

Compilatoare

Alocarea regiștrilor
Optimizări de nivel scăzut
Linking



Optimizari de nivel scazut

- Planificarea instructiunilor
- Alocarea registrilor
- Optimizarea fluxului de control
- Peephole

Alocarea registrilor

- Programele utilizeaza 'valori' – care sunt definite/calculate intr-un loc si folosite in mai multe locuri; trebuie stocate intre definiri si folosiri.
 - Prima optiune: stocare in memorie la fiecare definire, incarcare din memorie la fiecare folosire
 - Mai bine: stocare si folosire direct din registri (mult mai rapid, code size de obicei mai bun, mai putina putere consumata – dar numarul de registri este limitat).
- Scopul alocarii registrilor (ca optimizare) este reducerea traficului cu memoria/folosirea cat mai eficienta a registrilor pusi la dispozitie de procesor
- Scopul NU este sa se foloseasca un numar minim de registri pentru un anumit program
- Probabil optimizarea cu cel mai mare impact dintre toate
- Toate optimizarile trebuie sa coopereze pentru ca rezultatul RA sa fie bun

Alocarea registrilor (cont)

- Ce punem in registri
 - Temporari generati de compilator
 - Bucati din array-uri / structuri
(scalar replacement of aggregates)
 - Variabile locale definite in program
Pot fi refolosite in scopuri diferite (ex: contoare de bucla)

```
for i = ...  
  ... = i
```

```
for i = ...  
  ... = i
```



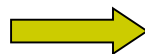
```
for i1 = ...  
  ... = i1
```

```
for i2 = ...  
  ... = i2
```

Web-uri

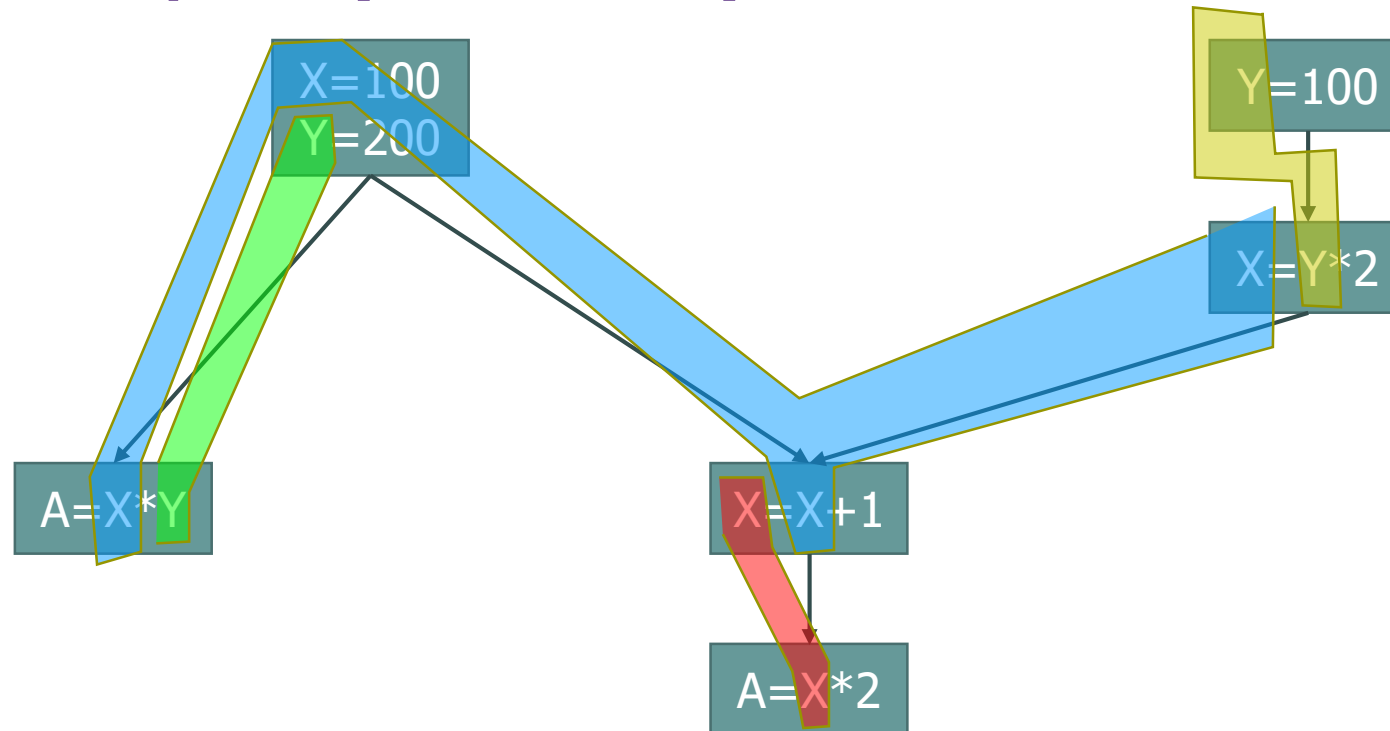
- O modalitate de “renaming”
- Contin pentru o definire, toate folosirile ei
- Daca doua definiri ‘ajung’ la aceeasi folosire (o folosire e expusa la doua definiri) atunci ele sunt in acelasi web
- Analogii / diferente fata de SSA?

```
if (...)  
  X = ...  
else  
  X = ...  
... = X  
  
X = ...  
... = X
```



```
if (...)  
  X1 = ...  
else  
  X1 = ...  
... = X1  
  
X2 = ...  
... = X2
```

Exemplu (web-uri)



- Def-use chains pentru fiecare variabilă
- O componentă conexă formează un web

Abordari pentru RA

- Alocare locala
 - Valoarea web-urilor se tine in memorie la inceputul si sfarsitul fiecarui basic block
 - Se aloca pe rand web-urile folosite la registri disponibili
 - Cand nu mai sunt registri disponibili – se de-aloca (se scrie la loc in memorie) web-ul a carui urmatoare folosire e cea mai indepartata, dintre toate web-urile ‘alocate’
- Programare liniara
 - Se formuleaza problema ca o problema de programare liniara pe numere intregi (ILP)
 - Se rezolva (optim) folosind un ‘solver’ disponibil
 - Foarte lent – nu prea e folosibil in practica
- Cea mai frecvent folosita metoda – colorarea grafurilor (Chaitin)
 - Fiecare web e un nod, se coloreza graful cu N culori unde N e nr. de registri

