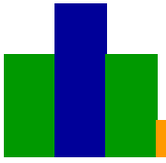
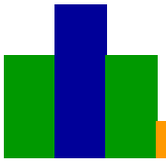


Lecţia 3:
Limbajul maşinii de calcul

G Stefănescu — Universitatea Bucureşti

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

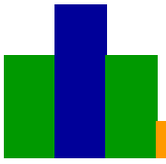
După: D. Patterson and J. Hennessy, Computer Organisation and Design



Limbajul masinii de calcul

Cuprins:

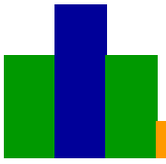
- *Operatii (aritmetice, logice, etc.)*
- Operanzi (date)
- Reprezentarea instructiunilor in hardware
- Instructiuni de decizie
- Suport hardware pentru proceduri
- Concluzii, diverse, etc.



MIPS

Instructiuni in limbajul masinii / MIPS:

- Maşina de calcul are un limbaj propriu folosind *instrucţiuni* dintr-un *set de instrucţiuni* specificat.
- Vom folosi MIPS-ul - un set tipic pentru limbajele de cod maşină actuale.



Operatii simple

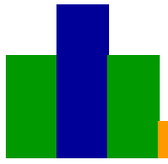
Operatii:

- operaţii aritmetice simple
- *cu 3 operanzi*: 2 argumente + 1 rezultat

Exemple:

Tip	Instructiune	Exemple	Semantica	Comentarii
Arit.	add	`add $s1,$s2,$s3`	`$s1=$s2+$s3`	3 operanzi
Arit.	subtract	`sub $s1,$s2,$s3`	`$s1=$s2-$s3`	3 operanzi

Principiul I: Simplitatea favorizează uniformitatea.
Avand operatii mai simple, arhitectura este mai uniforma, usor de folosit si eficientizat, cu grad mare de scalabilitate.



MIPS - operatii aritmetice

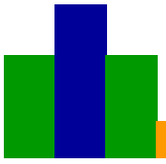
Instructiuni aritmetice:

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

  Semantică: Se adună conţinutul regiştrilor `reg2, reg3`, iar rezultatul se pune în registrul `reg1`.

- `sub reg1, reg2, reg3` (*subtract*)

  Semantică: Se scade conţinutul regiştrului `reg3` din conţinutul registrului `reg2`, iar rezultatul se pune în registrul `reg1`.



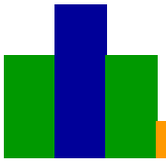
Exemplu

Exemplu:

```
f = (g+h)-(i+j);
```

se translateaza in

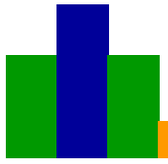
```
add t0,g,h;                          # var temp t0 contine g + h
add t1,i,j;                          # var temp t1 contine i + j
sub f,t0,t1;          # f este t1 - t0, deci (g + h) - (i + j)
```



Limbajul masinii de calcul

Cuprins:

- Operatii (aritmetice, logice, etc.)
- *Operanzi (date)*
- Reprezentarea instructiunilor in hardware
- Instructiuni de decizie
- Suport hardware pentru proceduri
- Concluzii, diverse, etc.



Registri

Operanzi (acces direct): De unde luăm datele (operanzii)?

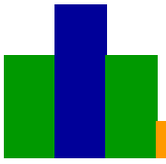
- Putem folosi *regiştri* specifici (de la 0 la 31), pentru access rapid.
- Datele se accesează *direct* (sunt în procesor).

Exemplul anterior devine

```
add $t0,$s1,$s2;                  # reg temp $t0 contine g + h
add $t1,$s3,$s4;                   # reg temp t1 contine i + j
sub $s0,$t0,$t1;  # f este $t1 - $t0, deci (g + h) - (i + j)
```

Principiul II: Mai mic inseamna mai rapid.

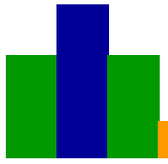
Operatiile nu se fac direct pe datele din memorie, ci pe cele din registri; un numar mic de registri favorizeaza o frecventa de ceas mai mare.



Memorie

Operanzi (acces indirect):

- Dacă datele sunt în memorie, procedura este indirectă, folosind instrucţiuni de *transfer de date* (data transfer);
- Pentru aceasta avem nevoie de *adresa de memorie*;
- Transferul: *Memorie* $\rightarrow$ *Registri* se numeşte *încărcare* (load); instrucţiunea este `lw` (load word);
- Transferul: *Registri* $\rightarrow$ *Memorie* se numeşte *memorare* (store); instrucţiunea este `sw` (store word);



