry signals are:
  - not accessible
  - accessible

$$OVFL = [\, A_{(n-1)} = B_{(n-1)} \,] \text{ and } [\, A_{(n-1)} \neq S_{(n-1)} \,]$$
$$= A_{(n-1)}\, B_{(n-1)}\, S_{(n-1)}' + A_{(n-1)}'\, B_{(n-1)}'\, S_{(n-1)}$$

Overflow signal generation

- The overflow signal OVFL is generated differently, according whether the internal carry signals are:
  - not accessible
  - accessible

$$OVFL = C_{(n)} \oplus C_{(n-1)}$$

Adders

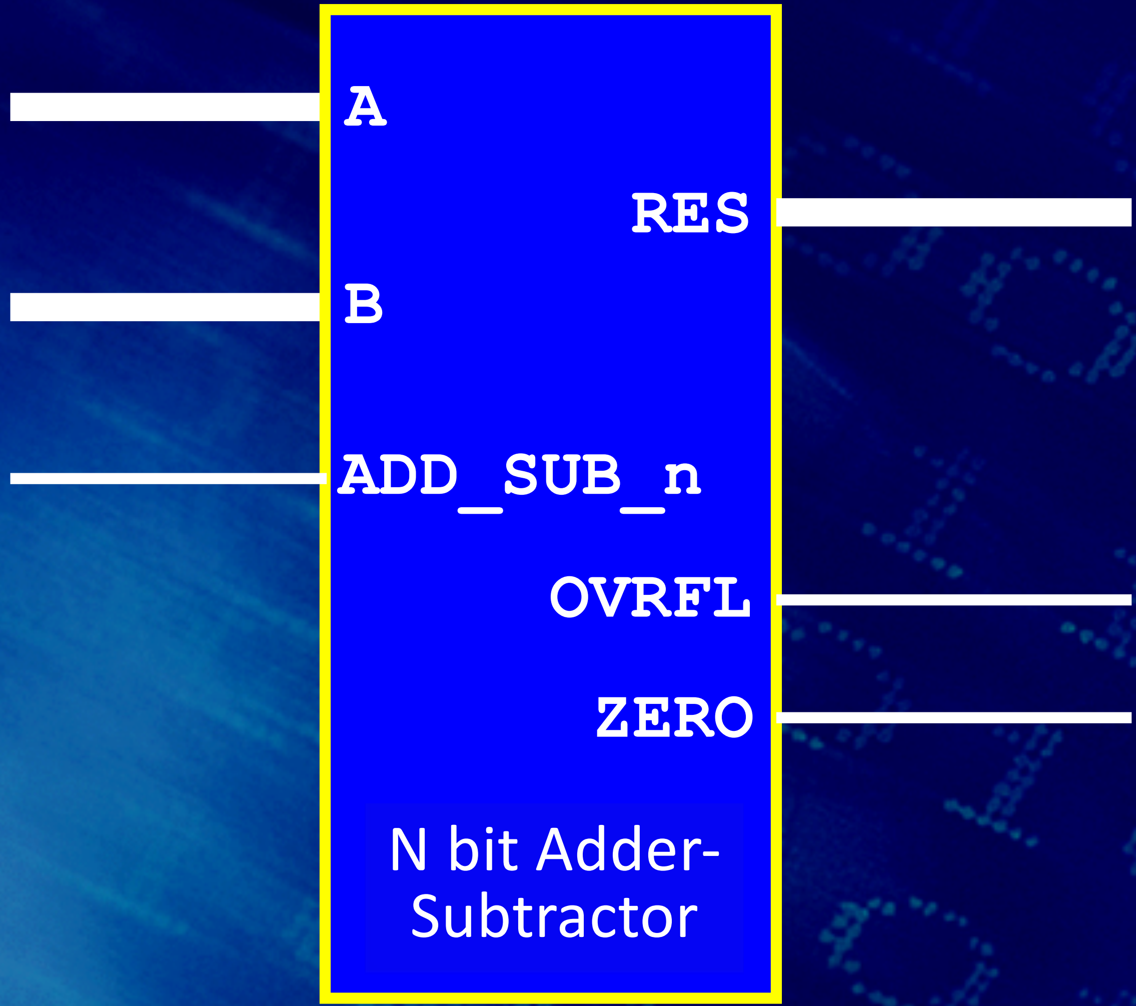
We shall consider:

- ***half adder***
- ***full adder***
- ***N-bit adder***
- ***N-bit adder-subtractor***

N-bit Adder-Subtractor

An adder-Subtractor is a combinational block capable of adding/subtracting two n-bits inputs operands, detecting overflow conditions. It has

- ***2 n-bits* data inputs, both labeled from *n-1* to *0***
- ***1* control input to select one of the two operations**
- ***1 n-bits* data output**
- ***1* control output asserted when an overflow condition is occurred.**


A
RES
B
ADD_SUB_n
OVRFL
ZERO
N bit Adder-
Subtractor

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Adder-Subtractor
- Encoder
- Decoder
- Multiplier
- Comparator
- ALU
- ROM
- I/O interfaces
- 1's counter

Multiplexer

A multiplexer is a combinational block capable of forcing its output to the current value of one of its inputs, according to the values of some control signals.

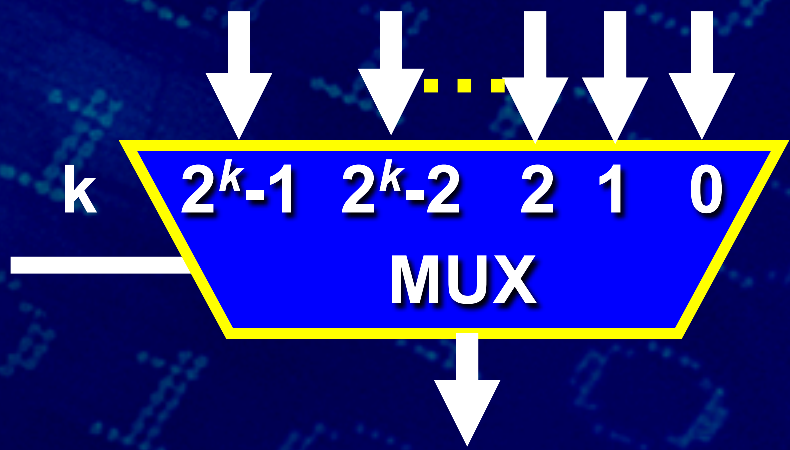
They are usually used to switch one of many inputs to a single output.

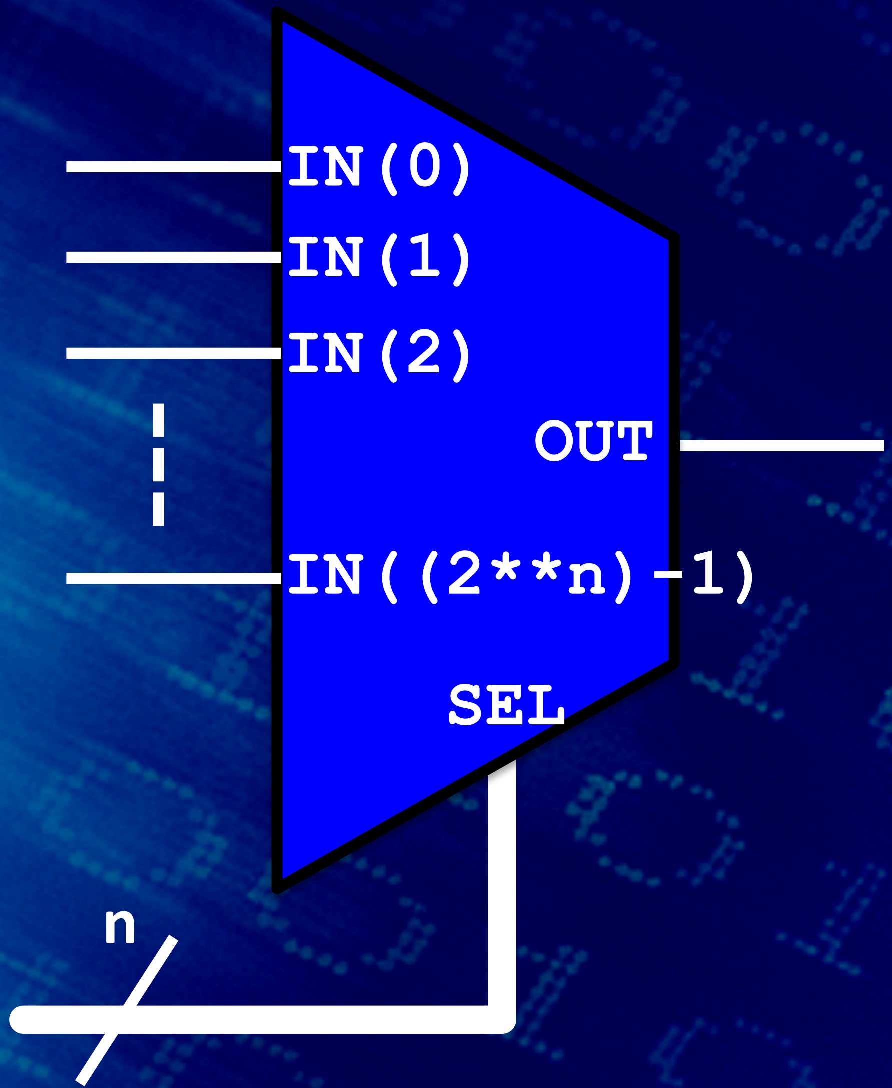
Multiplexer

A multiplexer has:

- ***2^k*** **data inputs, labeled from** ***2^k-1*** **to** ***0***
- ***k*** **control inputs**
- ***1*** **data output.**

It gets the value present on the input labeled 2^i, where i $(0 \le i \le k\text{-}1)$ is the binary number present on the control inputs




IN(0)
IN(1)
IN(2)
OUT
IN((2**n)-1)
SEL
n

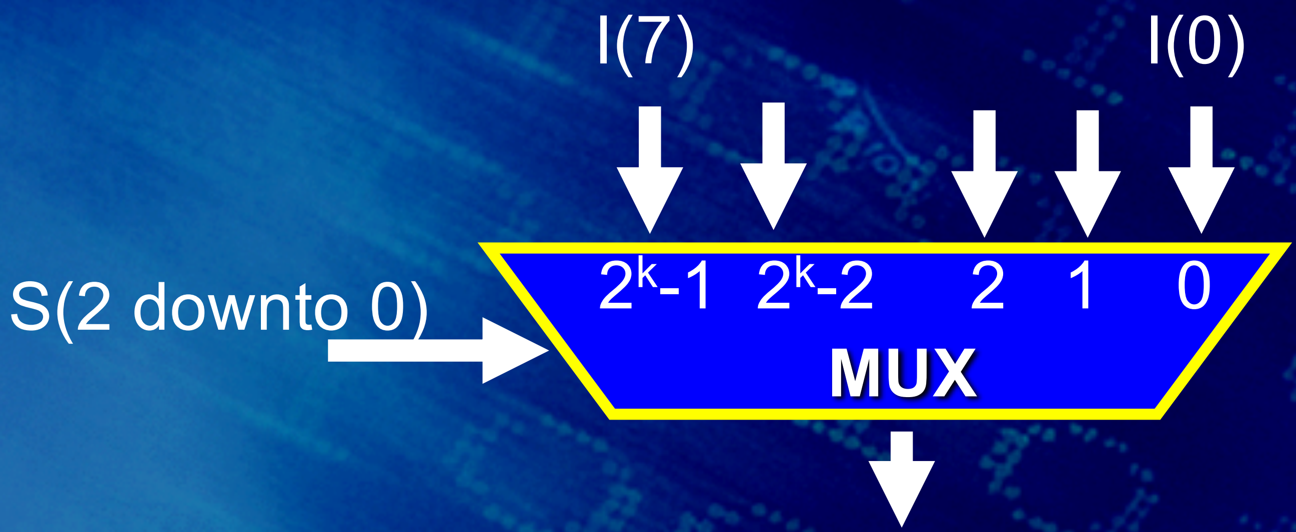
VHDL description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity mux is
generic( k  : integer := 3 );
port(MUX_IN :  in std_logic_vector( (2**k)-1 downto 0 );
        SEL :  in  unsigned(k-1 downto 0);
        MUX_OUT :  out std_logic);
end mux;
```

