Limbaje formale și tehnici de compilare

Curs 12: mașini virtuale bazate pe regiștri: structură, generare de cod, implementare

Regiștri

- **Registru** – O celulă de memorie, care poate conține o valoare
- Diverse MV pot defini regiștrii în moduri diferite:
  - **Globali** – vizibili din tot programul, la fel ca regiștrii unui microprocesor
  - **Locali** – similari variabilelor locale ale unei funcții
- În general, regiștrii sunt alocați automat de compilator și se folosesc pentru stocarea valorilor intermediare
- Unele MV definesc regiștri speciali pentru anumite tipuri de date, de exemplu pentru numere reale
- Vom nota regiștrii cu **#idx**, unde indexul este un număr întreg (#1, #2, …)
- Unele MV pot defini operațiile să fie efectuare doar între regiștri (ex: ADD #7,#3,#8), pe când alte MV acceptă operanzi micști (ex: ADD #7,#3,0.5)

MV bazate pe regiștri (MVR)

- Sunt MV care folosesc regiștri pentru stocarea rezultatelor intermediare sau pentru anumite operații specifice
- Exemple: LLVM, Lua VM, Parrot
- Este mai greu de generat cod pentru MVR decât pentru MVS (Mașină Virtuală bazată pe Stivă)
- MVR se pretează mai bine la optimizări ulterioare de cod și din acest motiv se folosesc pentru reprezentarea codului intermediar (IR – Intermediate Code), ca de exemplu în LLVM
- MVR nu folosesc la definire conceptul de stivă, dar implementările lor pot folosi stive

Definirea MVR folosită la acest curs

- Regiștrii sunt zone de memorie care pot stoca orice tip primar de date
- Regiștrii sunt variabile locale introduse de compilator. Ei sunt dispuși în cadrul funcției imediat după variabilele locale.
- Folosind notația **#idx**, indexarea fiind față de FP, se poate accesa orice variabilă din cadrul funcției: argumente, variabile locale, regiștri
- Operanzi pot fi adrese de variabile globale, regiștri sau constante (ex: DIV.i #1, x, 2). Pe prima poziție este destinația rezultatului.
- Pentru specificarea tipului (int, float), acesta se pune ca sufix după numele operației (ex: ADD.f)

Exemplu de cod MVR

```
int min(int x, int y){
    int r;
    if(x<y)r=x;
        else r=y;
    return r;
    }
```

```
// int k=min(7,21);
CALL   k, min, 7, 21
```

```
            ENTER   2       // nr_var_locale + nr_reg
            LESS.i  #2, #-3, #-2
            JF      #2, L1
            SET.i   #1, #-3
            JMP     L2
L1:         SET.i   #1, #-2
L2:         RET.i   2, #1  // nr_param, valoare return
```

FP	Valoare
-4	call_dst
-3	x
-2	y
-1	ret_addr
0	old_FP
1	r
2	reg #2

- **ENTER** ***nr_loc*** – alocă memorie locală (nr_var_locale+nr_reg)
- **RET.*tip*** ***nr_param, valoare_returnată*** – eliberează cadrul funcției și returnează valoarea specificată, având tipul dat
- **JF** și **JT** au pe prima poziție valoarea de testat (întotdeauna de tip *int*), iar pe a doua poziție eticheta destinație
- **SET.*tip*** ***dst, src*** – copiază *src* în *dst*

Etapele unui apel de funcție

- **Se apelează funcția** folosind **CALL**:
  - Depune în stivă adresa destinației
  - Depune pe stivă toate argumentele
  - Depune pe stivă adresa de revenire (adresa următoarei instrucțiuni de după CALL)
  - Face JMP la codul funcției

