Laborator 5

1. Conditional jumps

Afișarea mesajului să se facă numai dacă conținutul registrului eax este mai mare decât cel din ebx. Modificați și valoarea registrelor pentru a face în continuare afișarea mesajului "Hello, World!".

2. More hellos

Modificați programul astfel încât să mai afișeze încă un mesaj ('Goodbye, World!'). => mov eax, 2

Folosind instrucțiuni de tip jump, modificați programul astfel încât să afișeze de N ori 'Hello, World!', unde N este dat prin intermediul registrului ECX. Evitați ciclarea la infinit.

%include "printf32.asm"

section .data

myString: db "Hello, World!", 0

myGoodbyeString: db "Goodbye, World!", 0

section .text

global main

extern printf

main:

mov ecx, 6 ; N = valoarea registrului ecx

mov eax, 2

mov ebx, 1

cmp eax, ebx

jg print ; TODO1: eax > ebx?

ret

print:

PRINTF32 `%s\n\x0`, myString

; TODO2.2: afisati "Hello, World!" de N ori

dec ecx

cmp ecx, 0

jg print

; TODO2.1: afisati "Goodbye, World!"

PRINTF32 `%s\n\x0`, myGoodbyeString

ret

3. Grumpy jumps

Modificați valorile registrelor EAX si EBX astfel încât la rularea lui să se afișeze mesajul Well done!. Pentru a determina valorile necesare pentru registrele EAX si EBX vă recomandăm să folosiți GDB.

%include "printf32.asm"

section .data

wrong: db 'Not today, son.',0

right: db 'Well done!',0

section .text

global main

extern printf

main:

mov eax, 0xdeadc0de ; TODO3.1: modify eax register => mov eax, 0x1

mov ebx, 0x1337ca5e ; TODO3.1: modify ebx register

=> mov ebx, 0x4

mov ecx, 0x5 ; hardcoded; DO NOT change

cmp eax, ebx ; verifica eax=0x1 > ebx=0x4

jns bad ; nu se verifica cond => nu afiseaza bad

cmp ecx, ebx ; verifica ecx=0x5 < ebx=0x4

jb bad ; nu se verifica cond => nu afiseaza bad

add eax, ebx

xor eax, ecx

jnz bad

good:

PRINTF32 `%s\n\x0`, right

ret

bad:

PRINTF32 `%s\n\x0`, wrong

ret

4. Sets

Se implementeaza operații pe mulțimi ce pot conține elemente între 0 și 31. Un mod eficient de a face asta (atât din punct de vedere al spațiului cât și al vitezei) ar fi să reprezentăm mulțimile astfel încât un registru să reprezinte o mulțime. Fiecare bit din registru va reprezenta un element din mulțime (dacă bit-ul i este setat atunci mulțimea conține elementul i)

Exemplu: dacă eax ar conține reprezentarea mulțimii {0,2,4}, valoarea registrului ar fi 2^0 + 2^2 + 2^4 = 1 + 4 + 16 = 21. Documentați-vă despre instrucțiunile disponibile pe arhitectura x86.

- Aveți definite 2 mulțimi. Ce valori conțin? Realizați reuniunea celor 2 mulțimi.

- Folosiți instrucțiunea or pentru a adăuga două elemente noi în mulțime.

Folosiți-vă de faptul că mulțimile curente, deși au “spațiu” pentru 32 de biți, au doar 8 biți folosiți. Dacă veți face or cu un număr mai mare de 255 (0xff, 2^8-1) care are doi biți activi, veți adăuga practic două elemente noi la mulțime.

- Faceți intersecția celor 2 mulțimi.

- Determinați elementele care lipsesc din mulțimea eax pentru ca aceasta să fie completă.

Adică trebuie să faceți complementul numărului folosind instrucțiunea not.

- Eliminați un element din prima mulțime.

- Faceți diferența între mulțimi.