Colorarea grafurilor

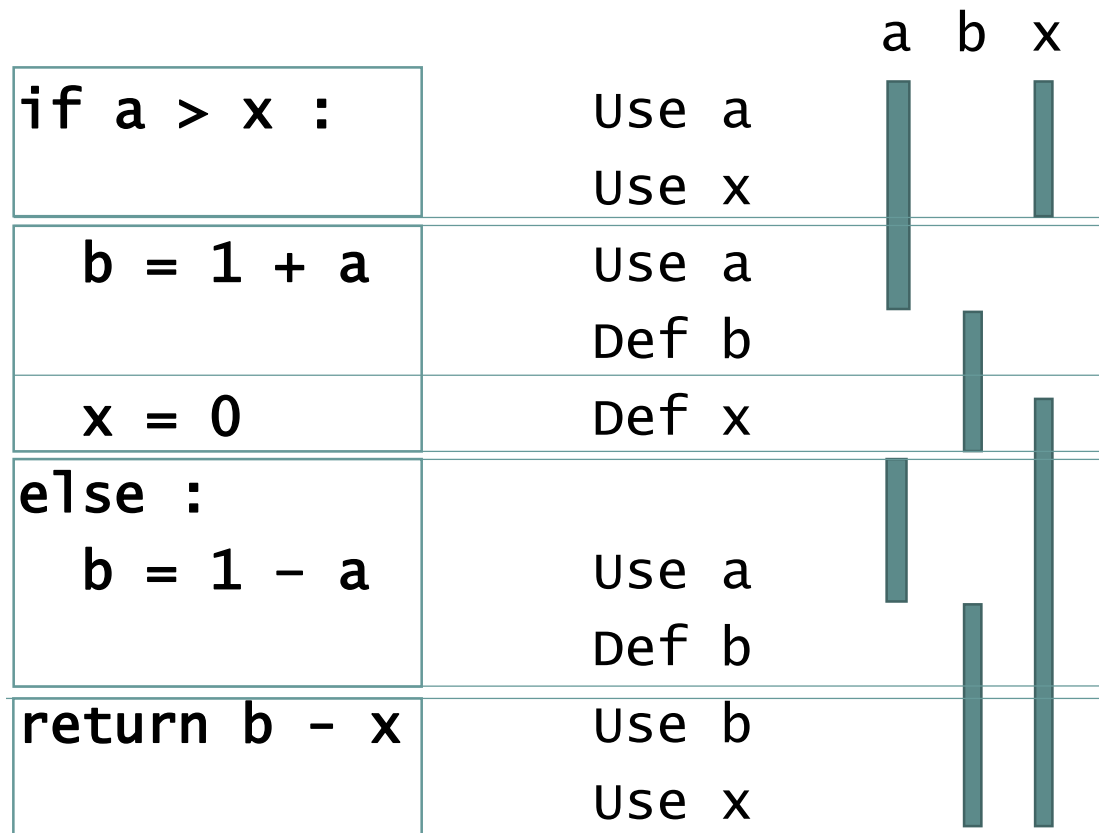
- Fiecare variabilă (web) este un nod într-un graf
- Se colorează graful cu N culori, unde N este numărul de regiștri
 - problema clasică în teoria grafurilor
 - NP, dar exista euristici destul de bune
- Dacă graful nu se poate colora, se elimină noduri
- Fiecare variabilă va fi memorată în registrul ce are culoarea asociată nodului.
- Variabilele corespunzătoare nodurilor eliminate vor fi ținute în memorie și nu în regiștri (*spill*)

Cum calculăm muchiile?

Seturi convexe/live ranges

- Un set S de instrucțiuni este convex atunci când pentru oricare A, B din S , dacă există o cale fezabilă de la A la B ce include o instrucțiune C , atunci C e și ea în S
- Live range-ul unui web – setul convex minimal de instrucțiuni care include toate definițiile și utilizările
 - Intuitiv: “zona din program unde web-ul e în viață”
- Calcul: analiza variabilelor în viață (*live vars*)
- Numărul de variabile în viață simultan: presiunea pe regiștri (*register pressure*)

Live ranges – exemplu

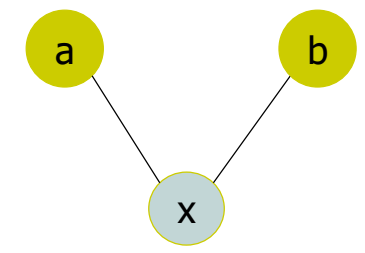


Graful de interferenta

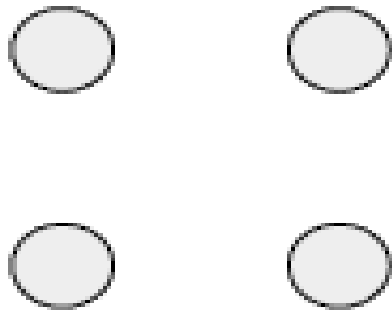
- Daca live-range-urile a doua web-uri se intersecteaza, spunem ca web-urile interfera
 - nu pot folosi acelasi registru
 - arc in graful de interferenta
(nu pot primi aceeasi culoare...)

Exemplu

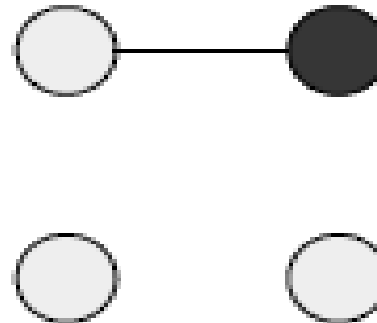
		a	b	x
if a > x :	Use a			
	Use x			
b = 1 + a	Use a			
	Def b			
x = 0	Def x			
else :				
b = 1 - a	Use a			
	Def b			
return b - x	Use b			
	Use x			



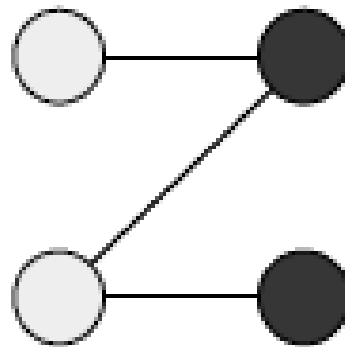
Exemple de colorare



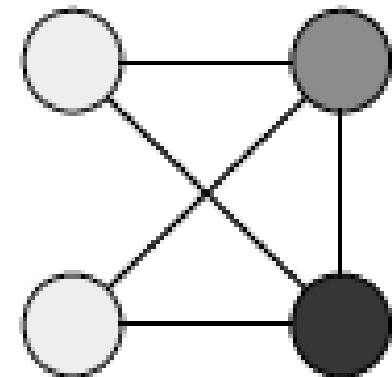
1 Color



2 Colors



Still 2 Colors



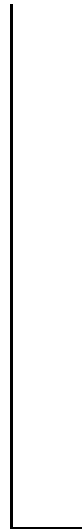
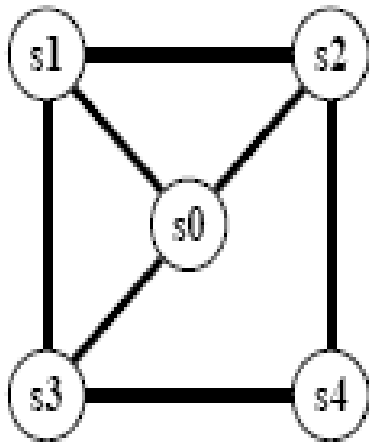
3 Colors

