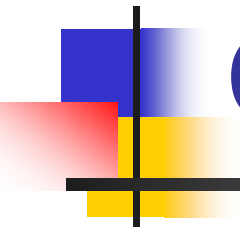
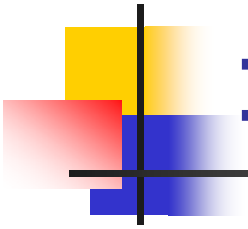


Circuite integrate digitale

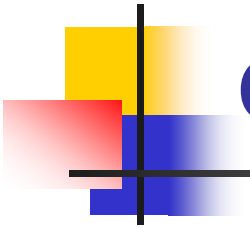
Curs 2



Ideile principale din cursul 1

- Definirea sistemelor digitale, prin contrast cu cele analogice
- Conversia analog – digital
- Definirea formală a sistemelor şi circuitelor digitale
- Introducere în Verilog HDL
- Aritmetică binară: reprezentarea numerelor în baza 2

- Tema 1...



Curs 2

- Un exemplu de circuit digital: *pixel corrector*
- Aplicaţii ale sistemelor şi circuitelor digitale

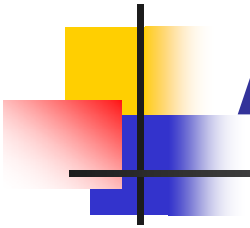
- Circuitele logice: exemple
- Funcţii şi algebră logică

- Aritmetică binară (continuare): operaţii, C2, virgulă mobilă

Exemplu de circuit digital: *pixel corrector*

- http://dcae.pub.ro/gstefan/digital_circuits.html

- Tema 2: desenaţi schema bloc a unui multiplicator pentru numere binare de 8 biţi, care foloseşte algoritmul înmulţirii "de mână"

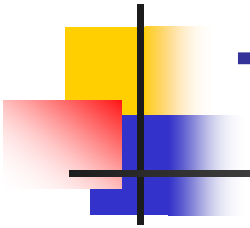


Aplicaţii ale circuitelor digitale

- sisteme de calcul: de la ipaduri şi laptopuri la supercomputere
- televiziune, comunicaţii, telefonie...
- procesări de semnale, inclusiv audio-video (camere video, sisteme audio)
- aparatură electrocasnică
- sisteme de comandă şi control

Aplicaţii (pieţe de desfacere)

- Automobile
  - Sistemul de aprindere, controlul motorului, ABS
- Aparate de uz casnic sau personal
  - TV, jucării, telefoane mobile, aparate casnice...
- Control industrial
  - Sisteme de control, roboţi industriali,....
- Medicină
  - Pompe de infuzie, maşină de dializă, proteze,...
- Reţele
  - Rutere, huburi,...
- Birou
  - Faxuri, fotocopiatoare, printere, monitoare...

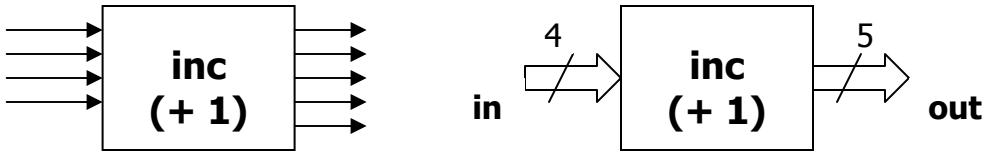


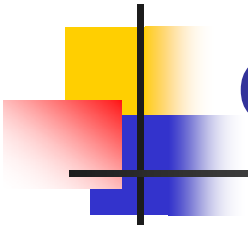
Tipuri de aplicaţii

- Procesare de date
- Transmisie de date
- Calcul
- Sisteme de comandă şi control

Funcţii logice

- Toate funcţiile circuitelor digitale se realizează pornind de la funcţii logice.
- Exemplu: incrementatorul **out = in + 1**
  - incrementarea este o funcţie aritmetică
  - se implementează cu ajutorul funcţiilor logice
    - out [0] = ~ in [0]
    - out [1] = in [1] & (~in [0]) + (~ in[1]) & in [0]
    - out [i] se modifică doar dacă toţi biţii anteriori ai lui in sunt 1





