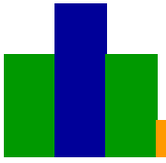
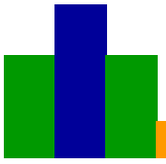


Lecţia 5:
Aritmetica pentru calculator - I

G Stefănescu — Universitatea Bucureşti

Arhitectura sistemelor de calcul, Sem.1
Octombrie 2016—Februarie 2017

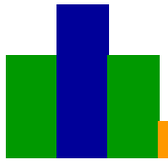
După: D. Patterson and J. Hennessy, Computer Organisation and Design



Aritmetica pentru calculator

Cuprins:

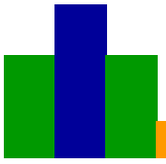
- *Numere (cu si fara semn)*
- Adunare si scadere
- Operatii logice
- Unitatea aritmetica si logica
- Adunare rapida
- Inmultire
- Impartire
- Operatii cu numere reale
- Concluzii, diverse, etc.



Aritmetica pentru calculator

Subiecte: Intrebări la care vom căuta răspuns sunt:

- Cum se *reprezintă numerele întregi*, în particular cele *negative*?
- Cum se *operează* cu ele *în calculator*?
- Care este *cel mai mare număr* reprezentabil în calculator?
  Ce se întâmplă dacă prin operare se obţin *numere mai mari*?
- Cum se operează cu *numere raţionale* ori *reale*?
- Cum se *adună*, *scad*, *înmulţesc*, şi *împart* numerele *în hardware*?
- Ce a fost cu *defectul de la Pentium*?
- Etc.



Numere naturale

Numere naturale:

- Numerele naturale se reprezintă uzual în *baza 10*, e.g.,

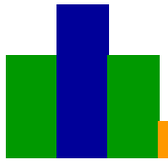
$$2005 = 2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$$

- Reprezentarea în *baza 2* este similară, e.g., numărul anterior se reprezintă astfel

$$\begin{aligned} & 2005_{\texttt{zece}} \\ & = 11111010101_{\texttt{doi}} \\ & = 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 \\ & \qquad + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \end{aligned}$$

- Poziţiile se notează de la dreapta la stânga, i.e.,

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



..Numere naturale

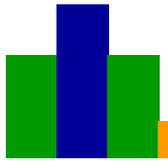
Numere naturale (cont.)

- Cuvintele MIPS au 32 de biţi, deci teoretic putem reprezenta numerele de la 0 la $2^{32} - 1 (= 4.294.967.295_{\texttt{zece}})$, e.g.

$$
\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\texttt{doi}} = && 0_{\texttt{zece}} \\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\texttt{doi}} = && 1_{\texttt{zece}} \\
&\vdots \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\texttt{doi}} = && 4.294.967.295_{\texttt{zece}}
\end{aligned}
$$

- Dacă prin operaţii aritmetice (e.g., înmulţire) se depăşeşte limita maximă $2^{32} - 1$ se spune că apare *overflow* (*depăşire*).

- Hardware-ul poate fi făcut fie *să detecteze* overflow-ul, fie *să-l ignore*.

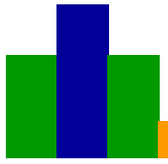


Numere intregi

Numere intregi:

- Numerele *întregi* se reprezintă cu *semn* şi *magnitudine*.
- Dacă semnul ar fi un bit separat, zero ar avea două reprezentări (± 0), generând confuzie...
- Se preferă următoare *reprezentare prin complementul la 2*

$$
\begin{aligned}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\texttt{doi}} &= 0_{\texttt{zece}} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\texttt{doi}} &= 1_{\texttt{zece}} \\
&\vdots \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\texttt{doi}} &= 2.147.483.647_{\texttt{zece}} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\texttt{doi}} &= -2.147.483.648_{\texttt{zece}} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\texttt{doi}} &= -2.147.483.647_{\texttt{zece}} \\
&\vdots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\texttt{doi}} &= -2_{\texttt{zece}} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\texttt{doi}} &= -1_{\texttt{zece}}
\end{aligned}
$$



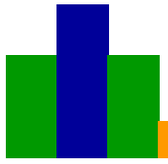
..Numere intregi

Numere intregi (cont.)

- Valoarea numerelor cu semn se obţine aplicând direct formula

$$x_{31} \times (-2^{31}) + x_{30} \times 2^{30} + x_{29} \times 2^{29} + \ldots + x_0 \times 2^0$$

- Alternativ, când $x_{31} = 1$ magnitudinea se obţine *complementând* faţă de 2 poziţiile 30-0 şi *adăugând 1*.
- La load (încărcare):
  - dacă se operează cu *operatori cu semn (signed)*, se face *extensia semnului*, completând toate pozitiile libere din faţă (dacă sunt) cu semnul, pentru a păstra valoarea;
  - dacă se lucrează cu *operatori fără semn (unsigned)* se completează cu 0.
- La fel, operaţiile de comparare pot da rezultate diferite dacă operatorii sunt consideraţi cu semn, ori fără.



..Numere intregi

Numere intregi (cont.) Exemple de operaţii

- Dacă regiştri `$s0,$s1` conţin

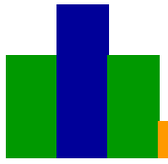
  `1111 1111 1111 1111 1111 1111 1111 1111` şi

  `0000 0000 0000 0000 0000 0000 0000 0001`

  instrucţiunile

```
slt  $t0,$s0,$s1;      # comparare cu semn
sltu $t0,$s0,$s1;    # comparare fara semn
```

  setează pe `$t0` cu 1 în primul caz ($-1 < 1$) şi cu 0 în al doilea ($2^{32} - 1 > 1$).



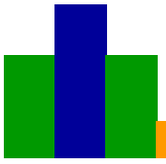
..Numere intregi

Numere intregi (cont.) Trucuri de operare rapidă:

- *Negarea unui număr:*
  Negarea unui număr se obţine complementând toţi biţii 31-0 şi adăugând 1.

- *Extensia cu semn şi negarea comută:*
  Dacă *neg* este funcţia de negare şi $ex_{m\to n}$ extensia cu semn de la m la n biţi, atunci $neg(ex_{m\to n}(p)) = ex_{m\to n}(neg(p))$.

- *Reprezentări hexazecimale:*
  Numerele $0,1,\ldots,9,10,11,12,13,14,15$ au *reprezentările hexazecimale* $0,1,\ldots,9,a,b,c,d,e,f$, respectiv.



Aritmetica pentru calculator

Cuprins:

- Numere (cu si fara semn)
- *Adunare si scadere*
- Operatii logice
- Unitatea aritmetica si logica
- Adunare rapida
- Inmultire
- Impartire
- Operatii cu numere reale
- Concluzii, diverse, etc.

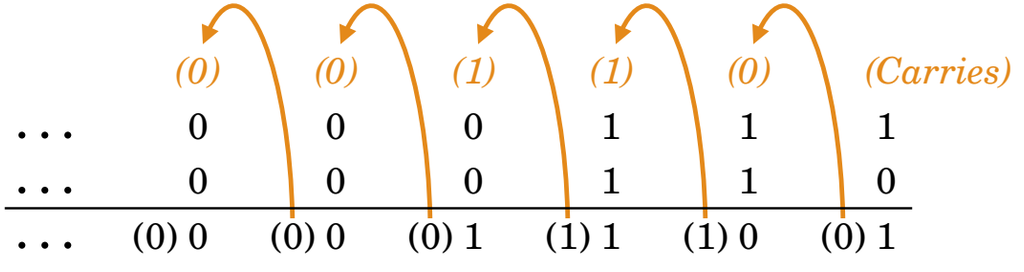
Adunare si scadere

Adunare si scadere:

- *Adunare:* Adunarea se face uzual:

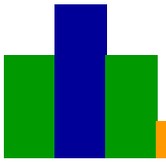
$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\texttt{doi}} & = \quad 7_{\texttt{zece}} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\texttt{doi}} & = \quad 6_{\texttt{zece}} \\ \hline & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\texttt{doi}} & = \quad 13_{\texttt{zece}} \end{array}$$

Algoritmul este ilustrat în desen, incluzând cifrele de transfer:



- *Scădere:* Folosind reprezentare cu complement, scăderea se reduce la adunare:

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\texttt{doi}} & = \quad 7_{\texttt{zece}} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\texttt{doi}} & = \quad -6_{\texttt{zece}} \\ \hline & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\texttt{doi}} & = \quad 1_{\texttt{zece}} \end{array}$$



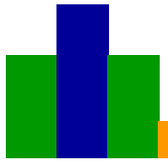
..Adunare si scadere

Depasire (overflow):

- Se pot obţine depăşiri în următoarele condiţii:

Operatie	Operand A	Operand B	Rezultat indicand overflow
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

- Operaţiile `add,addi,sub` *produc excepţii* la depăşire.
- Operaţiile `addu,addiu,subu` *nu produc excepţii* la depăşire.

