

Compilatoare

Generarea codului obiect



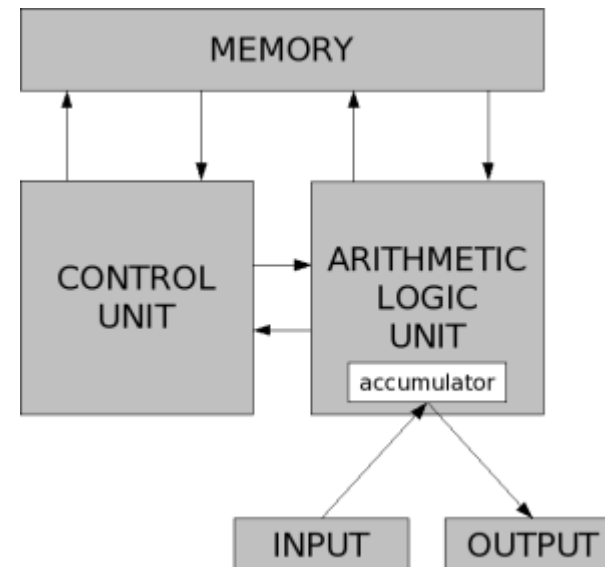
Computer architecture

A computer architecture is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture.

- Computer Architecture =
Instruction Set Architecture + Machine Organization +
 - ISA
- ... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.
- Amdahl, Blaaw, and Brooks, 1964
- “Portiunea din calculator vizibila unui programator”
 - ISA vs. ABI

Arhitectura 'standard'

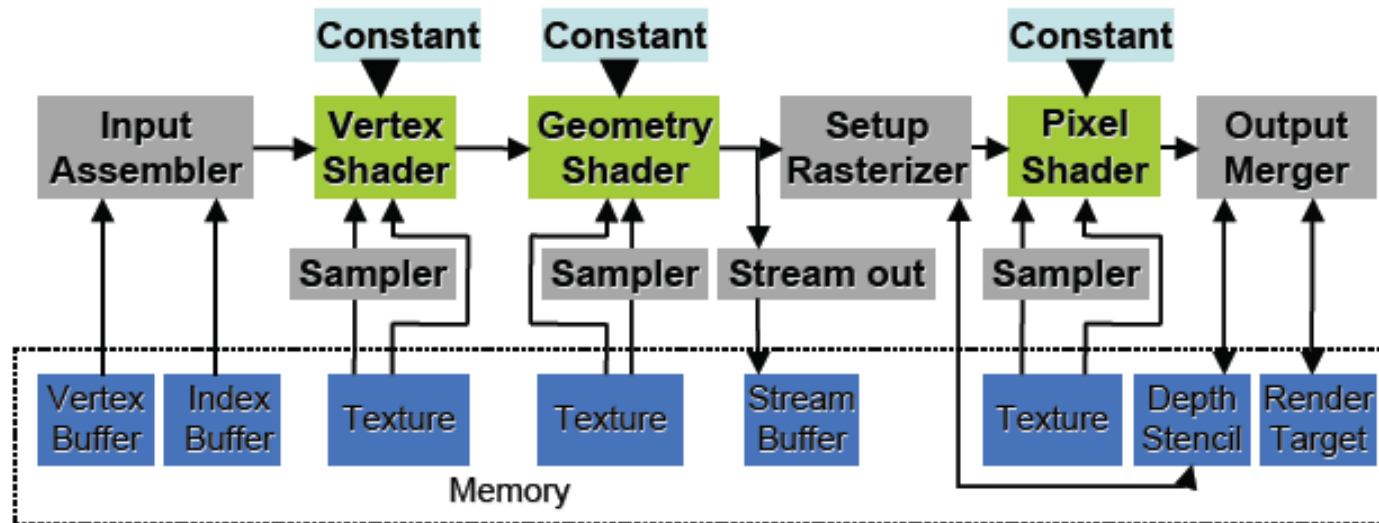
- Von Neumann:
 - Majoritatea procesoarelor din ziua de azi



- Harvard
 - Procesoare industriale - microcontrolere (PIC), DSP (Hawk, Blackfin)
- Mixt – memorii cache separate de instructiuni si date