Pentru a vă ajuta în afișare puteți folosi macro-ul PRINTF32. De exemplu:

PRINTF32 `Reuniunea este: \x0`

PRINTF32 `%u\n\x0`, eax

%include "printf32.asm"

section .text

global main

extern printf

main:

; Cele doua multimi se gasesc in eax si ebx

mov eax, 139

mov ebx, 169

PRINTF32 `%u\n\x0`, eax ; afiseaza prima multime

PRINTF32 `%u\n\x0`, ebx ; afiseaza cea de-a doua multime

; TODO1: reuniunea a doua multimi

mov edx, eax ; se adauga in edx toate elem din eax

or edx, ebx ; se adauga in edx elem egale cu cele din edx

PRINTF32 `%u\n\x0`, edx

; TODO2: adaugarea unui element in multime

or eax, 0x300 ; Adaugam in multimele elementele 8 si 9

; 0x300 <==> ...11|0000|0000

PRINTF32 `%u\n\x0`, eax

; TODO3: intersectia a doua multimi

mov edx, eax ; se adauga in edx toate elem din eax

and edx, ebx ; se adauga in edx toate elementele pe care le avea ebx

PRINTF32 `%u\n\x0`, edx

; TODO4: complementul unei multimi

mov edx, eax

not edx ; se neaga biti din edx

PRINTF32 `%u\n\x0`, edx

; TODO5: eliminarea unui element

; Vom elimina elementul 3 din prima multime

mov edx, 1 ; se incarca val 1 in edx

shl edx, 3 ; se shifteaza la stanga cu 3 pozitii pe bitii edx ului

not edx ; se neaga bitii

and eax, edx ; se verifica daca elementul se afla in eax

PRINTF32 `%u\n\x0`, eax

; TODO6: diferenta de multimi EAX-EBX

; Vom folosi multimile initiale pentru diferenta

mov eax, 139 ; eax = 10001011

mov ebx, 169 ; ebx = 10101001

not ebx ; ebx = 01010110

and eax, ebx ; 1+1 = 1, else 0

PRINTF32 `%u\n\x0`, eax

xor eax, eax

ret

5. Min

Calculați minimul dintre numerele din 2 registre (eax și ebx) folosind o instrucțiune de salt și instrucțiunea xchg.

%include "printf32.asm"

section .text

global main

extern printf

main:

;cele doua numere se gasesc in eax si ebx

mov eax, 4

mov ebx, 1

; TODO: aflati minimul

cmp eax, ebx

jl print\_min

xchg eax, ebx ; interschimba eax si ebx

print\_min:

PRINTF32 `%d\n\x0`, eax ; afiseaza minimul

xor eax, eax

ret

6. Fibonacci

Calculați al N-lea număr Fibonacci, unde N este dat prin intermediul registrului eax.

%include "printf32.asm"

section .text

global main

extern printf

main:

mov ecx, 7 ; vrem sa aflam al N-lea numar; N = 7

; TODO: calculati al N-lea numar fibonacci (f(0) = 0, f(1) = 1)

mov eax, 0

mov ebx, 1

fibonacci:

dec ecx ; decrementam cu 1 <=> ecx - 1

test ecx, ecx ; se verifica daca ecx a ajuns la 0

je print

add eax, ebx ; se face suma dintre eax si ebx

xchg eax, ebx ; se interschimba valorile pentru a trece mai departe in sir

jmp fibonacci

print:

PRINTF32 `%d\n\x0`, ebx

xor eax, eax ; se seteaza poti bitii pe 0 pentru a se returna in main

ret

Laborator 6

0. Recapitulare: Descompunerea unui număr în puteri ale lui 2

Numărul va fi pasat prin registrul eax.

De exemplu, pentru eax = 211, programul vostru va afișa:

1

2

16

64

128

%include "../utils/printf32.asm"

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov eax, 211 ; to be broken down into powers of 2

mov ebx, 1 ; stores the current power

; TODO - print the powers of 2 that generate number stored in EAX

etic:

cmp ebx, eax ; verifica daca puterea a depasit numarul

jg exit ; daca da, iese din bucla

test ebx, eax ; verifica daca exista o corespundenta intre nr si puterea curenta

; daca rez este 0 => trece la urm putere

jz incr

PRINTF32 `%u\n\x0`, ebx

incr:

shl ebx,1 ; o înmulțire a puterii curente a lui 2 prin deplasarea valorii în stânga cu un bit (shl ebx, 1).

jmp etic ; verifica următoarea putere a lui 2

exit:

leave

ret

1. Înmulțirea a două numere reprezentate pe un octet

Pentru a le putea accesa folosim o construcție de tipul byte [register].