```
architecture beh of  mux is
begin
    P1: process( MUX_IN, SEL )
     begin
          MUX_OUT <=  MUX_IN(conv_integer(SEL ));
      end process;
end beh;
```

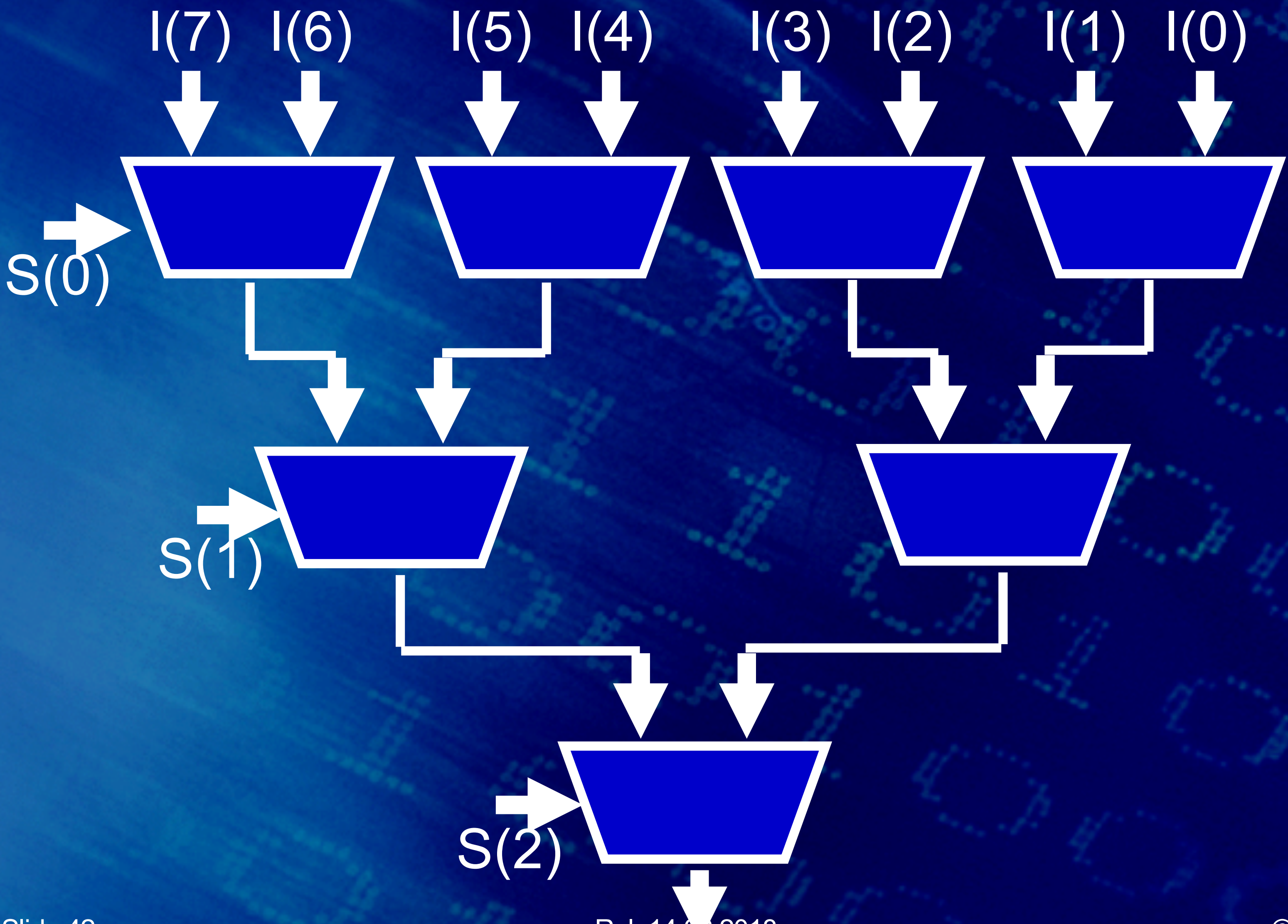
Exercize

How implementing an 8 → 1 mux

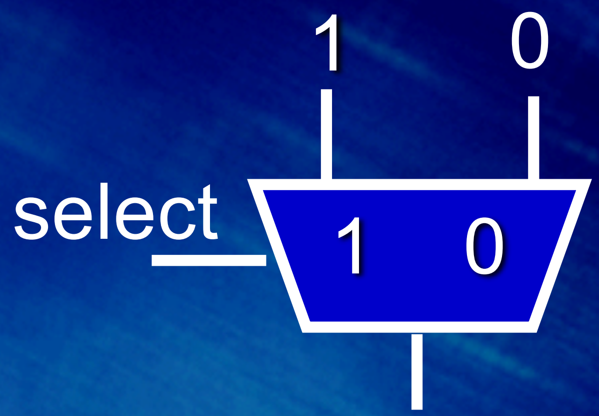


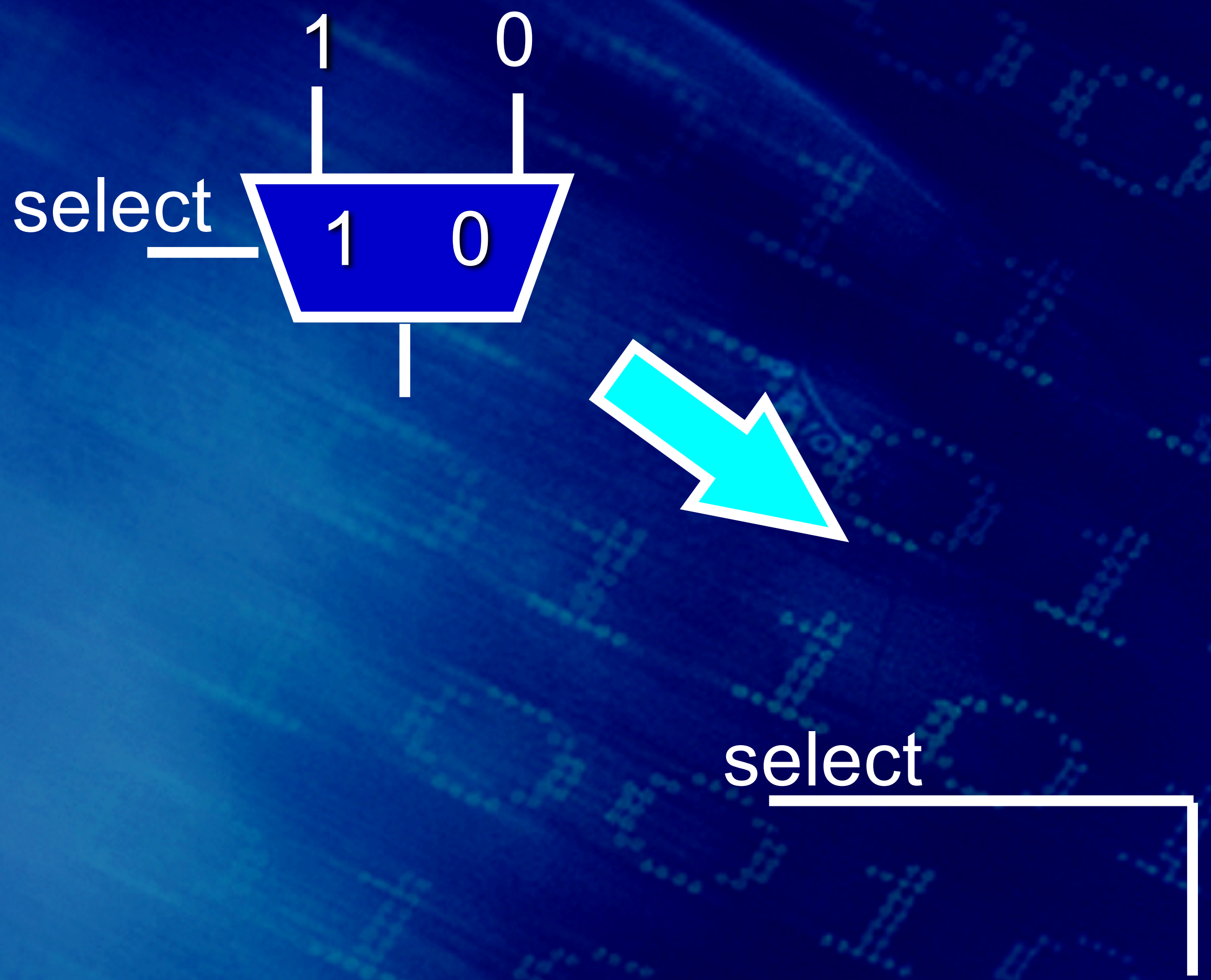
resorting to 2 → 1 mux's, only ?

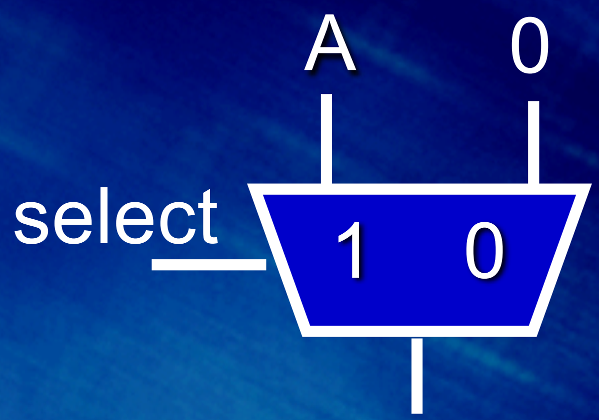
Solution

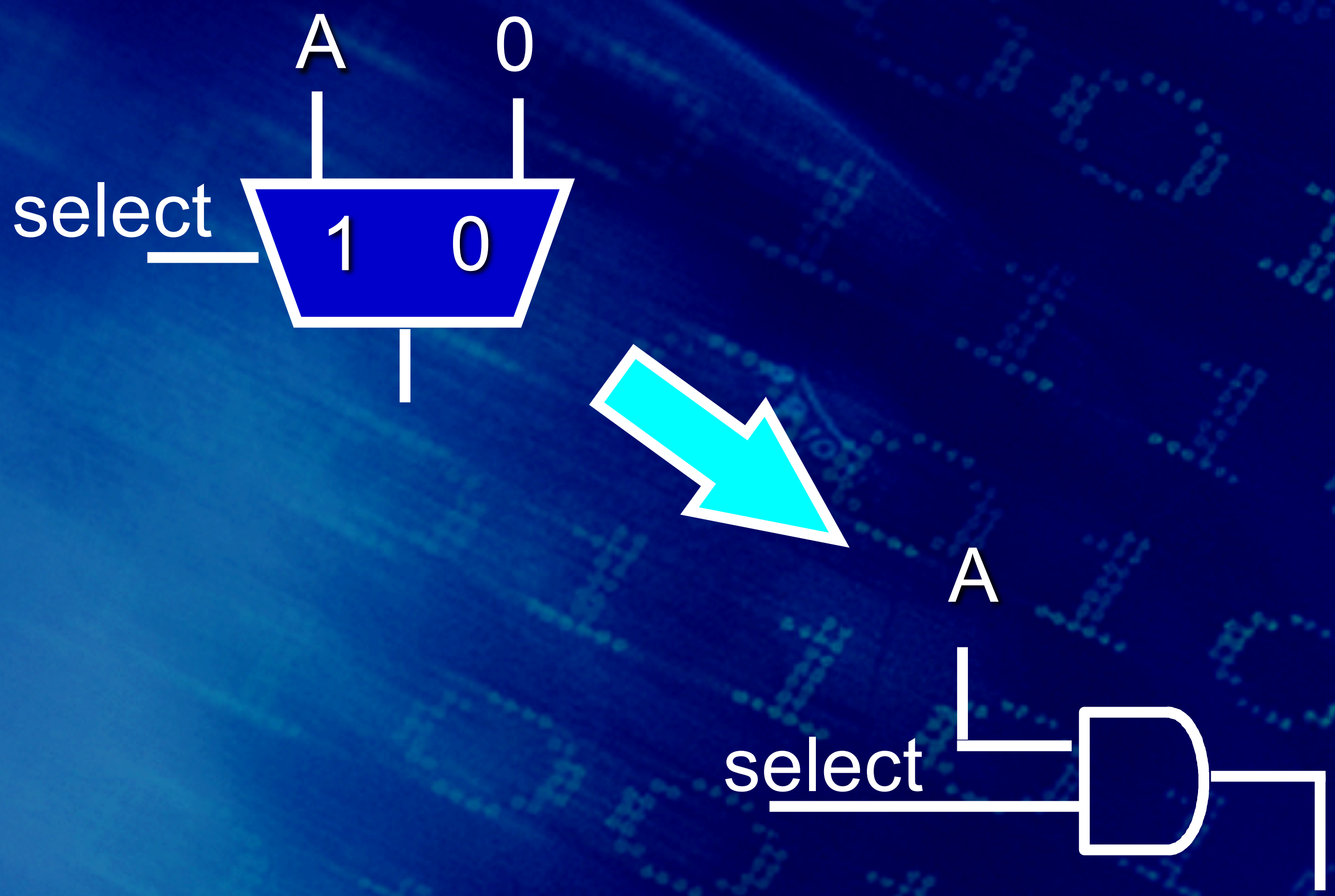


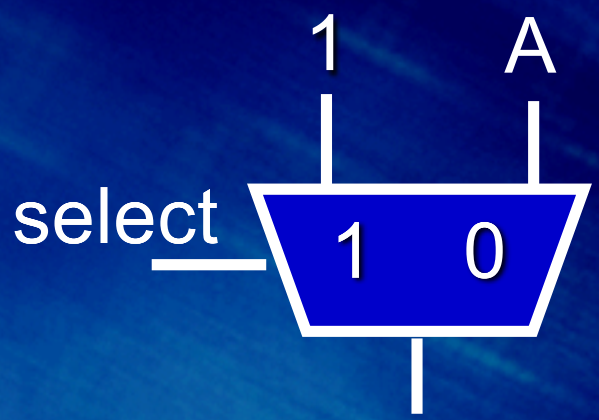
Some optimizations on mux

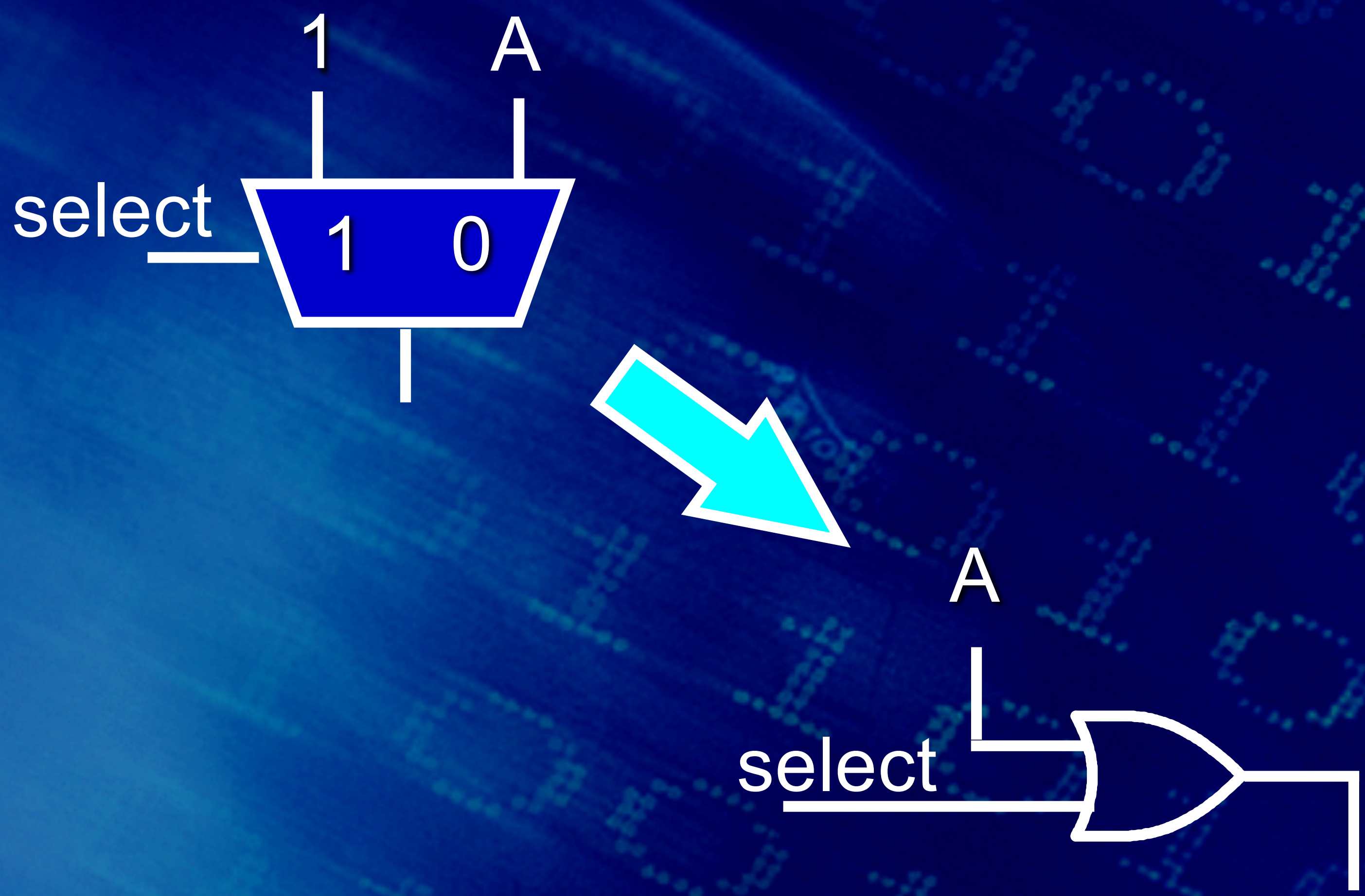

1
0
select
1
0

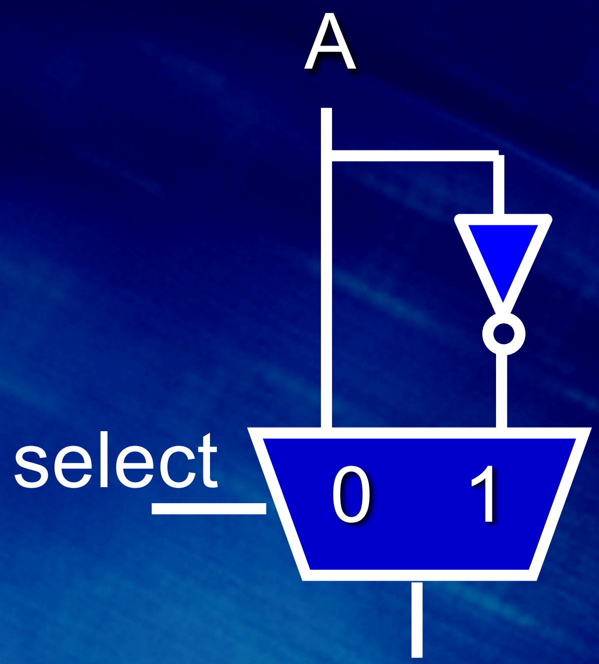

1
0
select
1
0
select

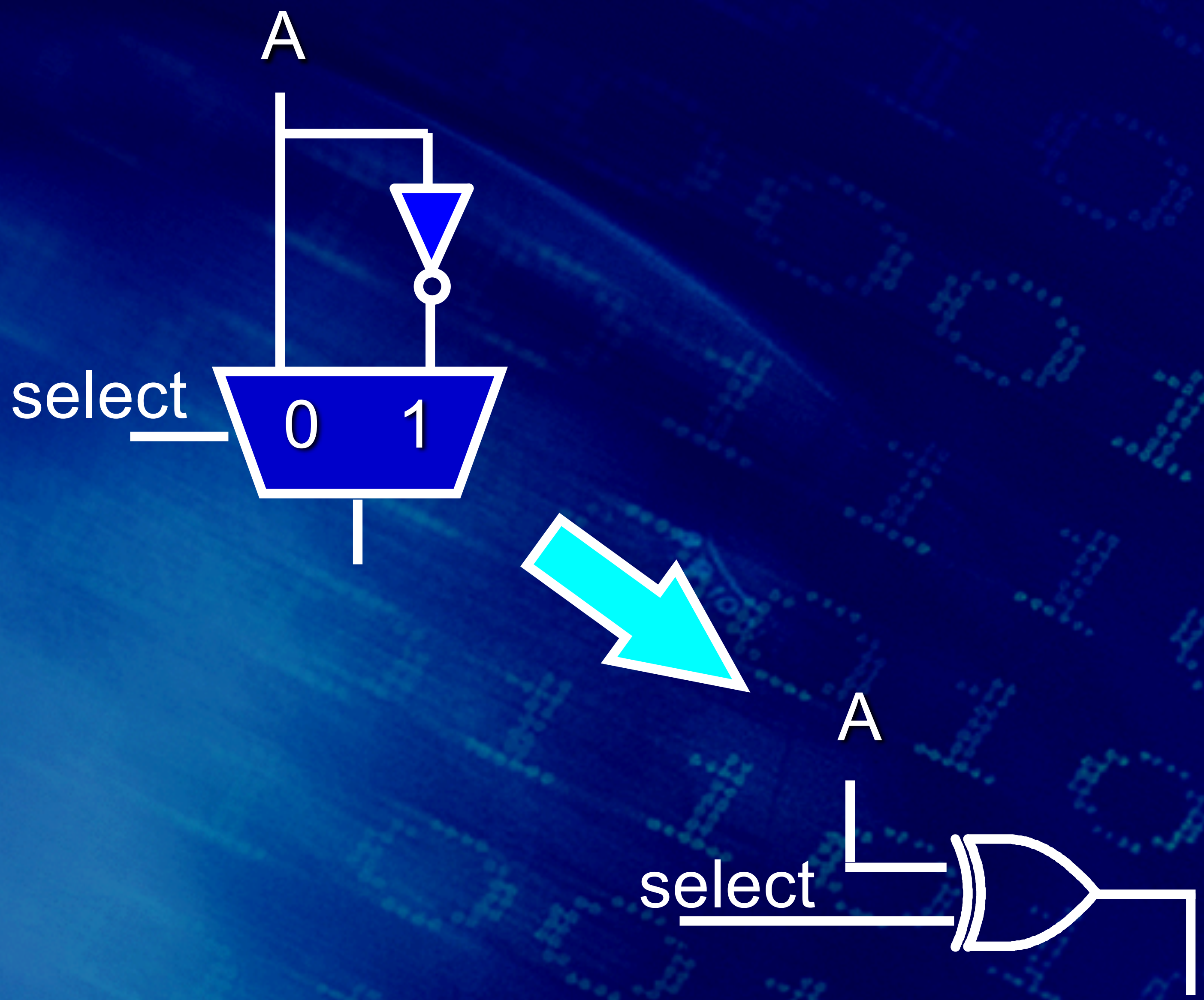

A
0
select
1
0


A
0
select
1
0
A
select


1
A
select
1
0


1
A
select
1
0
A
select


A
select
0
1


A
select
0
1
A
select

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Multiplier
- Adder-Subtractor
- Encoder
- Decoder
- ROM
- Comparator
- ALU
- I/O interfaces
- 1's counter

Demultiplexer

A demultiplexer is a combinational block capable of acting as the "reverse" of a multiplexer, thus forcing one of its output to the current value of its inputs, according to the values of some control signals.

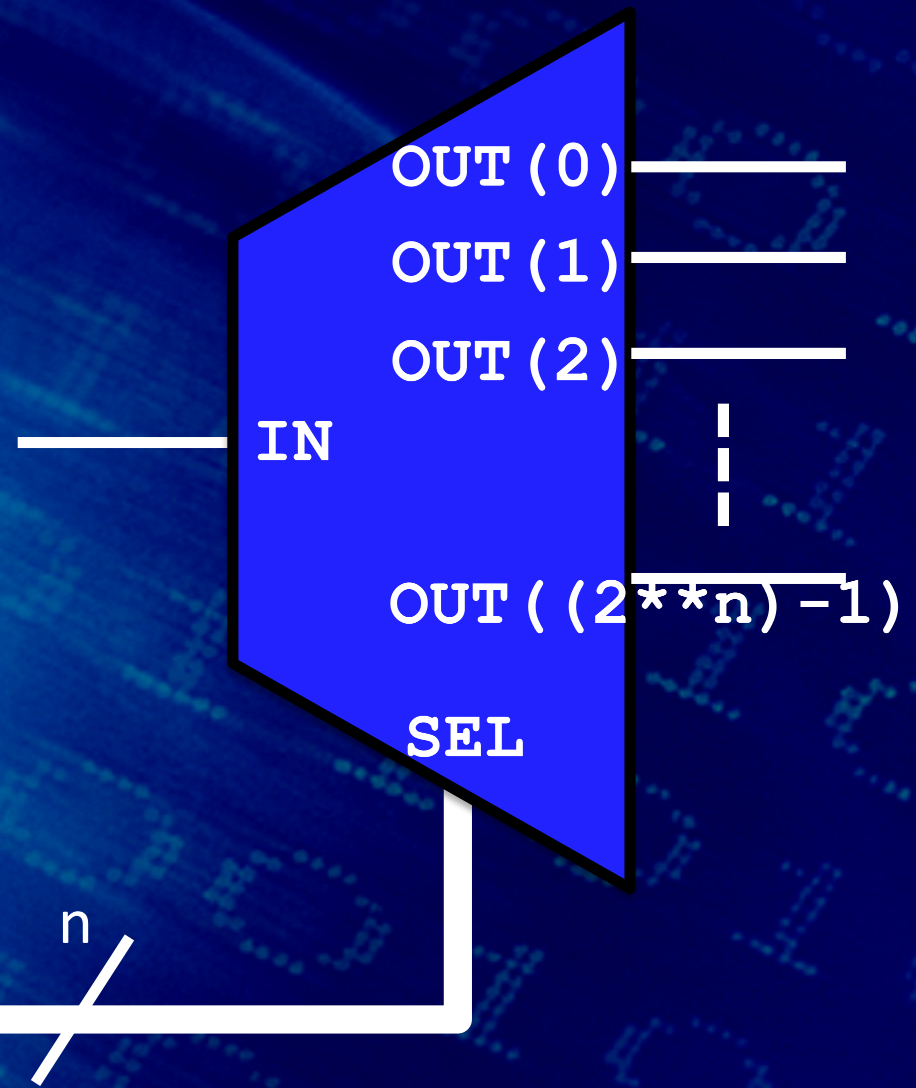
The remaining unselected outputs get a value that is implementation dependent.

Demultiplexer

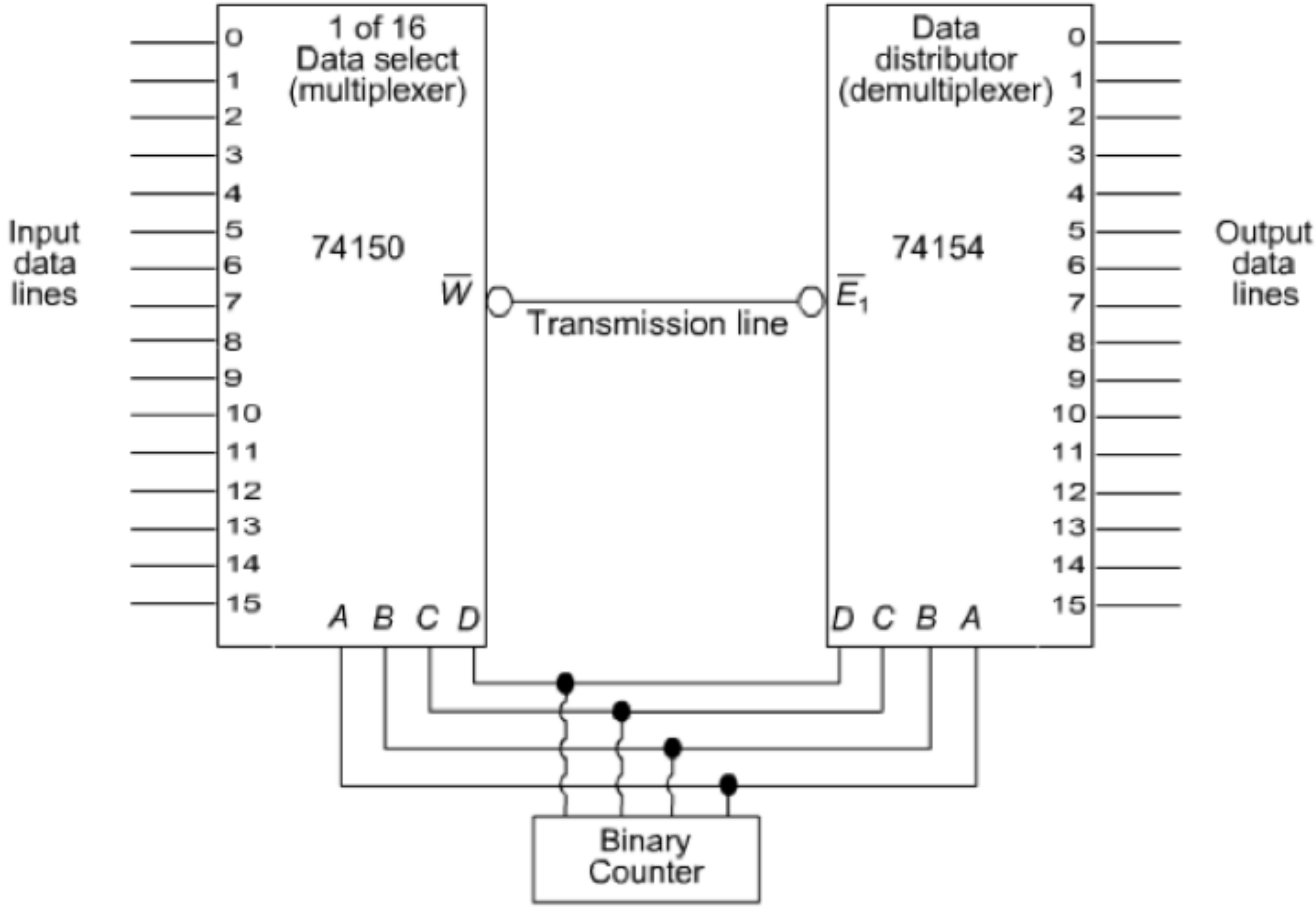
A demultiplexer has:

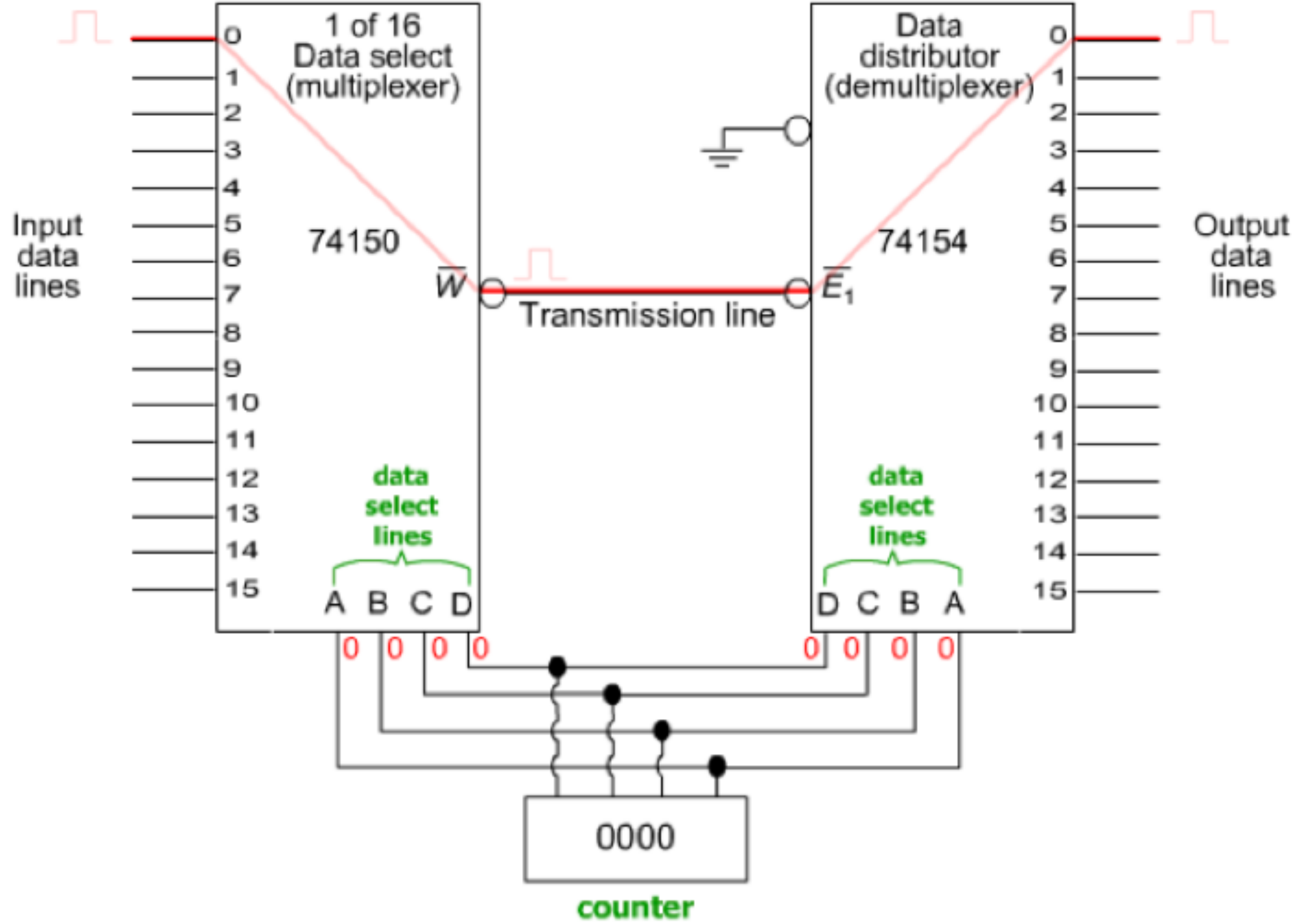
- ***2^k*** **data outputs, labeled from** ***2^k-1*** **to** ***0***
- ***k*** **control inputs**
- ***1*** **data input**