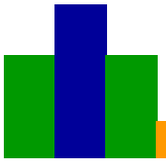


..Adunare si scadere

Depasire (cont.)

- MPIS nu are teste condiţionale pentru depăşire, dar astfel de teste pot fi simulate.
- Exemplu: Pentru a detecta *overflow la adunare cu semn* codul poate fi următorul:

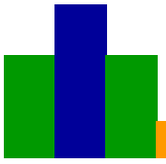
```
addu $t0,$t1,$t2;                          # $t0 = suma (fara semn)
xor  $t3,$t1,$t2;                        # verifica semnele $t1,$t2
slt  $t3,$t3,$zero;                   # $t3 = 1, daca difera semnele
bne  $t3,$zero, No_overflow;             # $t1,$t2 - semne diferite;
                                                    # n-am overflow
xor  $t3,$t0,$t1;                         # verifica semnele $t1, suma;
                                  # $t3 negativ, daca sunt diferite
slt  $t3,$t3,$zero;             # $t3 = 1, daca semnul sumei difera
bne  $t3,$zero, Overflow;                                 # am overflow
                        # ($t1,$t2 semne egale, suma semn diferit)
```



Operatii aritmetice MIPS

Operatii aritmetice MIPS (Legenda: A = Instructiune aritmetica):

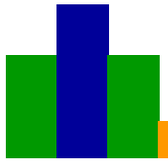
Tip	Instructiune	Exemple	Semantica	Comentarii
A	add	`add $s1,$s2,$s3`	`$s1=$s2+$s3`	3 operanzi; overflow detectat
A	subtract	`sub $s1,$s2,$s3`	`$s1=$s2-$s3`	3 operanzi; overflow detectat
A	add immediate	`addi $s1,$s2,100`	`$s1=$s2+100`	+ constanta; overflow detectat
A	add unsigned	`addu $s1,$s2,$s3`	`$s1=$s2+$s3`	3 operanzi; overflow nedetectat
A	subtract unsigned	`subu $s1,$s2,$s3`	`$s1=$s2-$s3`	3 operanzi; overflow nedetectat
A	add immediate unsigned	`addiu $s1,$s2,100`	`$s1=$s2+100`	+ constanta; overflow nedetectat
A	move to coprocesor register	`mfc0 $s1,$epc`	`$s1=$epc`	copiaza reg. EPC si alti registri pentru rezolvarea exceptiei



Aritmetica pentru calculator

Cuprins:

- Numere (cu si fara semn)
- Adunare si scadere
- *Operatii logice*
- Unitatea aritmetica si logica
- Adunare rapida
- Inmultire
- Impartire
- Operatii cu numere reale
- Concluzii, diverse, etc.



Operatii pe biti

Shift-uri:

- Se *deplasează* conţinutul unui registru cu *un număr de poziţii într-o direcţie*, poziţiile *vide* completându-se cu *0*-uri.
- Exemplu: Prin *deplasare la stânga cu 8 poziţii* registrul

  `0000 0000 0000 0000 0000 0000 0000 1101`

  devine

  `0000 0000 0000 0000 0000 1101 0000 0000`

- In MIPS denumirile sunt

  `sll` *(shift left logical)*

  şi, dual,

  `srl` *(shift right logical)*.

MIPS - operatii logice

Instructiuni logice:

- `sll reg1, reg2, const` (*shift left logical*)

  Semantică: Se deplasează conţinutul registrului `reg2` cu `const` poziţii la stânga, iar rezultatul se pune în `reg1`.

  Cod maşină (constanta se codifică în câmpul *shamt*):

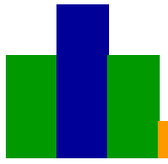
0	0	reg1	reg2	cons	0

- `srl reg1, reg2, const` (*shift right logical*)

  Semantică: Se deplasează conţinutul registrului `reg2` cu `const` poziţii la dreapta, iar rezultatul se pune în `reg1`.

  Cod:

0	0	reg1	reg2	cons	2



..Operatii pe biti

Operatii logice pe biti:

- Se pot face operaţii logice AND, OR *bit-cu-bit*, fie cu doi operatori în regiştri (anume `and,or`), fie cu un operator în registru şi o constantă dată ca valoare imediată (anume `andi,ori`).
- Exemplu: Dacă regiştrii cu operatori conţin

  `0000 0000 0000 0000 0000 1101 0000 0000`

  `0000 0000 0000 0000 0011 1100 0000 0000`

  rezultatul lui AND este

  `0000 0000 0000 0000 0000 1100 0000 0000.`
- Unul din operatori poate fi privit ca o *masca*: păstrează anumite poziţii din celălalt operator (unde el are 1), restul devine 0.
- Rezultatul lui OR este

  `0000 0000 0000 0000 0011 1101 0000 0000.`

MIPS - operatii logice

Instructiuni logice (cont.)

- `and reg1, reg2, reg3` (*and*)

  Semantică: `reg1` = `reg2` *AND* `reg3` (bit-cu-bit)

  Cod:

0	reg2	reg3	reg1	0	36

- `or reg1, reg2, reg3` (*or*)

  Semantică: `reg1` = `reg2` *OR* `reg3` (bit-cu-bit)

  Cod:

0	reg2	reg3	reg1	0	37

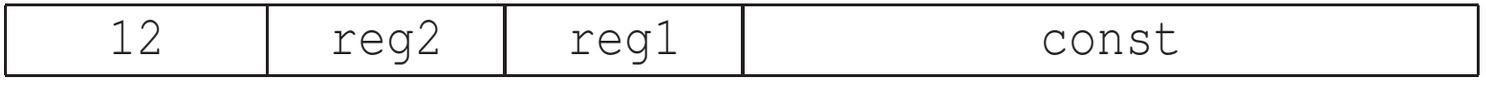
MIPS - operatii logice

Instructiuni logice (cont.)

- `andi reg1, reg2, const` (*and immediate*)

  Semantică: `reg1` = `reg2` *AND* `const` (bit-cu-bit)

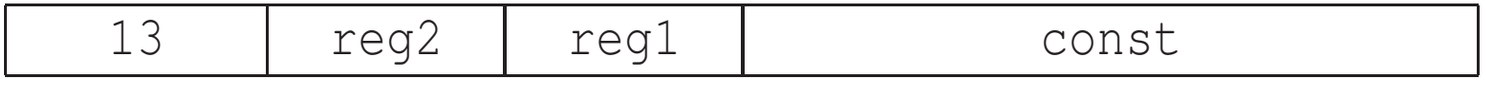
  Cod:

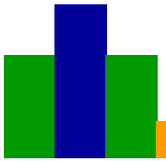


- `ori reg1, reg2, const` (*or immediate*)

  Semantică: `reg1` = `reg2` *OR* `const` (bit-cu-bit)

  Cod:

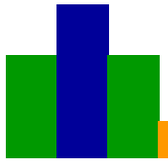




Aritmetica pentru calculator

Cuprins:

- Numere (cu si fara semn)
- Adunare si scadere
- Operatii logice
- *Unitatea aritmetica si logica*
- Adunare rapida
- Inmultire
- Impartire
- Operatii cu numere reale
- Concluzii, diverse, etc.