Accesarea memoriei

Exemplu: Dacă variabilele `g,h` sunt în regiştri `$s1,$s2`, iar adresa de început a vectorului `A` este în `$s3`,

```
g = h + A[8];
```

se translateaza in

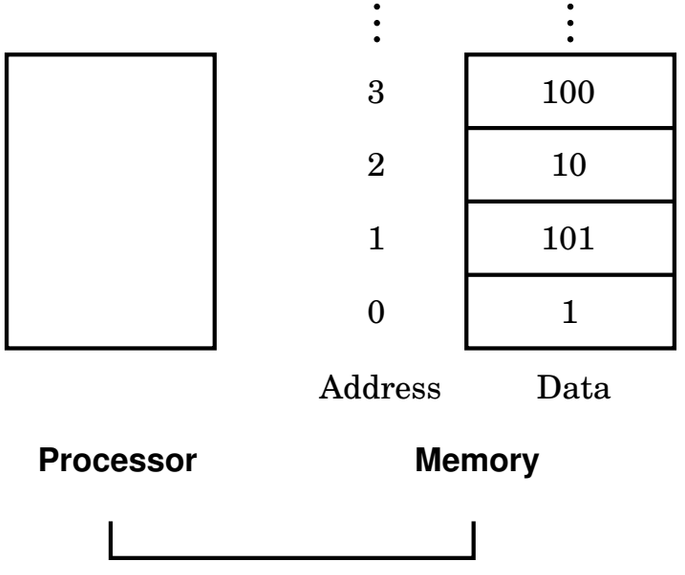
```
lw $t0,8($s3);                    # reg temp $t0 contine A[8]
add $s1,$s2,$t0;                                  g = h + A[8]
                (format provizoriu, cu deplasare neajustată)
```

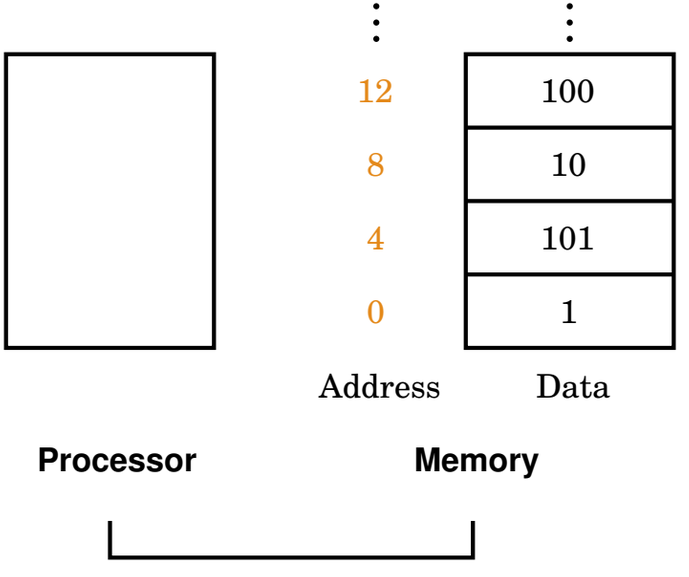
Intr-o instrucţiune de tipul `lw $t0,8($s3)`, adresa de început a vectorului este cea data de *registrul de bază* (`$s3`), iar constanta (`8`) reprezintă *deplasarea* (offset-ul) de la începutul vectorului la elementul curent.

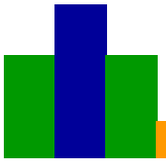
Adresarea memoriei

Adresarea memoriei:

- în stânga, avem adresare directă, prin numere/indici;
- în dreapta, avem adresare prin adrese de memorie;
- multe arhitecturi ofera adresa la nivel de *octet* (byte - B) (1 byte = 8b (biţi); 1 *cuvânt* (word) = 4B); *restrictie de aliniere* - în MIPS adresa cuvintelor începe la un multiplu de 4.





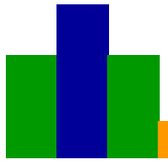


MIPS - instructiuni de transfer

Instructiuni MIPS (noi):

Tip	Instructiune	Exemple	Semantica	Comentarii
Tr.	load word	`lw $s1,100($s2)`	`$s1= Mem[$s2+100]`	Date din mem. in reg.
Tr.	store word	`sw $s1,100($s2)`	`Mem[$s2+100]= $s1`	Date din reg. in mem.

Legenda: Tr. = Instructiune de transfer de date.



..MIPS - instructiuni de transfer

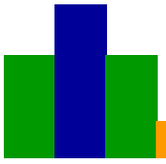
Instructiuni de transfer:

- `lw reg1, cons(reg2)` (*load word*)

  Semantică: Se adună `cons` la conţinutul regiştrului `reg2`. Rezultatul se interpretează ca adresă de memorie, iar cuvântul aflat acolo se copiază în `reg1`.

- `sw reg1, cons(reg2)` (*store word*)

  Semantică: Se adună `cons` la conţinutul regiştrului `reg2`. Rezultatul se interpretează ca adresă de memorie, iar conţinutul lui `reg1` se copiază la acea adresă.