The output labeled 2^{i}, where i ($0 \le i \le k\text{-}1$) is the binary number present on the control inputs, gets the value present on the input


OUT(0)
OUT(1)
OUT(2)
IN
OUT((2**n)-1)
SEL
n

Example of use


Input data lines
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
1 of 16 Data select (multiplexer)
74150
$\overline{W}$
A B C D
Transmission line
Data distributor (demultiplexer)
74154
$\overline{E}_1$
D C B A
Output data lines
Binary Counter



1 of 16
Data select
(multiplexer)
74150
Input
data
lines
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
$\overline{W}$
data
select
lines
A B C D
0 0 0 0
Transmission line
Data
distributor
(demultiplexer)
74154
$\overline{E}_1$
Output
data
lines
data
select
lines
D C B A
0 0 0 0
0000
counter

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Multiplier
- Adder-Subtractor
- Encoder
- Decoder
- ROM
- Comparator
- ALU
- 1's counter
- I/O interfaces

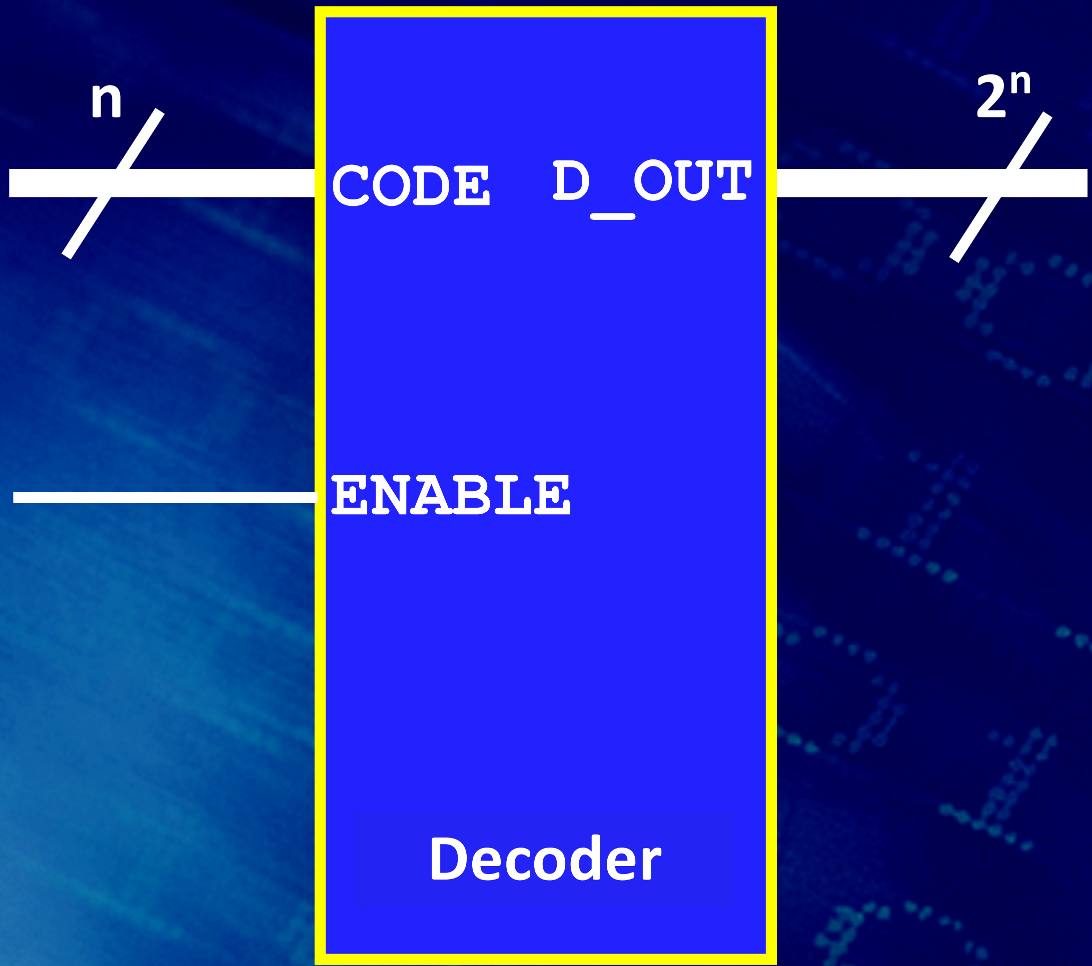
Decoder

A *decoder* $k \rightarrow 2^k$ has:

- *1 n-bit* data input
- 2^n outputs, labeled from *0* to 2^n*-1*
- *1* enable.

When enabled, just the output labeled *j* is asserted, *j* being the value present on the data input.

When disabled, no output is asserted.


n
2^n
CODE
D_OUT
ENABLE
Decoder

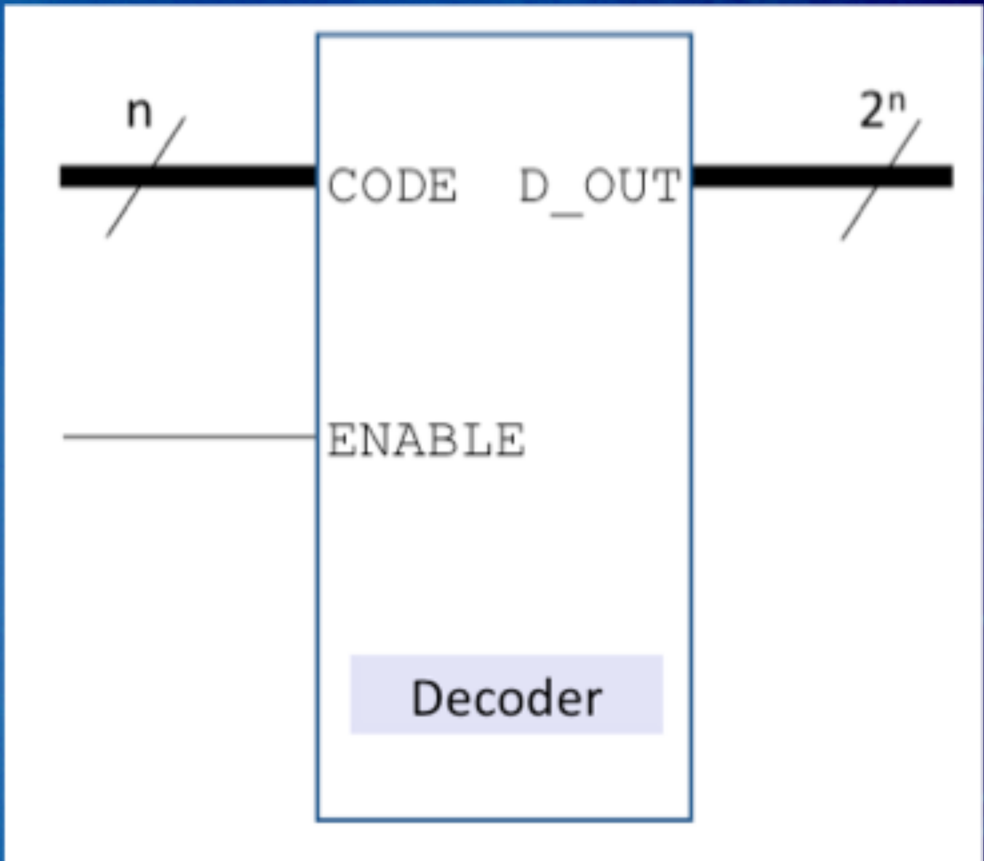
VHDL description

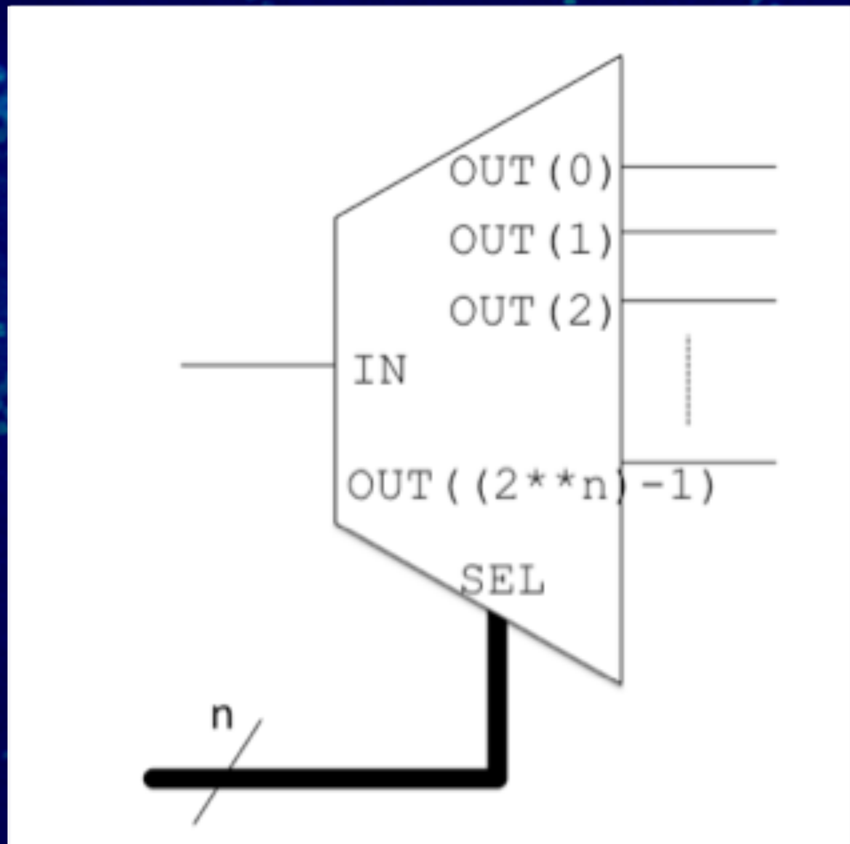
```vhdl
library  ieee;
use ieee.std_logic_1164.all;
use  ieee.std_logic_arith.all;
entity decoder is
generic( n: integer := 3);
port(
   CODE : in  unsigned(n-1 downto 0);
   ENABLE : std_logic ;
   DOUT : out std_logic_vector( (2**n)-1 downto 0)
);
end decoder;
```