Atunci cănd facem înmulțire procesul este următorul, așa cum este descris și aici:

-> Plasăm deînmulțitul în registrul de deînmulțit, adică:

dacă facem operații pe un byte (8 biți, un octet), plasăm deînmulțitul în registrul AL;

-> dacă facem operații pe un cuvânt (16 biți, 2 octeți, plasăm deînmulțitul în registrul AX;

-> dacă facem operații pe un dublu cuvânt (32 de biți, 4 octeți), plasăm deînmulțitul în registrul EAX.

-> Înmulțitorul este transmis ca argument mnemonicii mul. Înmulțitorul trebuie să aibă aceeași dimensiune ca deînmulțitul.

-> Rezultatul este plasat în două registre (partea high și partea low).

Testați programul. Încercați alte valori pentru num1 și num2.

2. Înmulțirea a două numere

Actualizați zona marcată cu TODO în fișierul multiply.asm pentru a permite înmulțirea și a numelor de tip word și dword, adică num1\_dw cu num2\_dw, respectiv num1\_dd și num2\_dd.

Pentru înmulțirea numerelor de tip word (pe 16 biți), componentele sunt dispuse astfel:

-> În registrul AX se plasează deînmulțitul.

-> Argumentul instrucțiunii, înmulțitorul, mul (posibil un alt registru) este pe 16 biți (fie valoare fie un registru precum BX, CX, DX).

-> Rezultatul înmulțirii este dispus în perechea DX:AX, adică partea “high” a rezultatului în registrul DX, iar partea “low” a rezultatului în registrul AX.

Pentru înmulțirea numerelor de tip dword (pe 32 biți), componentele sunt dispuse astfel:

-> În registrul EAX se plasează deînmulțitul.

-> Argumentul instrucțiunii, înmulțitorul, mul (posibil un alt registru) este pe 32 biți (fie valoare fie un registru precum EBX, ECX, EDX).

-> Rezultatul înmulțirii este dispus în perechea EDX:EAX, adică partea “high” a rezultatului în registrul EDX, iar partea “low” a rezultatului în registrul EAX.

La afișarea rezultatului folosiți macro-ul PRINTF32 pentru a afișa cele două registre care conțin rezultatul:

-> Registrele DX și AX pentru înmulțirea numerelor de tip word.

-> Registrele EDX și EAX pentru înmulțirea numerelor de tip dword.

%include "../utils/printf32.asm"

section .data

num1 db 43 ; numar pe un octet

num2 db 39

num1\_w dw 1349 ; numar pe un cuvant

num2\_w dw 9949

num1\_d dd 134932 ; numar pe un dublu cuvant

num2\_d dd 994912

section .text

extern printf

global main

main:

push ebp ; initializam stiva

mov ebp, esp

; Facem înmulțiri pentru diferite dimensiuni de date

; Multiplication for db

mov al, byte [num1] ; incarcam num1 in reg al (deinmultitul)

mov bl, byte [num2] ; incarcam num2 in reg bl (inmultitorul)

mul bl ; ax = al \* bl

; Print result in hexa

xor ebx, ebx ; anulam ebx pt urm instructiune

mov bx, ax ; incarcam rezultatul in bx

PRINTF32 `Rezultatul este: 0x%hx\n\x0`, eax

; Multiplication for dw

mov ax, word [num1\_w]

mov bx, word [num2\_w]

mul bx

; Print result in hexa

xor ebx, ebx

mov bx, dx

xor ebx, ebx

mov bx, ax

PRINTF32 `Rezultatul este: 0x%hx%hx\n\x0`, edx, eax

; Multiplication for dd

mov eax, dword [num1\_d]

mov ebx, dword [num2\_d]

mul ebx

; Print result in hexa

PRINTF32 `Rezultatul este: 0x%x%x\n\x0`, edx, eax

leave

ret

3. Suma primelor N numere naturale

%include "../utils/printf32.asm"

section .data

num dd 100

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov ecx, [num] ; Use ecx as counter for computing the sum.

xor eax, eax ; Use eax to store the sum. Start from 0.

add\_to\_sum:

add eax, ecx

loop add\_to\_sum ; Decrement ecx. If not zero, add it to sum.

mov ecx, [num]

PRINTF32 `Sum(%u): %u\n\x0`, ecx, eax

leave

ret

4. Suma pătratelor primelor N numere naturale

Registrele eax și edx le veți folosi la înmulțirea pentru ridicarea la putere (în instrucțiunea mul). Astfel că nu veți mai putea folosi (ușor) registrul eax pentru stocarea sumei pătratelor. Pentru a reține suma pătratelor aveți două variante:

(mai simplu) Folosiți registrul ebx pentru a reține suma pătratelor.