Euristica pentru colorare

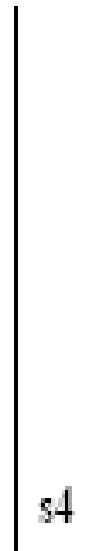
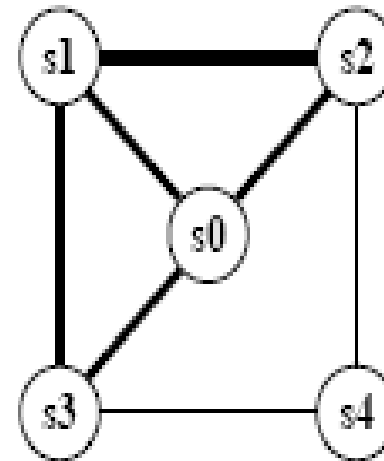
- Daca $\text{grad} < N$ ($\text{grad} = \#$ de vecini)
 - Nodul poate fi INTOTDEAUNA colorat
 - Dupa ce se coloreaza vecinii, e cel putin o culoare ramasa pentru el
- Si daca $\text{grad} \geq N$, s-ar putea ca nodul sa fie colorabil (nu e sigur)
- Algoritm:
 - Sterge nodurile cu $\text{grad} < N$, pune-le in ordine intr-o stiva (vor fi colorate mai tarziu)
 - Cand toate nodurile au $\text{grad} \geq N$
 - Gaseste un candidat pentru spill (nu i se da culoare)
 - Chaitin – spill; Briggs – pune-l pe stiva, intarzie decizia de spill (vezi “diamond graph”)
 - Cand am golit graficul, incepe colorarea
 - Ia nodul din varful stivei
 - Atribuie-i o culoare diferita de cea a vecinilor (daca e posibil)

Exemplu

$N = 3$

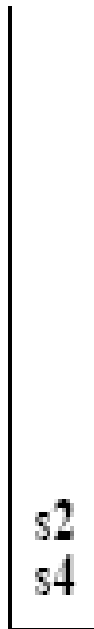
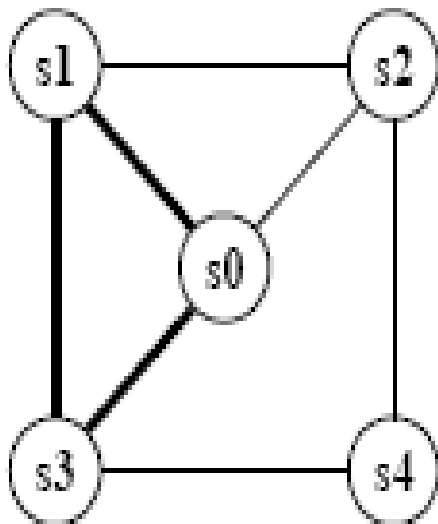


$N = 3$

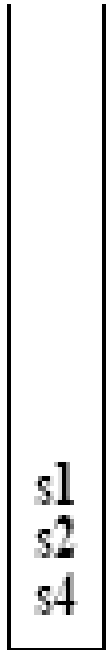
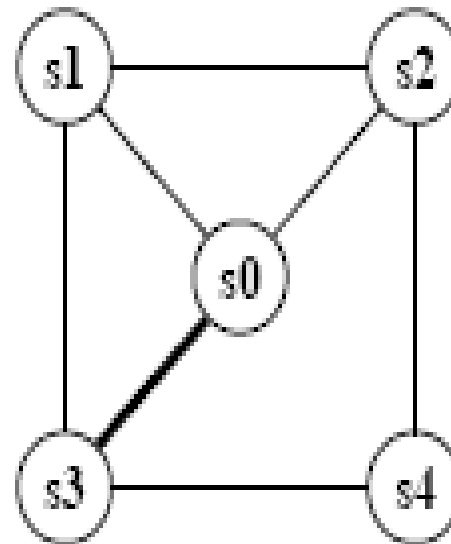


Exemplu(cont.)

$N = 3$

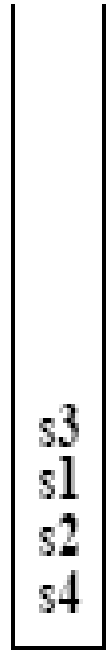
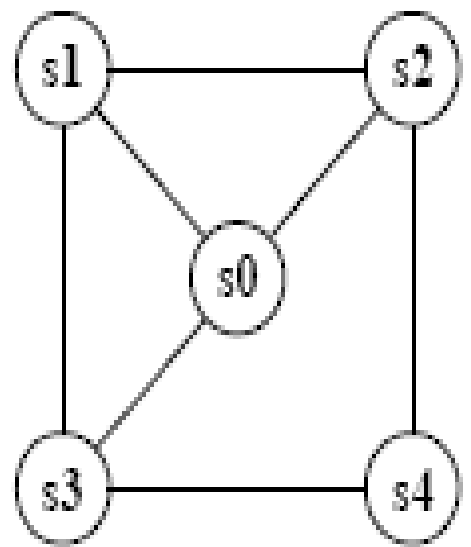


$N = 3$

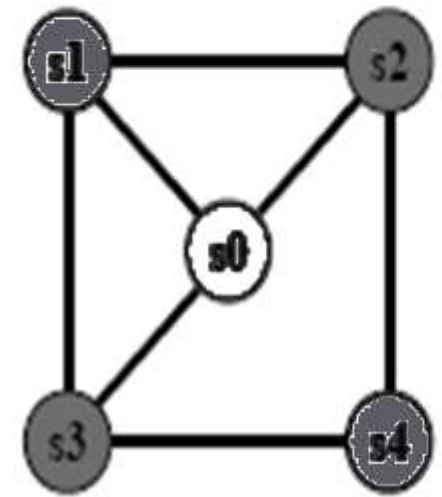


Exemplu(cont.)