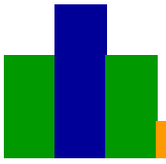
Operatii in memorie

Exemplu: Dacă variabila `h` este în regiştrul `$s2`, iar adresa de început a vectorului `A` este în `$s3`,

```
A[12] = h + A[8];
```

se translateaza in

```
lw $t0,32($s3);                  # reg temp $t0 contine A[8]
add $t0,$s2,$t0;               # reg temp $t0 contine h + A[8]
sw $t0,48($s3);                  # memoreaza h + A[8] in A[12]
                                                 (format corect)
```



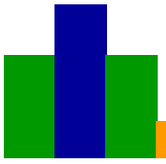
Operatii cu vectori

Exemplu: Dacă variabilele `g,h,i` sunt în regiştri `$s1,$s2,$s4`, iar adresa de început a vectorului `A` este în `$s3`,

```
g = h + A[i];
```

se translateaza în

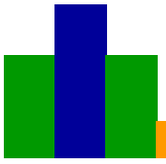
```
add $t1,$s4,$s4;                    # reg temp $t1 = 2 * i
add $t1,$t1,$t1;                          # $t1 = 4 * i
add $t1,$t1,$s3;          # $t1 contine adresa lui A[i]
lw $t0,0($t1);              # reg temp $t0 = A[i]
add $s1,$s2,$t0;           # reg temp g = h + A[i]
```



Operanzi MIPS, pe scurt

Operanzi MIPS:

Nume	Exemple	Comentarii
32 registri	`$t0-$t7, $s0-$s7, etc.`	Locatii pentru acces rapid la date - registri. Operatiile aritmetice se fac numai cu datele din registri.
2^{30} cuvinte de memorie	`Mem[0], ..., Mem[4294967292]`	Date accesate de instructiunile de transfer. Se foloseste acces la octet (byte). Adresele cuvintelor sunt multipli de 4.

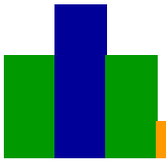


Interfata Hardware-Software

Interfata Hardware-Software:

Regiştri vs. memorie:

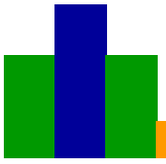
De regulă există mai multe variabile decât registri.
Variabilele mai putin utilizate se pun în memorie.
Există un permanent transfer între registri si memeorie care trebuie eficientizat.



Limbajul masinii de calcul

Cuprins:

- Operatii (aritmetice, logice, etc.)
- Operanzi (date)
- *Reprezentarea instructiunilor in hardware*
- Instructiuni de decizie
- Suport hardware pentru proceduri
- Concluzii, diverse, etc.



Reprezentarea instructiunilor in calculator

Formatul instructiunilor, exemplu:

- In calculator se foloseşte *baza 2* (semnale electrice înalte ori joase).
- Există o convenţie de translatare a numelor regiştrilor în *numere*; e.g., regiştri `s0-$s7` au numerele 16-23; regiştri `t0-$t7` au numerele 8-15; etc.

Exemplu: Codul instrucţiunii

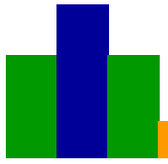
```
add $t0,$s1,$s2;
```

este

0	17	18	8	0	32

ori în binar,

000000	10001	10010	01000	00000	100000



..Reprezentarea instructiunilor

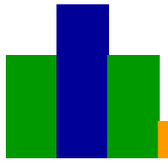
Formatul instructiunilor hardware:

- *Formatul-R* conţine 6 câmpuri

6 biti	5 biti	5 biti	5 biti	5 biti	6 biti
op	rs	rt	rd	shamt	funct

Câmpurile MIPS au următoarele semnificaţii *aici*:

- *op*: este operaţia de bază din instrucţiune - “opcod”;
- *rs*: registrul pentru primul argument;
- *rt*: registrul pentru al doilea argument;
- *rd*: registrul pentru rezultat (destinaţie);
- *shamt*: “shift amount” (folosit la operaţiile de deplasare (shiftare));
- *func*: funcţia; combinată cu `op` indica ce funcţie se aplică;



..Reprezentarea instructiunilor

Problemă: Dacă o instrucţiune necesită câmpuri de lungime mai mare, cum le poate obţine?

Solutie: Se face un compromis între nevoia de format fix (32b) şi nevoia de flexibilitate a formatului instrucţiunilor, anume

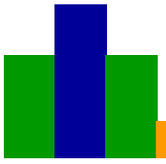
Avem mai multe formate de instrucţiuni, toate pe 32b, folosind aceleaşi câmpuri, eventual combinate.

Formatul instructiunilor hardware:

- *Formatul-I* combină ultimile 3 câmpuri, anume este

6 biti	5 biti	5 biti	16 biti
op	rs	rt	address

Adresa variază între $\{-2^{15}, +2^{15}\}$*, deci se pot accesa 32768 B (octeti).*



..Reprezentarea instructiunilor

Formatul-I al instructiunilor: Exemplu

```
lw $t0,32($s3);
```

este (`$s3 = 19; $t0 = 8`)

35	19	8	32

ori în binar,

100011	10011	01000	0000000000100000

Să notăm câteva diferenţe

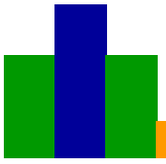
- Registrul de argumente `rt` (al doilea) din formatul-R devine registru de rezultat în formatul-I.
- Operaţia este complet definită de câmpul `op`.
- Codul include o valoare *imediată* ce se poate utiliza direct.