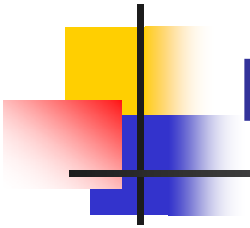
Cele mai importante funcţii logice

- inversarea (opusul valorii de intrare)

- SI: este adevărată numai dacă toate argumentele sunt adevărate
- SAU: este adevărată dacă cel puţin un argument este adevărat
- XOR: suma modulo 2

Simboluri

funcţie logică	simbol	operator Verilog
ŞI	· (se poate omite)	&
SAU	+	|
XOR	$\oplus$	^
NOT	’	~

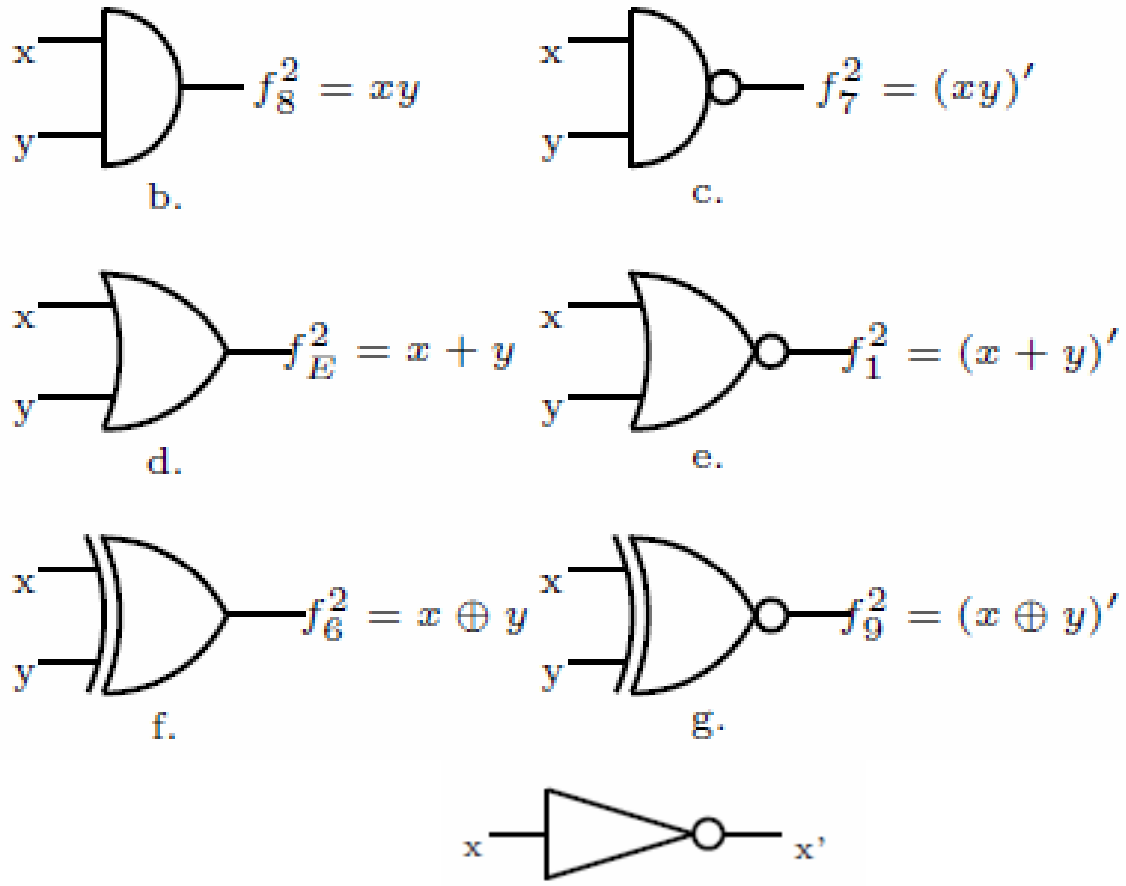


Denumiri uzuale

- Inversor: NOT

- SI (AND); SI negat (NAND)
- SAU (OR); SAU negat (NOR)
- XOR ; NXOR sau XNOR

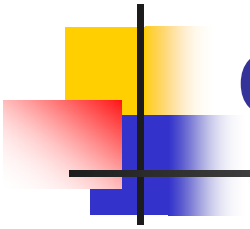
Porţi logice



x	y	xy	(xy)'