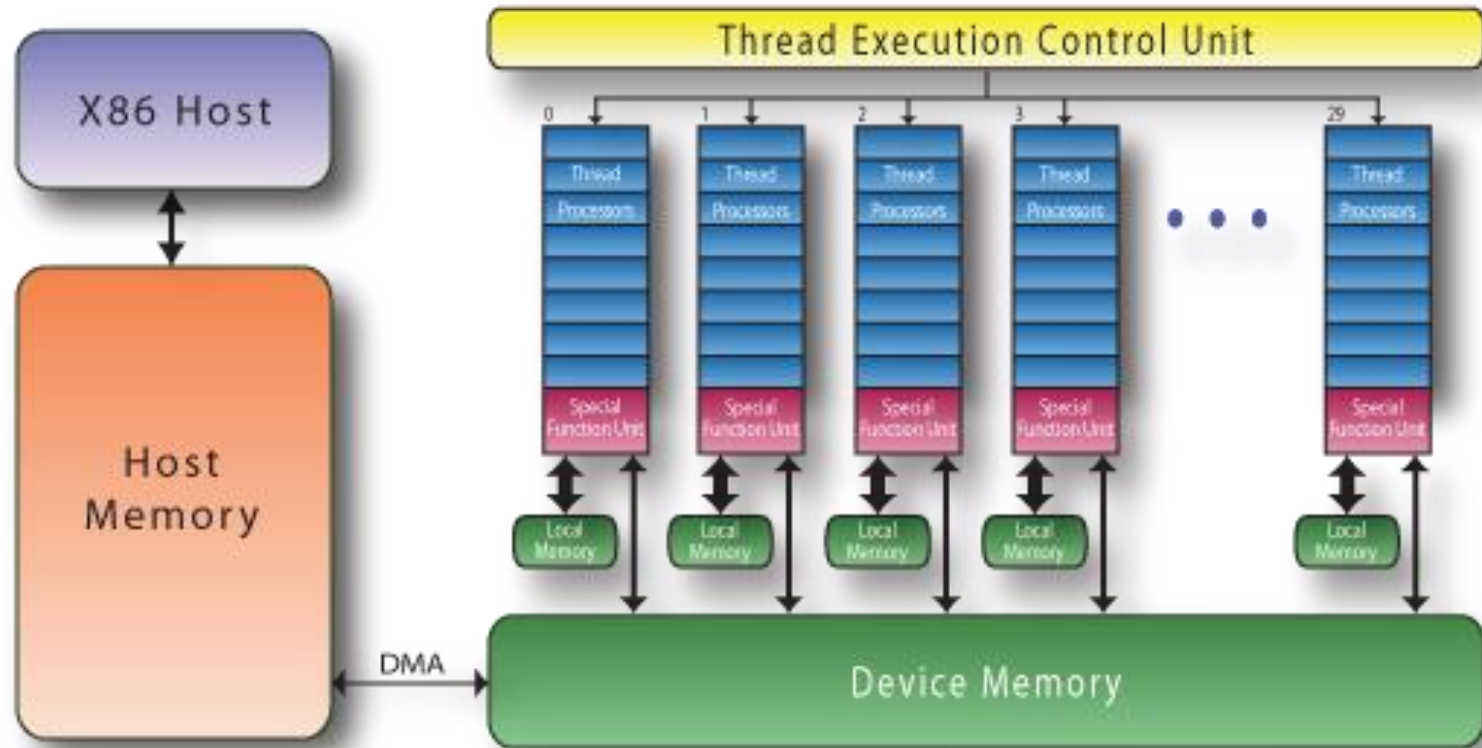
Nu este singura arhitectura

- Arhitecturi vectoriale
- Masiv paralele; fine threading – e.g. GPU



DirectX 10 – sursa: David Blythe Microsoft

General Purpose GPU



Sursa: NVidia

Tipuri de ISA

- Stack, acumulator, register
 - register-memory –x86,
 - register-register (load/store) –PPC
 - Memory-memory (VAX)
- Compilatoarele vor in general 'general purpose registers'
- Decizia influenteaza encodarea instructiunilor si numarul de instructiuni necesare pentru a face ceva