CALL dst, fn, arg_1, ,arg_n

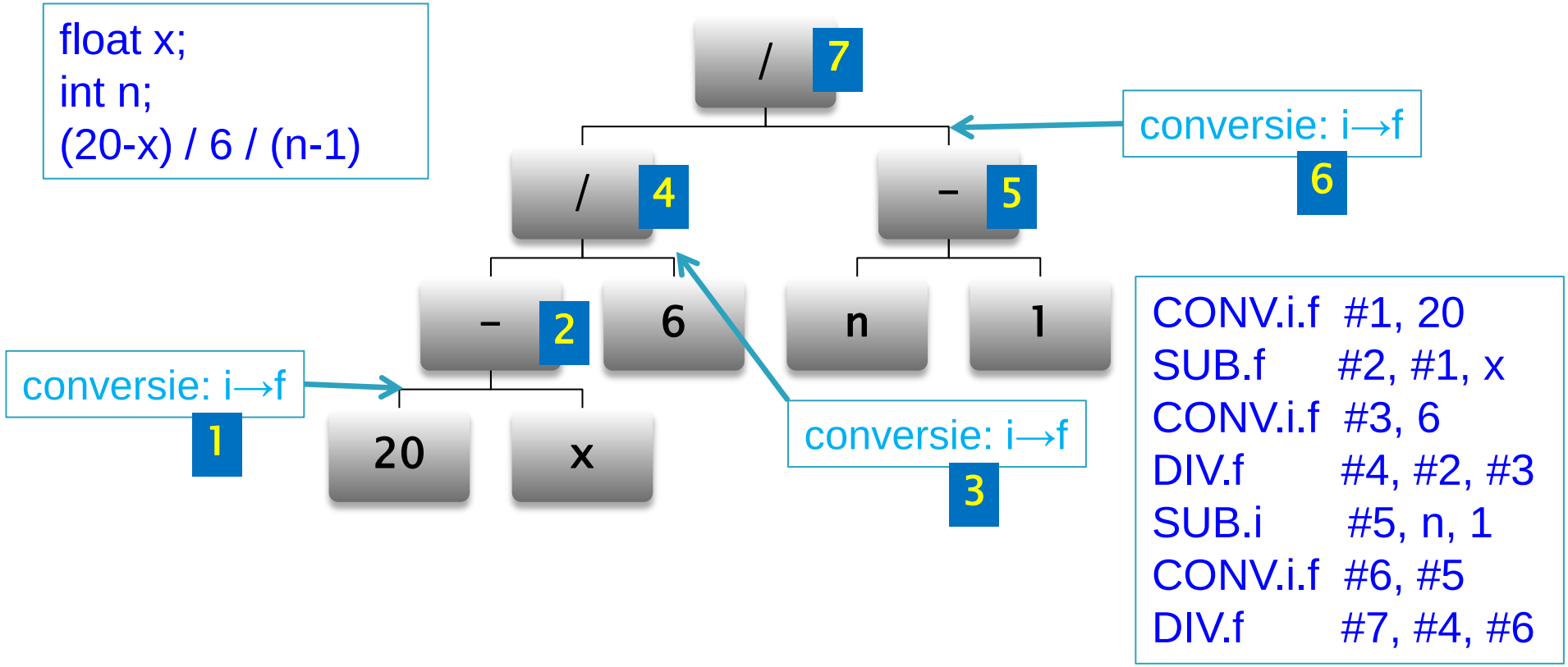
- **La intrarea în funcție cu ENTER**:
  - Se salvează vechiul FP în stivă, apoi în FP se va pune indexul vârfului stivei
  - Se adună la vârful stivei nr. de variabile locale și de regiștri, pentru a se face loc pentru ele după vechiul FP

ENTER nr_var_locale+nr_reg

- **Revenirea din funcție cu RET**:
  - Reface FP la valoarea anterioară apelului
  - Șterge din stivă tot cadrul funcției
  - Depune valoarea specificată de RET la adresa destinației

RET nr_arg, val

Generare de cod pentru o expresie



- Parcurgem arborele în postordine și:
  - Atribuim fiecărei operații, inclusiv conversiilor, un index unic, începând cu primul index liber față de FP. Acest index este registrul în care se va depune rezultatul operației respective
  - Generăm codul pentru operația respectivă