..Reprezentarea instructiunilor

Formatul instructiunilor:

Coduri binare pentru instrucţiunile de până acum

Instructiune	Format	op	rs	rt	rd	shamt	func
`add`	R	0	reg	reg	reg	0	32
`sub`	R	0	reg	reg	reg	0	34
`lw`	I	35	reg	reg	address		
`sw`	I	43	reg	reg	address		



Cod binar

Exemplu: Dacă variabila `h` este în regiştrul `$s1`, iar adresa de început a vectorului `A` este în `$t1`,

```
A[300] = h + A[300];
```

se translateaza în

```
lw $t0,1200($t1);              # reg temp $t0 contine A[300]
add $t0,$s2,$t0;                    # $t0 contine h + A[300]
sw $t0,1200($t1);       # se memoreaza h + A[300] in A[300]
```

..Cod binar

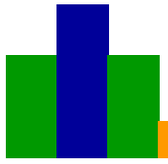
Exemplu (cont.)

In cod numeric este

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

iar în cod binar este

100011	01001	01000	00000	10010	110000
000000	10010	01000	01000	00000	100000
101011	01001	01000	00000	10010	110000

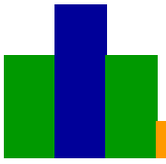


Principii de bază

Principii de bază:

- Instrucţiunile se reprezintă ca numere.
- Programele se pot pune în memoria calculatorului, citi, ori modifica, ca şi numerele.
- Rezultatul: conceputul de *program-memorat*.
- Rezultat teoretic: *probleme nerezolvabile* (problema opririi - aplicarea unui program lui însuşi duce la contradicţii)

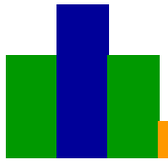
Multe avantaje: Datele si programele sunt la fel; în consecinta, hardware-ul se simplifica; pot aplica programe pe alte programe, privite ca date (e.g., compilatoare); etc.



Limbajul masinii de calcul

Cuprins:

- Operatii (aritmetice, logice, etc.)
- Operanzi (date)
- Reprezentarea instructiunilor in hardware
- *Instructiuni de decizie*
- Suport hardware pentru proceduri
- Concluzii, diverse, etc.



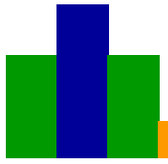
Instructiuni de decizie

Instructiuni de decizie:

- Programele autonome (în particular cele memorate) pot
  - alege între mai multe acţiuni posibile;
  - repeta instrucţiuni decizând când se termină; etc.
- Construcţiile de bază pentru astfel de programe sunt `if`
  - `if` (conditie booleana) `then` (instructiune) `else` (instructiune);

şi `goto`:

- (eticheta) `goto` (eticheta);



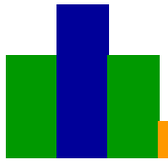
Universalitatea programelor

Universalitatea programelor:

- Teoretic, se poate demonstra că *programele cu regiştri* de tipul *plus-1 / minus-1 / test-la-0* sunt *universale*;
- Mai precis, folosind regiştri (ca variabile peste numere naturale) şi instrucţiuni de tipul

```
et:  x = x + 1;
et:  x = x - 1;
et:  if ( x == 0) goto et1 else goto et2;
```

  putem reprezenta toate programele posibile;
- Instrucţiunea `goto` a fost în mare eliminată din limbajele de programare curente (avem *"programare structurata"*), dar limbajele de tip maşină încă o folosesc frecvent.



MIPS - instructiuni de decizie

Instructiuni de decizie:

Ramificatii conditionale (conditional branches)

- `beq reg1, reg2, L1` (*branch if equal*)

  Semantică: Se trece la instrucţiunea cu eticheta `L1`, dacă valorile din regiştri `reg1` şi `reg2` sunt egale.

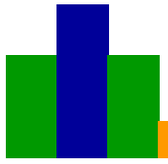
- `bne reg1, reg2, L1` (*branch if not equal*)

  Semantică: Se trece la instrucţiunea cu eticheta `L1`, dacă valorile din regiştri `reg1` şi `reg2` nu sunt egale.

Comparare şi asignare

- `slt reg1, reg2, reg3` (*set on less than*)

  Semantică: Se compară conţinutul registrului `reg2` cu al conţinutul registrului `reg3`. Dacă este mai mic, `reg1` se face `1`, altfel se face `0`.



..MIPS - instructiuni de decizie

Instructiuni de decizie (cont.)

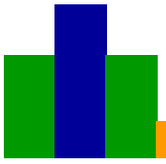
Salt neconditionat

- `j L1` (*jump*)

  Semantică: Se face salt necondţionat la instrucţiunea cu eticheta `L1`.