Unitatea aritmetica si logica

ALU (Arithmetic and Logic Unit):

- *ALU* (*unitatea aritmetica si logica*) este zona principala de calcul din computer.
- In ALU sunt efectuate atât *operaţiile aritmetice* (adunare, scădere, înmulţire, împărţire), cât şi *operaţiile logice* (shift-uri, AND, OR).
- Circuitele necesare sunt pur *combinaţionale* (i.e., nu necesită memorie).
- Cum MIPS are cuvinte pe 32 biţi, folosim *ALU de lărgime 32*.

ALU pe 1 bit

ALU pe 1 bit: Operaţiile logice (AND, OR, NOT, IF_ZERO) se realizează direct, fiind reprezentate în hardware:

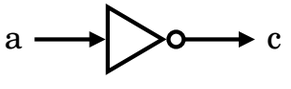
1. AND gate (c = a · b)

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (c = a + b)

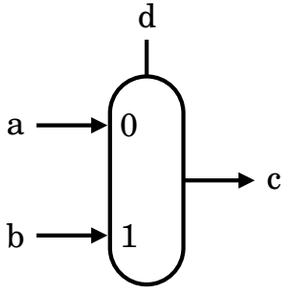
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = $\bar{a}$)



a	c = ā
0	1
1	0

4. Multiplexor
(if d = = 0, c = a;
else c = b)

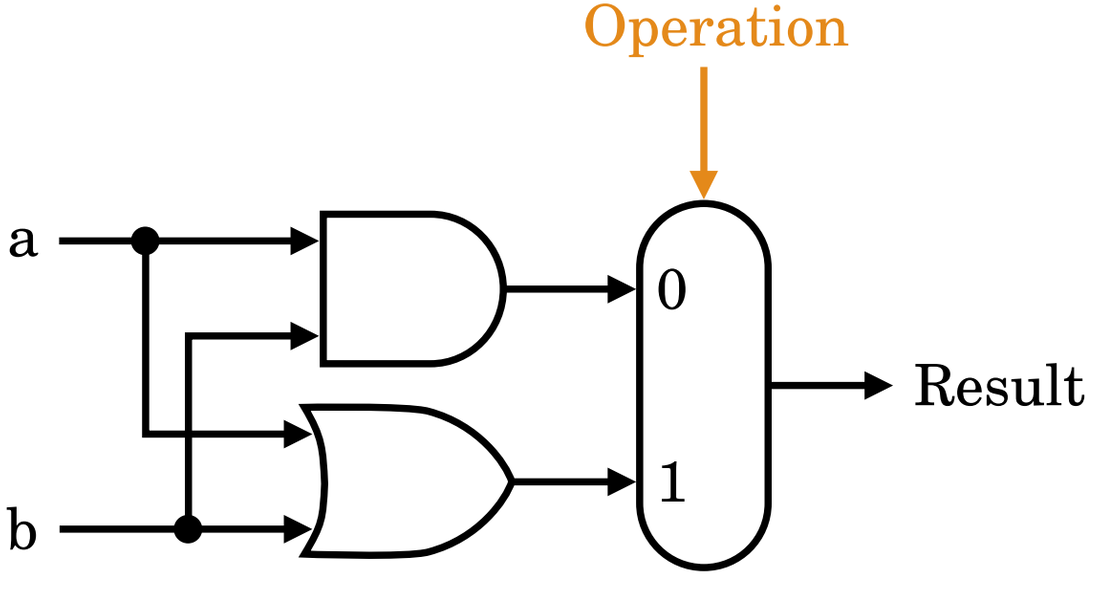


d	c
0	a
1	b

..ALU pe 1 bit

ALU pe 1 bit (cont.)

- Operaţiile logice AND şi OR pot fi cuplate într-o *unitate logică pe 1 bit* pentru {AND, OR} ca în figură

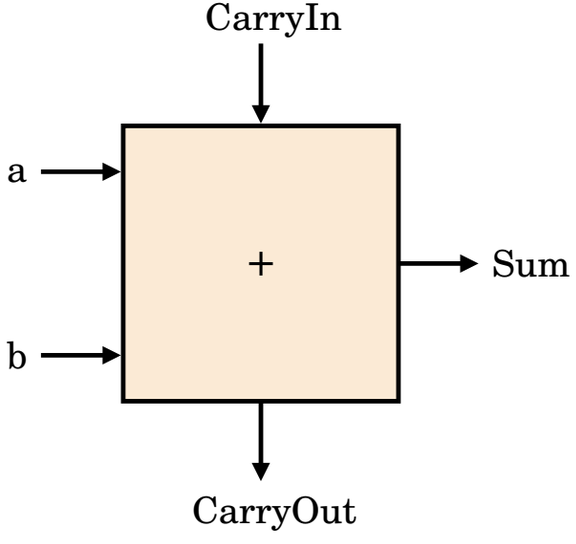


- Semnalul de control pentru multiplexor Operation selectează ce operaţie AND ori OR se execută.

Adunare

Adunare:

- La adunare, folosim cifrele de transfer `CarryIn` şi `CarryOut`.
- Pasul de bază foloseşte un modul ca în figură, cu tabela de adevăr alăturată:



a	b	CarryIn	CarryOut	Suma
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

- Notând $ci = \texttt{CarryIn}, co = \texttt{CarryOut}$, forma normală cu *sume de produse* pentru `CarryOut` este

$$co = \overline{a} \cdot b \cdot ci + a \cdot \overline{b} \cdot ci + a \cdot b \cdot \overline{ci} + a \cdot b \cdot ci$$

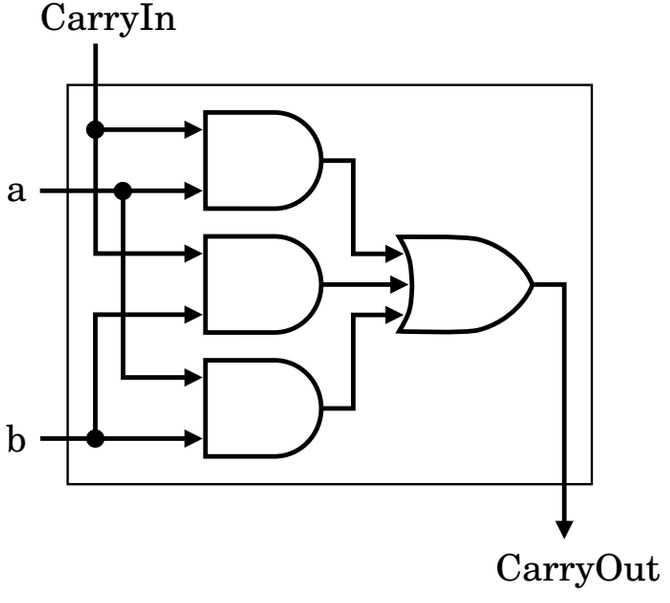
Adunare

Adunare (cont.)

- O *formula redusă* echivalentă pentru `CarryOut` este

$$co = b \cdot ci + a \cdot ci + a \cdot b$$

- Formula rezultă se translatează în următorul circuit

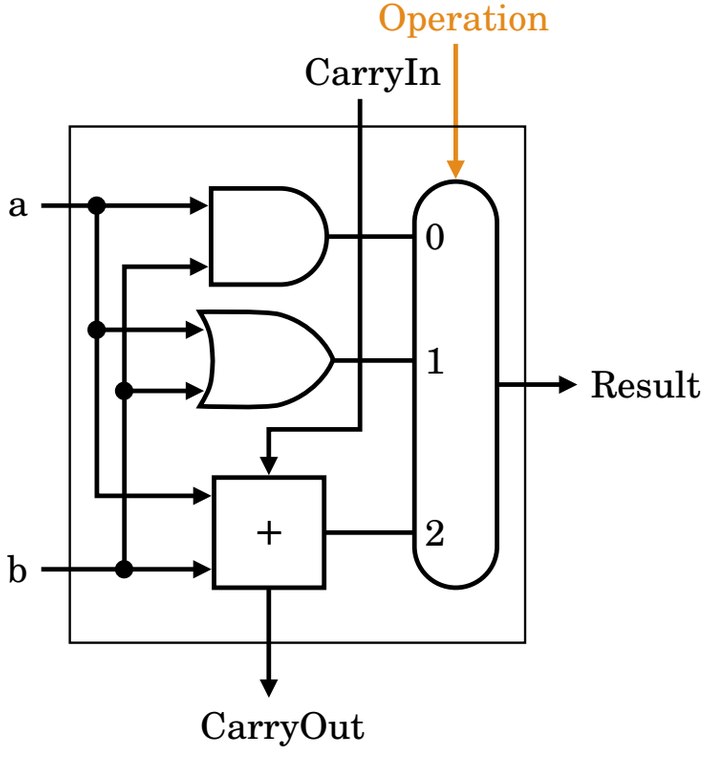


- Analog se obţine un circuit pentru `Sum`.

