Complemente de programare 1
C07: Integrarea cu alte limbaje de programare (ASM, Python)

conf. dr. ing. Elena Șerban șef lucr. dr. ing. Alexandru Archip

Universitatea Tehnică „Gheorghe Asachi" din Iași
Facultatea de Automatică și Calculatoare

Calculatoare și Tehnologia informației (licență, an I) – 2021 – 2022

E-mail: elena.serban@academic.tuiasi.ro | alexandru.archip@academic.tuiasi.ro

Cuprins

Cuprins

Arhitectura de bază a unui sistem de calcul

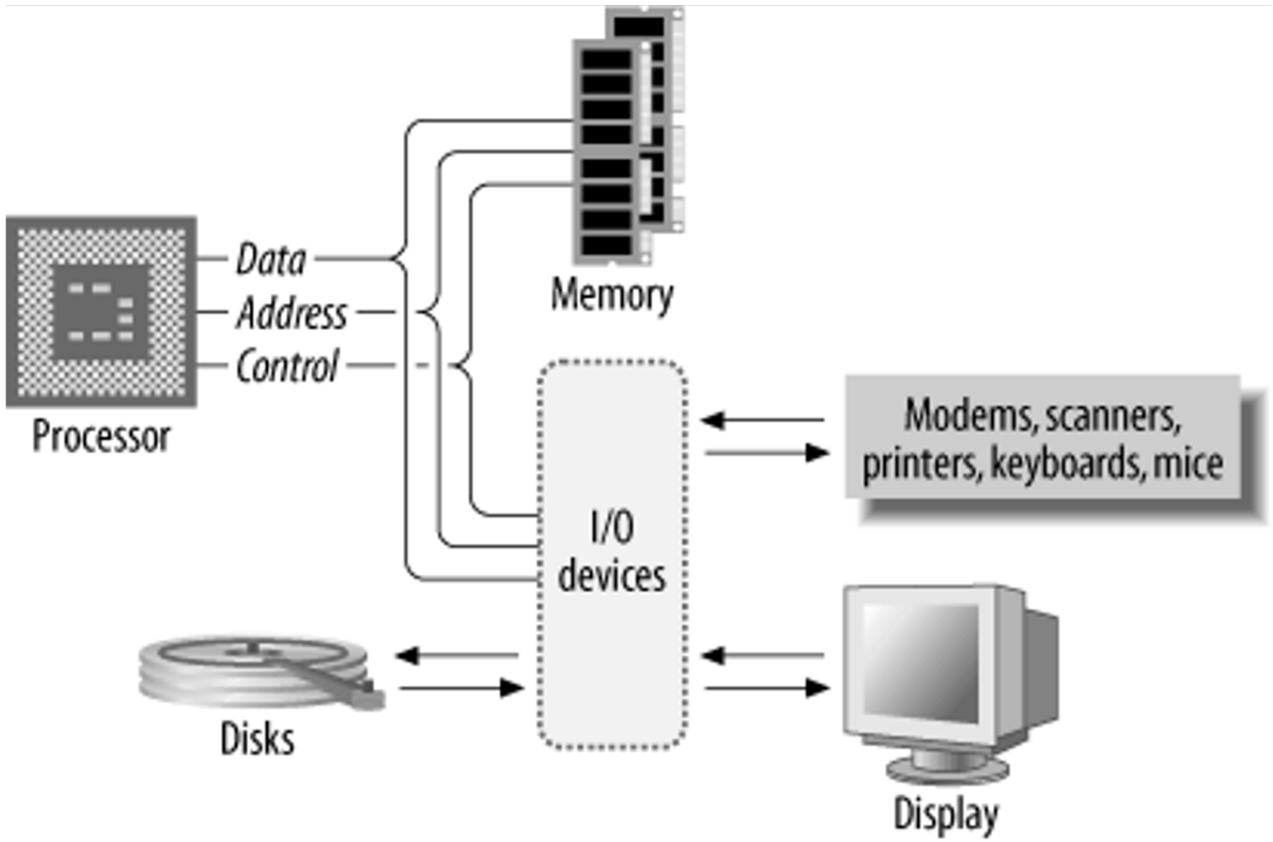

Figura 1. Arhitectura de bază a unui sistem de calcul [1]