- `j reg1` (*jump register*)

  Semantică: Se face salt necondţionat la instrucţiunea a cărei adresă de memorie este conţinută în registrul `reg1`.



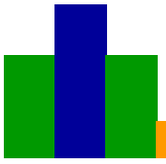
If

Exemplu: Dacă `f,g,h,i,j` sunt variabile memorate în regiştri `$s0-$s4` codul C

```
      if (i == j) go to L1;
      f = g + h;
L1:   f = f - i;
```

se translateaza în

```
      beq $s3,$s4,L1;              # treci la L1 daca i egal j
      add $s0,$s1,$s2;     # f = g + h (sarita cand i egal j)
L1:   sub $s0,$s0,$s3;    # f = f - i (executata intotdeauna)
```

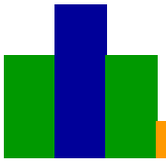


Interfata Hardware-Software

Interfata Hardware-Software:

Etichete:

Compilatoarele
crează etichete,
foloseste ramificatii, goto, etc.
Programarea la nivel înalt
evită acest lucru,
devenind mult mai simplă.



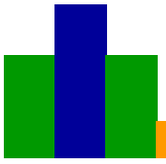
If-then-else

Exemplu: Dacă `f,g,h,i,j` sunt variabile memorate în regiştri `$s0-$s4` codul C *if-then-else*

```
if (i == j) f = g + h; else f = g - h;
```

se translateaza în

```
        bne $s3,$s4,Else;           # treci la Else daca i ≠ j
        add $s0,$s1,$s2;      # f = g + h (sarita cand i ≠ j)
        j Exit;                                 # treci la Exit
Else:   sub $s0,$s1,$s2;       # f = g - h (sarita cand i = j)
Exit:
```



Reprezentarea instructiunilor in calculator

Observatie: Instrucţiunea de salt necondiţionat are ca unic argument adresa, deci se pot folosi mai multe câmpuri din codul instrucţiunii.

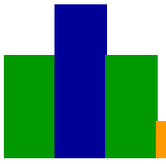
Formatul instructiunilor hardware:

- *Formatul-J* combină ultimile 5 câmpuri, anume este

6 biti	26 biti
op	address

Adresa variază acum între $\{-2^{25}, +2^{25}\}$.

- Spre deosebire de instrucţiunea de salt `j` de format-J, instrucţiunea de salt la registru `jr` are formatul-R.



Repetitii

Exemplu (bucle): Dacă `g,h,i,j` sunt variabile memorate în regiştri `$s1-$s4`, iar `A` este vector cu 100 elemente cu adresa de început în `$s5` codul C pentru o *bucla*

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

se translateaza în

```
Loop  add $t1,$s3,$s3;          # reg temp $t1 = 2 * i
      add $t1,$t1,$t1;                  # $t1 = 4 * i
      add $t1,$t1,$s5;         # $t1 = adresa lui A[i]
      lw $t0,0($t1);             # reg temp $t0 = A[i]
      add $s1,$s1,$t0;                  # g = g + A[i]
      add $s3,$s3,$s4;                     # i = i + j
      bne $s3,$s2,Loop;     # treci la Loop daca i ≠ h
```

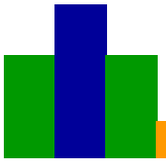
While

Exemplu (while): Dacă `i,j,k` sunt variabile memorate în regiştri `$s3-$s5`, iar `A` este vector cu 100 elemente cu adresa de început în `$s6` codul C pentru o bucla *while*

```
while (A[i] == k);
    i = i + j;
```

se translateaza în

```
Loop:  add $t1,$s3,$s3;          # reg temp $t1 = 2 * i
       add $t1,$t1,$t1;                  # $t1 = 4 * i
       add $t1,$t1,$s6;        # $t1 = adresa lui A[i]
       lw $t0,0($t1);            # reg temp $t0 = A[i]
       bne $t0,$s5,Exit;  # treci la Exit daca A[i] ≠ k
       add $s3,$s3,$s4;                    # i = i + j
       j Loop                             # go to Loop
Exit:
```

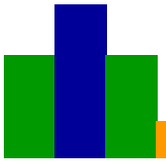


Interfata Hardware-Software

Interfata Hardware-Software:

Blocuri de bază:

Secventele de instructiuni
fără ramificatii
sunt fundamentale (blocuri de bază).
Compilarea
începe cu identificarea
acestor blocuri.



Switch

Exemplu (switch): Dacă `f-k` sunt variabile memorate în regiştri `$s0-$s5`, iar `$t2` conţine `4` codul C pentru o instrucţiune *switch*

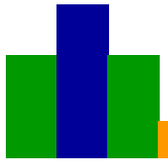
```
switch (k) {
    case 0:  f = i + j; break;
    case 1:  f = g + h; break;
    case 2:  f = g - h; break;
    case 3:  f = i - j; break;
}
```

se translateaza după cum urmează (presupunem `$t4` conţine adresa de început a unui zone ce conţine etichetele `L1-L4`):

Switch

Exemplu (switch) (cont.)