..ALU pe 1 bit

ALU pe 1 bit (cont.)

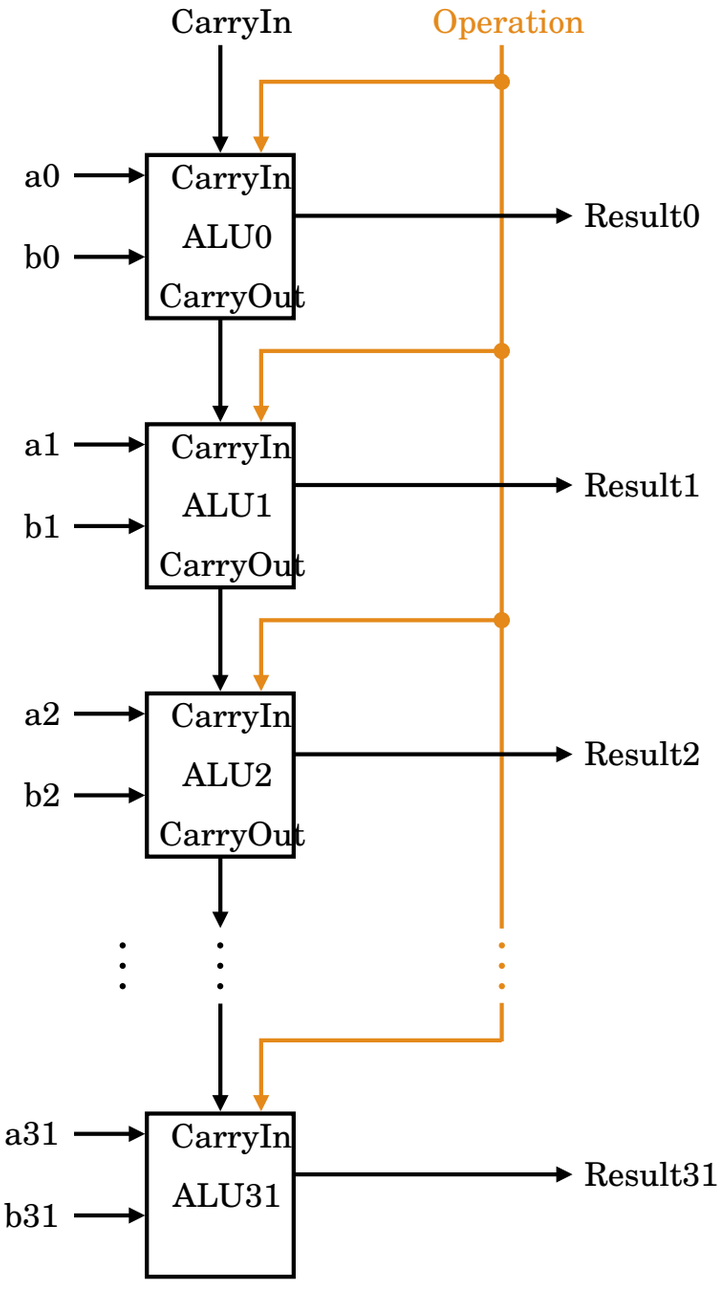
- Circuitele de mai sus pentru `CarryIn` şi `Sum` dau un circuit pentru modulul +.
- Impreună cu ALU precedent pentru {`AND, OR`}, găsim o *unitate aritmetică-logică pe 1 bit* pentru {`AND,OR,+`}



ALU pe 32 biti

ALU pe 32 biti:

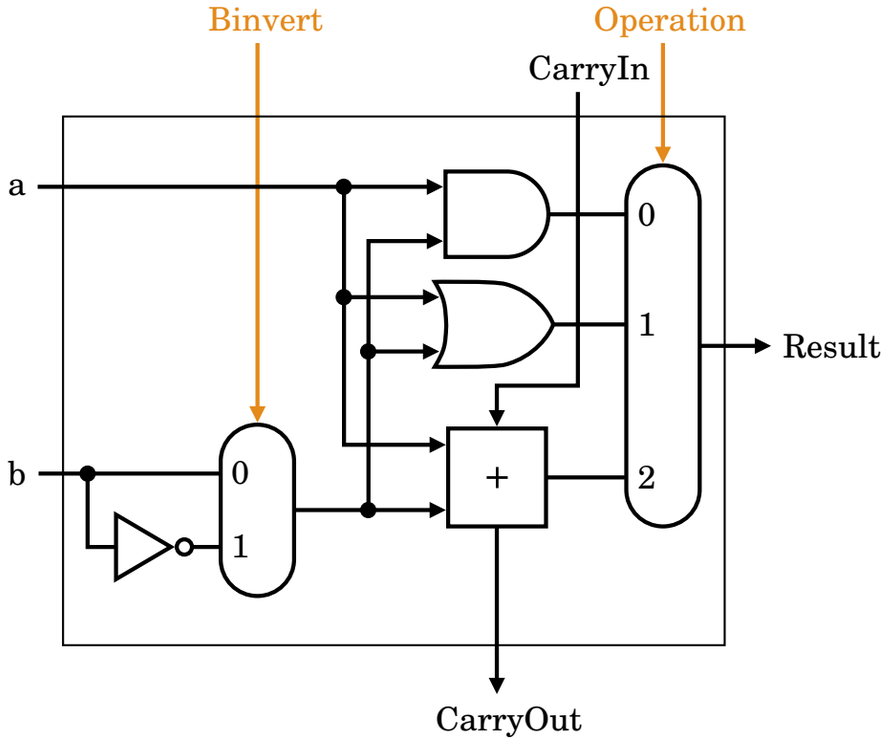
- Folosim 32 de ALU-uri pe 1 bit de tip {`AND`, `OR`, `+`}, conectate ca în figură.
- Sumatorul rezultat conectând `CarryOut`-ul unui modul la `CarryIn`-ul modulului următor se numeşte *ripple-carry adder* (sumator cu transport succesiv).
- Pentru sumator, `CarryIn`-ul modulului 0 este 0, iar `CarryOut`-ul modulului 31 se ignoră (aici).



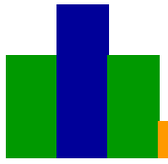
..ALU pe 32 biti

Adaugam scaderea:

- Scăderea $a - b$ se reduce la adunarea lui a cu $-b$.
- Cu reprezentarea numerelor negative prin complementul la doi, $-b$ se obţine inversând biţii lui b şi adunând 1.



- In hardware:
  - biţi inversaţi se obţin cu un multiplexor, folosind controlul `Binvert` ce alege între b şi $\overline{b}$;
  - adunarea lui 1 se realizează simplu, *setând* `CarryIn`*-ul modului iniţial 0 la 1* (nu la 0).
- Figura descrie un *ALU pe 32 biti pentru* {`AND`,`OR`,`+`,`-`}.