Adresarea memoriei

- Endianess
- Dimensiunea accesului; aliniament
- Moduri de adresare
 - Registru, imediate
 - Indirect, displacement, indexat (cu registru)
 - Autoincrement/decrement
 - Direct(absolut)
 - Memory indirect – `add r1,@(r3)`
 - Scalat – pt. array-uri. (poate fi implicit pt. toate modurile indexate)
 - Special (DSP) – modulo, bitreverse (*)

Tipul si dimensiunea operanzilor

- Toate procesoarele suporta intregi, (8/16/32/64 biti)
- Floating-point (32/64/80 biti)
- BCD (pt. aplicatii financiare)
- Fixed-point (DSP- 16/32/40biti), saturare
- Vertex, pixel (pt. procesare de imagini)

Operatiile din setul de instructiuni

- Aritmetice / logice
 - Integer, Floating Point
 - Bit Manipulation
 - Grafice (operatii pe pixel/vertex)
 - Instructiuni vectoriale/SIMD
 - DSP (saturare, rounding, MAC, FFT, Viterbi)
 - Crypto (AES)
- Transfer (Load / Store)

Operatiile din setul de instructiuni

- Control
 - Salturi, apeluri de procedura
 - Bucle hardware
 - Instructiuni conditionate si predicare
- String (move, compare, search)
- Sincronizare (compare-exchange pt mutex)
- Sistem (OS call, VM management)
 - Nivele de acces
 - Cum virtualizezi? Paravirtualizare / binary translation / support hardware (VT-X)

Codificarea setului de instructiuni (encoding)

- Cerinte in conflict – multe resurse, code size mic
- Restrictii de codificare
 - Restrictii de impachetare pe VLIW/EPIC
- Instructiuni – cu dim. fixa sau variabila
- Mai multe moduri de operare
 - x86/x64, PPC/VLE, ARM/Thumb/Thumb2/ARM V8
- Hardware dedicat pt. decomprimare

La compilare: compromis intre dimensiune si performanta

Exemplu de codificare

ARM/Thumb16

LDR R0, [SP, #12]	1001 1 000 00001100	(Rt, imm8)
ADD R0, R0, 1	00011 10 001 000 000	(imm, Rn, Rd)
STR R0, [SP, #12]	1001 0 000 00001100	(Rt, imm8)
CMP R0, #7	001 01 000 00000111	(Rt, imm8)
BNE PC+129	1101 0001 10000001	(cond, imm8)

Limitari

- Adresarea stivei
- Dimensiunea constantelor
- Distanța maximă a unui salt

Cod intermediar vs. cod obiect

- Reduce diferenta semantica cod sursa – cod obiect
- Acelasi compilator pe mai multe procesoare
- Acelasi compilator pentru mai multe limbaje
- Unele optimizari se fac mai simplu pe limbaj intermediar

Tipuri de cod intermediar

- Arbori
 - Abstract Syntax Tree e un cod intermediar.
- "Limbaj pentru masina virtuala"
 - Stiva vs. registri virtuali
 - "Quadruples", "three-address code"- max. 3 operanzi/instructiune (Cel mai frecvent folosit)

id1:= id2 op id3	{ op este un operator binar aritmetic sau logic}
id1:= op id2	{ op este un operator unar aritmetic sau logic}
id1:= val	{ val este un scalar sau o constanta }
id1:= &id2	{ adresa unui obiect in memorie }
id1:= * id2	{ citire din memorie via pointer }
* id1:= id2	{ scriere din memorie via pointer }
id1:= id2[id3]	{ id2 – adresa unui array, id3 – index in array }
goto id	
if id1 oprel id2 goto id3	{ oprel este un operator relational }

Nivelul codului intermediar

Original

```
float a[10][20];  
a[i][j+2];
```

High IR

```
t1 = a[i, j+2]
```

Mid IR

```
t1 = j + 2  
t2 = i * 20  
t3 = t1 + t2  
t4 = 4 * t3  
t5 = addr a  
t6 = t5 + t4  
t7 = *t6
```

Low IR

```
r1 = [fp - 4]  
r2 = [r1 + 2]  
r3 = [fp - 8]  
r4 = r3 * 20  
r5 = r4 + r2  
r6 = 4 * r5  
r7 = fp - 216  
f1 = [r7 + r6]
```