Optimizarea numărului de regiștri

```
CONV.i.f  #1, 20
SUB.f     #2, #1, x
CONV.i.f  #3, 6
DIV.f     #4, #2, #3
SUB.i     #5, n, 1
CONV.i.f  #6, #5
DIV.f     #7, #4, #6
```



```
CONV.i.f  #1, 20
SUB.f     #1, #1, x
CONV.i.f  #2, 6
DIV.f     #1, #1, #2
SUB.i     #2, n, 1
CONV.i.f  #2, #2
DIV.f     #1, #1, #2
```

```
float x;
int n;
(20-x) / 6 / (n-1)
```

- Se poate constata că regiștrii care sunt surse într-o operație (ex: #2 și #3 din DIV.f #4, #2, #3) nu mai sunt folosiți ulterior
- Putem să eliberăm acești regiștri imediat după ce au fost folosiți și să-i refolosim ulterior, inclusiv ca destinație pentru operația respectivă

Generarea de cod pentru MVR

- Regulile sintactice care generează valori (ex: *expr*) vor avea două atribute sintetizate:
  - **tip** – tipul valorii generate
  - **arg** – valoarea generată (constantă, variabilă, registru)
- Dacă este nevoie de un registru, acesta va fi alocat cu funcția **getReg().** Dacă mai sunt regiștri liberi, **getReg()** îi va refolosi, altfel va aloca noi regiștri
- Eliberarea unui registru se face cu funcția **freeReg(arg).** Dacă **arg** este un registru, acesta va fi marcat ca liber. Pentru alte tipuri de argumente **freeReg** nu are niciun efect
- **tipRezultat=combine(tip1,tip2)** – funcția **combine**() returnează tipul rezultat dintr-o operație care are operanzii de tipurile **tip1** și **tip2**
- **posInstr()** returnează poziția la care va fi adăugată noua instrucțiune în vectorul de instrucțiuni (prima poziție liberă)

Generare de cod pentru expresii

```
enum Tip = { TipInt, TipFloat }
calculator = ( expr[t,arg] {addShow(t,arg);freeReg(arg);} ) * FINISH
expr[out tip:Tip, out arg:Arg] =
  expr[t1,arg1] {idx=posInstr();}
  ( ADD termen[t2,arg2] {/*1*/}
  | SUB termen[t2,arg2] {/*2*/} ) | term[tip,arg]
term[out tip:Tip, out arg:Arg] =
  term[t1,arg1] {idx=posInstr();}
  ( MUL factor[t2,arg2] {/*3*/}
  | DIVfactor[t2,arg2] {/*4*/} ) | factor[tip,arg]
factor[out tip:Tip, out arg:Arg] =
  INT[n] {tip=TipInt;arg=Arg_int(n.i);}
  | FLOAT[n] {tip=TipFloat;arg=Arg_float(n.f);}
  | LPAR expr[tip,arg] RPAR
```

```
// 1=Add, 2=Sub, 3=Mul, 4=Div
tip=combine(t1,t2);
arg1=needCast(idx,t1,tip,arg1);
arg2=needCast(posInstr(),t2,tip,arg2)
freeReg(arg1);
freeReg(arg2);
arg=getReg();
addAdd(tip,arg,arg1,arg2); // 1,2,3,4
```

- **arg_dst=needCast(idx,tip_src,tip_dst,arg_src)** – dacă este necesar, la indexul dat se inserează instrucțiunea:
  CAST.*tip_src.tip_dst* **arg_dst, arg_src**
  dacă **arg_src** este registru, acesta se refolosește ca **arg_dst**, altfel se alocă un nou registru pentru **arg_dst**

Exemplu de cod cu if și while

```
int cmmdc(int x,int y){
    while(x!=y){
        if(x>y)x=x-y;
            else y=y-x;
        }
    return x;
    }
```