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

x	y	x+y	(x+y)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

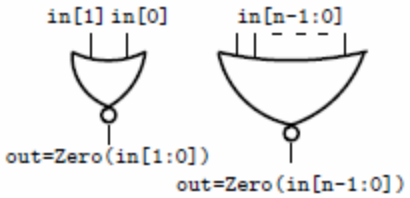
x	y	x⊕y	(x⊕y)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

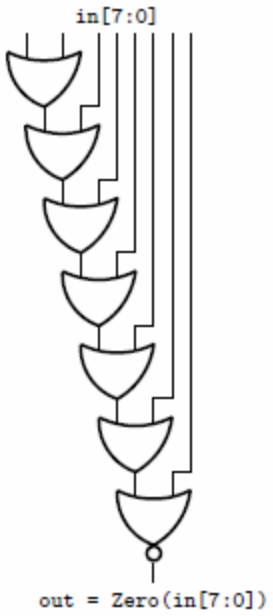


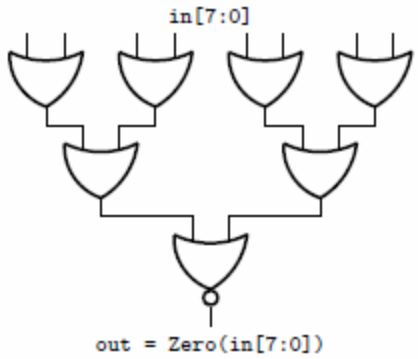
Circuite combinaţionale

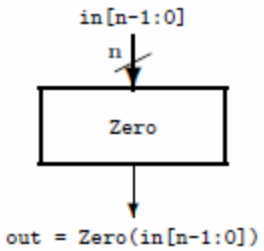
- Calculează funcţii logice mai complexe pornind de la funcţiile elementare
- Sunt construite din porţi logice
- Nu au nicio buclă de reacţie
- Exemple
  - Circuit care identifică valoarea (numărul) zero
  - Circuit de (multiplexor)
  - Sumator
  - Divizor