Arhitectura de bază a unui sistem de calcul

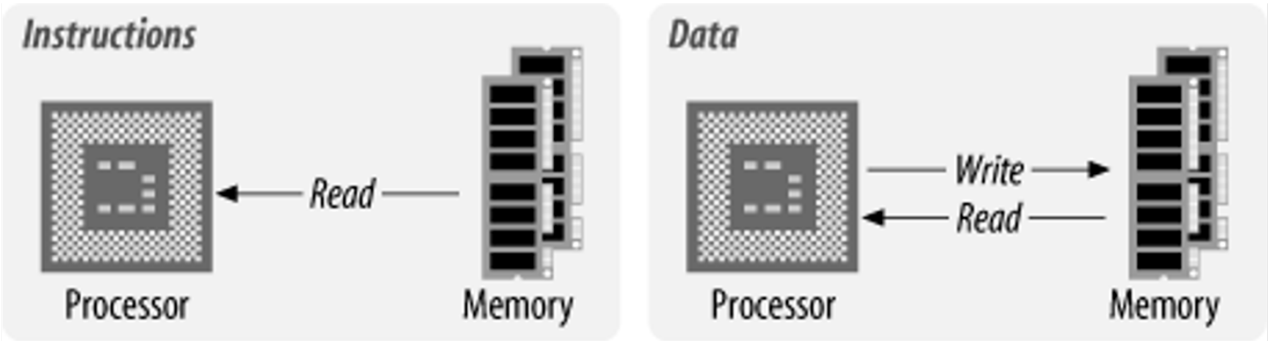

Figura 2. Fluxuri de date [1]

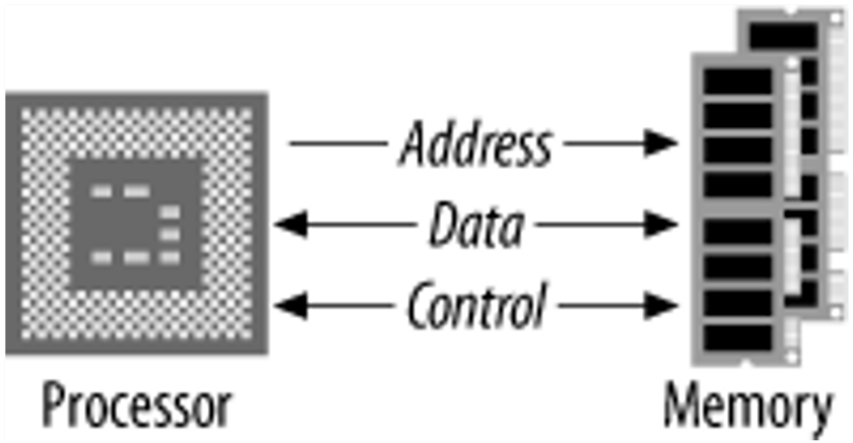

Figura 3. Magistrale [1]

Tipuri de arhitecturi pentru calculatoare

Arhitectura Princeton - mașina von Neumann

- Nu există diferență între date și instrucțiuni.
- Datele în sine nu au semnificație (0x4143 poate fi și instrucțiune și o dată – valoare într-un program).
- Datele și instrucțiunile împart aceeași memorie (secvențe de instrucțiuni dintr-un program pot fi tratate ca date în alt program) – este suficientă memorie pentru date și cod .
- Memoria este liniară (este ca un tablou unidimensional).
- Calculatoarele de uz general (PC, Mac)

Arhitectura Harvard

- Date și programele sunt stocate în memorii diferite
- Microcontrollere și aplicații embeded (ATMEGA328 din Arduino UNO: programele în Flash, datele în SRAM)