..ALU pe 32 biti

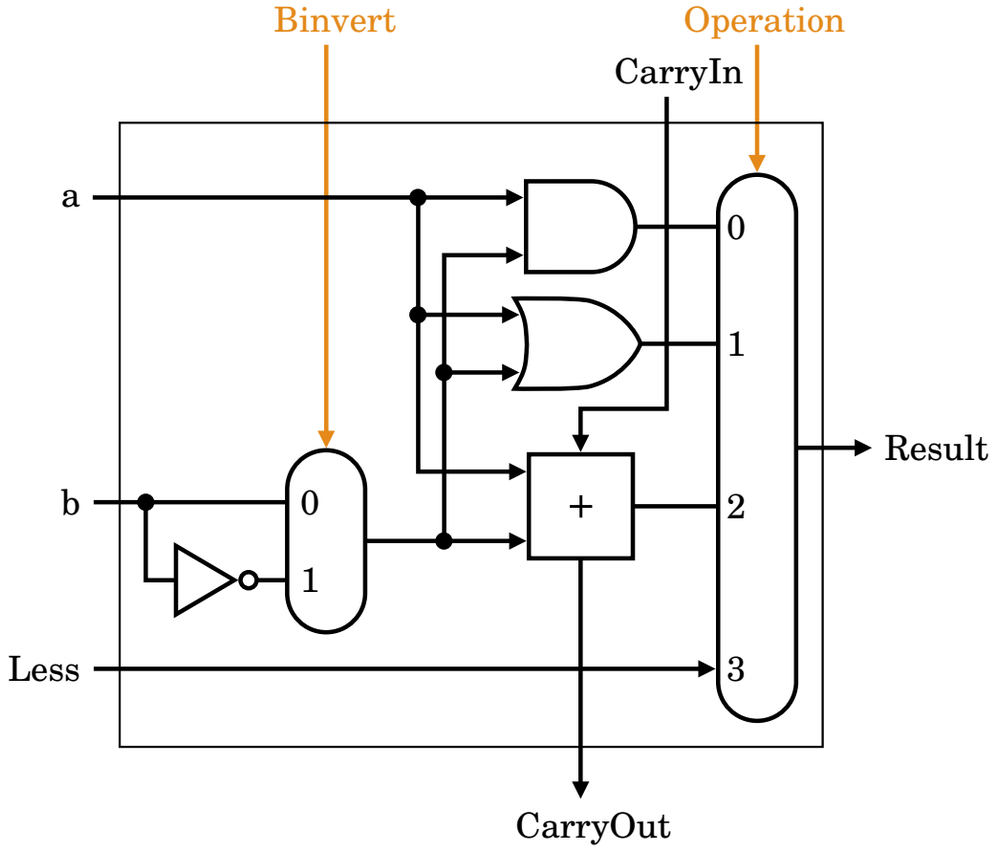
Adaugam "`set on less than`": Rezultatul c al comenzii `slt c,a,b` (i.e., 1 dacă $a < b$, altfel 0) se obţine astfel:

- Observăm că $(a-b) < 0 \Leftrightarrow a < b$, deci compararea lui a cu b se reduce la a compara cu 0 diferenţa $a-b$.
- Rezultatul testului este 1 dacă $a-b$ este negativ, altfel 0, deci c este exact *bitul de semn al lui $a-b$*.
- Folosim o "intrare" nouă (temporară) `Less` pentru toate modulele, care va produce biţii rezultatului comenzii; ulterior ea va fi conectata la rezultatul unui calcul intern.
- Bitul (cel mai semnificativ) de la modulului 31 de la $a-b$ va fi directionat la intrarea `Less` a modulului 0, restul biţilor `Less` fiind 0.

Nota: In prezenta overflow-ului, trebuiesc facute mici modificari.

..ALU pe 32 biti

ALU pe 1 bit cu `Less`:



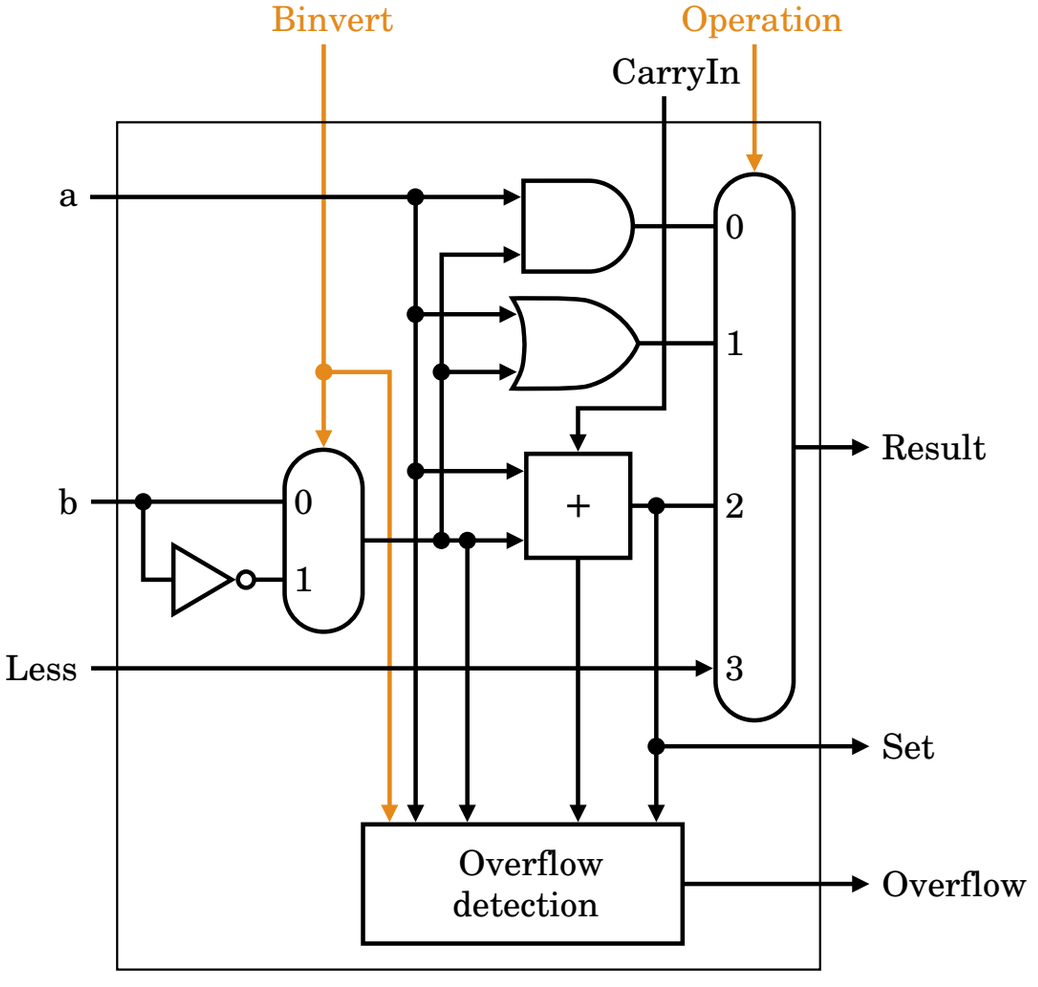
a.

- Multiplexorul are o intrare nouă `Less`, pentru `slt`.
- In circuitul final, "intrarea" `Less` va dispare, fiind calculată din a, b.

..ALU pe 32 biti

ALU pe 1 bit cu *overflow* (nedetaliat) şi extensia *pentru cel mai semnificativ bit*:

- Circuitul adaugă detectarea celui mai semnificativ bit `Set`.
- Se va folosi în ultima veriga a circuitului final (pentru modului 31).