Zero circuit: *n*-input NOR









fan-in = 2
depth = 1
size = 2

fan-in = n
depth = 1
size = n

fan-in = 8
depth = 7
size = 14

fan-in = 8
depth = 3
size = 14

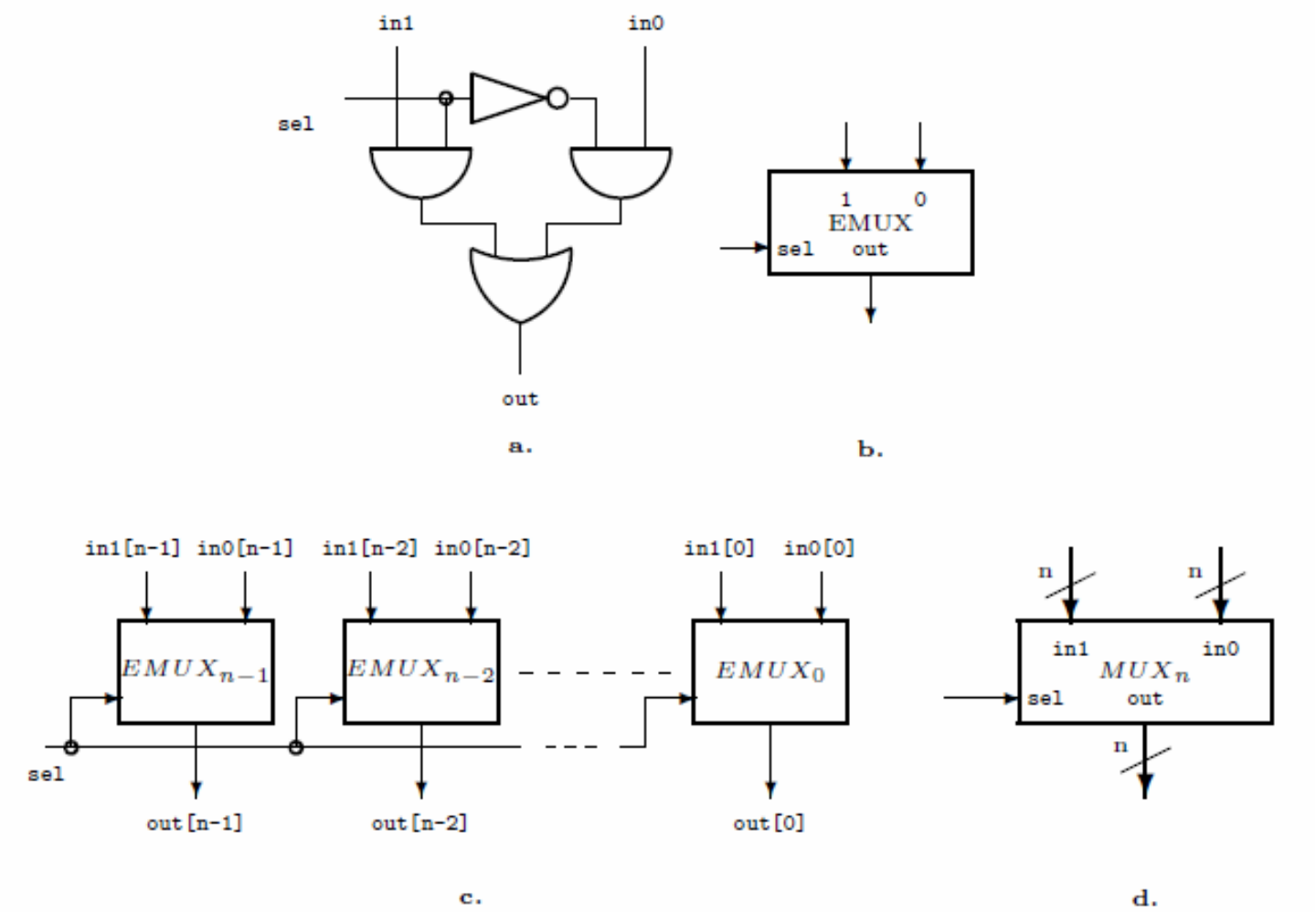
fan-in: număr intrări

depth (adâncime): numărul de niveluri logice

size (mărime): numărul total de intrări în toate porțile

Circuit de selecţie (multiplexor)