- HIR mentine structura limbajului
- MIR tinde sa fie independent de limbaj si masina
- LIR e dependent de masina

Mai mult de 3 operanzi

- Pentru apelul de subprograme se utilizează o secvență de instrucțiuni de forma:

```
param id.1  
param id.2  
...  
param id.n  
call id, n
```

- Expresiile cu mai mult de 3 operanzi se despart în expresii elementare

Example: GCC GENERIC

- GCC front-end, AST-based IR

```
float a[10][20];
```

```
@2695  var_decl      name: @2702  type: @2703  srcp: a.c:1
                    chan: @2704  size: @2705  algn: 32
                    used: 1
```

```
@2702  identifier_node strg: a      lngt: 1
```

```
@2703  array_type   size: @2705  algn: 32    elts: @2711
                    domn: @2712
```

```
@2705  integer_cst  type: @11   low : 6400
```

```
@2711  array_type   size: @2721  algn: 32    elts: @82
                    domn: @2722
```

```
@2721  integer_cst  type: @11   low : 640
```

```
@82    real_type     name: @78    size: @5     algn: 32
                    prec: 32
```

```
@78    type_decl    name: @81    type: @82    srcp: <built-in>:0
                    chan: @83
```

```
@81    identifier_node strg: float  lngt: 5
```

Example: GCC GIMPLE

- GCC High/Mid Level IR

```
float a[10][20];  
  
float f(i,j) {  
    return a[i][j+2];  
}
```

```
f (i, j)  
{  
    float D.1181;  
    int i.0;  
    int D.1183;  
  
    i.0 = i;  
    D.1183 = j + 2;  
    D.1181 = a[i.0][D.1183];  
    return D.1181;  
}
```

Example: GCC RTL

- GCC Low Level IR

```
(insn 5 4 6 3 a.c:4 (set (reg:SI 59 [ i.0 ])
  (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args))))

(insn 6 5 7 3 a.c:4 (set (reg:SI 62)
  (mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
    (const_int 4 [0x4])))))

(insn 7 6 8 3 a.c:4 (parallel [
  (set (reg:SI 58 [ D.1183 ])
    (plus:SI (reg:SI 62) (const_int 2 [0x2])))
  (clobber (reg:CC 17 flags))
])
```

```
i.0 = i;
D.1183 = j + 2;
```

```
(insn 5 2 6 a.c:4 (set (reg:SI 1 dx [orig:59 i.0 ]
  (mem/c/i:SI (plus:SI (reg/f:SI 6 bp)
    (const_int 8 [0x8])))) { *movsi_1 }

(insn 6 5 31 a.c:4 (set (reg:SI 0 ax [62])
  (mem/c/i:SI (plus:SI (reg/f:SI 6 bp)
    (const_int 12 [0xc])))) { *movsi_1 }

(insn 31 6 8 a.c:4 (set (reg:SI 2 cx [orig:58 D.1183 ]
  (plus:SI (reg:SI 0 ax [62])
    (const_int 2 [0x2]))) { *lea_1 } )
```

Example: LLVM IR

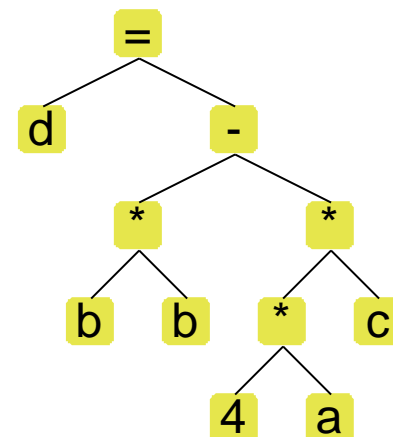
- High/Mid/Low Level IR

```
float a[10][20];  
  
float f(i,j) {  
    return a[i][j+2];  
}
```

```
@a = common global [10 x [20 x float]]  
        zeroinitializer, align 16  
  
define float @f(i32 %i, i32 %j) nounwind {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    store i32 %i, i32* %1, align 4  
    store i32 %j, i32* %2, align 4  
    %3 = load i32* %2, align 4  
    %4 = add nsw i32 %3, 2  
    %5 = sext i32 %4 to i64  
    %6 = load i32* %1, align 4  
    %7 = sext i32 %6 to i64  
    %8 = getelementptr inbounds  
        [10 x [20 x float]]* @a, i32 0, i64 %7  
    %9 = getelementptr inbounds  
        [20 x float]* %8, i32 0, i64 %5  
    %10 = load float* %9  
    ret float %10  
}
```