N = 3

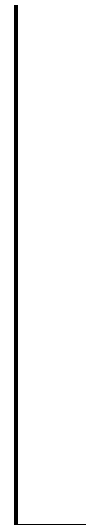
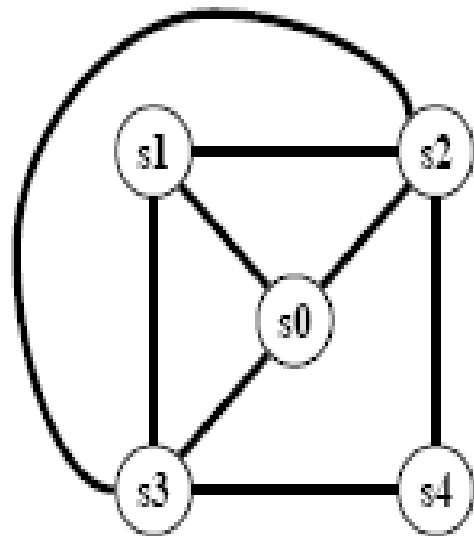


N = 3

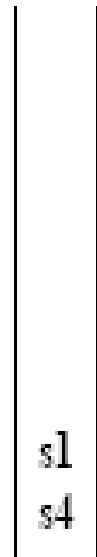
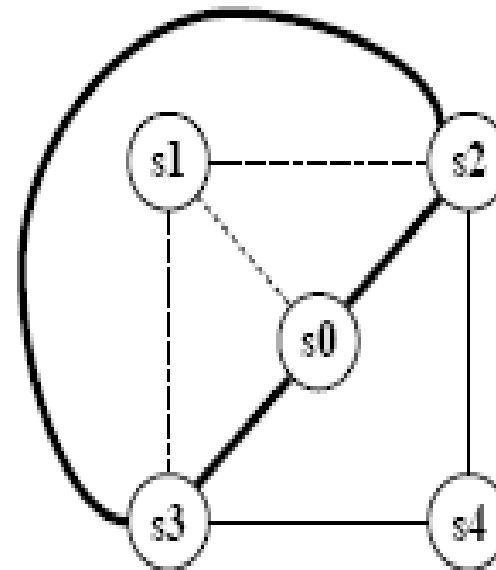


Alt exemplu

$N = 3$



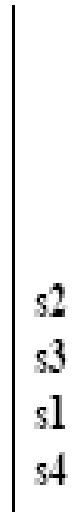
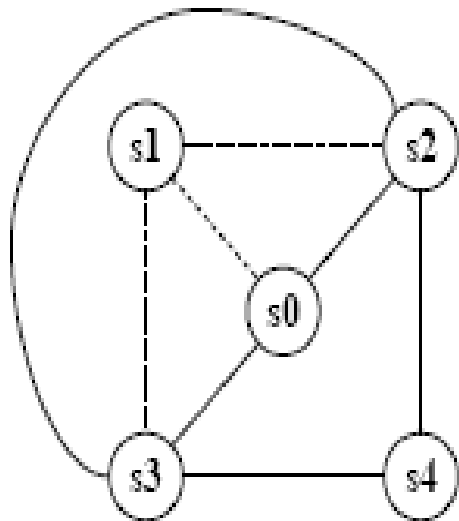
$N = 3$



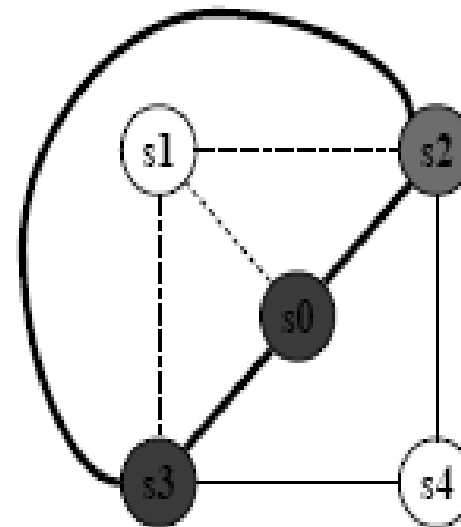
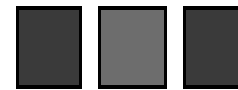
s1: Possible Spill

Alt exemplu (cont.)

N = 3



N = 3



s1: Actual Spill

Ce facem cand colorarea esueaza

- Spill: Plaseaza web-ul in memorie – ‘store’ dupa toate definiriile, ‘load’ inainte de toate folosirile
 - Dar instructiunile tot vor folosi registri...
 1. Reia colorarea – web-ul s-a transformat intr-o multime de weburi mici (cu mai putini vecini)
 2. Registri rezervati pentru spill
- Split: Imparte web-ul in bucati, reia colorarea (live range splitting)

Ce web alegem pentru spill?

- Unul cu mai mult de N vecini
- Unul care sa “deblocheze” procesul de colorare (sa nu mai fie necesar sa se faca spill si altor vecini)
 - Greu de determinat – alegem unul cu cat mai multi vecini
- Unul cu spill cost-ul cat mai mic (costul instructiunilor de load si store necesare)
- Pot fi cerinte contradictorii
 - Un web cu multe folosiri si live range intins are multi vecini, dar si spill cost mare.