`out = sel in1 + sel’ in0`



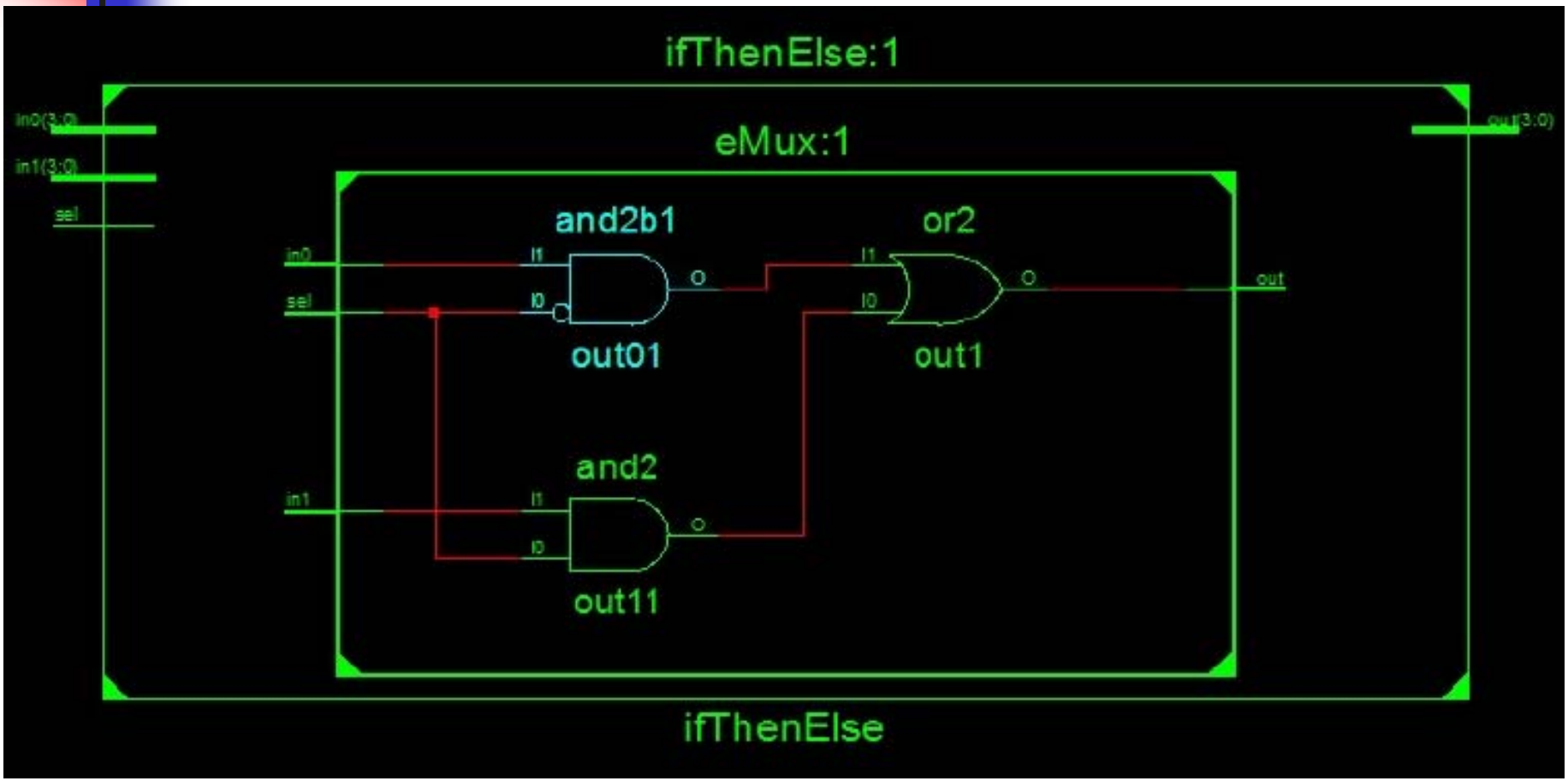
Behavioral:

```
module ifThenElse #(parameter n = 4)
        (output [n-1:0] out,
         input          sel,
         input  [n-1:0] in1, in0);

    assign out = sel ? in1 : in0;
endmodule
```

Structural:

```
module eMux(output out,
            input  sel, in1, in0

    wire        invSel;

    not inverter(invSel, sel);
    and and1(out1, sel, in1),
        and0(out0, invSel, in0);
    or  outGate(out, out1, out0)
endmodule
```

```
module ifThenElse #(parameter n = 4)
        (output [n-1:0] out,
         input          sel,
         input  [n-1:0] in1, in0);

    genvar i ;
    generate for (i=0; i<n; i=i+1)
     begin: eMUX
            eMux selector(.out(out[i]),
                          .sel(sel   ),
                          .in1(in1[i]),
                          .in0(in0[i]));

        end
    endgenerate
endmodule
```


ifThenElse:1
in0(3:0)
in1(3:0)
sel
eMux:1
and2b1
or2
in0
sel
I1
I0
O
out01
I1
I0
O
out1
out
out(3:0)
and2
in1
I1
I0
O
out11
ifThenElse