```
architecture beh of  decoder is
begin
P1: process( CODE, ENABLE )
   variable sel: integer range 0 to 2**n -1;
   begin
      sel:=conv_integer(CODE);
      for k  in DOUT 'range loop
         if (ENABLE='1') and (k = sel)  then
            DOUT (k) <= '1';
         else
            DOUT (k) <= '0';
         end if;
      end loop;
   end process;
end beh;
```

Decoders vs Demultiplexers

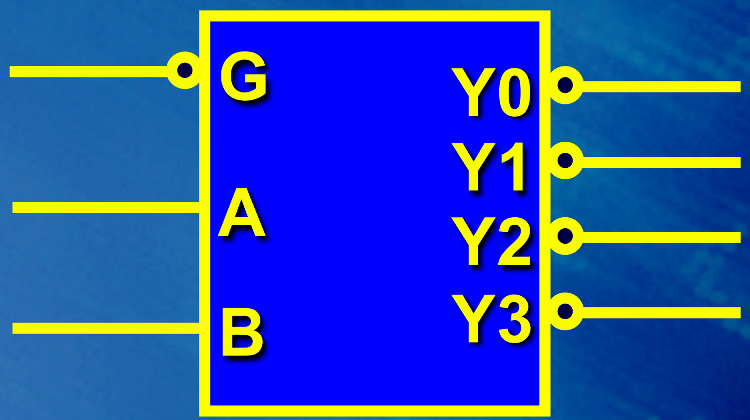
- **Decoders and Demultiplexers are conceptually interchangeable, simply swapping data & control signals.**





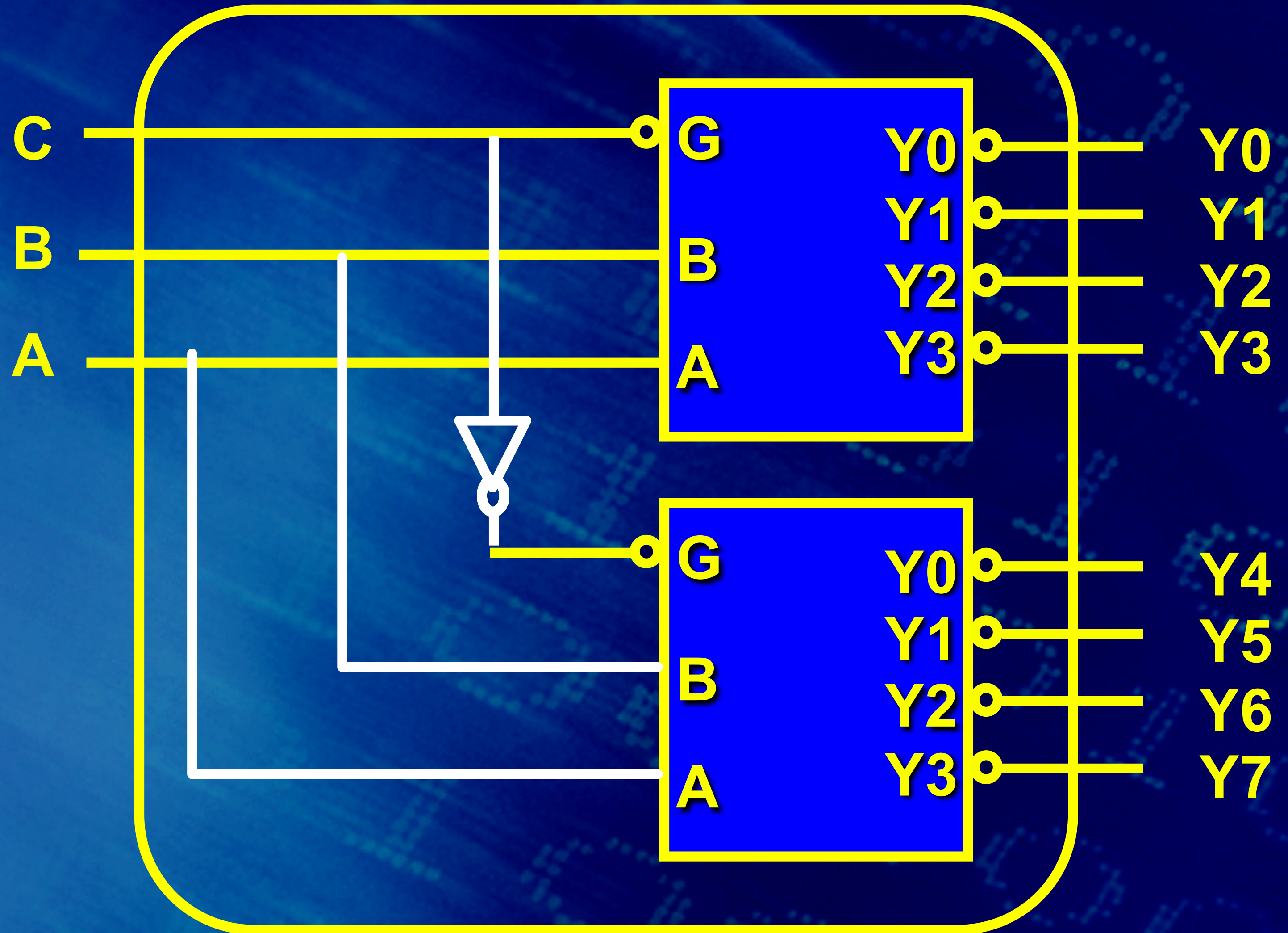
Exercise

Design a 3 → 8 decoder, resorting to the following 2 → 4 decoder:



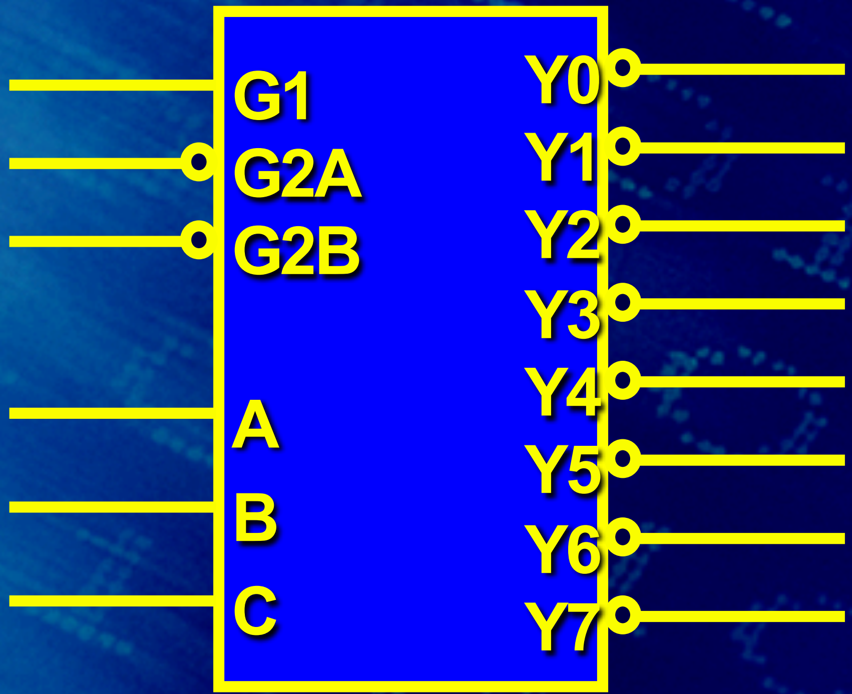
Inputs			Outputs			
G	B	A	Y3	Y2	Y1	Y0
1	-	-	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

Solution: decoder 3 → 8

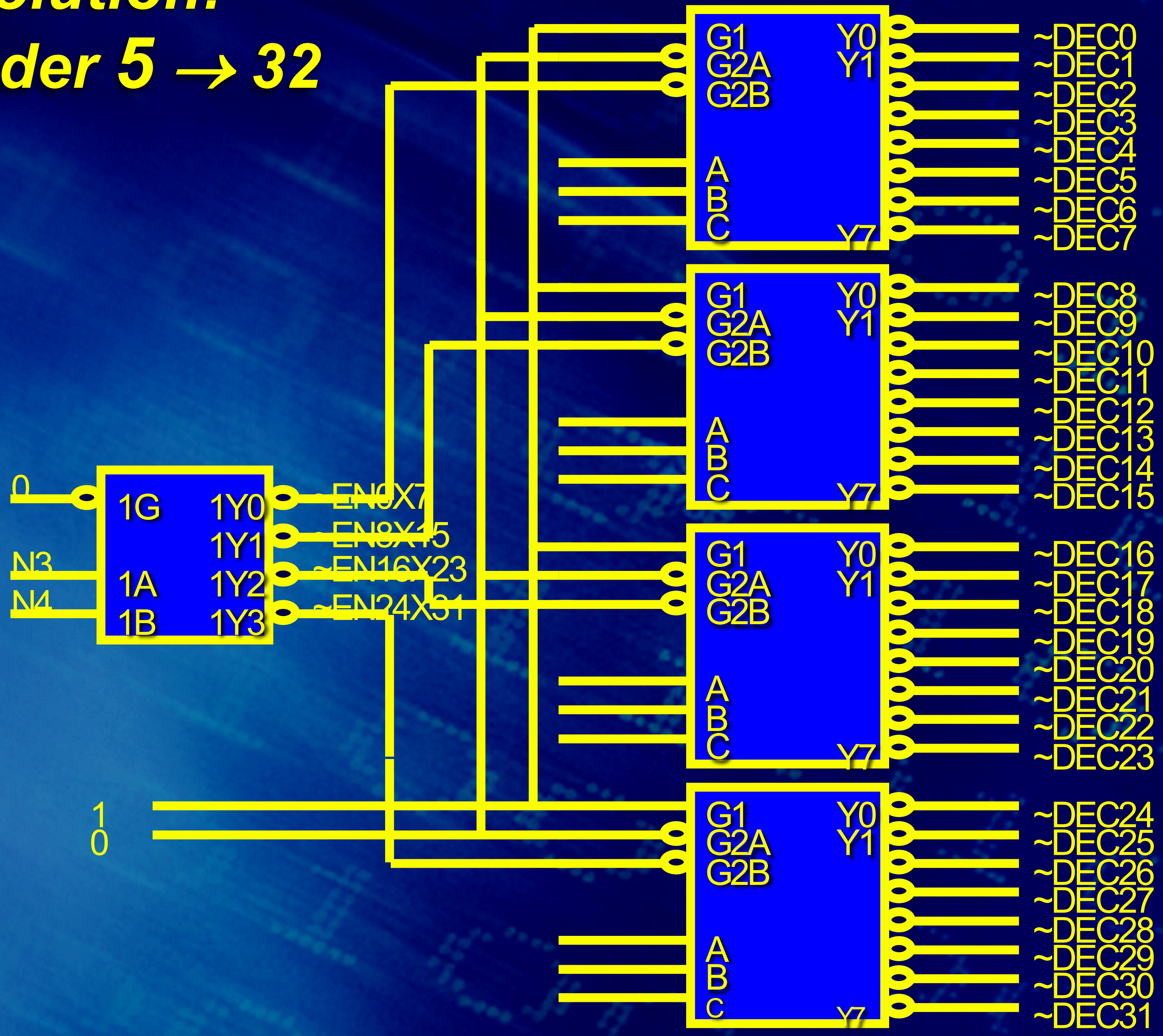


Exercise

Design a 5 → 32 decoder, resorting to the following 3 → 8 decoder:



Solution: decoder 5 → 32



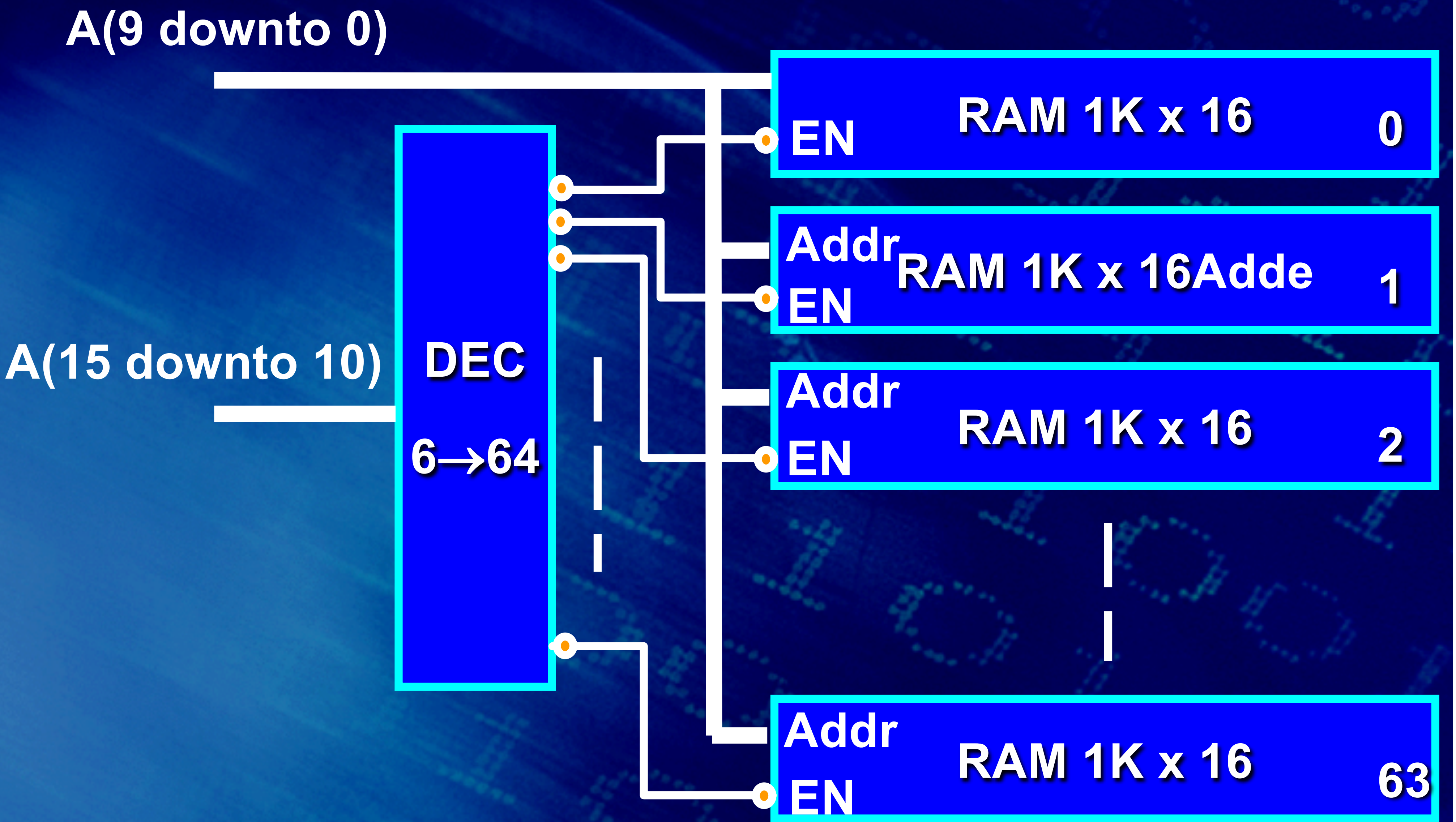
Example of use

- Design a 64K x 16 word-oriented RAM resorting to 1K x 16 RAM word-oriented modules.

Solution

- We need 64 modules of RAM
- To address 64 K words we need 64 K different addresses, so we need 16 bit.
- Of the 16 bits of the address:
  - The 6 most significant ones can be used to select 1 out of the 64 modules
  - The remaining 10 bits can be used to select the target word inside the selected module (1 K words –> 10 address bit)

Solution



A(9 downto 0)
A(15 downto 10)
DEC
6→64
EN
RAM 1K x 16
0
Addr
RAM 1K x 16Adde
1
EN
Addr
RAM 1K x 16
2
EN
Addr
RAM 1K x 16
63
EN

Design principle

- All the modules are connected to the data bus
- The least significant bits of the address bus are connected to the address inputs of all the modules and are used to select the word inside
- The most significant bits of the address bus are connected to a decoder, charged of properly enabling the memory modules.

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Multiplier
- Adder-Subtractor
- Encoder
- Decoder
- ROM
- Comparator
- ALU
- 1's counter
- I/O interfaces

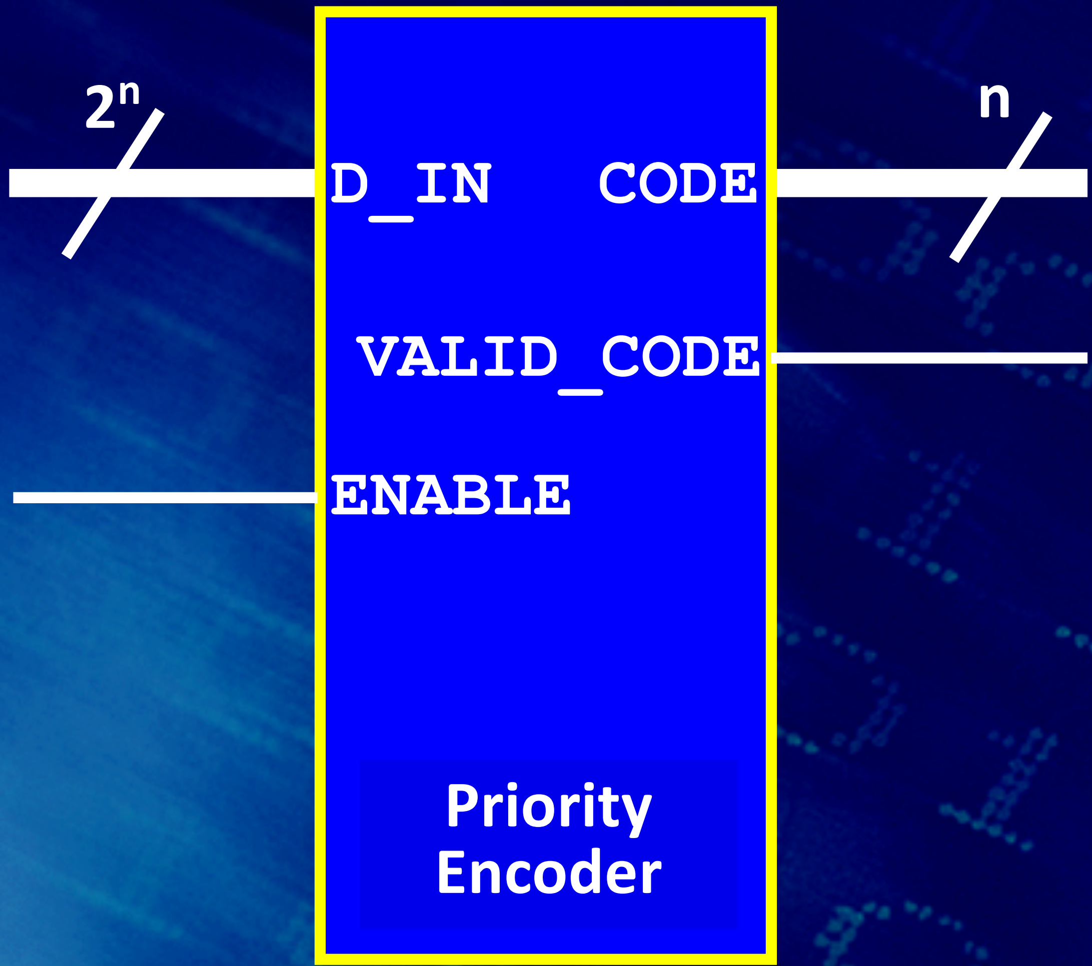
Priority Encoder

A ***priority encoder*** has:

- *2^k* inputs, labeled from *0* to *2^k-1* (each input is assigned a fixed priority)
- *k* outputs
- *1* “data valid” output
- *1* enable.

Priority Encoder (2)

- When enabled:
  – the outputs get a value corresponding to the binary encoding of *j* , *j* being the label of the asserted input having the highest priority
  – the “data valid” output is asserted.
- When disabled or when no input is asserted:
  – the “data valid” output is not asserted.


2n
n
D_IN
CODE
VALID_CODE
ENABLE
Priority Encoder

VHDL description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity priority_encoder is
generic(N: integer := 3);
port(M : in  std_logic_vector( (2**N)-1 downto 0);
      ENABLE : in std_logic ;
      VALID : out std_logic;
      CODE : out unsigned(N-1 downto 0));
end priority_encoder;
```

```
architecture beh of  priority_encoder is
begin
 P1: process( M, ENABLE )
   variable  k : integer range 0 to (2**N);
   variable found : boolean;
      begin
     k:= (2**N);    -- while statement's initial conditions
     found := false ; -- while statement's exit condition
```



```
while ((k > 0) and (not found))  loop
   k := k-1;
   if  M(k) = '1'  then
      found := true ;
   end if;
end loop;
CODE <=  conv_unsigned( k, N ) ;
if  found and ENABLE='1' then
   VALID <= '1';
else
   VALID <= '0';
end if;
end process;
end beh;
```

