

ARHITECTURA SISTEMELOR DE CALCUL SEMINAR 0x04

NOTIȚE SUPORT SEMINAR

Cristian Rusu

ÎNTREBĂRI SCURTE, EX. 1

a) adresa de memorie cea mai mare accesibilă este

ÎNTREBĂRI SCURTE, EX. 1

- a) adresa de memorie cea mai mare accesibilă este $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)
- b) instrucțiunea este *jne etichetă*, unde *jne* are opcode 0110
- adresa de memorie cea mai mare accesibilă este

ÎNTREBĂRI SCURTE, EX. 1

- a) adresa de memorie cea mai mare accesibilă este $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)
- b) instrucțiunea este *jne etichetă*, unde *jne* are opcode 0110
- adresa de memorie cea mai mare accesibilă este 2^{28}

ÎNTREBĂRI SCURTE, EX. 1

- a) adresa de memorie cea mai mare accesibilă este $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)
- b) instrucțiunea este *jne etichetă*, unde *jne* are opcode 0110
- adresa de memorie cea mai mare accesibilă este $2^{28} = 0.25\text{ GB}$
- c) avem instrucțiunea *add R1, R2*
- opcode este 0011

ÎNTREBĂRI SCURTE, EX. 1

- a) adresa de memorie cea mai mare accesibilă este $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)
- b) instrucțiunea este *jne etichetă*, unde *jne* are opcode 0110
- adresa de memorie cea mai mare accesibilă este $2^{28} = 0.25\text{ GB}$
- c) avem instrucțiunea add R1, R2
- opcode este 0011
 - operația suportă putem avea $2^{14} = 16384$ regiștri diferiți
- d) avem instrucțiunea add R1, R2, R3
- opcode este 0100

ÎNTREBĂRI SCURTE, EX. 1

- a) adresa de memorie cea mai mare accesibilă este $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)
- b) instrucțiunea este *jne etichetă*, unde *jne* are opcode 0110
- adresa de memorie cea mai mare accesibilă este $2^{28} = 0.25\text{ GB}$
- c) avem instrucțiunea add R1, R2
- opcode este 0011
 - operația suportă putem avea $2^{14} = 16384$ regiștri diferiți
- d) avem instrucțiunea add R1, R2, R3
- opcode este 0100
 - vom avea 9.33 biți pentru fiecare reprezentare a unui registru: două poziții suportă $2^9 = 512$ iar o poziție $2^{10} = 1024$

COD ASSEMBLY, EX. 2

- while loop

sum = 0;

i = 0;

while (i < 10) {

 sum = sum + i;

 i = i + 1;

}

```
.global main

main:
    ; initialize
    mov $0, i
    mov $0, sum

    ; while loop
et_loop:
    mov sum, %eax
    mov i, %ecx
    add %ecx, %eax
    mov %eax, sum

    inc i

    cmp $10, i
    jne et_loop

    ; afiseaza suma
    mov sum, %eax
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```


COD ASSEMBLY, EX. 2

- **while loop**

sum = 0;

i = 0;

while (i < 10) {

 sum = sum + i;

 i = i + 1;

}

- **rezultatul este?**

```
.global main

main:
    ; initialize
    mov $0, i
    mov $0, sum

    ; while loop
et_loop:
    mov sum, %eax
    mov i, %ecx
    add %ecx, %eax
    mov %eax, sum

    inc i

    cmp $10, i
    jne et_loop

    ; afiseaza suma
    mov sum, %eax
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

COD ASSEMBLY, EX. 2

- **while loop**

sum = 0;

i = 0;

while (i < 10) {

 sum = sum + i;

 i = i + 1;

}

- **rezultatul este?**

- 45

```
.global main

main:
    ; initialize
    mov $0, i
    mov $0, sum

    ; while loop
et_loop:
    mov sum, %eax
    mov i, %ecx
    add %ecx, %eax
    mov %eax, sum

    inc i

    cmp $10, i
    jne et_loop

    ; afiseaza suma
    mov sum, %eax
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

FOR LOOP, EX. 3

```
int sum = 0;  
int i = 0;  
for (i = 0; i < 10; i++)  
    sum += i;
```

- **care este diferența între `i++` și `++i`**
 - este rezultatul același?
 - e o variantă mai eficientă decât cealaltă?

FOR LOOP, EX. 3

```
int sum = 0;
int i = 0;
for (i = 0; i < 10; i++)
    sum += i;
```

- care este diferența între `i++` și `++i`
 - este rezultatul același?
 - e o variantă mai eficientă decât cealaltă?