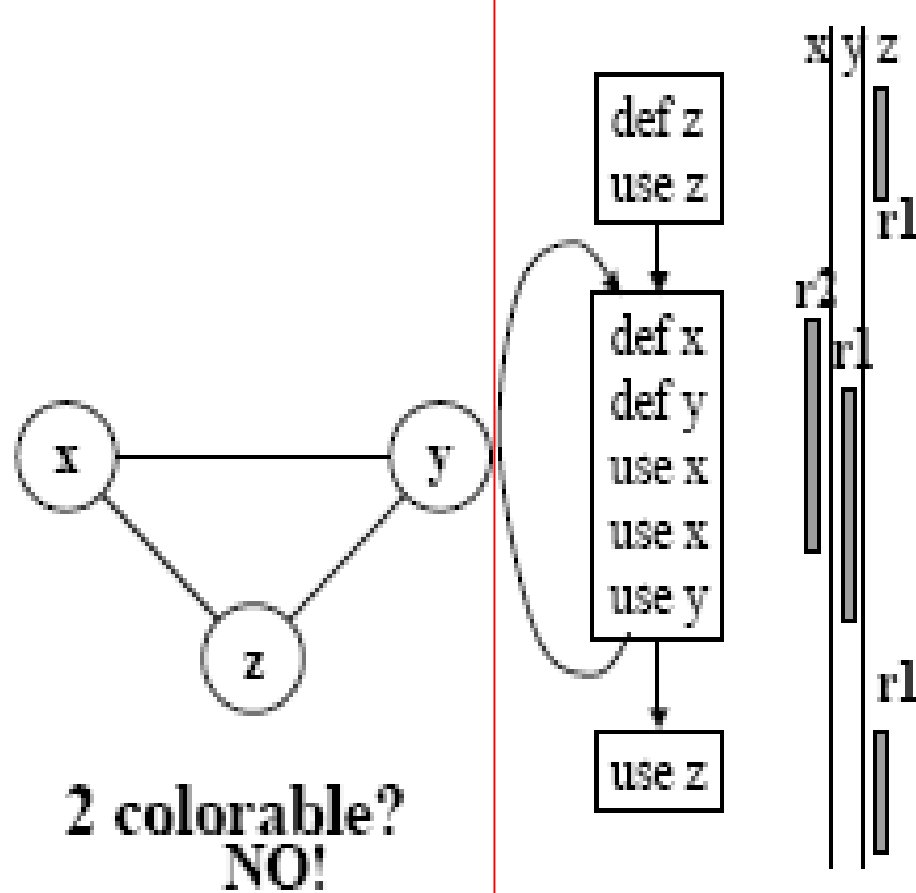
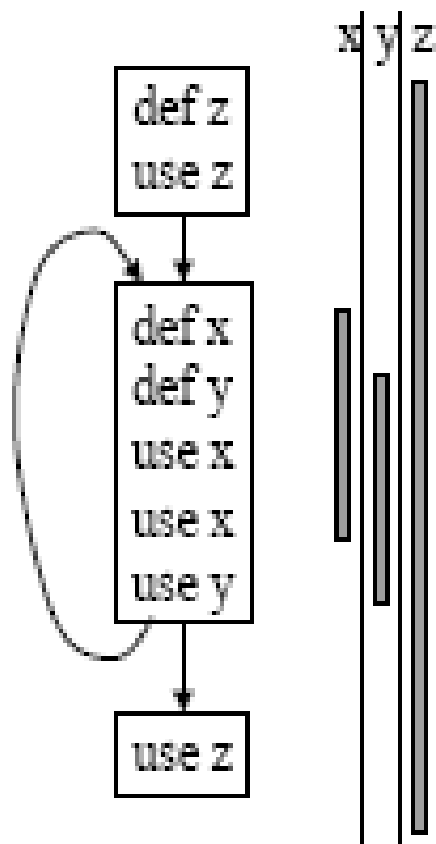
Spill cost

- Nu stim exact costul dinamic – nu stim ce branch-uri se vor executa
- Poate fi folosit 'profile information' sau o estimare statica (de ex. estimam ca bucelele se executa de 20 ori – e suficient dpdv. al costului)
- Rematerializare
 - Pentru reducerea costului, o valoare poate fi recalculata, in loc sa fie incarcata din memorie
 - Constanta, sau rezultatul unui calcul folosind valoarea din alt registru
 - Eliminarea unui load poate face store-ul corespunzator inutil (cand?)

Splitting ca alternativa la spill

- In general, “spill-everywhere” este prea conservativ – nu avem nevoie de load&store peste tot
- Alternativ – putem incerca sa segmentam live-range-ul unui web prin introducerea unor instructiuni de transfer, a.i. sa se faca spill doar la una dintre portiuni
 - Eventual, i se face spill din start – adica, in loc de “instructiuni de transfer” vom avea transfer in/din memorie in anumite puncte din program

Exemplu



Euristica de live range splitting

- Algorithm:
 - Identifica un punct din program unde graful nu e N-colorabil (punct unde # de weburi e $>N$)
 - Alege un web care nu e folosit pe cel mai mare interval ce cuprinde punctul respectiv
 - Imparte web-ul la capetele intervalului
 - Refa graful de interferenta
 - Reincearca sa colorezi graful
- O metoda mai buna/complementara:
alocarea ierarhica

Register Coalescing (combinarea nodurilor)

- Gaseste instructiuni de forma $x=y$, daca x si y nu interfera, le combina intr-un singur nod
- Avantaj: elimina instructiuni de transfer (similar cu 'copy propagation')
- Dezavantaj: poate creste gradul nodului combinat, un graf care era colorabil poate deveni necolorabil
- Euristici:
 - Briggs – uneste nodurile doar daca nodul rezultat are mai putin de k vecini care au mai mult de k vecini (adica, va fi sigur colorabil)
 - George & Appel – uneste daca pentru orice vecin t al lui x , fie t are mai putin de k vecini fie t are deja interferenta cu y (altfel zis, nu face graful "mai greu colorabil")
- Register targeting (precoloring) – o forma de coalescing, precoloreaza var. care au nevoie sa fie intr-un anumit reg. fizic (de ex, sunt parametri in apel de functie)
- Cand se face 'coalescing'? (inainte de colorare)

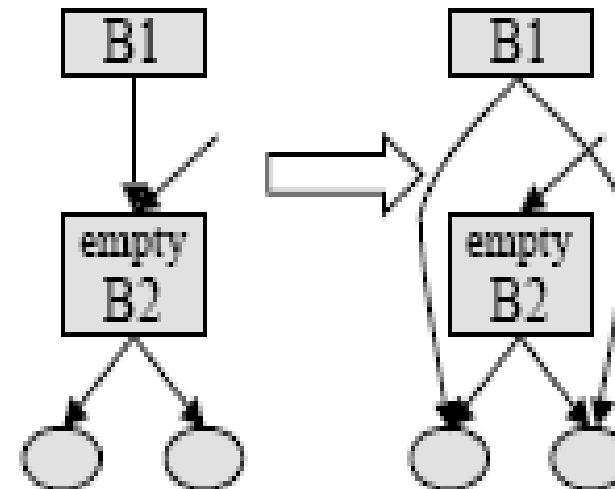
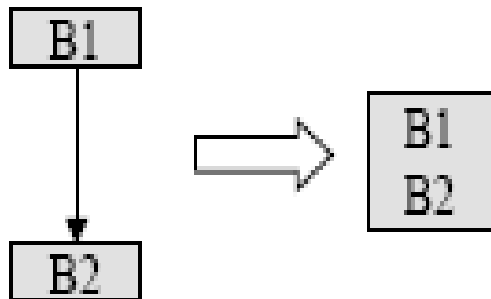
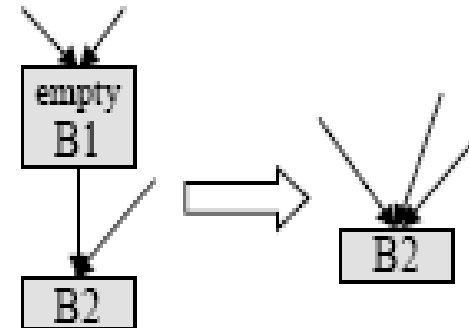
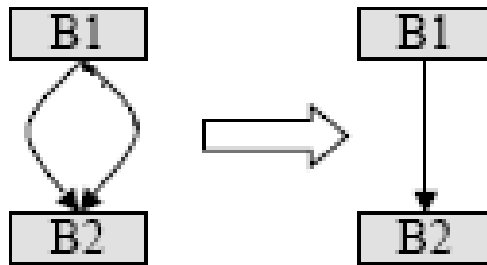
Interactiunea RA cu planificarea instructiunilor