Basic combinational blocks

- Multiplexer
- Demultiplexer
- Multiplier
- Adder-Subtractor
- Encoder
- Decoder
- Comparator
- ALU
- ROM
- 1's counter
- I/O interfaces

Rel. 14.06.2018

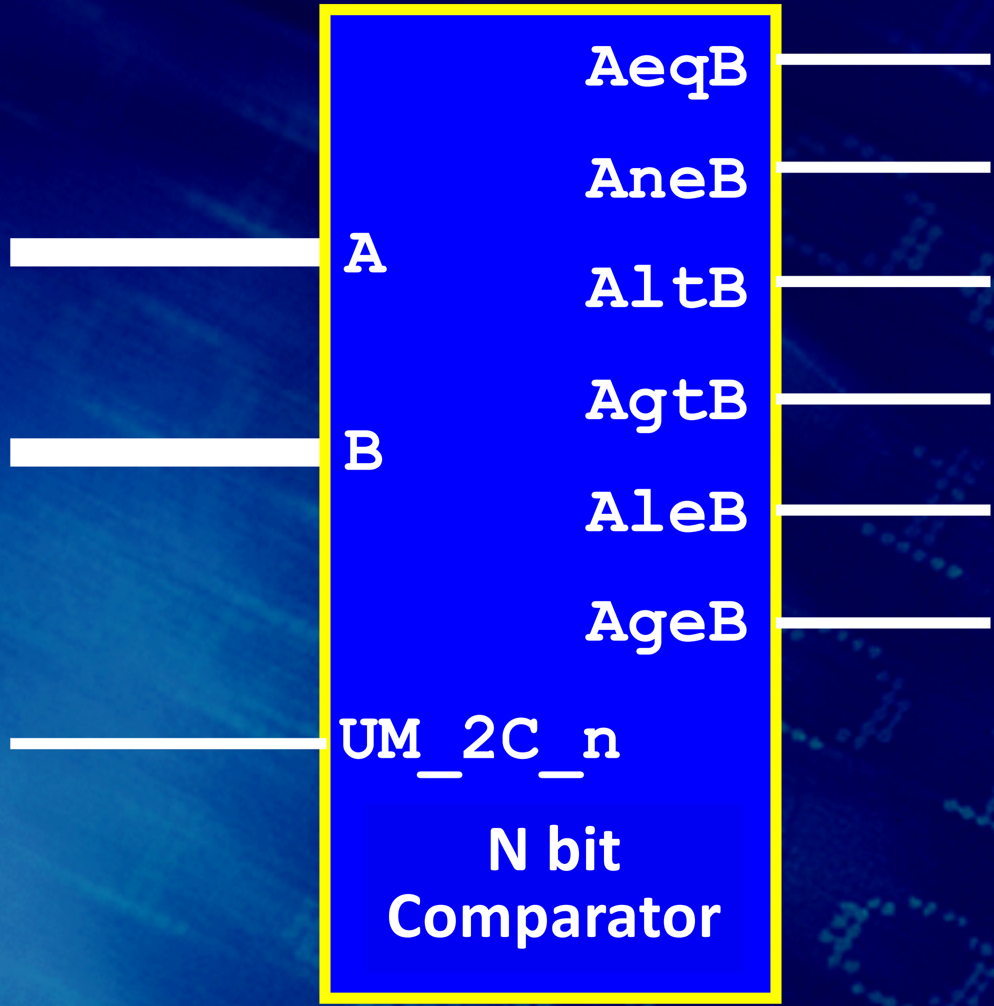
Comparator

A comparator gets 2 n-bit binary numbers:

- `A(n-1 downto 0)`
- `B(n-1 downto 0)`

and provides in output the result of the comparison between `A` and `B`.

A control input `UM_2C_n` specifies whether the input operands are unsigned or signed, respectively.


AeqB
AneB
A
AltB
AgtB
B
AleB
AgeB
UM_2C_n
N bit
Comparator

Comparator for 2C numbers

We have to implemented a comparator for numbers represented according to the 2’s complement notation, resorting to comparators for absolute numbers.

Rel. 14.06.2018

Solution

One can observe that:

$$A \ ? \ B \Leftrightarrow (A + C) \ ? \ (B + C)$$

where "?" is the target comparison operator.

Selecting a proper value for C can guarantee that the comparison A ? B is always performed between positive numbers.

Solution

When A and B are expressed resorting to n bits, since:

$$-2^{n-1} \leq A \leq 2^{n-1} - 1$$

one can choice:

$$C = 2^{n-1}$$

Rel. 14.06.2018

Solution

When A and B are expressed resorting to *n* bits, since:

$$-2^{n-1} \leq A \leq 2^{n-1} - 1$$

one can choice:

$$C = 2^{n-1}$$

Adding $C = 2^{n-1}$ is equivalent to complementing the most significant bit.

Thus:

A ? B ⇔ (A(n-1)' & A(n-2 downto 0)) ?
(B(n-1)' & B(n-2 downto 0))

How can we prove it intuitively?

Example

2	010	-2	110
3	011	-3	101

Example

2	010	-2	110
3	011	-3	101

2	3	<
2	-3	>
-2	3	<
-2	-3	>

Example

2	010	-2	110
3	011	-3	101

2	3	<	010	011
2	-3	>	010	101
-2	3	<	110	011
-2	-3	>	110	101

Example

2	010	-2	110
3	011	-3	101

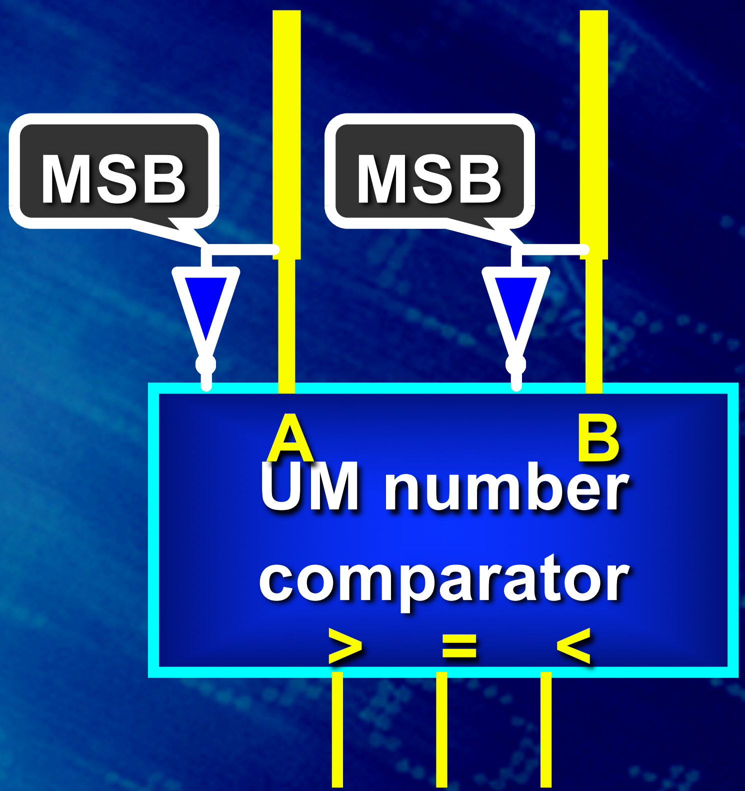
2	3	<	010	011	110	111
2	-3	>	010	101	110	001
-2	3	<	110	011	010	111
-2	-3	>	110	101	010	001

Example

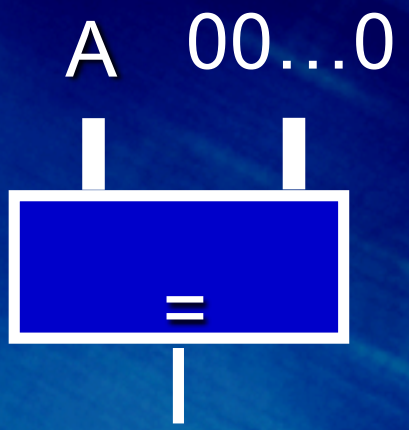
2	010		-2	110				
3	011		-3	101				

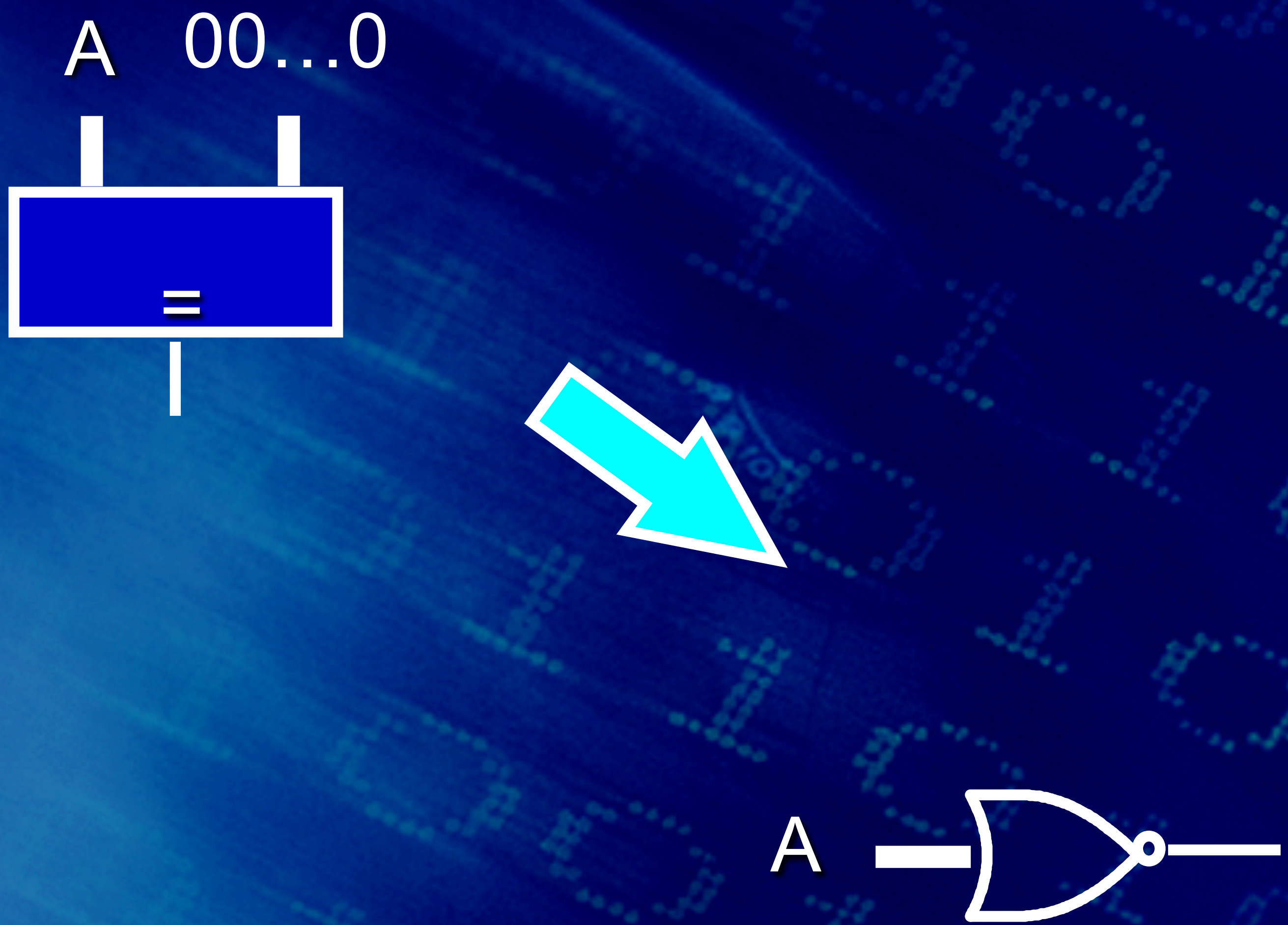
2	3	<	010	011	110	111	<
2	-3	>	010	101	110	001	>
-2	3	<	110	011	010	111	<
-2	-3	>	110	101	010	001	>