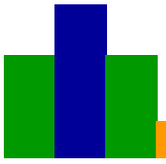
```
        slt $t3,$s5,$zero;                      # test k < 0
        bne $t3,$zero,$Exit;      # daca k < 0, treci la Exit
        slt $t3,$s5,$t2;                          test k > 4
        beq $t3,$zero,$Exit;      # daca k > 4, treci la Exit
        add $t1,$s5,$s5;             # reg temp $t1 = 2 * k
        add $t1,$t1,$t1;                        # $t1 = 4 * k
        add $t1,$t1,$t4;       # $t1 = adresa tabelei Jump[k]
        add $t0,0($t1);             # reg temp $t0 = Jump[k]
        jr $t0;                      # salt bazat pe reg $t0
L0:     add $s0,$s3,$s4              # k = 0, deci f = i + j
        j Exit                                # treci la Exit
L1:     add $s0,$s1,$s2              # k = 1, deci f = g + h
        j Exit                                # treci la Exit
L2:     sub $s0,$s1,$s2              # k = 2, deci f = g - h
        j Exit                                # treci la Exit
L3:     sub $s0,$s3,$s4              # k = 3, deci f = i - j
Exit:
```



Limbajul masinii de calcul

Cuprins:

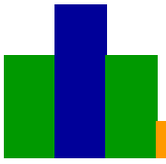
- Operatii (aritmetice, logice, etc.)
- Operanzi (date)
- Reprezentarea instructiunilor in hardware
- Instructiuni de decizie
- *Suport hardware pentru proceduri*
- Diverse, concluzii, etc.



Proceduri

Proceduri: Procedurile necesită următoarele operaţii:

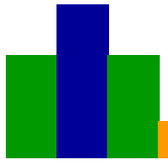
- *plasarea parametrilor* undeva spre a fi accesaţi de procedură;
- *transferul controlului* spre procedură;
- asigurarea de *resurse de memorie* pentru procedură;
- *executarea* codului procedurii;
- *plasarea rezultatului* undeva pentru a fi accesat de programul apelant;
- *returnarea controlului* spre programul apelant.



Suport MIPS pentru proceduri

Suport MIPS pentru proceduri: MIPS suportă invocarea de proceduri prin câteva convenţii:

- alocă 4 regiştrii `$a0-$a3` pentru a transfera *argumentele*;
- alocă 2 regiştri `$v1-$v2` pentru returnarea *rezultatelor*;
- alocă un registru `$ra` pentru *adresa instrucţiunii de return* după efectuarea procedurii.

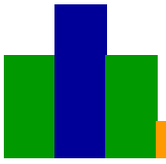


MIPS - instructiuni pentru proceduri

Jump and link: In plus, MIPS are o instrucţiune specială

- `jal Proc` (*jumn and link*)

  Semantică: Se trece la procedura `Proc` (prima instrucţiune din procedură). Adresa de continuare a programului apelant se reţine în registrul `$ra`. Procedura foloseşte o instrucţiune de terminare `jr $ra` care face acest transfer al controlului inapoi in programul apelant.

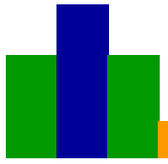


Registri suplimentari

Registri suplimentari:

- Pentru a putea continua calcul după invocarea procedurii trebuie să *salvăm* valorile vechilor regiştri spre a fi restaurate.
- Structura de date folosită este o *stivă* (stack), cu cele două operaţii *pune* (push) şi *scoate* (pop); o stiva are ordinea LIFO (last-in-first-out), anume se scot ultimele elemente puse.
- Pentru a gestiona stiva se foloseşte un registru special `$sp` (*stack pointer*).

Din motive istorice, stiva creste de la adrese mari la adrese mici.