Mașina von Neumann

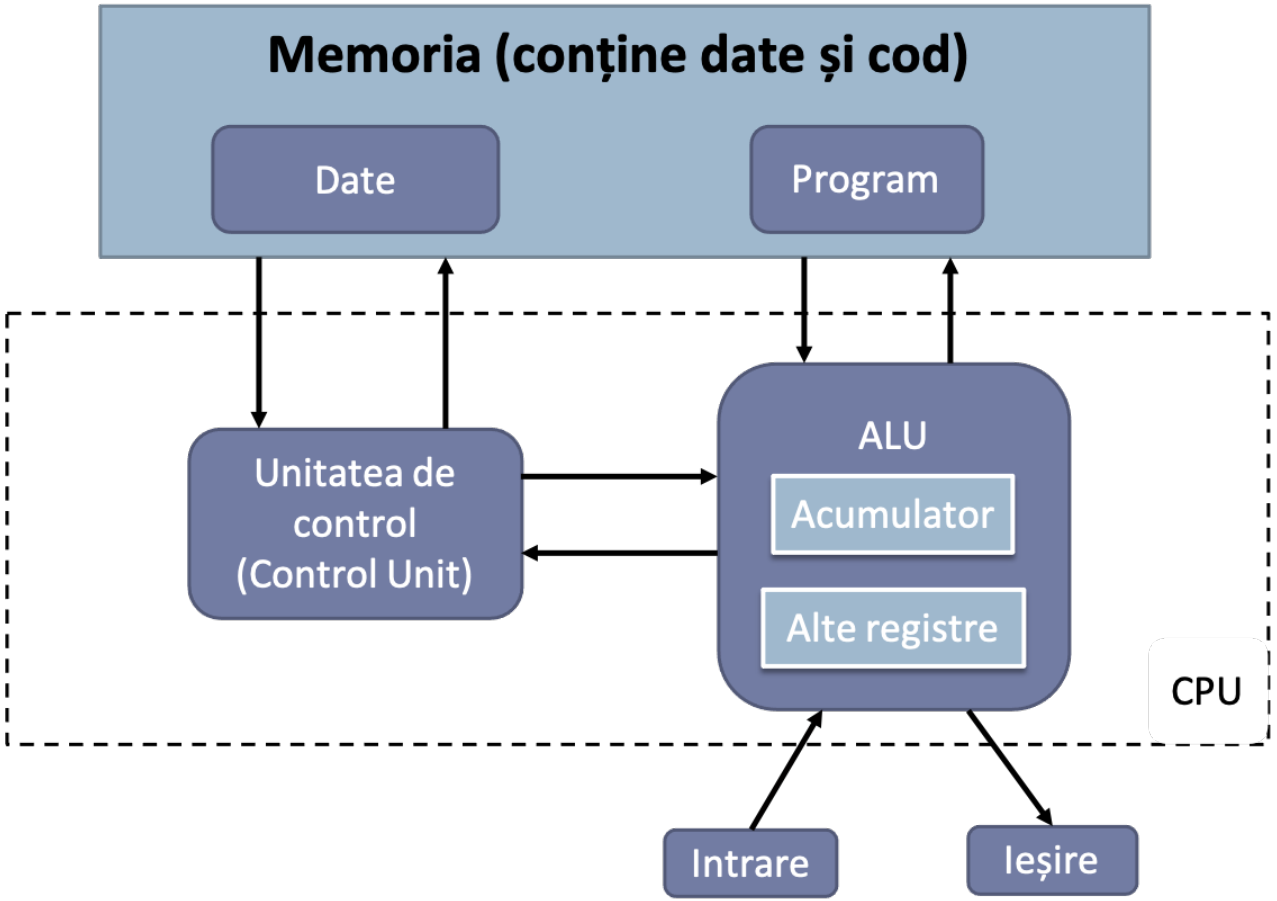

Figura 4. Mașina von Neumann

Procesorul x86-64 - Registre [2]

Registre generale

- 64 biți: RAX, RBX, RCX, RDX, R8 - R15
- 32 biți: EAX, EBX, ECX, EDX, R8d - R15d
- 16 biți: AX, BX, CX, DX,R8w - R15w
- 8 biți: AH, AL, BH, BL, CH, CL, DH, DL, R8b - R15b

Registre de segment - conțin adrese [3]

- CS – segmentul de cod
- DS – segmentul de date
- ES, FS, GS – extrasegment
- SS – segment de stivă

  *Observație:*În modul x86-64 registrele CS, DS, ES, SS nu sunt folosite (valoarea lor este 0(zero)).

Procesorul x86-64 - Registre [2]

Registre speciale

- DI/EDI/RDI, SI/ESI/RSI – registre index
- BP/EBP/RBP, SP/ESP/RSP – stiva (baza, vârful)
- EIP – registru care conține adresa următoarei instrucțiuni
- FLAGS/EFLAGS/RFLAGS – registrul indicatorilor

Cuprins

Sintaxa pentru asamblare GCC inline

Stiluri de a scrie în limbaj de asamblare

- Intel
- AT&T (folosit de GNU C Compiler - GCC)

Reguli de sintaxă

- **Denumirea registrelor** Numele registrelor sunt precedate de caracterul **%**
- **Ordinea operanzilor** Primul operand este sursa, al doilea operand este destinația.

```
mov eax, edx (Intel)
mov %edx, %eax (AT%T)
```

Sintaxa pentru asamblare GCC inline

Reguli de sintaxă

- **Dimensiunea operanzilor** este determinată de ultimul caracter al numelui instrucțiunii (op-code):
  - b - lucru pe octet (8 biți)
  - w - lucru pe cuvânt (16 biți)
  - l - lucru pe 32 biți
  - q - lucru pe 64 biți

  Forma corectă a instrucțiunii din slide-ul anterior:

```
movl %edx, %eax
```

- **Operand imediat** Acest tip de operand este marcat cu prefixul $.

```
addl $5, %eax
```

Sintaxa pentru asamblare GCC inline

Reguli de sintaxă

- **Operanzi din memorie** Lipsa prefixului unui operans înseamnă că este o adresă de memorie:
  Exemplu 1 - pune adresa variabilei `bar` în registrul `ebx`

```
movl $bar, %ebx
```

  Exemplu 2 - pune valoarea variabilei `bar` în registrul `ebx`

```
movl bar, %ebx
```

- **Indexarea** sau indirectarea se realizează punând registrul de index sau adresa celulei de memorie folosite la indirectare între paranteze.
  Exemplu

```
movl 8(%ebp), %eax
```

Cod inline

Observație

Exemplele sunt scrise pentru calculatoare cu procesoare de tip Intel x86; pentru alte microprocesoare trebuie folosite instrucțiunile specifice acestora.