- Daca RA se face inainte, tinde sa refoloseasca registri si sa adauge noi dependente
- Daca scheduling se face inainte, poate creste presiunea pe registri
 - Forward LS muta load-urile prea devreme
 - Backward LS muta store-urile prea tarziu
 - In plus – RA poate face spill – ar trebui replanificat

Optimizarea salturilor

- CFG poate fi 'reordonat' astfel incat sa reducem numarul de instructiuni de salt
- Salturile la alte instructiuni de salt sunt comune!
 - Un salt(conditionat sau nu) la un salt neconditionat poate fi inlocuit cu un salt la destinatia celui de-al doilea salt
 - Un salt neconditionat la un salt conditionat poate fi inlocuit cu o copie a saltului conditionat
 - Un salt conditionat la un salt conditionat poate fi inlocui cu un salt care mentina conditia primului salt si destinatia celui de-al doilea, daca putem demonstra ca cea de-a doua conditie e inclusa in prima
- Perechile de bb care au proprietatea ca 'b1 are un singur succesori – b2' si 'b2 are un singur predecesor – b1' pot fi unite intr-un singur bb (eliminarea saltului neconditionat la blocul urmator)

Cateva transformari (grafic)



Exemplu (eliminarea saltului la blocul urmator)

- Eliminarea saltului la blocul urmator trebuie facuta cu atentie...

L1: ...

a = b + c

goto L2

L6: ...

goto L4

L2:

b = c * 2

a = a + 1

if c < 0 goto L3

L5: ...

L1: ...

a = b + c

b = c * 2

a = a + 1

if c < 0 goto L3

goto L5

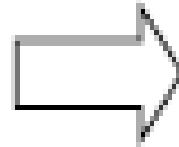
L6: ...

goto L4

L5: ...

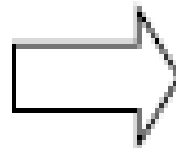
Alte exemple (optimizarea salturilor)

```
if a = 0 goto L1
...
L1: if a >= 0 goto L2
...
L2: ...
```



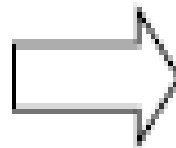
```
if a = 0 goto L2
...
L1: if a >= 0 goto L2
...
L2: ...
```

```
goto L1
L1: ...
```



```
L1: ...
```

```
if a = 0 goto L1
goto L2
L1: ...
```



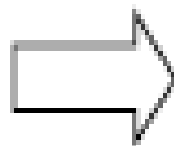
```
if a != 0 goto L2
L1: ...
```

Simplificarea buclelor

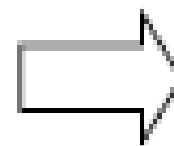
- O bucla cu corpul gol poate fi eliminata
 - Daca nu avem efecte laterale prin intermediul contorului sau putem sa "scoatem din bucla" efectele laterale
 - Problematic in cazul programatorilor care se bazeaza pe bucle goale pentru timing
- O bucla cu un numar de iteratii cunoscut (si suficient de mic) poate fi inlocuita cu cod secvential

Exemplu de simplificare

```
s = 0
i = 0
L1: if i > 4 goto L2
    i = i + 1
    s = s + i
    goto L1
L2: ...
```



```
s = 0
i = 0
i = i + 1
s = s + i
i = i + 1
s = s + i
i = i + 1
s = s + i
i = i + 1
s = s + i
L2: ...
```



```
i = 4
s = 10
L2: ...
```

Tranformarea while->repeat (inversarea buclelor)

- Are avantajul ca se executa o singura instructiune de salt pentru "inchiderea buclei"
- Trebuie sa putem demonstra ca bucla se va executa cel putin o data
- (low-level vs high-level)

```
for (i = 0; i < 100; i++) {  
    a[i] = i + 1;  
}
```

Numar de iteratii
cunoscut



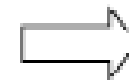
```
i = 0;  
while (i < 100) {  
    a[i] = i + 1;  
    i++;  
}
```



```
i = 0;  
repeat {  
    a[i] = i + 1;  
    i++;  
} until (i >= 100)
```

```
for (i = k; i < n; i++) {  
    a[i] = i + 1;  
}
```

Numar de iteratii
necunoscut

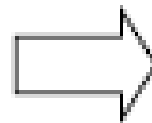


```
if (k >= n) goto L  
i = k;  
repeat {  
    a[i] = i + 1;  
    i++;  
} until (i >= n)  
L:
```

Unswitching

- E o transformare de flux de control care muta codul invariant afara din bucle
- Discutabil daca e de low-level sau high-level

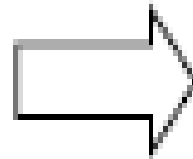
```
for (i = 1; i < 100; i++) {  
    if (k == 2)  
        a[i] = a[i] + 1;  
    else  
        a[i] = a[i] - 1;  
}
```



```
if (k == 2) {  
    for (i = 1; i < 100; i++)  
        a[i] = a[i] + 1;  
} else {  
    for (i = 1; i < 100; i++)  
        a[i] = a[i] - 1;  
}
```

Unswitching (exemplu2)

```
for (i = 1; i < 100; i++) {  
    if (k == 2 && a[i] > 0)  
        a[i] = a[i] + 1;  
}
```

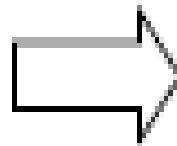


```
if (k == 2) {  
    for (i = 1; i < 100; i++) {  
        if (a[i] > 0)  
            a[i] = a[i] + 1;  
    }  
} else {  
    i = 100;  
}
```