```
int i = 1;
i++; // == 1 și i == 2
```

```
int i = 1;
++i; // == 2 și i == 2, deci compilatorul nu are nevoie de o variabilă temporară
```

FOR LOOP, EX. 3

- implementare mai eficientă
 - mai puțină memorie
 - mai puține accesări ale memoriei

```
.global main

main:
    ; initialize
    mov $0, i
    mov $0, sum

    ; while loop
et_loop:
    mov sum, %eax
    mov i, %ecx
    add %ecx, %eax
    mov %eax, sum

    inc i

    cmp $10, i
    jne et_loop

    ; afiseaza suma
    mov sum, %eax
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

FOR LOOP, EX. 3

- **implementare mai eficientă**
 - mai puțină memorie
 - mai puține accesări ale memoriei
- **totul în regiștri**
 - ca programul acesta să fie identic cu while
 - la sfârșit
 - `mov %eax, sum`
 - `mov $10, i`
- **se poate cu mai puține instrucțiuni?**

```
main:
    ; initializare
    xor %eax, %eax
    xor %ecx, %ecx

    ; while loop
et_loop:
    add %ecx, %eax

    inc %ecx

    cmp $10, %ecx
    jne et_loop

    ; afiseaza suma
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

FOR LOOP, EX. 3

- **implementare mai eficientă**
 - mai puțină memorie
 - mai puține accesări ale memoriei
- **totul în regiștri**
 - ca programul acesta să fie identic cu while
 - la sfârșit
 - `mov %eax, sum`
 - `mov $10, i`
- **se poate cu mai puține instrucțiuni?**
 - da, parcurgere inversă
- **se poate cu și mai puține instrucțiuni?**

```
main:
    ; initializare
    xor %eax, %eax
    mov $9, %ecx

    ; while loop
et_loop:
    add %ecx, %eax

    dec %ecx
    jnz et_loop

    ; afiseaza suma
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

FOR LOOP, EX. 3

- **implementare mai eficientă**
 - mai puțină memorie
 - mai puține accesări ale memoriei
- **totul în regiștri**
 - ca programul acesta să fie identic cu while
 - la sfârșit
 - `mov %eax, sum`
 - `mov $10, i`
- **se poate cu mai puține instrucțiuni?**
 - da, parcurgere inversă
- **se poate cu și mai puține instrucțiuni?**
 - da: `mov $45, %eax`

```
main:
    ; initializare
    xor %eax, %eax
    mov $9, %ecx

    ; while loop
et_loop:
    add %ecx, %eax

    dec %ecx
    jnz et_loop

    ; afiseaza suma
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

<https://godbolt.org/>

În godbolt încercați compilatoarele gcc și clang și flag-ul de optimizare -O3, iar pentru gcc puteți folosi și flag-ul -masm=att

FOR LOOP, EX. 3

- se poate mai eficient?
 - loop unrolling

```
int sum = 0;  
int i = 0;  
for (i = 0; i < 10; i++)  
    sum += i;
```

FOR LOOP, EX. 3

- se poate mai eficient?
 - loop unrolling

```
int sum = 0;
int i = 0;
for (i = 0; i < 10; i+=2) {
    sum += i;
    sum += i+1;
}
```

- de ce am vrea să facem așa ceva?

FOR LOOP, EX. 3

- se poate mai eficient?
 - loop unrolling

```
int sum = 0;
int i = 0;
for (i = 0; i < 10; i+=2) {
    sum += i;
    sum += i+1;
}
```

- de ce am vrea să facem așa ceva?
 - mai puține salturi

FOR LOOP, EX. 3

- se poate mai eficient?
 - loop unrolling

```
main:
    ; initialize
    xor %eax, %eax
    mov $10, %ecx

    ; while loop
et_loop:
    add %ecx, %eax
    dec %ecx

    add %ecx, %eax
    dec %ecx

    jnz et_loop

    sub $10, %eax

    ; afiseaza suma
    push %eax
    push $formatPrint
    call printf
    pop %ebx
    pop %ebx