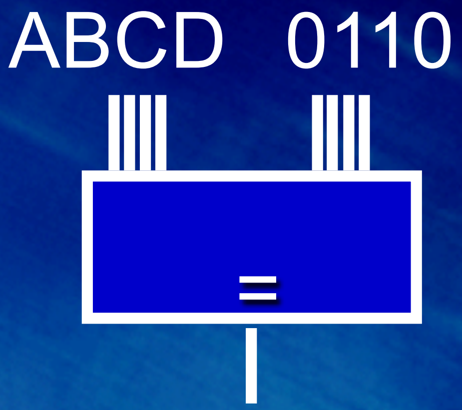
Solution

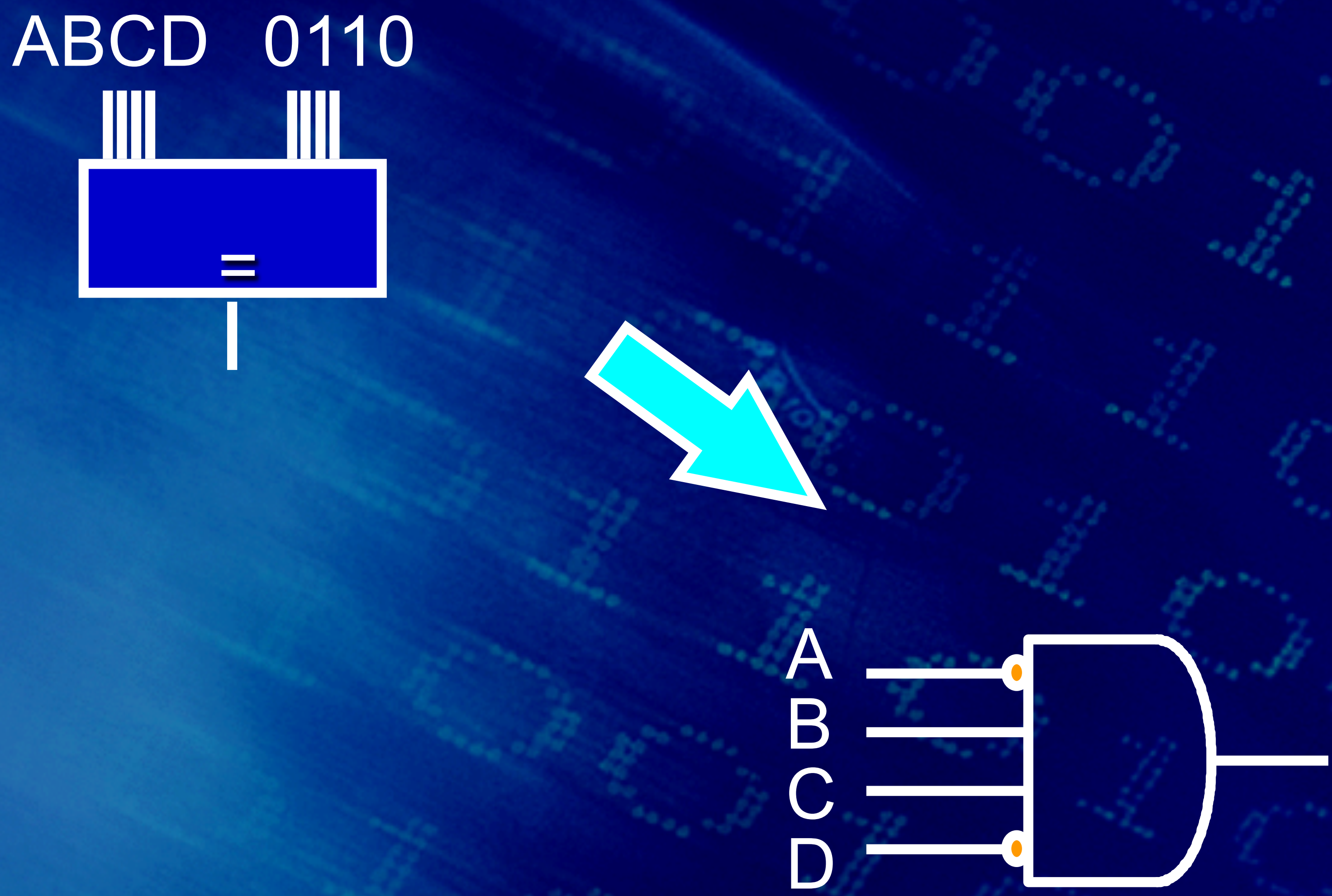


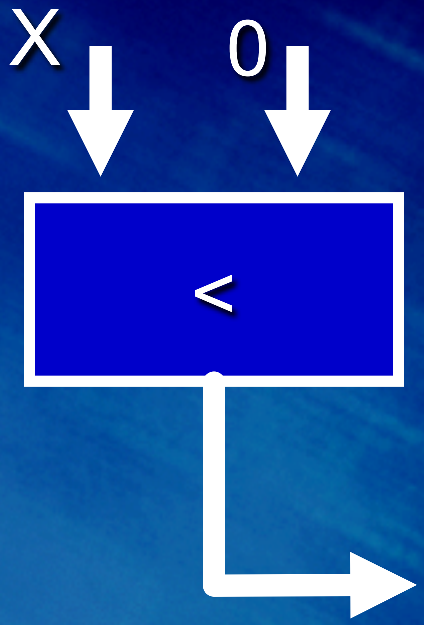
Some optimizations on comparators

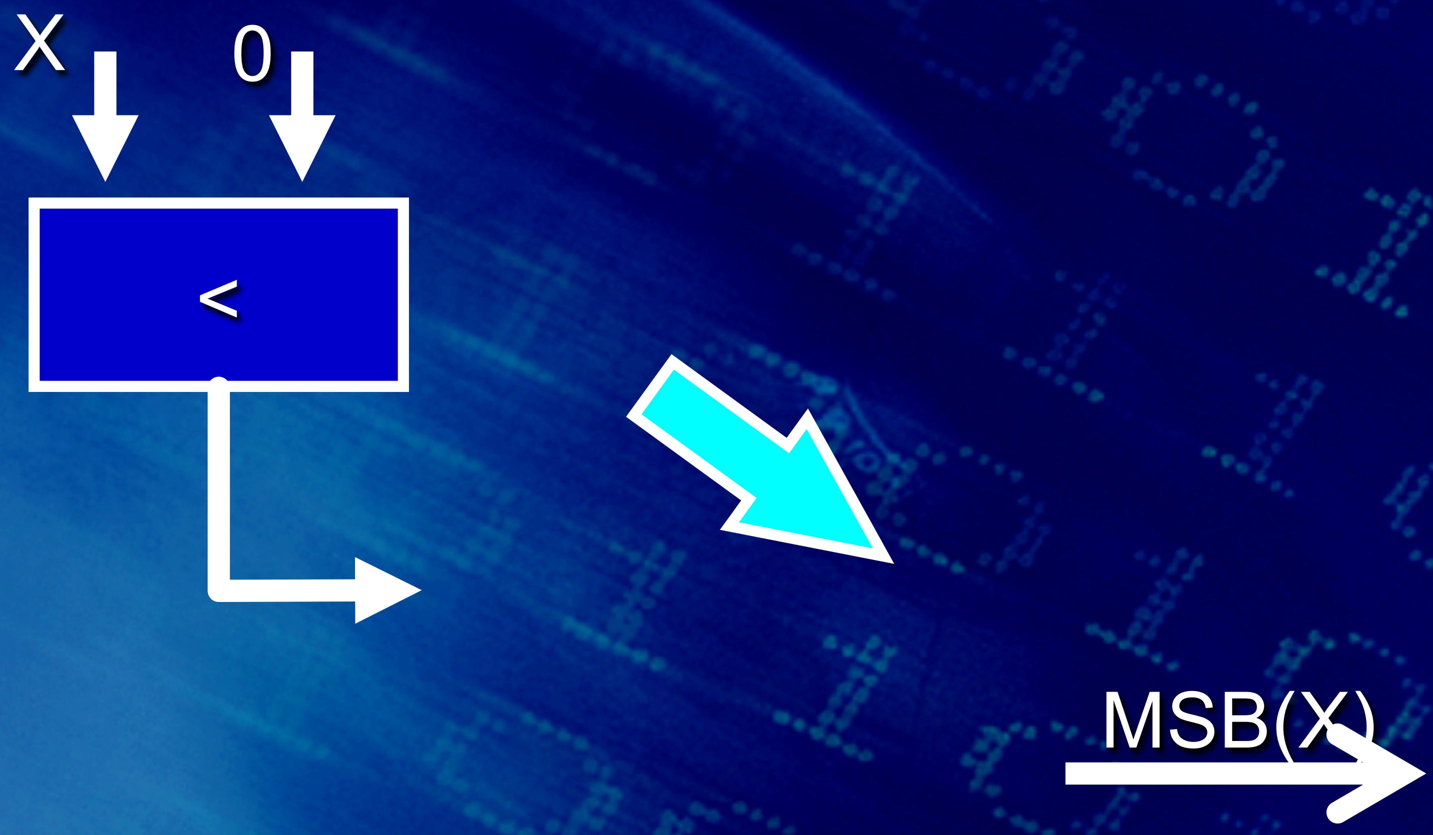

A
00…0
=


A
00…0
=
A


ABCD
0110
=


ABCD
0110
=
A
B
C
D


X
0
<


X
0
<
MSB(X)

Basic combinational blocks

Multiplexer

Demultiplexer

Multiplier

Adder-Subtractor

Encoder

Decoder

Comparator

ROM

ALU

I/O interfaces

1's counter

Counter of 1's occurrences

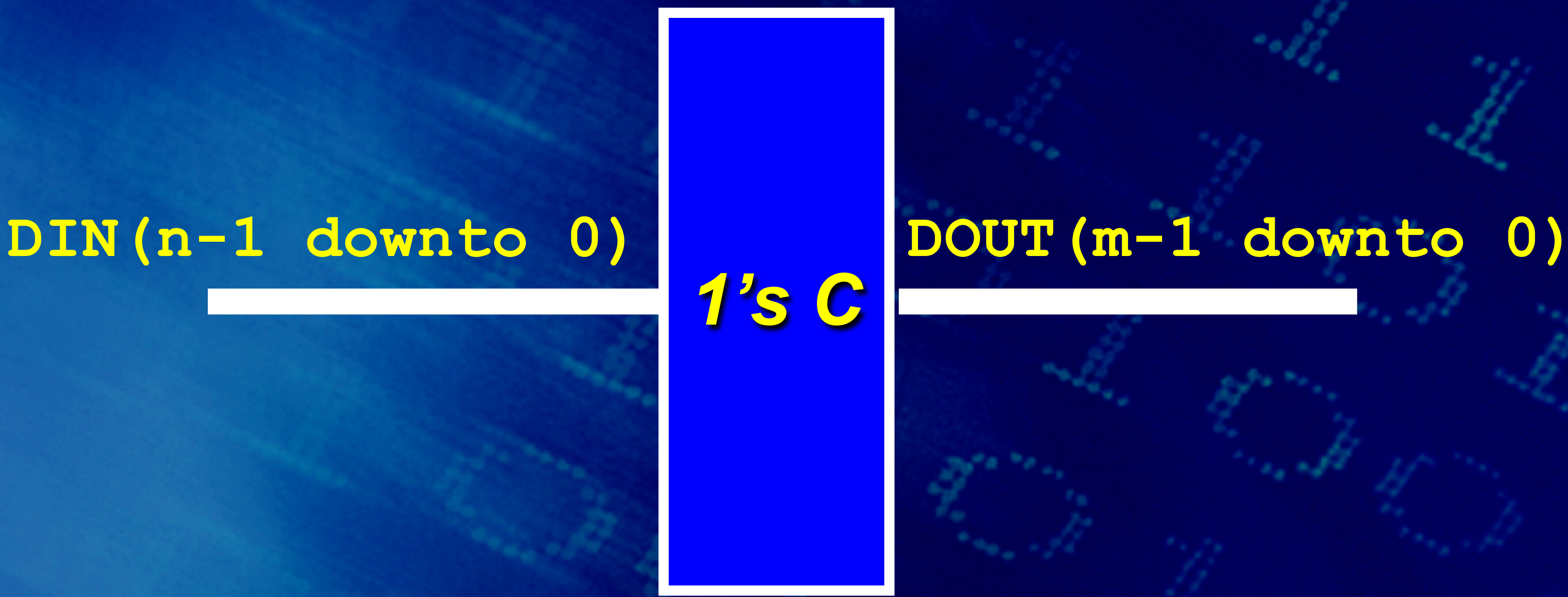
A *counter of 1's occurrences* has:

- *1 n-bit* data input
- *1 m-bit* data output, being $m = \lceil \log_2 (n+1) \rceil$

The device counts the number of occurrences of a '1' on its inputs and provides it on its outputs.

Counter of 1's occurrences (2)

Symbol



VHDL description

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;

entity count_1 is
generic (in_size: integer:=7;
         out_size: integer:=3);
port(
 din: in std_logic_vector(in_size -1 downto 0);
 dout: out std_logic_vector(out_size -1 downto 0)
);
end count_1;
```

```
architecture beh of count_1 is
begin
process (din)
   variable counter:integer range 0 to in_size -1;
begin
   assert (in_size =< 2**out_size)
     report “ERROR in sizes”   severity error;
   counter :=0;
   for i in in_size’range loop
     if din(i)='1' then
        counter := counter +1;
     end if;
   end loop;
   dout<= conv_std_logic_vector(counter, out_size);
end process;
end beh;
```

Basic combinational blocks

Multiplexer

Encoder

Demultiplexer

Adder-Subtractor

Decoder

Multiplier

ROM

Comparator

ALU

I/O interfaces

1's counter

ROM

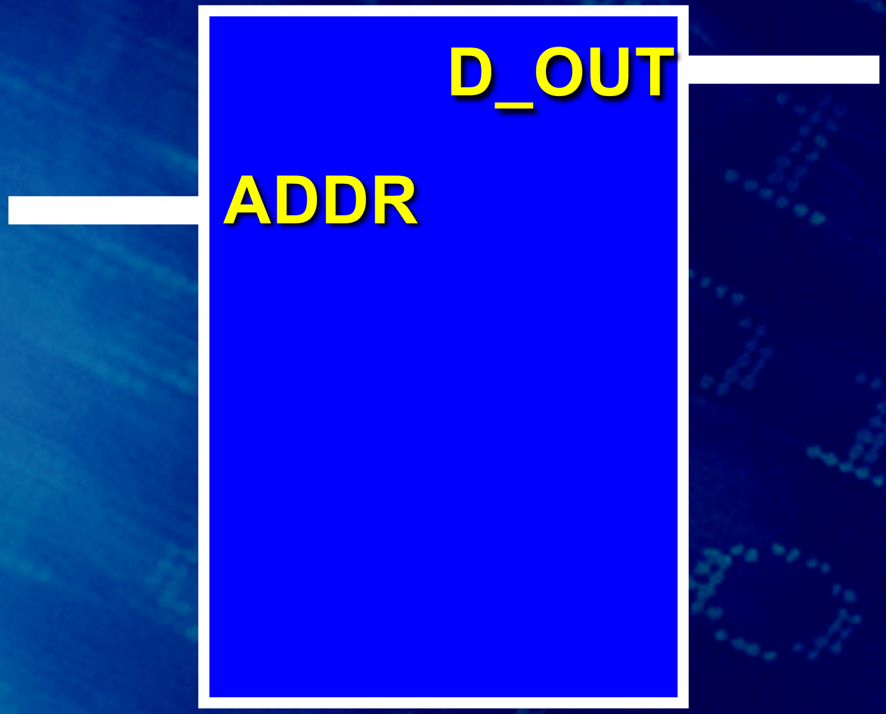
A $2^k \times n$ *ROM* (Read-Only Memory) has:

- k address inputs *ADDR*
- n data outputs *D_OUT*
- 2^k n-bit internal cells, storing permanent values.

The outputs gets the value stored in the j cell, j being the current value present on address inputs.

ROM (2)

Symbol



VHDL description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ROM is
generic (k: integer:=3; n: integer:=4);
port(
        ADDR :  in  unsigned(k-1 downto 0);
        D_OUT:  out  std_logic_vector(n-1 downto 0);
end ROM ;
```

```
architecture beh of ROM is
   type memory is array (0 to 2**k-1) of

         std_logic_vetor(n-1 downto 0);
Begin
P1: process (ADDR)
 variable mem: memory;
   begin
   mem:= (0 => “1011”,
             1 to 3 => “1010”,
             4 => “0110”,
             others => “0000”);
   D_OUT<= mem( conv_integer(ADDR));
   end process;
end behav;
```

Basic combinational blocks

Multiplexer

Demultiplexer

Multiplier

Adder-Subtractor

Encoder

Decoder

ALU

Comparator

ROM

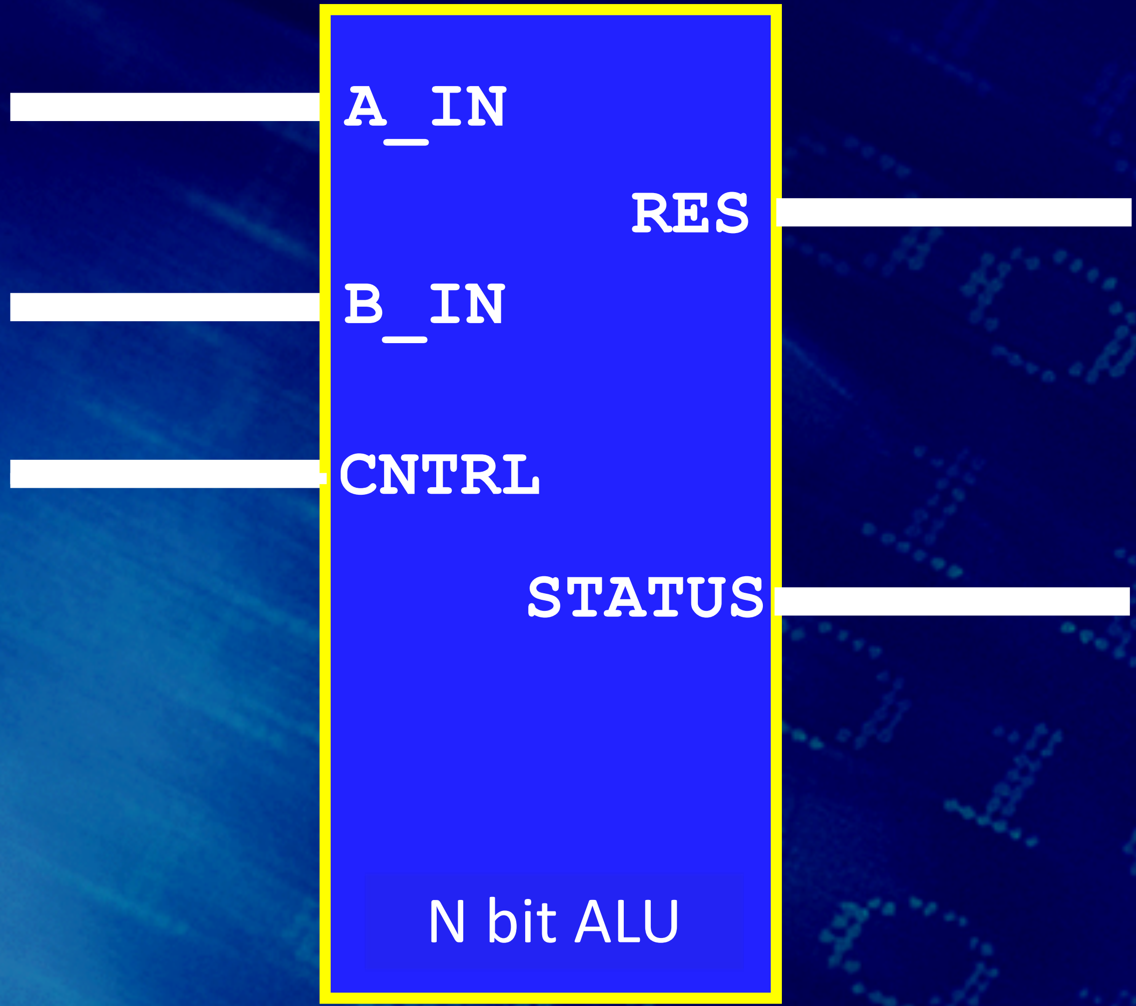
I/O interfaces

1's counter

ALU

An ***n-bit ALU*** **(Arithmetic Logic Unit) performs logic and/or arithmetic operations on 2 n-bit operands, under the control of ad-hoc control signals.**

Rel. 14.06.2018


A_IN
RES
B_IN
CNTRL
STATUS
N bit ALU

Functions implemented by the ALU '181

Input				Functions	
S3	S2	S1	S0	arithmetic (M=0)	logic (M=1)
0	0	0	0	F=A minus 1 plus CIN	F=A'
0	0	0	1	F=AB minus 1 plus CIN	F=A'+ B'
0	0	1	0	F=AB' minus 1 plus CIN	F=A'+ B
0	0	1	1	F=1111 plus CIN	F= 1111
0	1	0	0	F=A plus (A+B') plus CIN	F=A'B'
0	1	0	1	F=AB plus (A+B') plus CIN	F=B'
0	1	1	0	F=A minus B minus 1 plus CIN	F=A ⊕ B'
0	1	1	1	F=A+B' plus CIN	F=A+B'
1	0	0	0	F=A plus (A+B) plus CIN	F=A'B
1	0	0	1	F=A plus B plus CIN	F=A ⊕ B
1	0	1	0	F=AB' plus (A+B) plus CIN	F=B
1	0	1	1	F=A+B plus CIN	F=A+B
1	1	0	0	F=A plus A plus CIN	F=0000
1	1	0	1	F=AB plus A plus CIN	F=AB'
1	1	1	0	F=AB' plus A plus CIN	F=AB
1	1	1	1	F=A plus CIN	F=A

Basic combinational blocks

Multiplexer

Encoder

Demultiplexer

Adder-Subtractor

Multiplier

Decoder

Comparator

ALU

ROM

I/O interfaces

1's counter

Multiplier

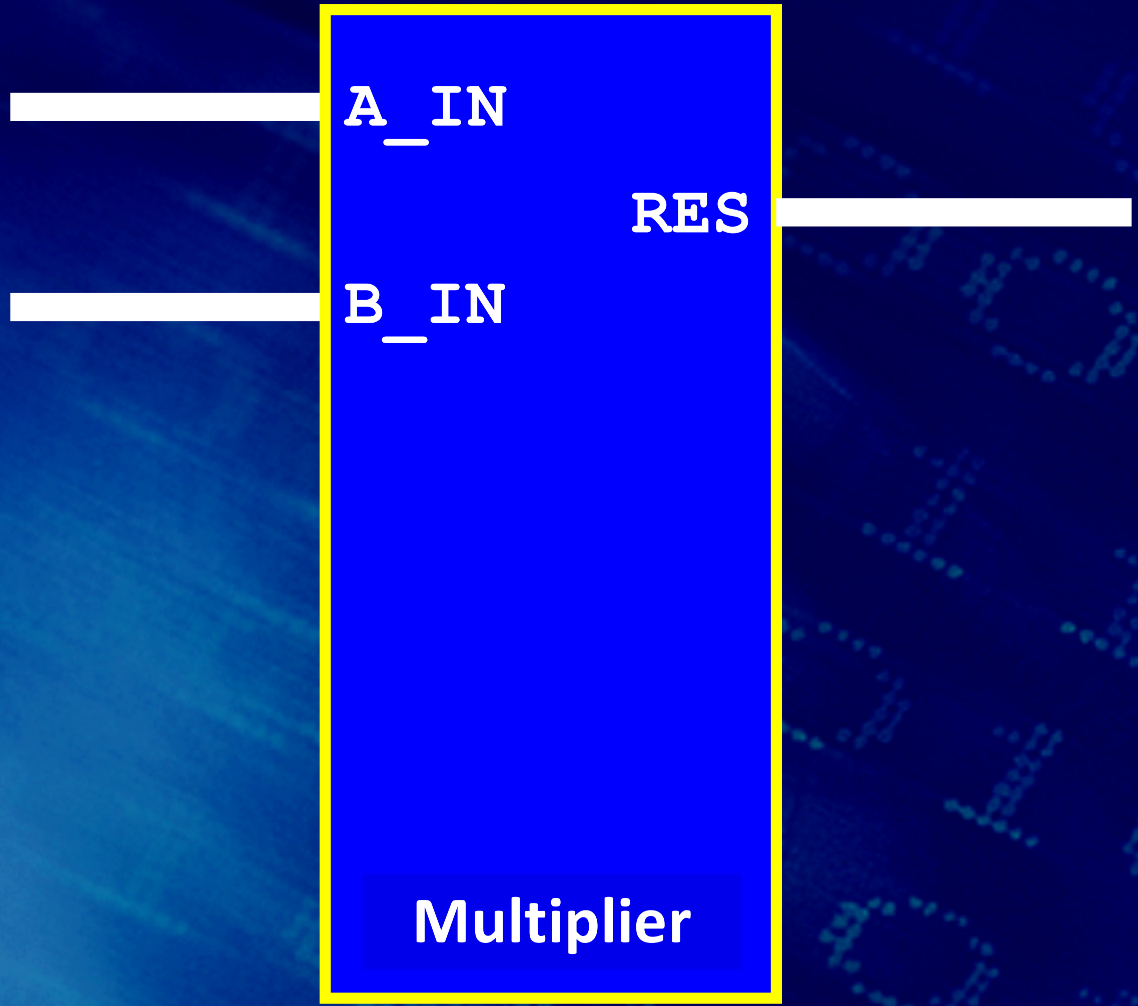
A *multiplier* has:

- *1 n-bits* data inputs, labeled from *n-1* to *0*
- *1 m-bits* data inputs, labeled from *m-1* to *0*
- *no* control signals
- *k-bits* data outputs, labeled from *k-1* to *0*

and provides in output the results of the product between the two input operands.