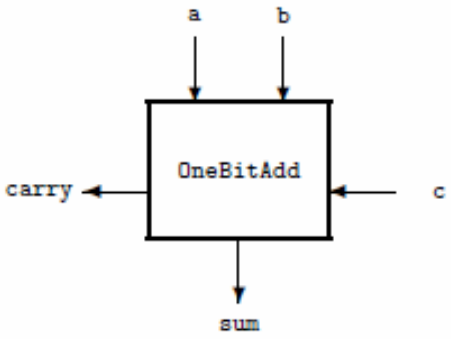
Sumator

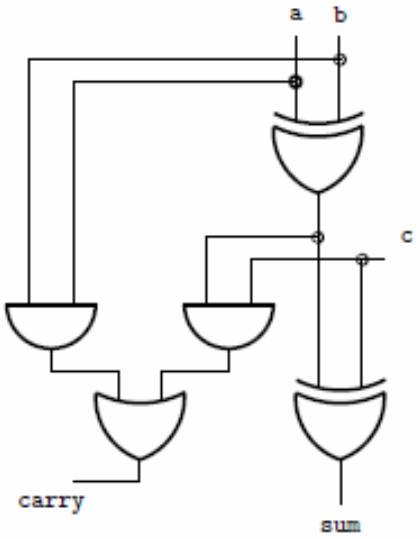
```
module adder #(parameter n = 4)// defines a n-bit adder
        (output [n-1:0] sum,   // the n-bit result
         output         carry, // carry output
         input          c,     // carry input
         input  [n-1:0] a, b); // the two n-bit numbers

    assign {carry, sum} = a + b + c;
endmodule
```

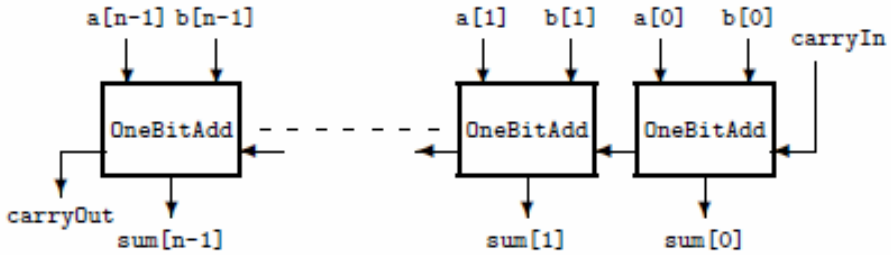
Pentru n= 1, sumatorul complet de un bit:

sum = a ⊕ b ⊕ c = (a ⊕ b) ⊕ c

carry = a b + c (a ⊕ b)







a	b	c	sum	carry
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

$$
\begin{aligned}
sum &= a'b'c + a'bc' + ab'c' + abc \\
&= a'(b'c + bc') + a(b'c' + bc) \\
&= a'(b \oplus c) + a(b \oplus c)' \\
&= a \oplus (b \oplus c) \\
&= a \oplus b \oplus c \\
carry &= a'bc + ab'c + abc' + abc = \\
&= (a'b+ab')c+ab(c'+c)= \\
&= (a \oplus b)c + ab
\end{aligned}
$$

Reprezentarea numerelor întregi în Verilog

- numere pozitive (întregi fără semn)
  - 8'b0000_0000 = 0
  - 8'b1111_1111 = +255
- Complement faţă de doi (numere întregi cu semn)
  - 8'b**1**_0000000 = -128
  - 8'b**0**_0000000 = 0
  - 8'b**0**_1111111 = +127

  - +127=0_1111111 ->1_0000000+1=1_0000001=-127

Divizare (împărţire la 2)

- Întregi pozitive: ***logic*** **right shift** (with **0** input)

```
8'b1001_1100 (156) => 8'b0100_1110 (78)
```

- Întregi cu semn: ***arithmetic*** right shift (with **sign** input)

```
8'b1_1011001(-70) => 8'b1_1101100 (-35)
```