Evaluarea de expresii

- Expresiile cu mai mult de 3 operanzi se despart in expresii elementare
 - $d = b^2 - 4ac$
- Pe masinile load-store – creeaza un nou temporar pentru fiecare rezultat intermediar
 - Se presupune deocamdata ca exista un numar infinit de registri
 - Se poate lucra cu un numar finit prin tehnici simple de alocare, dar e mai bine sa lasam un pas special de optimizare sa se ocupe de asta
 - Index: nr. instructiunii care a produs valoarea
- Generarea se face printr-o traversare a arborelui sintactic

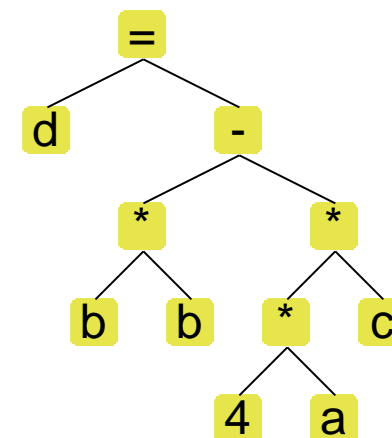


```

t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
d := t4
    
```

Expresii pe masinile cu stiva

- Instructiuni cu 0 adrese:
 - push, pop, aritmetice;
 - operatiile binare scot doua valori de pe stiva, pun la loc una
- Pe masinile cu stiva:
 - Pt a evalua o variabila – incarca valoarea
 - Pt a evalua o constanta – push valoarea
 - Pt a evalua o expresie
 - Evaluateaza stanga
 - Evaluateaza dreapta
 - Aplica operatorul



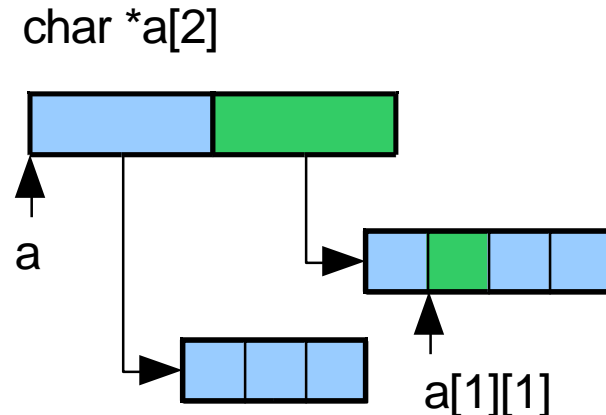
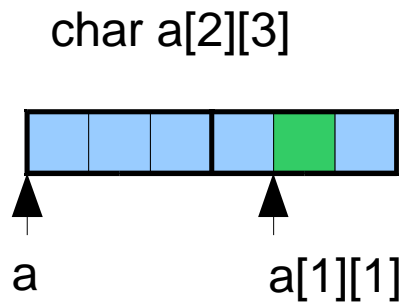
Load b
 Load b
 Mu1t
 Push 4
 Load a
 Mu1t
 Load c
 Mu1t
 Sub
 Store d

Tipuri simple

- Integer – atentie la cross-compiling
- Caractere – tipuri de codificari; Unicode
- Floating point
 - Reprezentare standard (IEEE 754)
 - Valori speciale: ± 0 , $\pm \text{inf}$, NotANumber
 - Overflow, dar si underflow
 - Care este cel mai mic numar pozitiv reprezentabil?
 - Numere denormalizate
 - Rotunjire, trunchiere, si aritmetica pe intervale

Tipuri compuse

- A :array($L1..L1+M-1, L2..L2+N-1$) of integer;
 - $A[i,j]$ e de fapt $tmp=(i-L1)*N+j-L2$; $A[tmp]$
 - Se poate tine $A-L1*N-L2$ in symbol table, accesam direct in functie de i si j
 - Atribuirea devine de fapt o bucla dubla / memcpy



Aliniere, padding

- A:record (structuri)