(mai complicat) Înainte de a opera registrul eax salvați valoarea sa pe stivă (folosind instrucțiunea push), apoi faceți operațiile necesare și apoi restaurați valoarea salvată (folosind instrucțiunea pop).

Pentru verificare, suma pătratelor primelor 100 de numere naturale este 338350.

Varianta simpla

%include "../utils/printf32.asm"

section .data

num dd 100

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov ecx, [num] ; Use ecx as counter for computing the sum.

xor ebx, ebx ; Use ebx to store the sum. Start from 0.

add\_to\_sum:

mov eax, ecx

mul eax

add ebx, eax

loop add\_to\_sum ; Decrement ecx. If not zero, add it to sum.

mov ecx, [num]

PRINTF32 `Sum(%u): %u\n\x0`, ecx, ebx

leave

ret

Varianta complicata

%include "../utils/printf32.asm"

section .data

num dd 100

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov ecx, [num] ; Use ecx as counter for computing the sum.

xor eax, eax ; Use eax to store the sum. Start from 0.

add\_to\_sum:

push eax

mov eax, ecx

mul eax

mov edx, eax

pop eax

add eax, edx

loop add\_to\_sum ; Decrement ecx. If not zero, add it to sum.

mov ecx, [num]

PRINTF32 `Sum(%u): %u\n\x0`, ecx, eax

leave

ret

5. Suma elementelor dintr-un vector (array) reprezentate pe un octet (bytes, reprezentare pe 8 biți)

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 10

section .data

byte\_array db 8, 19, 12, 3, 6, 200, 128, 19, 78, 102

word\_array dw 1893, 9773, 892, 891, 3921, 8929, 7299, 720, 2590, 28891

dword\_array dd 1392849, 12544, 879923, 8799279, 7202277, 971872, 28789292, 17897892, 12988, 8799201

big\_numbers\_array dd 20000001, 3000000, 3000000, 23456789, 56789123, 123456789, 987654321, 56473829, 87564836, 777777777

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter.

xor eax, eax ; Use eax to store the sum.

xor edx, edx ; Store current value in dl; zero entire edx.

add\_byte\_array\_element:

mov dl, byte [byte\_array + ecx - 1]

add eax, edx

loop add\_byte\_array\_element ; Decrement ecx, if not zero, add another element.

PRINTF32 `Array sum is %u\n\x0`, eax

leave

ret

6. Suma elementelor dintr-un vector

Când veți calcula adresa unui element din array, veți folosi construcție de forma:

base + size \* index

În construcția de mai sus:

-> base este adresa vectorului (adică word\_array sau dword\_array)

-> size este lungimea elementului vectorului (adică 2 pentru vector de word (16 biți, 2 octeți) și 4 pentru vector de dword (32 de biți, 4 octeți)

-> index este indexul curent în cadrul vectorului

Suma elementelor celor trei vectori trebuie să fie:

-> sum(byte\_array): 575

-> sum(word\_array): 65799

-> sum(dword\_array): 74758117

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 10

section .data

byte\_array db 8, 19, 12, 3, 6, 200, 128, 19, 78, 102

word\_array dw 1893, 9773, 892, 891, 3921, 8929, 7299, 720, 2590, 28891

dword\_array dd 1392849, 12544, 879923, 8799279, 7202277, 971872, 28789292, 17897892, 12988, 8799201

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter.

xor eax, eax ; Use eax to store the sum.

xor edx, edx ; Store current value in dl; zero entire edx.

add\_byte\_array\_element:

mov dl, byte [byte\_array + ecx - 1]

add eax, edx

loop add\_byte\_array\_element ; Decrement ecx, if not zero, add another element.

PRINTF32 `Array sum is %u\n\x0`, eax

mov ecx, ARRAY\_SIZE

xor eax, eax

xor edx, edx

add\_word\_array\_element:

mov dx, word [word\_array + 2 \* ecx - 2]

add eax, edx

loop add\_word\_array\_element ; Decrement ecx, if not zero, add another element.