    ; flush
    push $0
    call fflush
    pop %ebx

    ; exit
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

PIPELINE HAZARDS, EX. 4

a) $\%eax \leftarrow \%ebx + \%ecx$

$\%eax \leftarrow \%ebx + \%edx$

b) $\%ebx \leftarrow \%ecx + \%eax$

$\%eax \leftarrow \%edx + \%eax$

c) $\%eax \leftarrow \%ebx + \%ecx$

$\%edx \leftarrow \%eax + \%edx$

d) $\%eax \leftarrow 6$

$\%eax \leftarrow 3$

$\%ebx \leftarrow \%eax + 7$

PIPELINE HAZARDS, EX. 4

a) **%eax** \leftarrow %ebx + %ecx

%eax \leftarrow %ebx + %edx **WAW**

b) %ebx \leftarrow %ecx + %eax

%eax \leftarrow %edx + %eax

c) %eax \leftarrow %ebx + %ecx

%edx \leftarrow %eax + %edx

d) %eax \leftarrow 6

%eax \leftarrow 3

%ebx \leftarrow %eax + 7

PIPELINE HAZARDS, EX. 4

a) **%eax** \leftarrow %ebx + %ecx

%eax \leftarrow %ebx + %edx **WAW**

b) %ebx \leftarrow %ecx + **%eax**

%eax \leftarrow %edx + %eax **WAR**

c) %eax \leftarrow %ebx + %ecx

%edx \leftarrow %eax + %edx

d) %eax \leftarrow 6

%eax \leftarrow 3

%ebx \leftarrow %eax + 7

PIPELINE HAZARDS, EX. 4

a) **%eax** \leftarrow %ebx + %ecx

%eax \leftarrow %ebx + %edx **WAW**

b) %ebx \leftarrow %ecx + **%eax**

%eax \leftarrow %edx + %eax **WAR**

c) **%eax** \leftarrow %ebx + %ecx

%edx \leftarrow **%eax** + %edx **RAW**

d) %eax \leftarrow 6

%eax \leftarrow 3

%ebx \leftarrow %eax + 7

PIPELINE HAZARDS, EX. 4

a) **%eax** \leftarrow %ebx + %ecx

%eax \leftarrow %ebx + %edx **WAW**

b) %ebx \leftarrow %ecx + **%eax**

%eax \leftarrow %edx + %eax **WAR**

c) **%eax** \leftarrow %ebx + %ecx

%edx \leftarrow **%eax** + %edx **RAW**

d) **%eax** \leftarrow 6

%eax \leftarrow 3 **WAW**

%ebx \leftarrow **%eax** + 7 **RAW** , rezultatul poate fi 10 sau 13

BRANCH PREDICTION, EX. 6

- ce face algoritmul?

```
while (na > 0 && nb > 0)
{
    if (*A++ <= *B++) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}
```

BRANCH PREDICTION, EX. 6

- ce face algoritmul?
 - merge (interclasare)
- câte instrucțiuni de salt avem?

```
while (na > 0 && nb > 0)
{
    if (*A++ <= *B++) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}
```

BRANCH PREDICTION, EX. 6

- ce face algoritmul?
 - merge (interclasare)
- câte instrucțiuni de salt avem?
 - 4

```
while (na > 0 && nb > 0)
{
    if (*A++ <= *B++) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}
```

BRANCH PREDICTION, EX. 6

- ce face algoritmul?
 - merge (interclasare)
- câte instrucțiuni de salt avem?
 - 4
- predicția pentru fiecare?

```
while (na > 0 && nb > 0) 1
{
    if (*A++ <= *B++) 2 {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) { 3
    *C++ = *A++; --na;
}

while (nb > 0) { 4
    *C++ = *B++; --nb;
}
```

BRANCH PREDICTION, EX. 6

- **ce face algoritmul?**
 - merge (interclasare)
- **câte instrucțiuni de salt avem?**
 - 4
- **predicția pentru fiecare?**
 - Salt 1: sare mereu
 - Salt 2: în general, nu știm
 - Salt 3: sare mereu
 - Salt 4: sare mereu

```
while (na > 0 && nb > 0)
{
    if (*A++ <= *B++) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}
```