- se alocă memorie secvențial pentru câmpurile structurii;
- de obicei, se aliniaza;
- nu ne putem baza pe 'layout'



```
struct { char c1; int i; char c2};
```

- Bit fields



```
struct { char c1:6; char c2:6;  
        short i1:6; short i2:6;};
```

Generarea de cod pentru atribuire

- Genereaza adresa partii stangi
 - Variabila (tip scalar? poate fi tinuta in registru)
 - Membru (expresie cu ".")
 - Adresa structurii + offsetul in cadrul structurii.
 - Element (expresie cu "[]")
 - Adresa array + valoare index * dimensiune element.
- Genereaza valoarea partii drepte
- Atribuie (*addr)=val
 - Se presupune ca am transformat deja casturile implicite in casturi explicite

Generarea de cod pentru if

<pre>if cond then then_statements else else_statements; end if;</pre>	<pre>t1 = cond if not t1 goto else_label {quadruples for then_statements} goto endif_label else_label: {quadruples for else_statements} endif_label:</pre>
---	--

- Genereaza etichete
- Genereaza instructiuni pentru nodurile fii
- Plaseaza etichete in cod
 - (poate avea nevoie de etichete si pt then -> cand?)
- Daca avem `elif`, e practic "else if"; eticheta de `endif` poate fi mostenita de la parinte

Evaluarea booleana partiala

- Se trateaza expresiile booleene ca instructiuni de tip 'if then else'
 - If (B1 || B2) S1 else S2
=> **if(B1) goto S1 else if (B2) S1 else S2;**
 - If (B1 && B2) S1 else S2
=> **if(B1) then if(B2) then S1 else goto S2 else S2;**
 - Practic, se mostenesc etichetele de 'then' si 'else' de la if-ul "parinte"

Generarea de cod pt. while

```
while (cond) {  
s1;  
if (cond2) break;  
s2;  
if (cond3) continue;  
s3;  
};
```

```
start_loop:  
    if (!cond) goto end_loop  
    quadruples for s1  
    if (cond2) goto end_loop  
    quadruples for s2  
    if (cond3) goto start_loop  
    quadruples for s3  
    goto start_loop  
end_loop:
```

- Genereaza doua etichete: start_loop, end_loop
- Restul codului – ca mai sus

Alte tipuri de bucle

- Bucle numerice

- Semantica: bucla nu se executa daca intervalul variabilei e vid – deci testul se executa initial
- E de fapt o bucla while:
- For J in expr1..expr2
 - > **J=expr1; while(J<=expr2){...;J++;}**
 - Trebuie avut grija la "continue" – J++ se executa in acest caz!

- Bucle repeat...until

- Sunt bucle 'while', dar se copiaza o data corpul buclei inaintea ei

Plasarea testului la sfarsit

K in expr1 .. Expr2 loop

```

t1 = expr1
t2 = expr2
K = t1 - 1
goto test_label
start_label:
    quadruples for S1
test_label:
    K = K + 1
    if K <= t2 goto start_label
end_label:

```

S1;
end loop;

- Generarea de bucle hardware
 - Procesoare cu o instructiune de 'loop'
 - Se detecteaza secventa de mai sus in codul intermediar

Generarea de cod pentru switch

- Daca intervalul e mic si majoritatea cazurilor sunt definite, se poate crea tabela de salt ca vector de etichete

```
case x is
when up: y := 0;
when down : y := 1;
end case;

table label1, label2 ...
jumpi table+x
label1:
y = 0
goto end_case
label2:
y = 1
goto end_case
end_case:
```

- Altfel, se foloseste o serie de if-uri

Operatii complexe

- Ridicarea la putere – cazurile simple pot fi tratate eficient
 - $x^2 = x*x$; $x^4 = x^2*x^2$ (strength reduction)
- Cazurile complicate – apel de functii de biblioteca
- Tipurile care nu sunt suportate nativ – tot prin functii de biblioteca
- Intrinseci – operatii complexe la nivel inalt, simple la nivelul codului obiect
 - DSP (mac), bitscan, instr. vectoriale

Generatoare de generatoare de cod

- Setul de instructiuni asamblare poate fi reprezentat ca un set de reguli de rescriere a arborelui sintactic
 - Sablon, nod nou, cost, actiune
 - replacement \leftarrow template (cost) = {action}
- Codul asamblare e generat in procesul de reducere al arborelui la un singur nod
- Alg. care cauta acoperirea de cost minim – generator de generator de cod.
- Setul de reguli (“gramatica”) = schema de traducere a arborelui