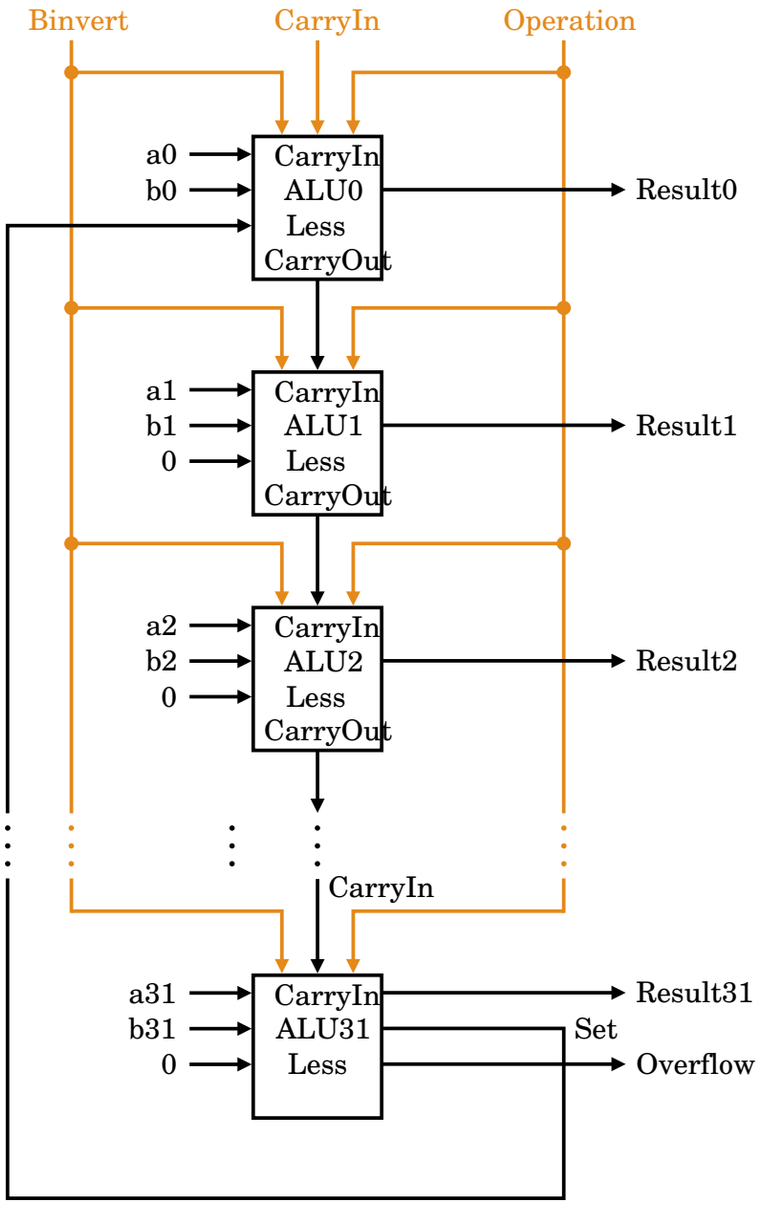


b.

..ALU pe 32 biti

ALU pe 32 biti:

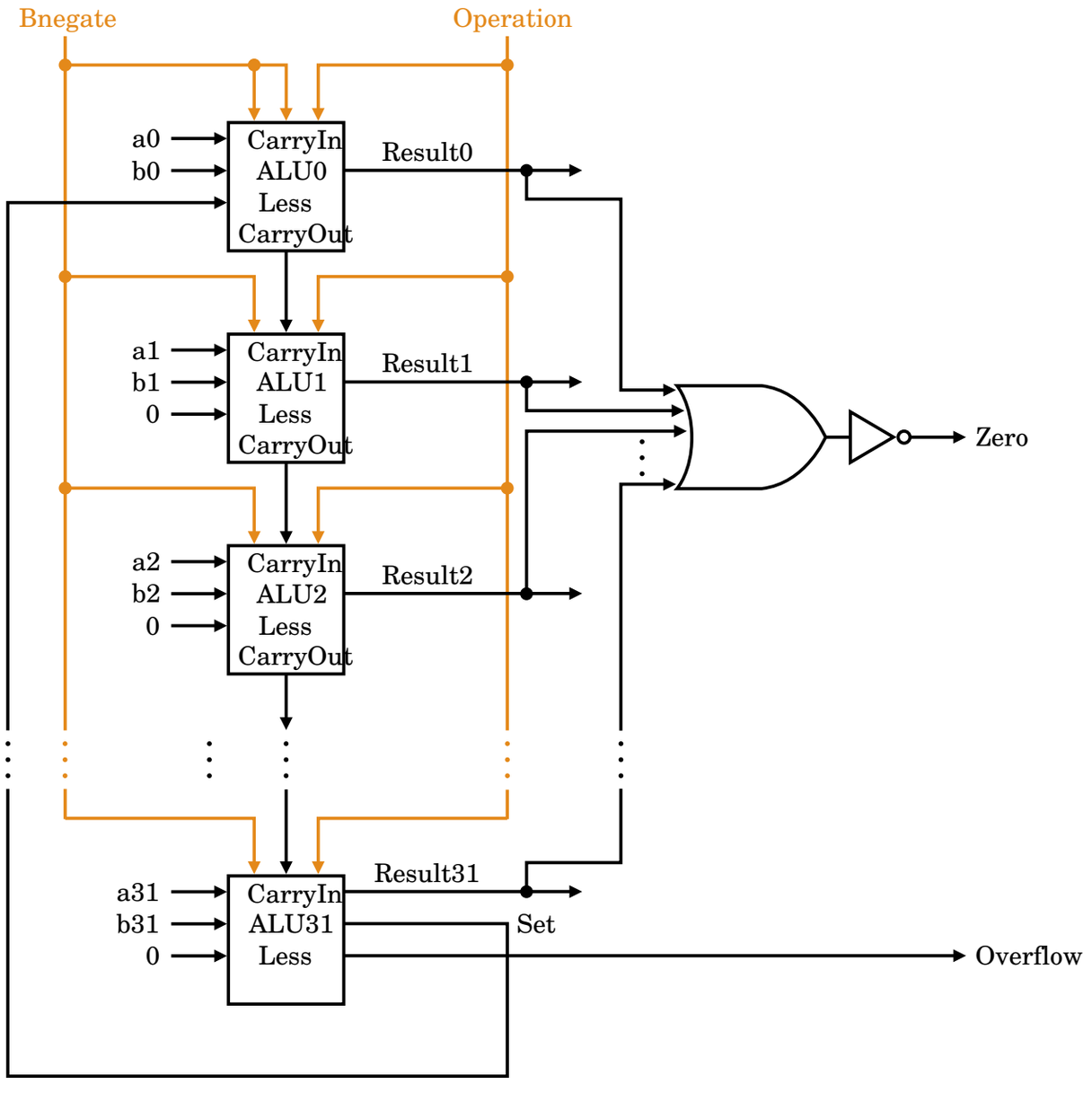
- cu `set on less than`
- şi *detecţie de overflow*.



..ALU pe 32 biti

ALU pe 32 biti cu *test la zero*:

- adăugăm testul dacă *rezultatul este zero* (folosit la instrucţiunile condiţionale)



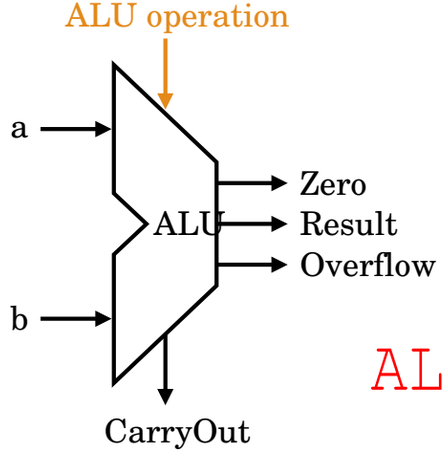
ALU pe 32 biti (final)

ALU pe 32 biti *final*:

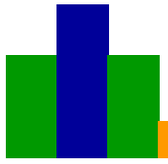
- Liniile de control combină `Binvert` şi `CarryIn` în `Bnegate`. Tabela de comandă este:

Bnegate	Operation	Function
0	00	and
0	01	or
0	10	add
1	10	subtract
1	11	set on less than

- Obţinem *ALU final pe 32 biţi*, notat simbolic



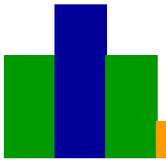
`ALU Operation` este (`Bnegate,Operation`)



..ALU pe 32 biti (final)

Comentarii:

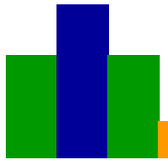
- In componenta finală `a`, `b`, `Result` au câte 32 de biţi, `ALU Operation` are 3, iar `Zero`, `Overflow`, `CarryOut` câte 1 bit.
- Formal, tipul nu diferenţiază intrările (ori ieşirile) între ele, e.g.,
$$ALU : 32 + 32 + 3 \rightarrow 1 + 32 + 1 + 1$$
- ... dar, conceptual, o componentă are două direcţii ortogonale: una de *calcul* şi una de *control* (ori *interacţie*).
- In ALU de mai sus, *calea de calcul* este relaţia $(\texttt{a}, \texttt{b}) \mapsto \texttt{Result}$.
- Partea de *control de intrare* indică ce operaţie are loc în procesarea curentă; partea de *control de ieşire* conţine indicaţii despre excepţii ori pentru controlul altor componente.