BRANCH PREDICTION, EX. 6

- ce face algoritmul?
 - merge (interclasare)
- câte instrucțiuni de salt avem?
 - 4
- predicția pentru fiecare?
 - Salt 1: sare mereu
 - Salt 2: în general, nu știm
 - Salt 3: sare mereu
 - Salt 4: sare mereu
- cum eliminăm Saltul 2?
 - $\text{int cmp} = (*A \leq *B)$
 - $\text{int min} = *B \wedge ((*B \wedge *A) \& (-\text{cmp}))$
 - $*C++ = \text{min}$
 - $A += \text{cmp}, na -= \text{cmp}$
 - $B += !\text{cmp}, nb -= !\text{cmp}$

```
while (na > 0 && nb > 0)
{
    if (*A++ <= *B++) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}
```

TO UPPER, EX. 7

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

TO UPPER, EX. 7

- un algoritm simplu de toUpper()

```
void toUpper(char *buff, int count) {  
    for (int i = 0; i < count; ++i)  
    {  
        if (buff[i] >= 'a' && buff[i] <= 'z')  
            buff[i] -= 32;  
    }  
}
```

- branchless?

TO UPPER, EX. 7

- un algoritm simplu de toUpper()

```
void toUpper(char *buff, int count) {  
    for (int i = 0; i < count; ++i)  
    {  
        if (buff[i] >= 'a' && buff[i] <= 'z')  
            buff[i] -= 32;  
    }  
}
```

- branchless? mai bine?

```
void toUpper(char *buff, int count) {  
    for (int i = 0; i < count; ++i)  
    {  
        buff[i] = buff[i]*!((buff[i] >= 'a' && buff[i] <= 'z'))  
            + (buff[i] - 32)*(buff[i] >= 'a' && buff[i] <= 'z');  
    }  
}
```

TO UPPER, EX. 7

- un algoritm simplu de toUpper()

```
void toUpper(char *buff, int count) {  
    for (int i = 0; i < count; ++i)  
    {  
        if (buff[i] >= 'a' && buff[i] <= 'z')  
            buff[i] -= 32;  
    }  
}
```

- branchless?

```
void toUpper(char *buff, int count) {  
    for (int i = 0; i < count; ++i)  
    {  
        buff[i] -= 32*(buff[i] >= 'a' && buff[i] <= 'z');  
    }  
}
```

COD ASSEMBLY. EX. 7 (VECHI)

```
        .globl f
f:
        movl    $1, %r8d
        jmp     .LBB0_1
.LBB0_6:
        incl    %r8d
.LBB0_1:
        movl    %r8d, %ecx
        imull   %ecx, %ecx
        movl    $1, %edx
.LBB0_2:
        movl    %edx, %edi
        imull   %edi, %edi
        movl    $1, %esi
        .align  16, 0x90
.LBB0_3:
        movl    %esi, %eax
        imull   %eax, %eax
        addl    %edi, %eax
        cmpl    %ecx, %eax
        je      .LBB0_7
        cmpl    %edx, %esi
        leal    1(%rsi), %eax
        movl    %eax, %esi
        jl      .LBB0_3
        cmpl    %r8d, %edx
        leal    1(%rdx), %eax
        movl    %eax, %edx
        jl      .LBB0_2
        jmp     .LBB0_6
.LBB0_7:
        pushq   %rax
.Ltmp0:
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   printf
        movl    $1, %eax
        popq    %rcx
        retq

.L.str:
        .asciz  "%d %d\n"
        .size   .L.str, 7
```

COD ASSEMBLY. EX. 7 (VECHI)

```
        .globl f
f:
        movl    $1, %r8d
        jmp     .LBB0_1
.LBB0_6:
        incl    %r8d
.LBB0_1:
        movl    %r8d, %ecx
        imull   %ecx, %ecx
        movl    $1, %edx
.LBB0_2:
        movl    %edx, %edi
        imull   %edi, %edi
        movl    $1, %esi
        .align  16, 0x90
.LBB0_3:
        movl    %esi, %eax
        imull   %eax, %eax
        addl    %edi, %eax
        cmpl    %ecx, %eax
        je      .LBB0_7
        cmpl    %edx, %esi
        leal    1(%rsi), %eax
        movl    %eax, %esi
        jl      .LBB0_3
        cmpl    %r8d, %edx
        leal    1(%rdx), %eax
        movl    %eax, %edx
        jl      .LBB0_2
        jmp     .LBB0_6
.LBB0_7:
        pushq   %rax
.Ltmp0:
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   printf
        movl    $1, %eax
        popq    %rcx
        retq

.L.str:
        .asciz  "%d %d\n"
        .size   .L.str, 7
```

verifică $x^2 + y^2 = z^2$, cu condiția $x \leq y$