Exemplu: reguli

Regula	Instructiune	Cost
reg r1 \rightarrow const c	mov r1, c	2
reg r1 \rightarrow read (a)	mov r1, a	2
$\lambda \rightarrow$ write a := (reg r1)	mov a, r1	2+r1
$\lambda \rightarrow$ write (reg r1) := (reg r2)	mov [r1], r2	1+r1+r2
reg r1 \rightarrow read (reg r2)	mov r1, [r2]	1+r2
reg r1 \rightarrow read ((const c) + (reg r2))	mov r1, c[r2]	2+r2
reg r1 \rightarrow (reg r1) + (read ((const c) + (reg r2)))	add r1, c[r2]	2+r1+r2
reg r1 \rightarrow (reg r1) + (reg r2)	add r1, r2	1+r1+r2
reg r1 \rightarrow (const 1) + (reg r1)	inc r1	1+r1

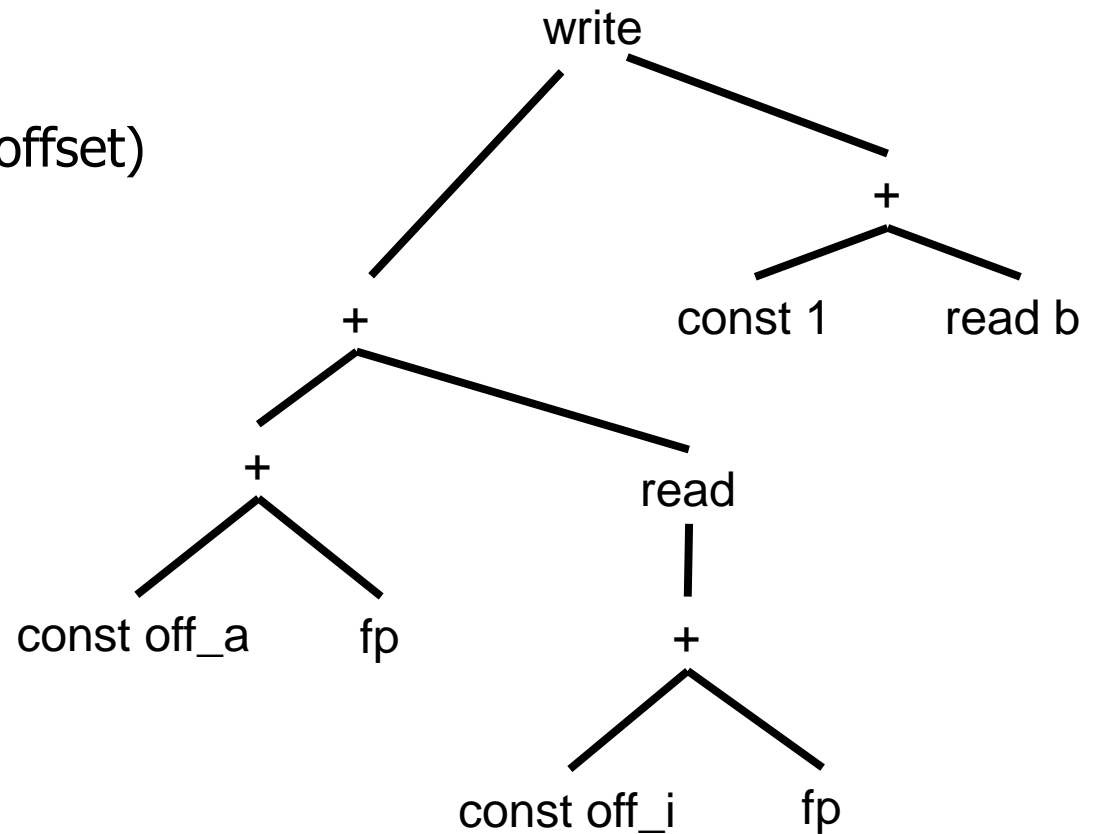
Reguli, rescriere arbore

	Rewrite rule	Cost	Instruction
(1)	$reg_i \leftarrow const_c$	2	MOV #c, Ri
(2)	$reg_i \leftarrow mem_a$	2	MOV a, Ri
(3)	$\lambda \leftarrow$ <pre> := / \ mem_a reg_i </pre>	$2 + cost.reg_i$	MOV Ri, a
(4)	$\lambda \leftarrow$ <pre> := / \ ind global_b reg_i </pre>	$2 + cost.reg_i$	MOV b, * Ri
(5)	$reg_i \leftarrow$ <pre> ind + / \ const_c reg_j </pre>	$2 + cost.reg_j$	MOV c(Rj), Ri
(6)	$reg_i \leftarrow$ <pre> + / \ reg_i ind + / \ const_c reg_j </pre>	$2 + cost.reg_i + cost.reg_j$	ADD c(Rj), Ri
(7)	$reg_i \leftarrow$ <pre> + / \ reg_i reg_j </pre>	$1 + cost.reg_i + cost.reg_j$	ADD Rj, Ri
(8)	$reg_i \leftarrow$ <pre> + / \ reg_i const_1 </pre>	$1 + cost.reg_i$	INC Ri

Exemplu arbore

$$a[i] = b + 1$$

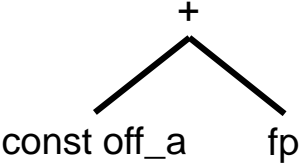
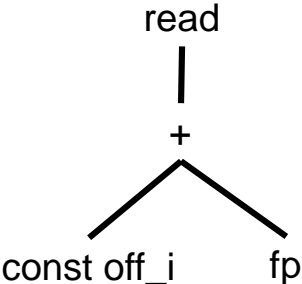
- $a, i \rightarrow$ pe stiva (fp + offset)
- $b \rightarrow$ in .data



Cum se face generarea de cod

- Intr-un prim pas, se acopera arborele cu sabloane, a.i. sa se obtina un cost minim