Aritmetica pentru calculator

Cuprins:

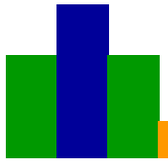
- Numere (cu si fara semn)
- Adunare si scadere
- Operatii logice
- Unitatea aritmetica si logica
- *Adunare rapida*
- Inmultire
- Impartire
- Operatii cu numere reale
- Concluzii, diverse, etc.



Adunare rapida

Adunare rapida:

- Operaţiile de bază (adunare, etc.) se execută de un număr imens de ori. Pot fi executate mai rapide?
- Relativ la adunare, metoda *carry-ripple* anterioară este pur *secvenţială*: modului $k+1$ aşteaptă `CarryOut`-ul de la modulul k spre a-l folosi drept `CarryIn`-ul său.
- O metodă de eficientizare ar fi să *operăm în paralel* şi să *anticipăm deplasările* (cifrele de transport).



..Adunare rapida

Adunare rapida cu hardware infinit:

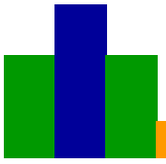
- Fie ci, co prescurtări pentru `CarryIn,CarryOut`. Avem formulale de calcul

$$ci_1 = co_0 = b_0 \cdot ci_0 + a_0 \cdot ci_0 + a_0 \cdot b_0$$

$$\begin{aligned} ci_2 = co_1 &= b_1 \cdot ci_1 + a_1 \cdot ci_1 + a_1 \cdot b_1 \\ &= b_1 \cdot (b_0 \cdot ci_0 + a_0 \cdot ci_0 + a_0 \cdot b_0) + a_1 \cdot (b_0 \cdot ci_0 + \\ &a_0 \cdot ci_0 + a_0 \cdot b_0) + a_1 \cdot b_1 \\ &= b_1 \cdot b_0 \cdot ci_0 + b_1 \cdot a_0 \cdot ci_0 + b_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_0 \cdot \\ &ci_0 + a_1 \cdot a_0 \cdot ci_0 + a_1 \cdot a_0 \cdot b_0 + a_1 \cdot b_1 \end{aligned}$$

  Si aşa mai departe...

- Teoretic, cu hardware “infinit” putem obţine rezultatul direct cu un circuit pe 2 nivele (folosind câte porţi vrem).



..Adunare rapida

Sumator cu carry-lookahead:

- Observăm că

$$ci_{k+1} = a_k \cdot b_k + (a_k + b_k) \cdot ci_k$$

deci este util să introducem notaţiile

$$g_k = a_k \cdot b_k$$
$$p_k = a_k + b_k$$

(g_k este deplasare *generată*, p_k este deplasare *propagată*).

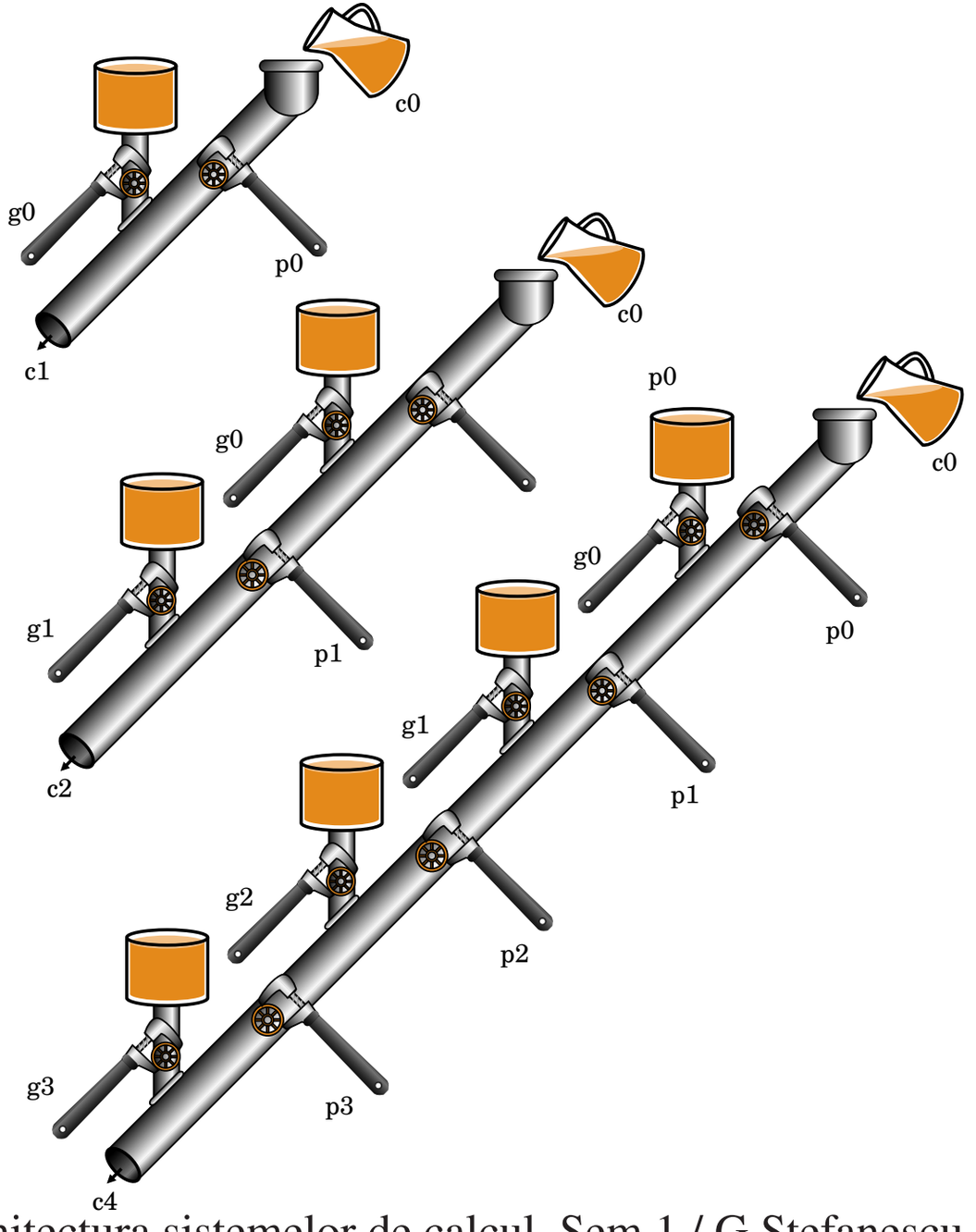
- Pentru 4 biţi avem:

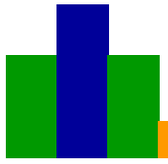
$$ci_1 = g_0 + p_0 \cdot ci_0$$
$$ci_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot ci_0$$
$$ci_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot ci_0$$
$$ci_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot ci_0$$

..Adunare rapida

Sumator carry-lookahead (cont.)

- Analogie de tip apă-şi-conducte privind generarea şi propagarea deplasărilor.
- c_k este 1 când un g_i este 1 şi toţi p_j dintre i şi k sunt 1
  (un bit de deplasare este generat şi apoi propagat)





..Adunare rapida

Sumator carry-lookahead (cont.)