Conversia if-urilor/instructiuni conditionate

- Multe arhitecturi au instructiuni care se pot executa doar daca o conditie e adevarata (cel putin instructiuni de transfer...)
- Utilizarea acestor instructiuni transforma dependentele de control in dependente de date, maresta basic-blocurile si creste numarul oportunitatilor pentru planificator

```
if a > b goto L1
max = b
goto L2
L1: max = a
L2: ...
```



```
t1 = a > b
max = b
max = (t1) a
```

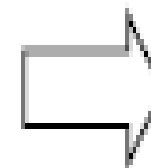
Tail merging

- Se aplica la bb-uri care au a acelasi set de succesori si au un set de instructiuni comune "pe coada"
- Se inlocuiesc ultimele instructiuni dintr-un bloc cu salt la ultimele instructiuni din celalalt

```

...
r1 = r2 + r3
r4 = r3 shl 2
r2 = r2 + 1
r2 = r4 - r2
goto L1
...
r5 = r4 - 6
r4 = r3 shl 2
r2 = r2 + 1
r2 = r4 - r2
L1: ...

```



```

...
r1 = r2 + r3
goto L2
...
r5 = r4 - 6
L2: r4 = r3 shl 2
r2 = r2 + 1
r2 = r4 - r2
L1: ...

```

Prezicerea statica a salturilor

- 'Branch prediction' se foloseste pentru a prezice daca o instructiune conditionata de salt va fi executata (se va sari) sau nu
- Procesoarele moderne se bazeaza pe BP pentru a ghici ce instructiuni trebuie citite dupa o instructiune de branch
- Prezicerea statica – compilatorul prezice daca saltul se va executa sau nu (plaseaza 'prezicerea' in instructiunea de salt, sau ca o instructiune separata)
- Prezicerea dinamica – hardware-ul tine minte comportamentul salturilor executate recent, si prezice pe baza istoriei
- O regula simpla de prezicere zice ca "salturile inapoi se vor executa, iar cele inainte nu"

Optimizari 'peephole'

- O metoda eficienta si des folosita de a 'curata' codul
- Ideea de baza: descopera imbunatatiri locale uitandu-se la o fereastră (de obicei mica) din cod (peephole = gaura cheii)
- De obicei fereastră (peephole) e o secventa de instructiuni aflate una dupa alta (dar nu neaparat)
- Optimizorul detecteaza anumite sabloane de instructiuni/secvente de cod si le inlocuieste cu secvente de cod echivalente, dar mai eficiente
 - "Reguli de rescriere" $i_1, \dots, i_n \rightarrow j_1, \dots, j_m$ unde RHS e versiunea imbunatatita de LHS

Exemple de peephole

- `move r1,r2; move r2,r1 -> move r1,r2`
- `addiu r1, i ; addiu r1, j -> addiu r1, i+j`

<pre>store r1 => r0, 8 load r0, 8 => r15</pre>	➔	<pre>store r1 => r0, 8 move r1 => r15</pre>
--	---	---

<pre>addiu r1, 0 => r2 mult r3, r2 => r4</pre>	➔	<pre>mult r3, r1 => r4</pre>
--	---	---------------------------------

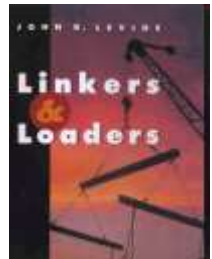
<pre>jumpL L1 L1: jumpL L2</pre>	➔	<pre>jumpL L2 L1: jumpL L2</pre>
----------------------------------	---	----------------------------------

Optimizari peephole (cont.)

- Foarte multe optimizari pot fi reformulate ca optimizari peephole (dar nu sunt neaparat mai eficiente in forma asta)
- La fel ca si majoritatea optimizarilor din compilatoare – optimizarile peephole trebuie aplicate repetat (iterativ) pentru a obtine efect maxim
- Cea mai frecventa utilizare a optimizarilor ‘peephole’ este la generarea de “machine idioms” – instructiuni specifice unui procesor, care fac o operatie complexa (ce s-ar face altfel folosind o secventa de instructiuni ‘general purpose’)

Linking

- Tipuri de fisiere continand cod obiect
 - Module, Biblioteci statice / dinamice, Executabile
- Sectiuni / Segmente
- Simboluri (exportate / importate)
 - Codificarea simbolurilor (mangling)
- Mai multe informatii: *Linkers and Loaders*, John R. Levine (www.iecc.com/linker)



Operatii

- Rezolvarea simbolurilor
 - La linkare sau la rulare (dynamic linking)
 - Stabilirea valorii simbolurilor (section+offset)
 - Asamblarea unui fisier continand doar modulele necesare
 - Root - Entry symbol
 - Cate functii sunt intr-o sectiune?
 - Poate trata o sectiune ca o secventa binara opaca.
 - Linking vs. partial linking
 - Simboluri '*Weak*'

Tipuri de sectiuni

- Cod, date initializate (PROGBITS)
- Date neinitializate (NOBITS)
- Resurse
- Informatii pt linker/loader (simboluri, relocari)
- Informatii suplimentare (debug)
- Permisuni – read, write, execute

Plasarea sectiunilor

- Reguli de asamblare: Linker Control File / Script

```
SECTIONS
```

```
{  
    . = 0x100000;  
    .text ALIGN(4096) : { *(.text*) }  
    .rodata ALIGN(4096) :  
    {  
        *(.rodata*)  
        start_ctors = .;  
        *(.ctor*)  
        end_ctors = .;  
    }  
    .data ALIGN(4096) : { *(.data*) }  
    .bss ALIGN(4096) : { *(.bss*) }  
}
```

Relocarea

- Inlocuirea simbolurilor cu adrese de memorie
- Ajustarea adreselor la incarcarea programului in memorie
- Operatii de relocare specifice procesorului – formeaza un limbaj.
 - Ex: R_386_PC32 (new value = old value + symbol – program counter)

```

30: 55          push   %ebp
31: 89 e5       mov    %esp,%ebp
33: 83 ec 08    sub   $0x8,%esp
36: 8b 45 10    mov   0x10(%ebp),%eax
39: 89 44 24 04  mov   %eax,0x4(%esp)
3d: 8b 45 0c    mov   0xc(%ebp),%eax
40: 89 04 24    mov   %eax,(%esp)
43: e8 00 00 00 00  call  _mul
48: 03 45 08    add   0x8(%ebp),%eax
4b: c9         leave
4c: c3         ret

```

```
int mac(int a, int b, int c)
{ return a + mul(b, c); }
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE          VALUE
00000044    DISP32        _mul
```