Forma generală

```
asm("cod_in_limbaj_de_asamblare");
```

sau

```
__asm__("cod_in_limbaj_de_asamblare");
```

Cod inline

Exemplu 1

```
asm("movl %ebx, %eax");
```

sau

```
__asm__("movb %ch, (%ebx)");
```

Cod inline

Observație

Dacă trebuie să scriem mai multe instrucțiuni de asamblare, se folosește ; la sfârșitul fiecărei instrucțiuni.

Exemplu 2

```
int main(void) {
    /* Add 10 and 20 and
    store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"
              );

    /* Subtract 20 from 10 and
    store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "subl %ebx, %eax;"
              );

    /* Multiply 10 and 20
    and store result into register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "imull %ebx, %eax;"
              );

    return 0 ;
}
```

Specificarea operanzilor

Putem specifica operanzii.
Forma generală:

```
asm ( "cod_in_asamblare"
            : operanzi de iesire                    /* optional */
            : operanzi de intrare                   /* optional */
            : lista registrelor modificate          /* optional */
);
```

Exemplu 3

```
int val;

asm ("movl %%eax, %0;" : "=r" ( val ));
```

Specificarea operanzilor

Exemplu 4

```
int no = 100, val ;
asm ("movl %1, %%ebx;"
     "movl %%ebx, %0;"
     : "=r" ( val )      /* iesire */
     : "r" ( no )        /* intrare */
     : "%ebx"            /* registru modificat */
 );
```

Specificarea operatorilor

Exemplu 5

```
int arg1, arg2, suma;

__asm__ ("addl %%ebx, %%eax;"
         : "=a" (suma)
         : "a" (arg1), "b" (arg2));
```

Exemplu complet

Exemplu 6

```
#include <stdio.h>

int main() {

    int arg1, arg2, suma, dif, prod, cat, rest;

    printf( "Introduceti doua numere intregi : " );
    scanf( "%d%d", &arg1, &arg2 );

    /* Realizeaza operatiile dorite */
    __asm__ ( "addl %%ebx, %%eax;"
        : "=a" (suma)
        : "a" (arg1) , "b" (arg2) );

    __asm__ ( "subl %%ebx, %%eax;"
    : "=a" (dif)
    : "a" (arg1) , "b" (arg2) );

    __asm__ ( "imull %%ebx, %%eax;"
    : "=a" (prod)
    : "a" (arg1) , "b" (arg2) );
```

Exemplu complet

Exemplu 6 - continuare

```
    __asm__ ( "movl␣$0x0,␣%%edx;"
              "movl␣%2,␣%%eax;"
              "movl␣%3,␣%%ebx;"
               "idivl␣%%ebx;"
               : "=a" (cat), "=d" (rest)
               : "g" (arg1), "g" (arg2) );

    printf( "%d␣+␣%d␣=␣%d\n", arg1, arg2, suma );
    printf( "%d␣-␣%d␣=␣%d\n", arg1, arg2, dif );
    printf( "%d␣*␣%d␣=␣%d\n", arg1, arg2, prod );
    printf( "%d␣/␣%d␣=␣%d\n", arg1, arg2, cat );
    printf( "%d␣%%␣%d␣=␣%d\n", arg1, arg2, rest );

    return 0 ;
}
```

Exemplu complet

Exemplu 7

```
#include <stdio.h>

int main(void)
{
    int a = 200;
    int b = 1;
    printf("a este %d, b este %d\n", a, b);
    __asm__ ("xchg %0, %1;"
             :"+r" (a), "+r" (b)
        );

    printf("a este %d, b este %d\n", a, b);

    return 0;
}
```

Ce ar mai fi de spus

- Python și C [7]
- GDB [8]
- Valgrind [9]

Resurse utile

1. Catsoulis, J, *Designing Embedded Hardware*, 2nd Edition, O'Reilly, 2005
2. ***, x64 Cheat Sheet
3. Shubham Dubey, Segmentation in Intel x64(IA-32e) architecture - explained using Linux, 2020
4. Zhirkov, I, *Low-Level Programming - C, Assembly and Program Execution on Intel 64 Architecture*, Apress, 2017
5. jain.pk,Using Inline Assembly in C/C++, 2006
6. ***, How to Use Inline Assembly Language in C Code
7. Jim Anderson, Python Bindings: Calling C or C++ From Python
8. Keith Seitz,The GDB developer's GNU Debugger tutorial, Part 1: Getting started with the debugger
9. ***, Valgrind