- Putem aplica procedura de mai sus la un nivel superior de abstracţie folosing *grupuri de biţi*.
- Pentru un grup de 4 biţi, formulele sunt un pic mai complicate. *Propagările* pe blocuri sunt:

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0;$$
$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4;$$
$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8;$$
$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12};$$

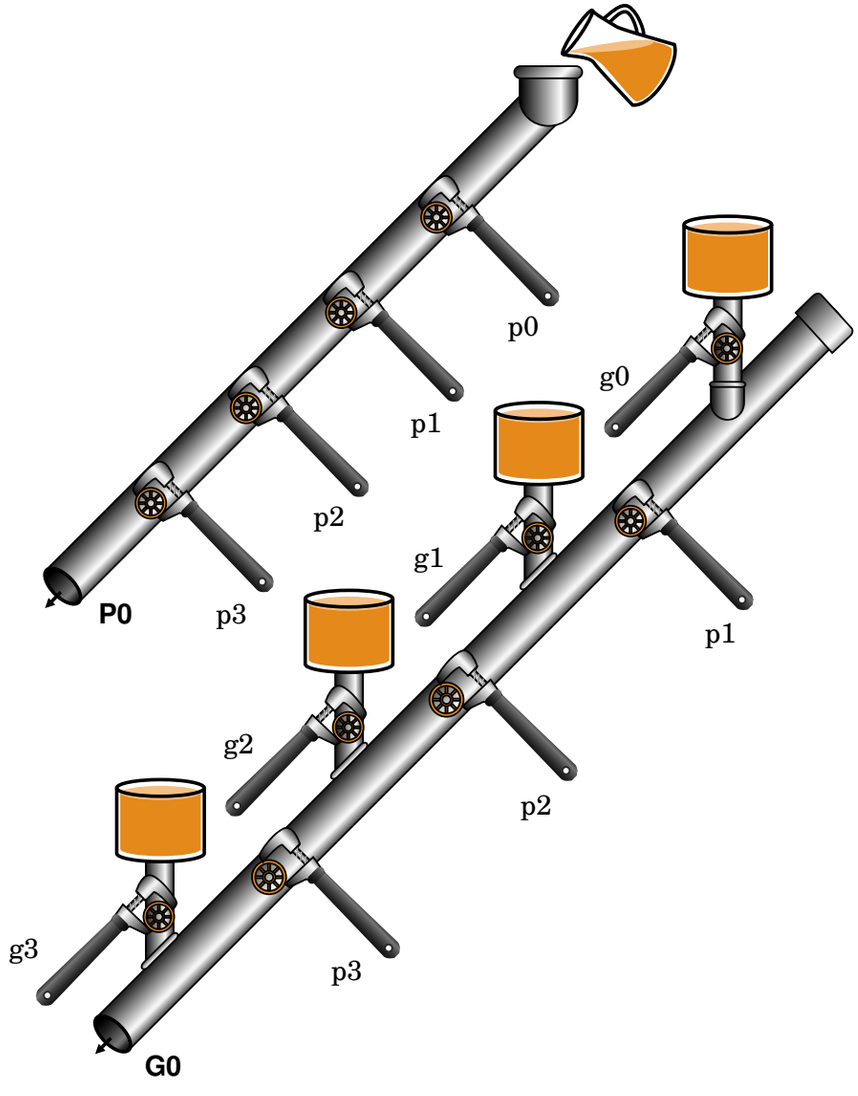
Generările pe blocuri sunt:

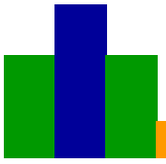
$$G_0 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$
$$G_1 = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4$$
$$G_2 = g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8$$
$$G3 = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}$$

..Adunare rapida

Sumator carry-lookahead (cont.)

- Analogie de tip apă-şi-conducte privind generarea şi propagarea deplasărilor pe *grupuri de biţi*.





..Adunare rapida

Sumator cu carry-lookahead:

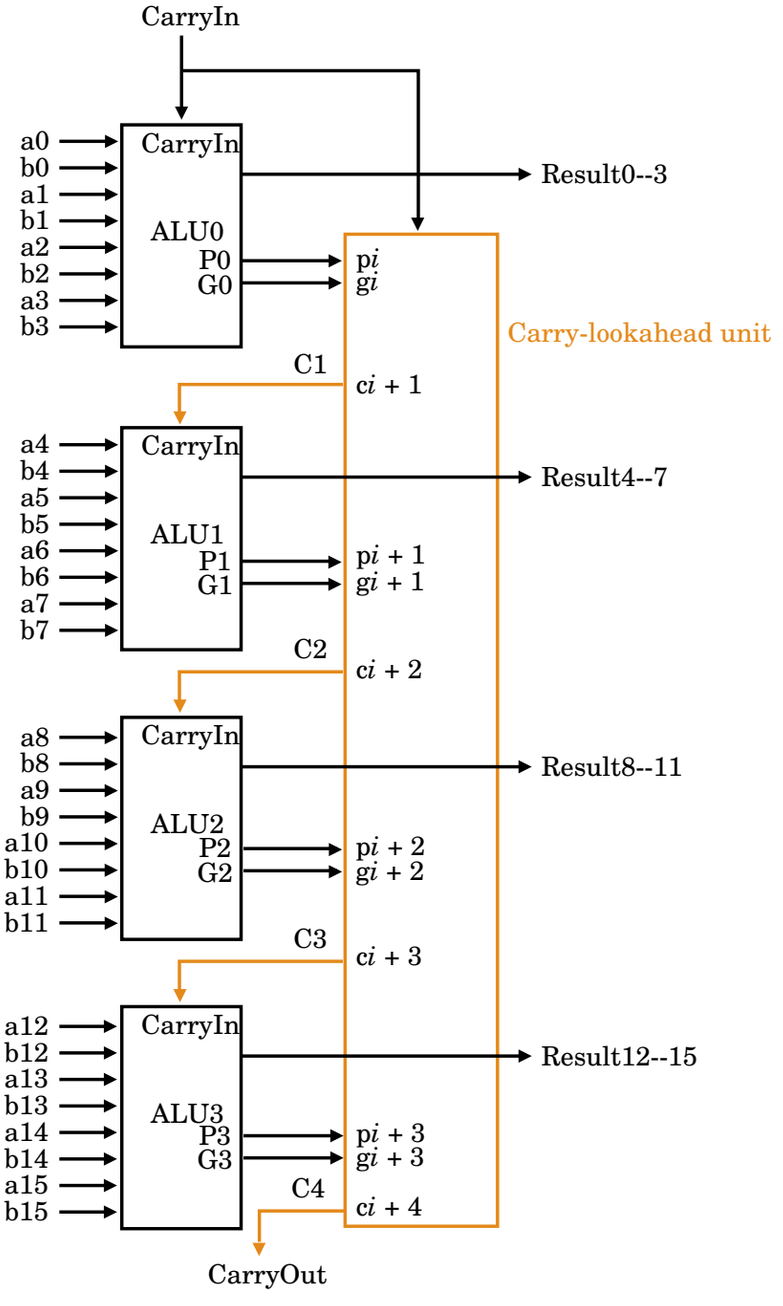
- Ecuaţiile finale pentru deplasări pe blocuri de 4 biţi sunt similare, dar folosesc generările şi propagările pe blocuri P, G:

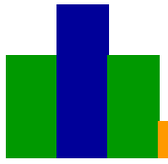
$$Ci_1 = G_0 + P_0 \cdot Ci_0$$
$$Ci_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot Ci_0$$
$$Ci_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot Ci_0$$
$$Ci_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot Ci_0$$

..Adunare rapida

Sumator cu carry-lookahead:

- In fine, desenul arată schema finală a sumatorului.
- Exemplul prezintă cazul unui sumator pe 16 biţi format din 4 ALU pe 4 biţi.



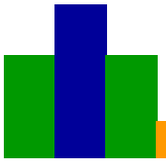


..Adunare rapida

Sumator cu carry-lookahead: Analiză:

- Să considerăm cazul unui sumator cu 16 biţi.
- In cazul secvenţial (ripple-carry adder), sunt necesare 16 adunări, fiecare necesitând 2 nivele de logică (porti AND, apoi OR). Deci timpul este de $16 \times 2 = 32$ unităţi.
- Cu carry-lookahead adder:
  - 1 nivel de logică este folosit pentru p_k, g_k;
  - 2 nivele de logică în `Carry-lookahead` pentru C_k;
  - 2 nivele de logică în fiecare ALU (pentru rezultatul final).
- Obţinem, în principiu, un sumator de $32/5 = 6.4$ ori mai rapid.

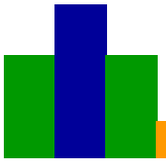
Nota: Sa notam ca sumatorul carry-lookahead este mai rapid, dar necesita mult mai multe porti, in particualar porti AND si OR generalizate.



Aritmetica pentru calculator

Cuprins:

- Numere (cu si fara semn)
- Adunare si scadere
- Operatii logice
- Unitatea aritmetica si logica
- Adunare rapida
- *Inmultire*
- *Impartire*
- *Operatii cu numere reale*
- *Concluzii, diverse, etc.*



Partea II

A se insera...