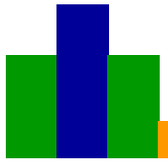


Translatarea unei proceduri

Exemplu (procedură): Procedura cu codul C

```
int lex (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

se translateaza după cum urmează (`g,h,i,j` vor fi în regiştri `$a0,$a1,$a2,$a3`; pentru `f` folosim `$s0`; presupunem cunoscut codul pentru adunat ori scăzut `4,8,12`):



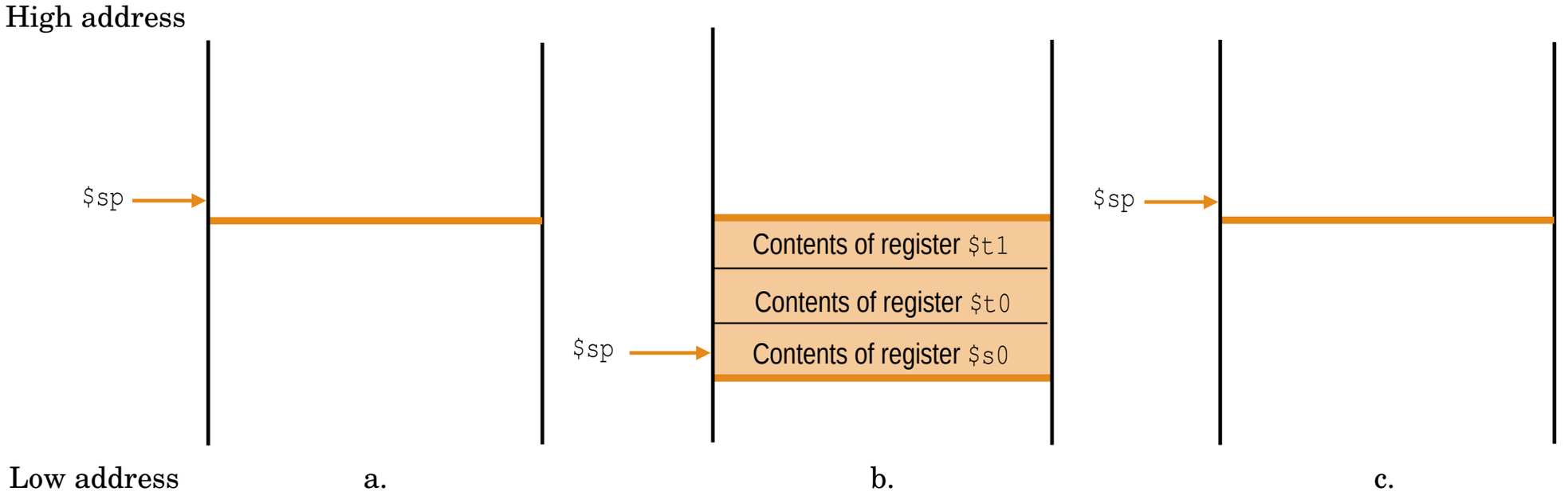
..Translatarea unei proceduri

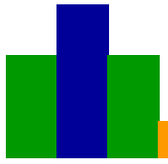
Exemplu (procedura) (cont.)