Formate obiect

- ISA cu segmente (.COM , DOS)
- Spatiu de memorie real (.EXE, DOS)
- Memorie virtuala + sectiuni (a.out, Unix)
- Formate moderne complexe:
 - ELF (Unix) – fisiere obiect (.o), biblioteci (.so), executabile
 - COFF, PE (Windows) – biblioteci (.dll), executabile (.exe)
 - Sectiuni aditionale (.rela, .debug), resurse

Biblioteci partajate

- Legate static/dinamic
 - Cod, date read-only – comune
 - Date rw – separat pentru fiecare aplicatie
- Apelul de functii din biblioteca
 - Stubs / thunks
 - “Lazy binding”
 - Incarcare explicita (dlopen / LoadLibrary)

Adresarea bibliotecilor partajate

- Spatiu de adrese rezervat – posibile conflicte
- Position Independent Code / Data (PIC/PID)
 - Adresarea indexata relativa la PC
 - Registru dedicat pt baza segmentului de date
 - Relocare la incarcarea bibliotecii
 - ELF: Global Offset Table
 - De ce? `static CData *pData = &myData;`
- PIC: Functiile de biblioteca apelate via thunks
 - ELF: Procedure Linkage Table
 - Suporta "Lazy binding"

Informatii de debug

- Simboluri locale (functii, variabile, parametri)
- Tipurile si structura lor
- Cadrului de stiva – dimensiune, continut
- Linii de cod \leftrightarrow adrese in `.code`
- Variabile \leftrightarrow locatii de memorie (in `.data` / `.stack` / registri)

Informatii de debug: DWARF

```
int mac(int a, int b, int c) {
    return a + mul(b, c);
}
```

```
10: 55          push   %ebp
11: 89 e5       mov    %esp,%ebp
13: 8b 45 10    mov    0x10(%ebp),%eax
16: 0f af 45 0c imul  0xc(%ebp),%eax
1a: 03 45 08    add   0x8(%ebp),%eax
1d: 5d         pop   %ebp
1e: c3         ret
```

Line Number Statements:

Set Address to 0x0

Copy

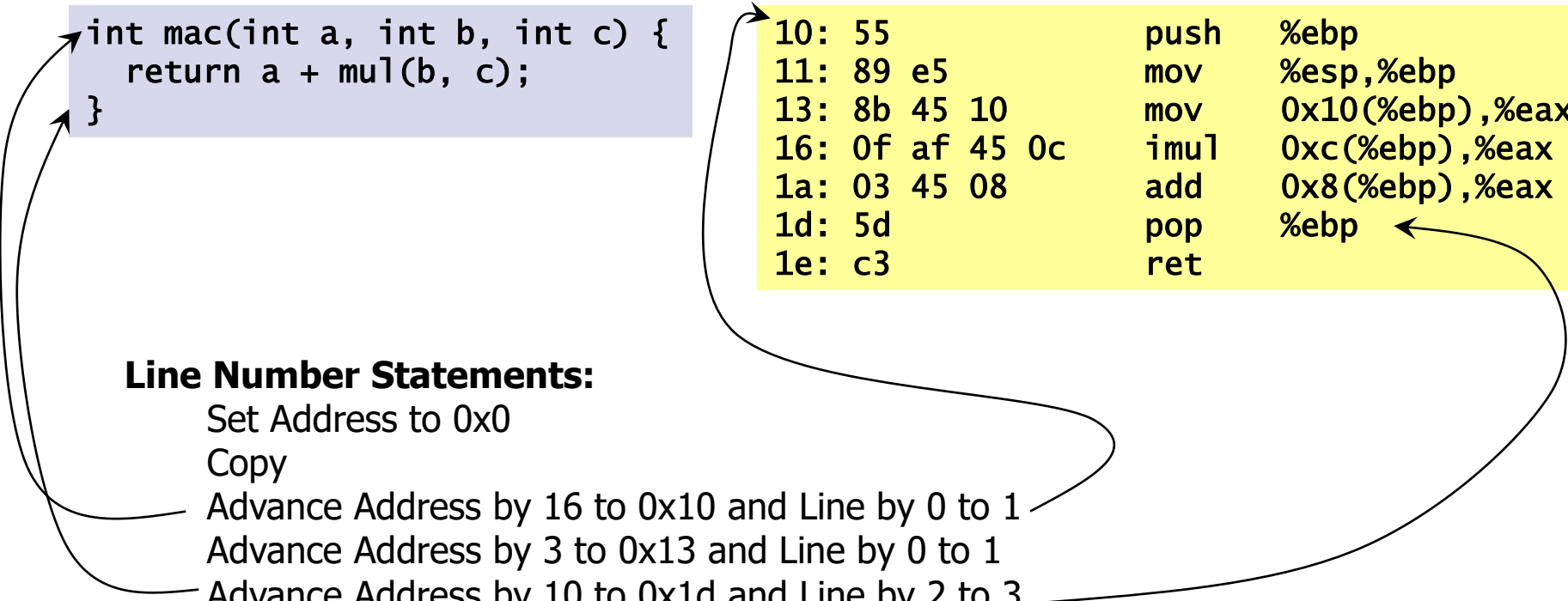
Advance Address by 16 to 0x10 and Line by 0 to 1

Advance Address by 3 to 0x13 and Line by 0 to 1

Advance Address by 10 to 0x1d and Line by 2 to 3

Advance PC by 2 to 0x1f

End of Sequence



Informatii de debug: DWARF

```

10: 55          push  %ebp
11: 89 e5       mov   %esp,%ebp
13: 8b 45 10    mov   0x10(%ebp),%eax
16: 0f af 45 0c imul  0xc(%ebp),%eax
1a: 03 45 08    add  0x8(%ebp),%eax
1d: 5d         pop  %ebp
1e: c3         ret
  
```

```

int mac(int a, int b, int c) {
    return a + mul(b, c);
}
  
```

10-11 (DW_OP_breg4: 4)
 11-13 (DW_OP_breg4: 8)
 13-1f (DW_OP_breg5: 8)

formal_parameter
 name : a
 decl_line : 1
 location : \$DW_OP_fbreg: 0
 type

formal_parameter
 name : b
 decl_line : 1
 location : \$DW_OP_fbreg: 4
 type

formal_parameter
 name : c
 decl_line : 1
 location : \$DW_OP_fbreg: 8
 type

base_type
 byte_size : 4
 encoding : signed
 name : int

DW_TAG_subprogram
 DW_AT_name : mac
 DW_AT_decl_line : 1
 DW_AT_low_pc : 0x10
 DW_AT_high_pc : 0x1f
 DW_AT_frame_base