FP	Valoare
-4	call_dst
-3	x
-2	y
-1	ret_addr
0	old_FP
1	reg #1

```
        ENTER    1
L1:     NOTEQ.i  #1,  #-3, #-2
        JF       #1, L4
        GRT.i     #1, #-3, #-2
        JF       #1, L2
        SUB.i    #1, #-3, #-2
        SET.i     #-3, #1
        JMP      L3
L2:     SUB.i    #1, #-2, #-3
        SET.i     #-2, #1
L3:     JMP      L1
L4:     RET.i     2, #-3
```

Implementare MVR – Val și stiva

```
typedef union _Val{
    int i;                  // numere intregi, indecsi in FP sau in instructiuni
    float f;                // numere reale
    union _Val *addr;       // adresa unei valori (folosita pentru RET)
    }Val;

Val stack[];        // stiva
Val *SP;            // Stack Pointer - pointer la valoarea din varful stivei
Val *FP;            // Frame Pointer - pointer la locatia unde s-a salvat vechiul FP
```

- **Val** – o celulă de memorie care conține o valoare

Implementare MVR – Arg

```
typedef enum { ArgKindCt,      // constante: 21, 3.14
               ArgKindAddr,    // o adresa, de exemplu a unei variabile globale
               ArgKindLocal    // orice locatie accesibila prin FP, inclusiv registri
               } ArgKind;

typedef struct{                // un argument de instructiune
    ArgKind kind;
    union{
        Val val;       // o constanta, pentru ArgKindCt
        Val *addr;     // o adresa, pentru ArgKindAddr
        int idx;       // indexul fata de FP, pentru ArgKindLocal
        };
    }Arg;
```

```
Val argVal(Arg arg){
    switch(arg.kind){
        case ArgKindCt:return arg.val;
        case ArgKindAddr:return *arg.addr;
        case ArgKindLocal:return FP[arg.idx];
        }
    }
```

```
Val *argAddr(Arg arg){
    switch(arg.kind){
        case ArgKindAddr:
            return arg.addr;
        case ArgKindLocal:
            return &FP[arg.idx];
        }
    }
```

Implementare MVR – Instr

```
typedef enum {ADD_I, NOT_F, CALL, RET} Code;
Instr instructions[];          // vector de instructiuni
Instr *IP;                     // Instruction Pointer
```

```
typedef struct{
    Code code;
    union{
        struct{
            Arg dst,left,right; // ex: ADD_I dst, left, right
            }binOp;             // toate tipurile de operatii binare, ex: ADD, SUB, ...
        struct{
            Arg dst;            // destinatia rezultatului returnat de CALL
            int idxFn, nArgs; // indexul functiei in instructions, nr de argumente
            Arg *args;          // vector de argumente alocat dinamic
            }call;
        struct{
            int nArgs;          // numarul de argumente al functiei
            Arg val;            // valoarea de returnat
            }ret;
        struct{
            Arg cond;           // conditia pentru JF si JT
            int idx;            // indexul instrucțiunii de salt
            }jmp;
        };
    }Instr;
```

Implementare MVR – execuție

```
// ADD_I  dst, left, right
i1=argVal(IP->binOp.left).i;
i2=argVal(IP->binOp.right).i;
*argAddr(IP->binOp.dst)=Val_int(i1+i2);
IP++;
```

```
// SET_I  dst, src
i=argVal(IP->unOp.src).i;
*argAddr(IP->unOp.dst)=Val_int(i);
IP++;
```

```
// CALL  dst,arg1,...,argn
pusha(argAddr(IP->call.dst));
for(i=0;i<IP->call.nArgs;++i)
    pushv(argVal(IP->call.args[i]));
pusha(IP-instructions+1);
IP=&instructions[IP->call.idxFn];
```

```
// RET  nArgs, val
SP=FP - IP->ret.nArgs - 3;
*SP[1].addr=argVal(IP->ret.val);
IP=&instructions[FP[-1].i];
FP=FP[0].addr;
```

```
// JF  cond, idx_label
if(getVal(IP->jmp.cond).i)IP++;
    else IP=&instructiuni[IP->jmp.idx];
```