To avoid overflow condition:

- k=m+n-1 for Signed operands
- k=m+n for Unsigned operands


A_IN
RES
B_IN
Multiplier

Multipliers

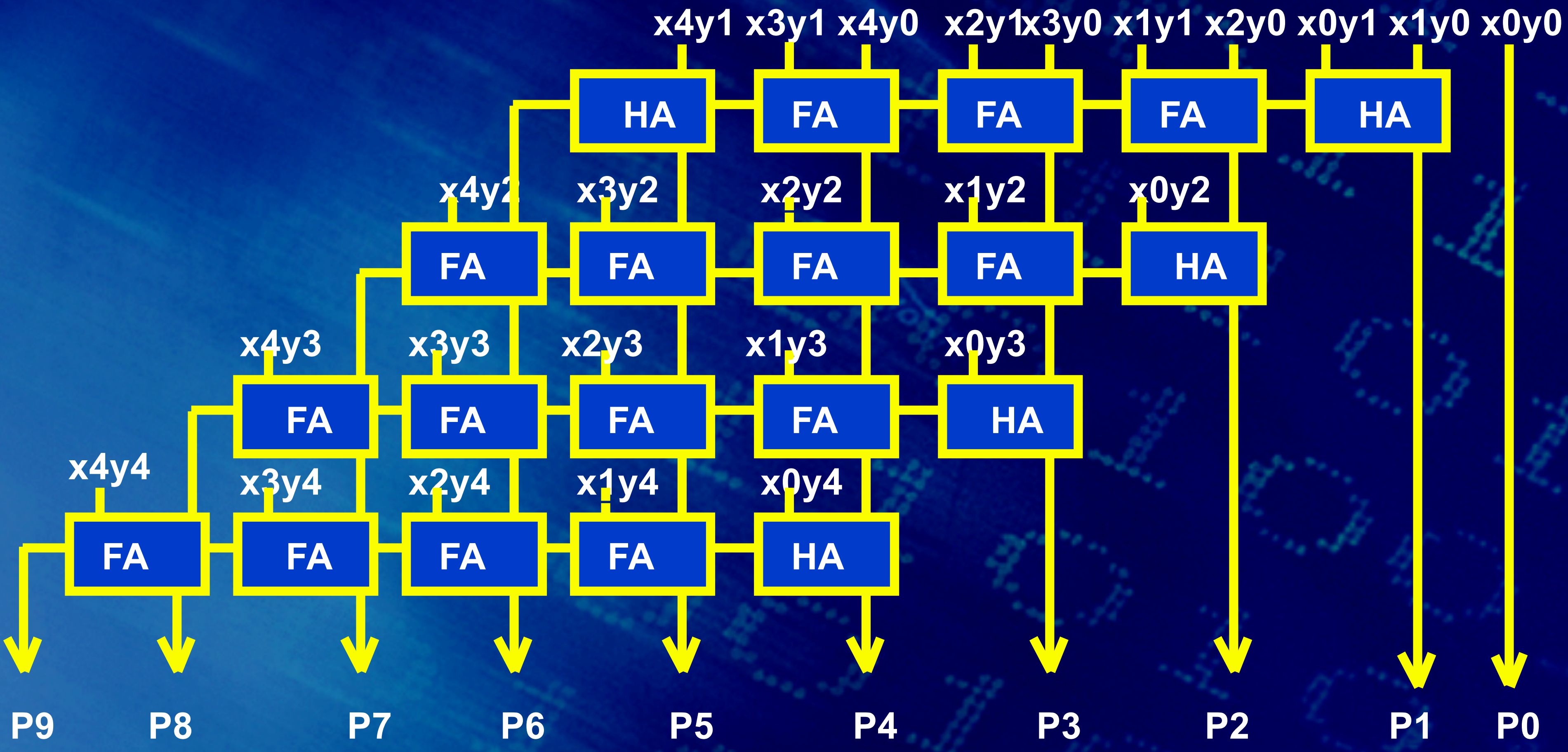
- Two architectures will be considered:
  - *carry propagate*
  - *based on elementary modules.*

Rel. 14.06.2018

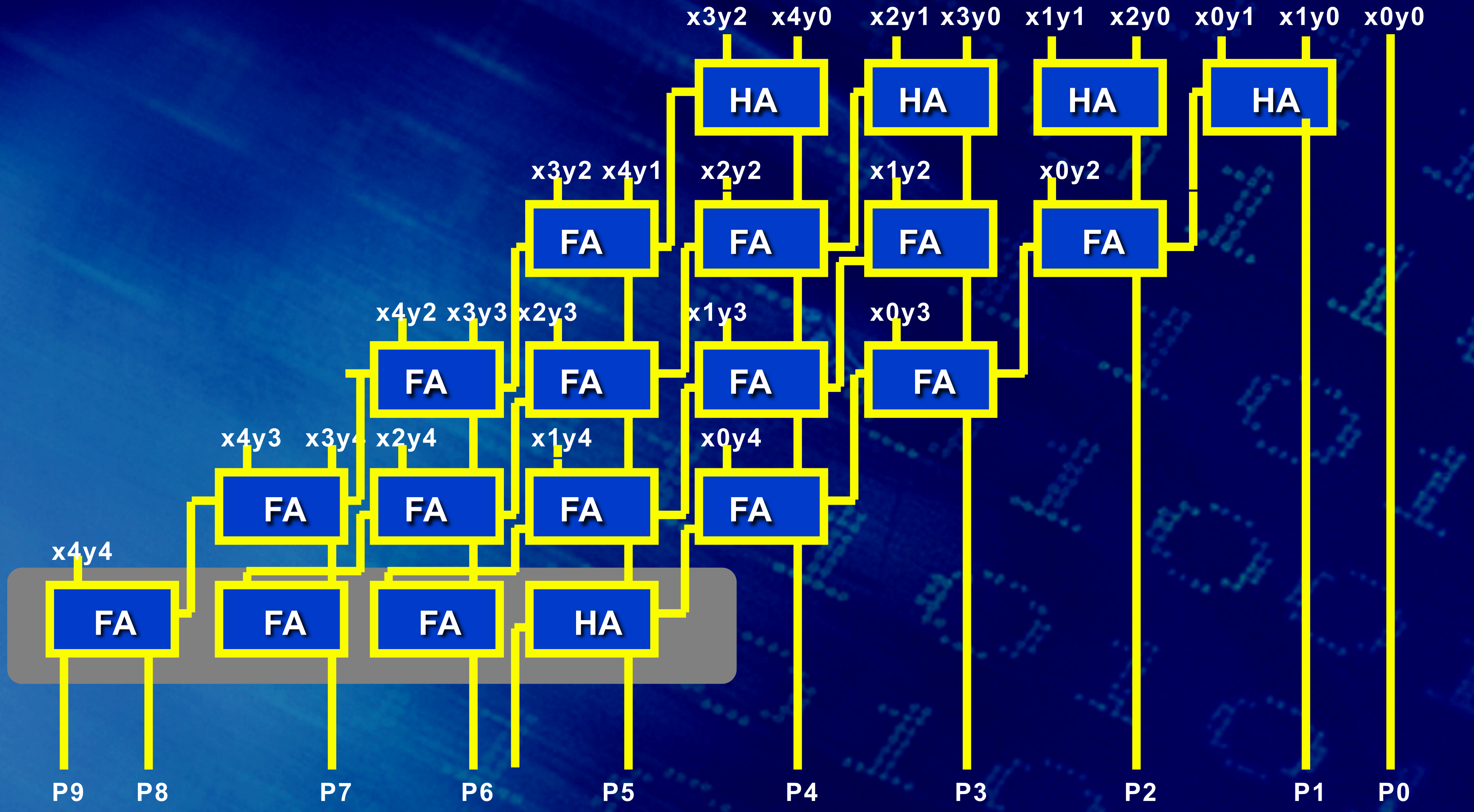
Product of two 5-bit binary numbers

					x4	x3	x2	x1	x0
					y4	y3	y2	y1	y0
					x4y0	x3y0	x2y0	x1y0	x0y0
				x4y1	x3y1	x2y1	x1y1	x0y1	
			x4y2	x3y2	x2y2	x1y2	x0y2		
		x4y3	x3y3	x2y3	x1y3	x0y3			
	x4y4	x3y4	x2y4	x1y4	x0y4				
p9	p8	p7	p6	p5	p4	p3	p2	p1	p0

5-bit carry-propagate multiplier

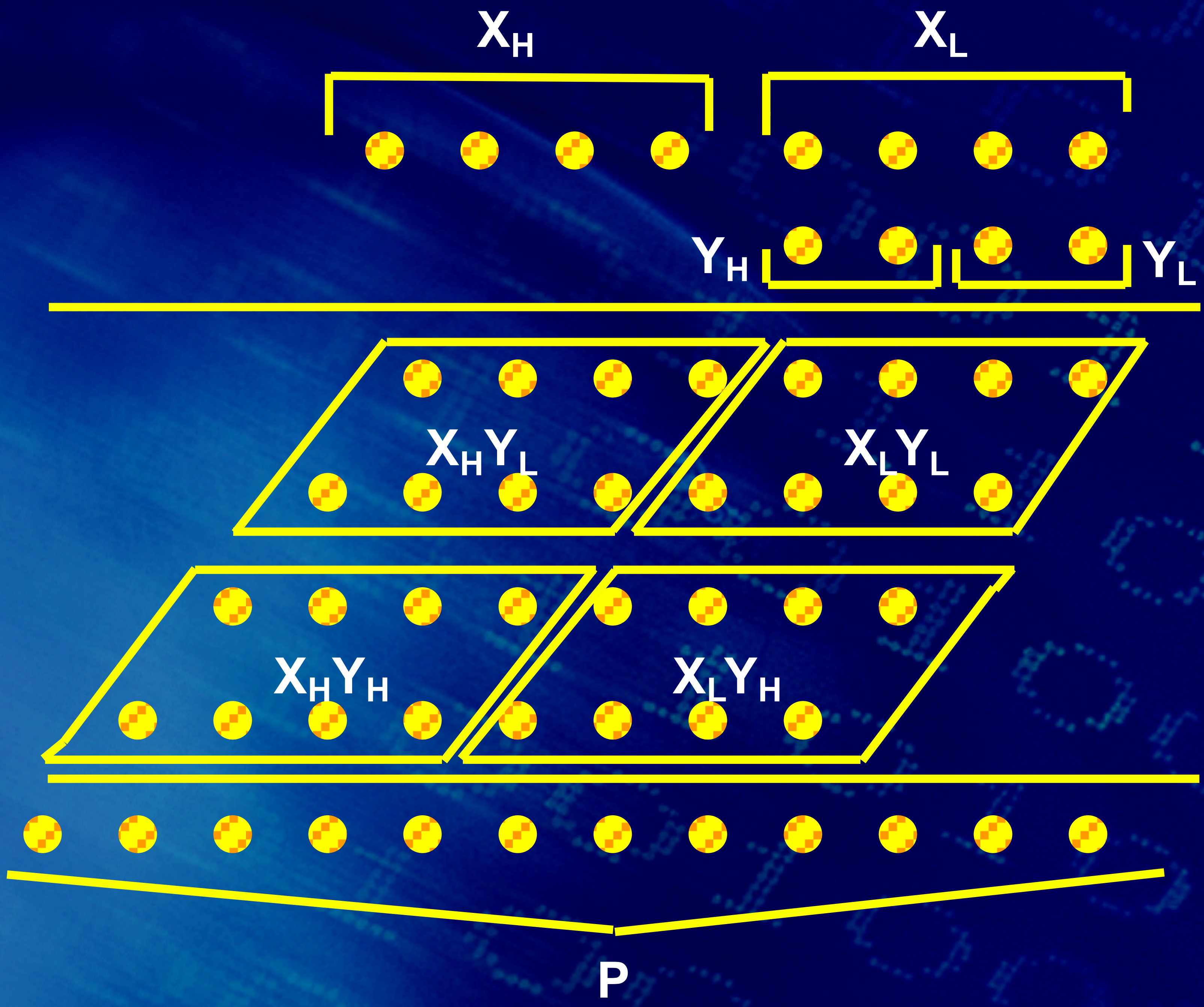


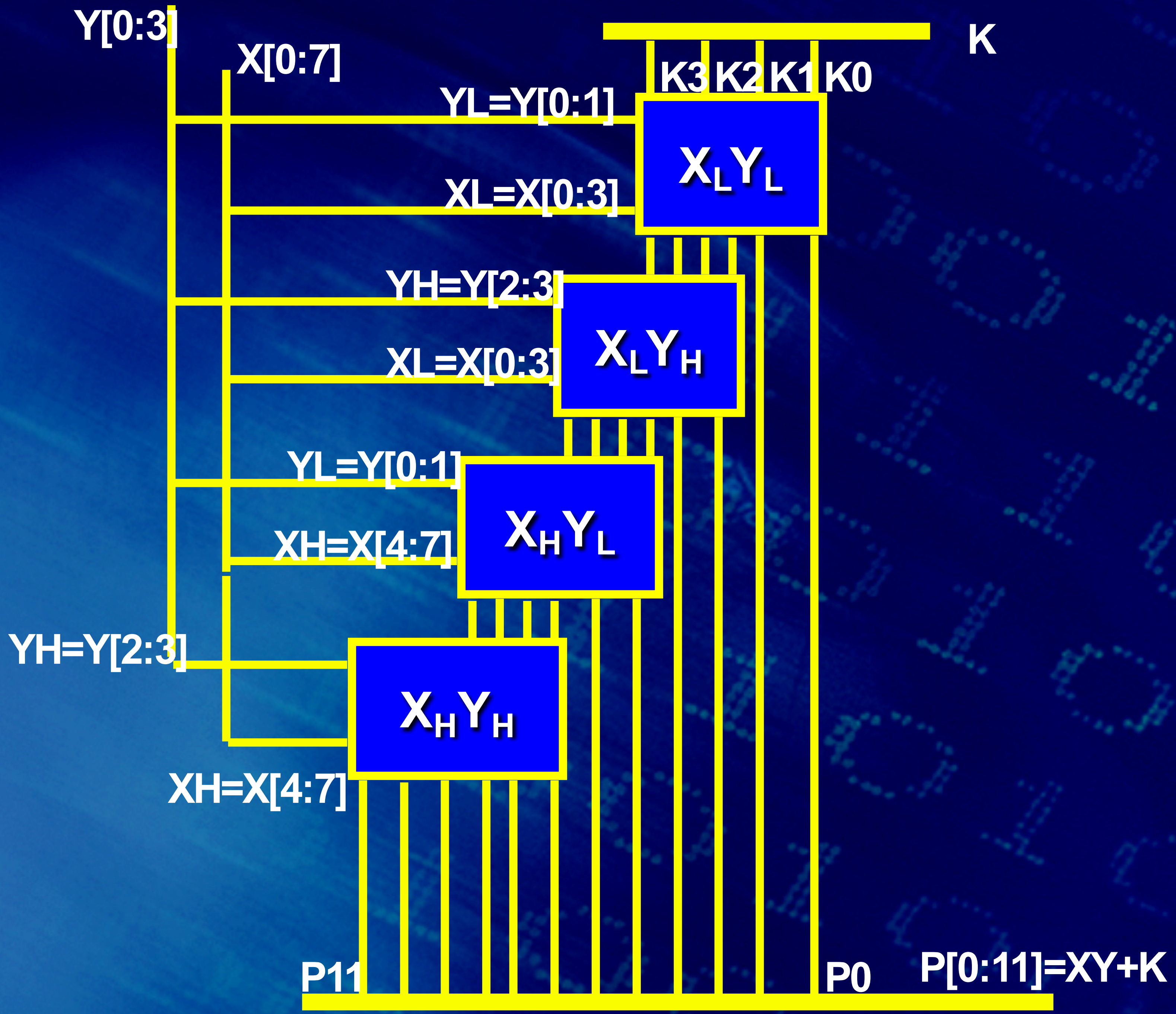
Optimized implementation



“Composed” multipliers

- To analyze the basic principle driving “composed” multipliers let’s consider an 8×4 multiplier, implemented by 4×2 multipliers.