```
lex:
        sub $sp,$sp,12;          # spatiu in stiva pentru 3 elem
        sw $t1,8($sp);             # salveaza continutul lui $t1
        sw $t0,4($sp);             # salveaza continutul lui $t0
        sw $s0,0($sp);             # salveaza continutul lui $s0
        add $t0,$a0,$a1;                          # $t0 = g + h
        add $t1,$a2,$a3;                          # $t1 = i + j
        add $s0,$t0,$t1;                # f = (g + h) - (i + j)
        add $v0,$s0,$zero;                              # return f
        lw $s0,0($sp);                        # restaureaza $s0
        lw $t0,4($sp);                        # restaureaza $t0
        lw $t1,8($sp);                        # restaureaza $t1
        add $sp,$sp,12;                 # sterge 3 elem din stiva
        jr $ra;                       # salt inapoi in prog apelant
```

Stack pointer

Stack-pointer: Figura indică stiva şi pointer-ul *înainte*, *în timpul*, şi *după* apelarea procedurii.



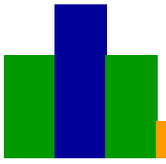


MIPS - registri suplimentari

MIPS - registri suplimentari: Pentru a evita salvarea şi restaurarea regiştrilor, MIPS face următoarele convenţii

- `$t0-$t9`: sunt 10 *regiştri temporari* ale căror valori *nu trebuie* conservate de procedura apelată;
- `$s0-$s7`: sunt 8 *regiştri salvati* ale căror valori *trebuie* conservate de procedura apelată (dacă sunt folosiţi);

In exemplul anterior doar `$s0` *trebuie salvat & restaurat.*



Frame pointer

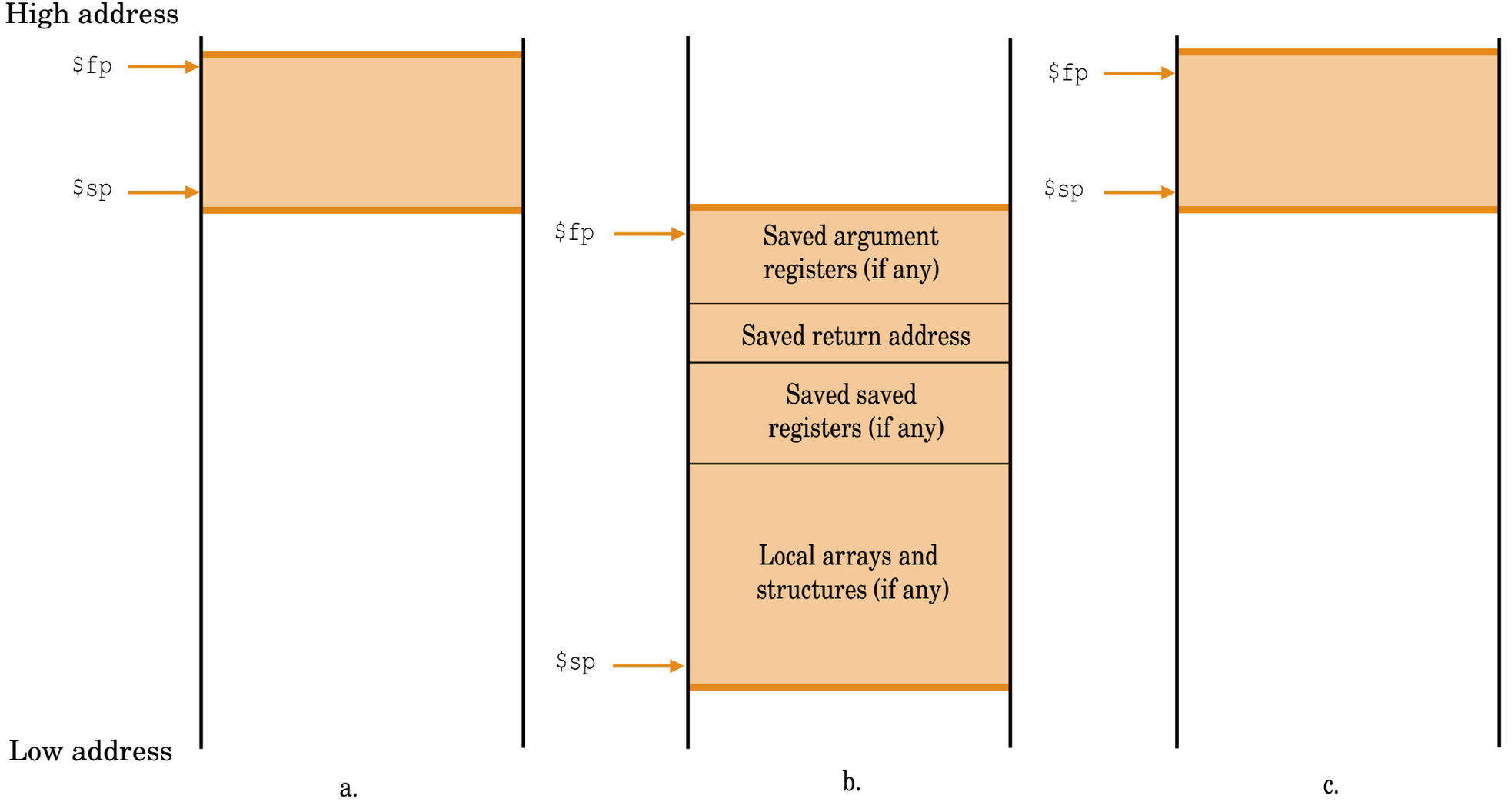
Frame pointer:

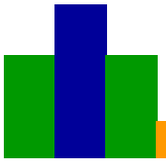
- Procedura poate necesita si date suplimentare care trebuiesc memorate în stiva (vectori, etc.).
- Se foloseşte încă un registru special `$fp` (*frame pointer*): structura dintre frame-pointer şi stack-pointer care conţine regiştrii salvaţi şi variabilele locale se numeşte *procedure frame* (cadrul procedurii).

Conventia MIPS este de a pune extra-parametrii procedurii (daca sunt) imediat sub pointer-ul `$fp`.

..Frame pointer

Frame pointer (cont.) Stiva şi stack- şi frame-pointerii *înainte*, *în timpul*, şi *după* apelarea unei proceduri.

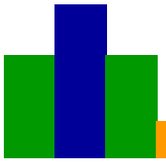




Operanzi MIPS, pe scurt

Operanzi MIPS:

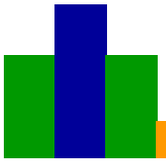
Nume	Exemple	Comentarii
32 registri	`$zero(0), $at(1), $v0-$v1(2-3), $a0-$a4(4-7), $t0-$t7(8-15), $s0-$s7(16-23), $t8-$t9(24-25), $k0-$k1(26-27), $gp(28), $sp(29), $fp(30), $ra(31)`	Locatii pentru acces rapid la date; `$zero` contine 0 `$v0-$v1`: valori pentru rezultate `$a0-$a4`: argumente `$t0-$t7;$t8-$t9`: valori temporare `$s0-$s7`: valori temporare salvate `$k0-$k1`: pentru sistemul de operare `$at,$gp,$sp,$fp,$ra`: asamblor, pointer global, pointer stiva, pointer cadru, adresa intoarcere
2^{30} cuvinte de memorie	`Mem[0], ..., Mem[4294967292]`	Date accesate de instructiunile de transfer; Se foloseste acces la octet (byte), deci adresele cuvintelor sunt multiplu de 4.



Limbajul masinii de calcul

Cuprins:

- Operatii (aritmetice, logice, etc.)
- Operanzi (date)
- Reprezentarea instructiunilor in hardware
- Instructiuni de decizie
- Suport hardware pentru proceduri
- *Concluzii, diverse, etc.*



Concluzii, diverse, etc.

A se insera...