- In al doilea pas, se executa codul asociat cu sabloanele – care va produce programul in limbaj de asamblare.
- Descrierea e similara cu gramaticile pt. parsare – dar algoritmul e fundamental diferit!

Generare de cod

Arbore	Regula	Instructiuni	Cost
const off_a	reg r1 \rightarrow const off_a	mov r1, off_a	2
	reg r1 \rightarrow (reg r1) + (reg fp)	mov r1, off_a add r1, fp	3
	reg r3 \rightarrow read (reg r2)	mov r2, off_i add r2, fp mov r3, [r2]	4
	reg r2 \rightarrow read ((const off_i) + (reg fp))	mov r2, off_i[fp]	2

Generare de cod

Arbore	Regula	Instructiuni	Cost
<pre> graph TD A["+"] --- B["+"] A --- C["read"] B --- D["const off_a"] B --- E["fp"] C --- F["+"] F --- G["const off_i"] F --- H["fp"] </pre>	reg r1 → (reg r1) + (read ((const off_i) + (reg fp)))	mov r1, off_a add r1, fp add r1, off_i[fp]	5
	reg r1 → (reg r1) + (reg r2)	mov r1, off_a add r1, fp mov r2, off_i[fp] add r1,r2	6
<pre> graph TD A["+"] --- B["const 1"] A --- C["read b"] </pre>	reg r3 → (reg r3) + (reg r4)	mov r3, 1 mov r4, b add r3, r4	5
	reg r3 → (const 1) + (reg r4)	mov r4,b inc r4	3

Generare de cod

Arbore	Instruțiuni	Cost
<pre> graph TD write[write] --- plus1[+] write --- plus2[+] plus1 --- const_off_a[const off_a] plus1 --- fp1[fp] plus2 --- const_1[const 1] plus2 --- read[read] read --- plus3[+] plus3 --- const_off_i[const off_i] plus3 --- fp2[fp] </pre>	<pre> mov r1, off_a add r1, fp add r1, off_i[fp] mov r4,b inc r4 mov[r1],r4 </pre>	<p style="text-align: center;">9</p>

Algoritmul de generare de cod

- Acoperirea optima e formata din acoperirea optima a fiilor + costul aplicarii unui sablon
- Principiul optimalitatii – programare dinamica
- Cautati iburg, lburg