X_H
X_L
Y_H
Y_L
X_HY_L
X_LY_L
X_HY_H
X_LY_H
P


Y[0:3]
X[0:7]
K
K3 K2 K1 K0
YL=Y[0:1]
XL=X[0:3]
XLYL
YH=Y[2:3]
XL=X[0:3]
XLYH
YL=Y[0:1]
XH=X[4:7]
XHYL
YH=Y[2:3]
XHYH
XH=X[4:7]
P11
P0
P[0:11]=XY+K

Basic combinational blocks

Multiplexer

Demultiplexer

Multiplier

Adder-Subtractor

Encoder

Decoder

ROM

Comparator

ALU

1's counter

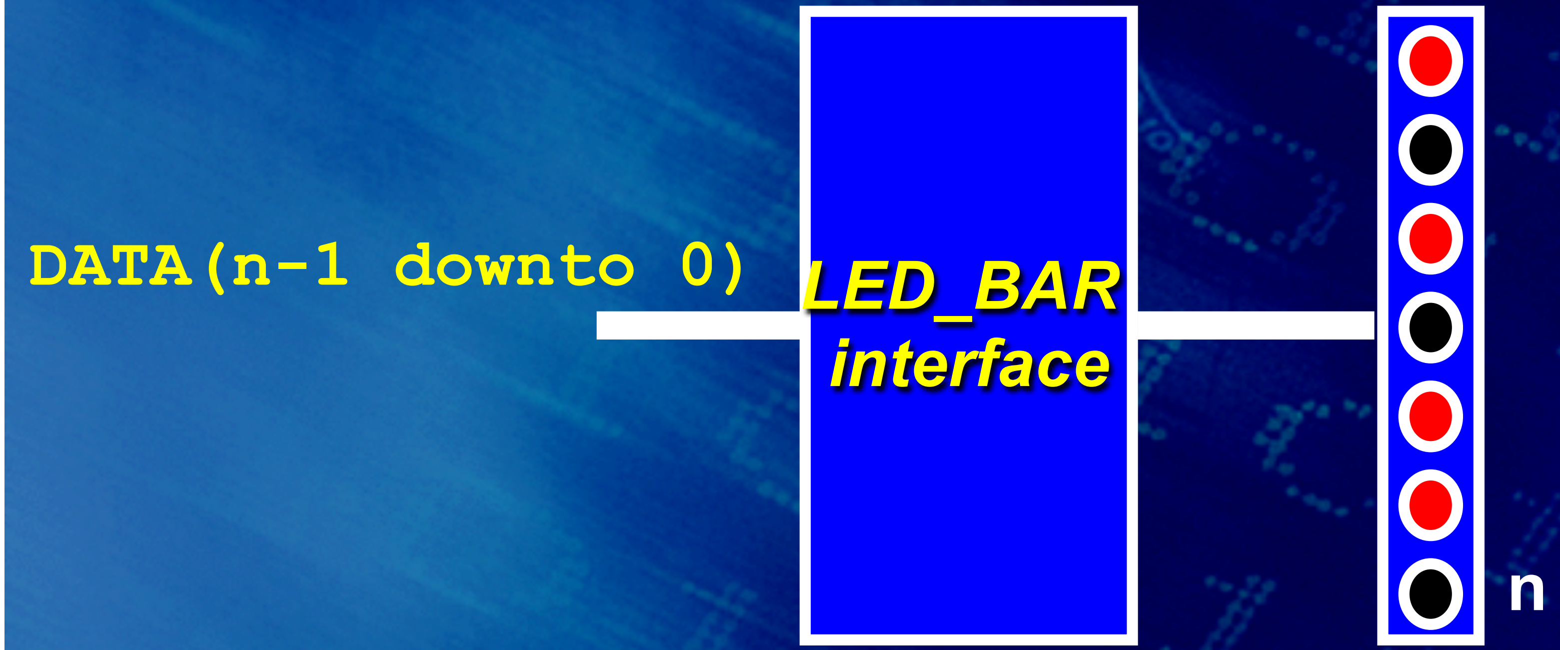
I/O interfaces

I/O interfaces

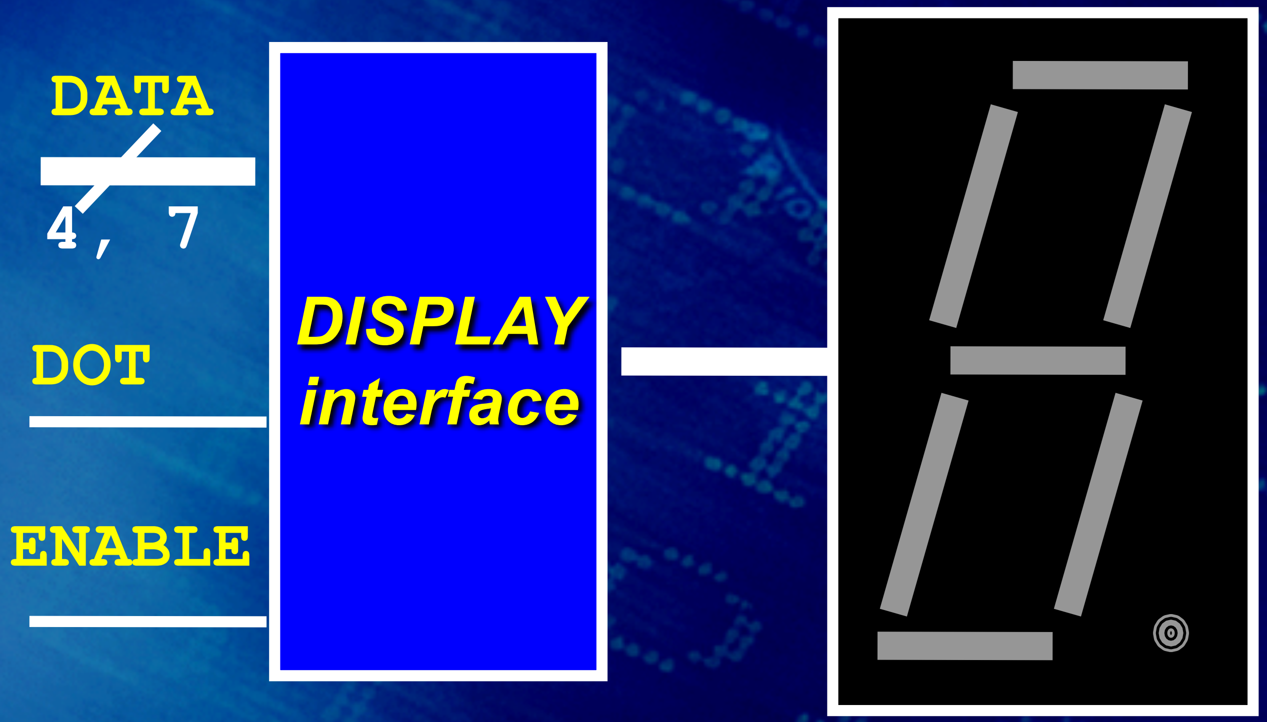
The combinational interfaces for the following I/O devices will be presented:

- ***led bar driver***
- ***7-segments display***
- ***dip-switch***
- ***keypad***

Led bar interface

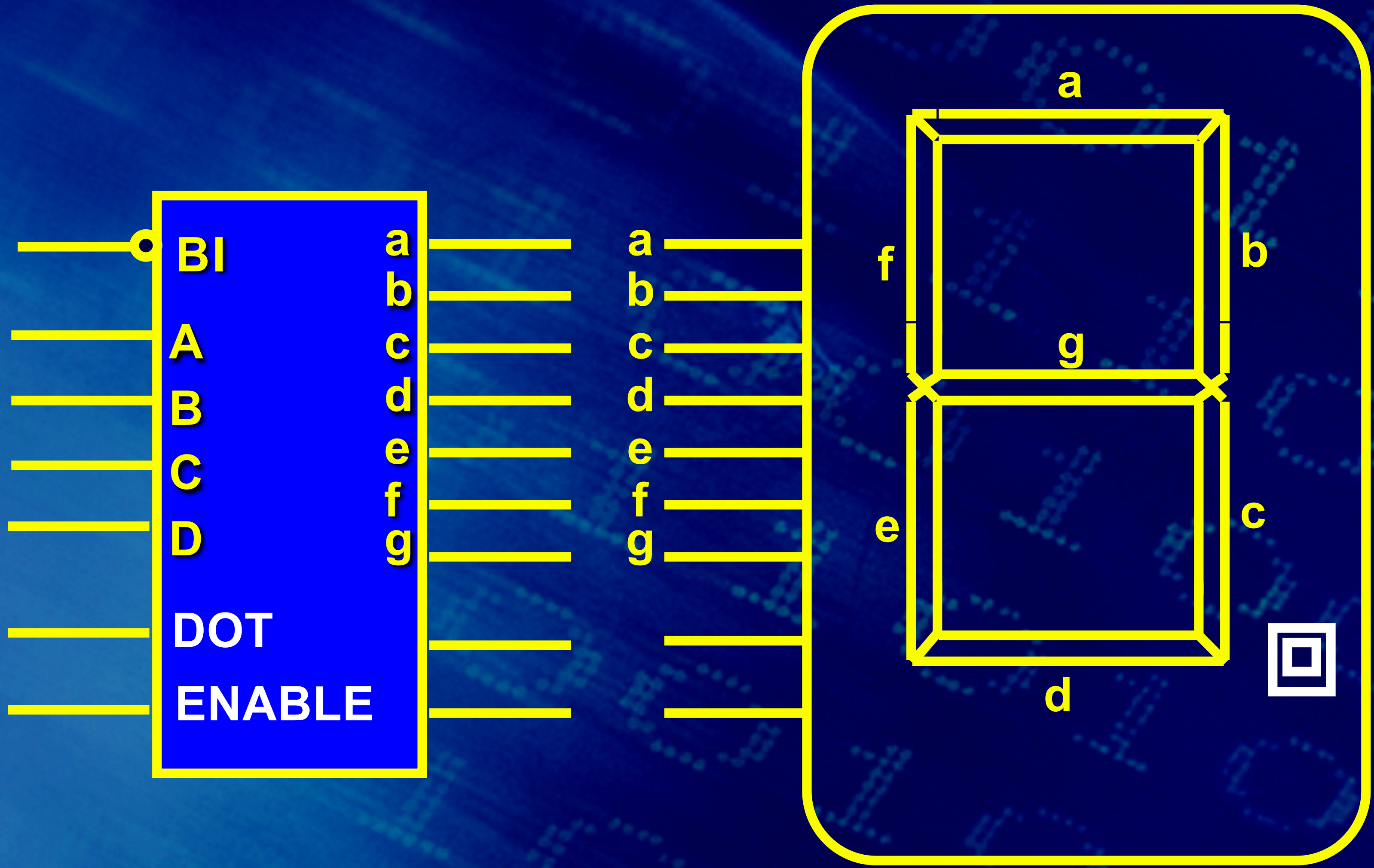


7-segments display interface

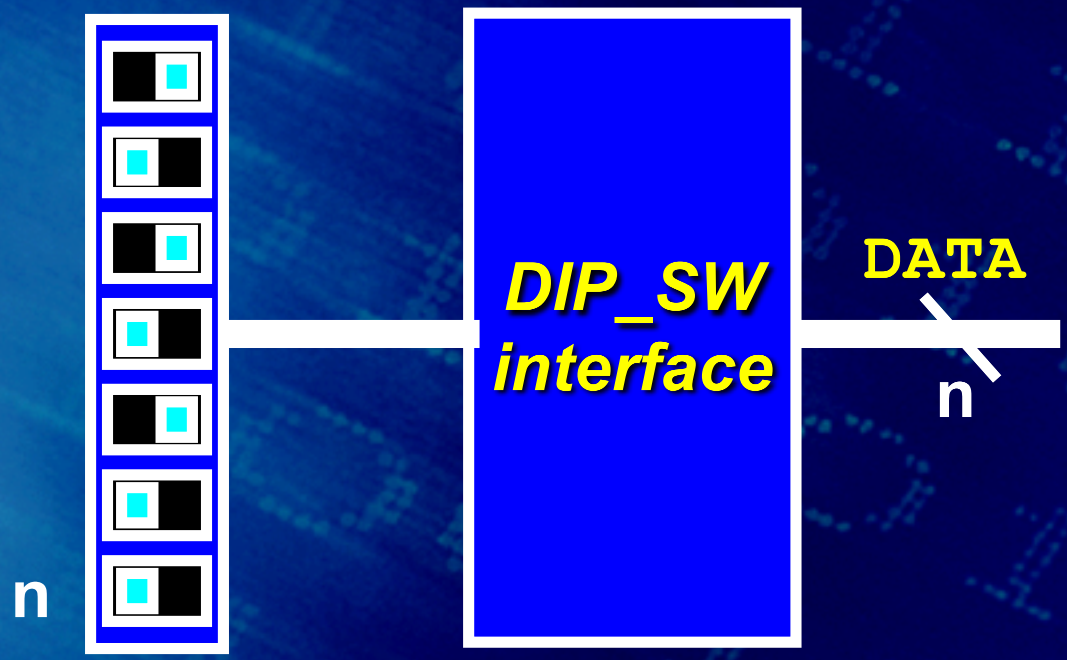


 Rel. 14.06.2018

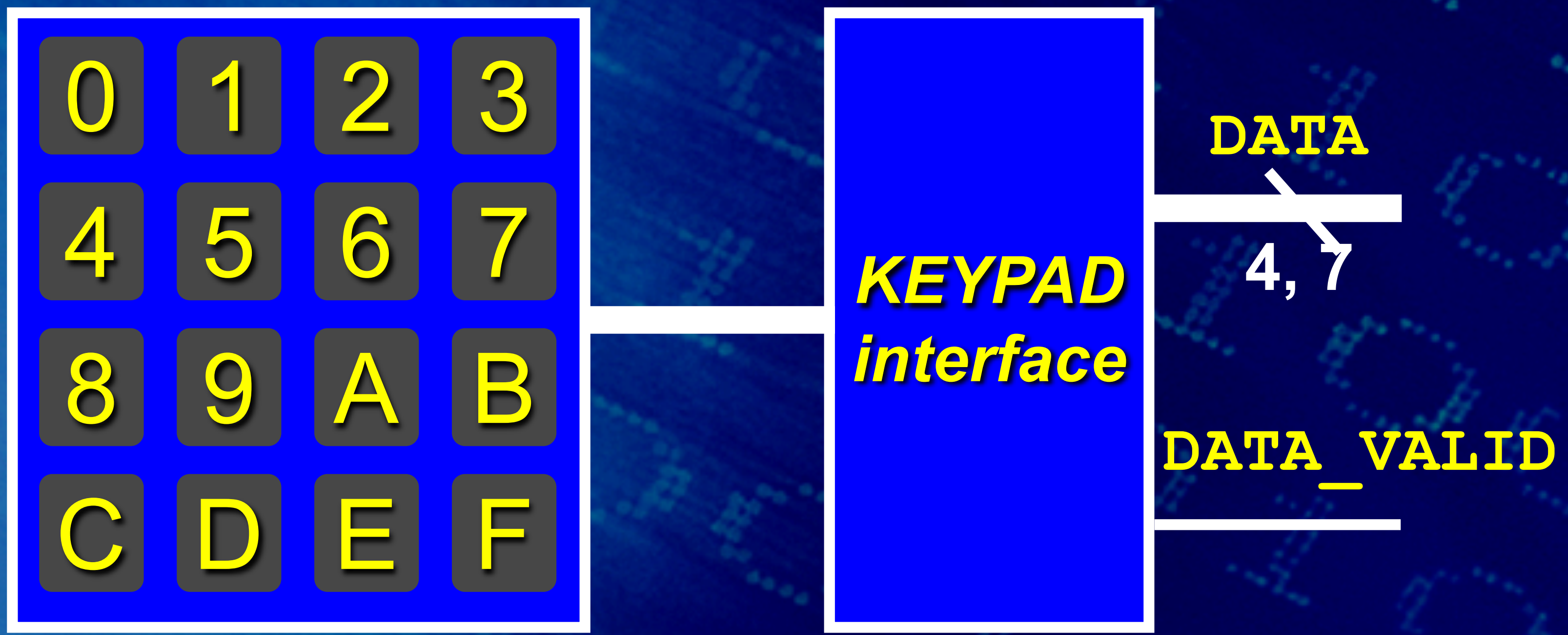
7-segments display interface

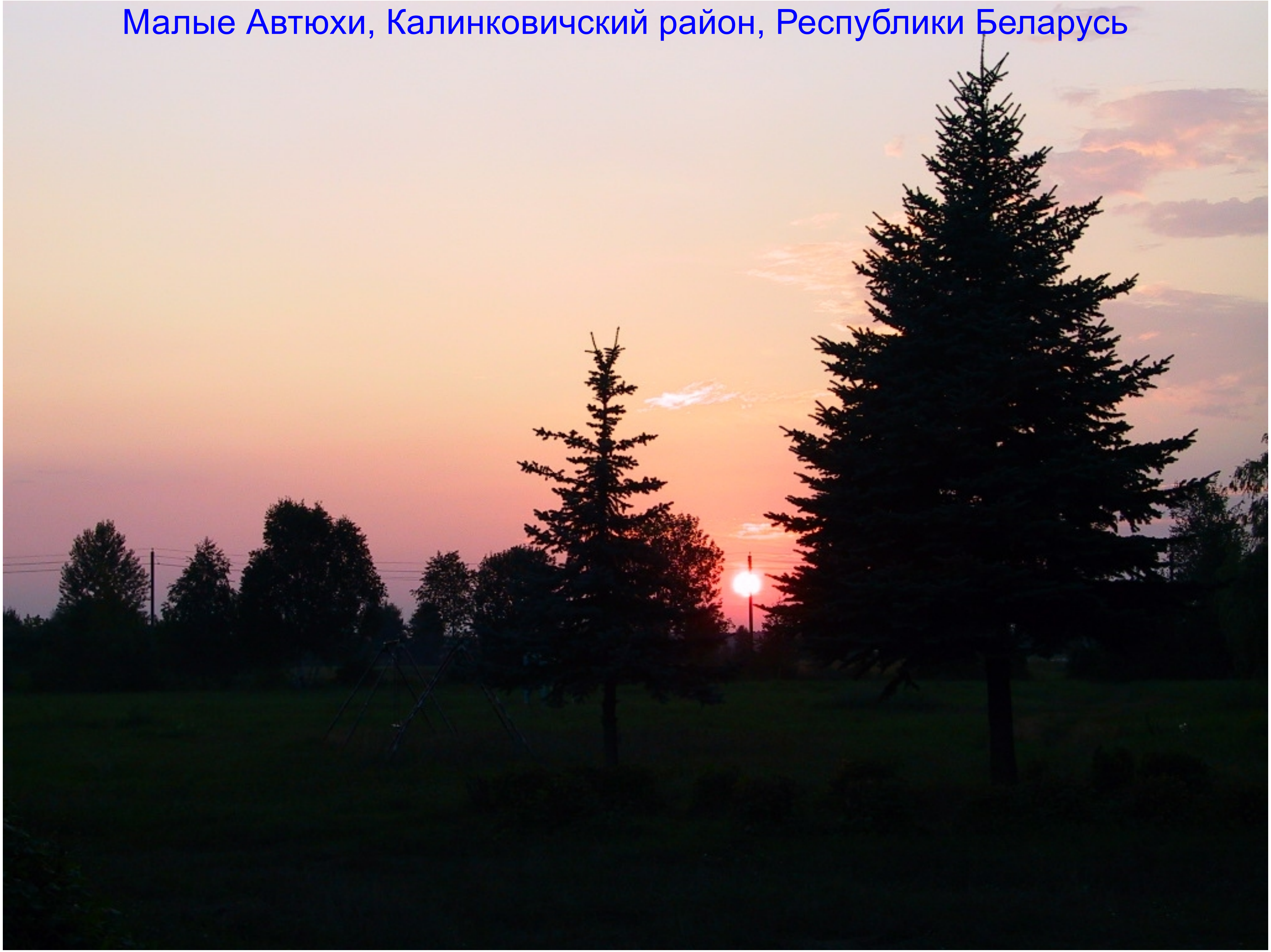


Dip-switch interface



Keypad interface




Малые Автюхи, Калинковичский район, Республики Беларусь