PRINTF32 `Array sum is %u\n\x0`, eax

mov ecx, ARRAY\_SIZE

xor eax, eax

xor edx, edx

add\_dword\_array\_element:

mov edx, dword [dword\_array + 4 \* ecx - 4]

add eax, edx

loop add\_dword\_array\_element ; Decrement ecx, if not zero, add another element.

PRINTF32 `Array sum is %u\n\x0`, eax

leave

ret

7. Suma pătratelor elementelor dintr-un vector

Puteți folosi vectorul dword\_array, având însă grijă ca suma pătratelor elementelor conținute să poată fi reprezentată pe 32 de biți.

Dacă folosiți construcția de mai jos (vector cu 10 elemente)

dword\_array dd 1392, 12544, 7992, 6992, 7202, 27187, 28789, 17897, 12988, 17992

suma pătratelor va fi 2704560839.

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 10

section .data

dword\_array dd 1392, 12544, 7992, 6992, 7202, 27187, 28789, 17897, 12988, 17992

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

xor eax, eax ; resetează registrul eax la zero

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter.

xor eax, eax ; Use eax to store the sum.

xor edx, edx ; Store current value in dl; zero entire edx.

add\_dword\_array\_element:

push eax ; salvează valoarea curentă a sumei pe stivă

mov eax, dword [dword\_array + 4 \* ecx - 4] ; încarcă elementul curent al vectorului în eax

mov edx, dword [dword\_array + 4 \* ecx - 4] ; încarcă elementul curent al vectorului în edx

mul edx ; înmulțește valorile pentru a obține pătratul elementului

mov edx, eax ; mută rezultatul (pătratul) în edx

pop eax ; restaurează valoarea originală a sumei din stivă

add eax, edx ; adaugă pătratul elementului la suma curentă

loop add\_dword\_array\_element ; decrementează ecx și continuă bucla dacă nu a ajuns la zero

PRINTF32 `Array sum is %u\n\x0`, eax

leave

ret

8. Împărțirea a două numere

Dacă împărțitorul este de tip byte (8 biți), componentele sunt dispuse astfel:

deîmpărțitul este plasat în registrul AX

-> argumentul instrucțiunii div are 8 biți și poate fi reprezentat de un registru sau de o valoare imediată

-> câtul este dispus în AL

-> restul este dispus în AH

Dacă împărțitorul este de tip word (16 biți), componentele sunt dispuse astfel:

-> deîmpărțitul este dispus în perechea DX:AX, adică partea sa high în registrul DX, iar partea low în AX

-> argumentul instrucțiunii div are 16 biți și poate fi reprezentat de un registru sau de o valoare imediată

-> câtul este dispus în AX

-> restul este dispus în DX

Dacă împărțitorul este de tip dword (32 de biți), componentele sunt dispuse astfel:

-> deîmpărțitul este dispus în perechea EDX:EAX, adică partea sa high în registrul EDX, iar partea low în EAX

-> argumentul instrucțiunii div are 32 de biți și poate fi reprezentat de un registru sau de o valoare imediată

-> câtul este dispus în EAX

-> restul este dispus în EDX

Dacă programul vă dă “SIGFPE. Arithmetic exception”, cel mai probabil aţi uitat să iniţializaţi partea superioară a deîmpărţitului (AH, DX sau EDX).

%include "../utils/printf32.asm"

section .data

dividend1 db 91

divisor1 db 27

dividend2 dd 67254

divisor2 dw 1349

dividend3 dq 69094148

divisor3 dd 87621

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

xor eax, eax ; inițializează EAX cu zero

mov al, byte [dividend1] ; încarcă deîmpărțitul 1 în AL

mov bl, byte [divisor1] ; încarcă deîmpărțitul 1 în AL

div bl ; împarte AL la BL, rezultatul se află în AL, restul în BL

xor ebx, ebx ; inițializează EBX cu zero

mov bl, al ; mută rezultatul (câtul) în BL

PRINTF32 `Quotient: %hhu\n\x0`, ebx ; afișează câtul

xor ebx, ebx ; inițializează EBX cu zero

mov bl, ah ; mută restul în BL

PRINTF32 `Remainder: %hhu\n\x0`, ebx ; afișează restul

mov edx, [dividend2] ; încarcă deîmpărțitul 2 în EDX

mov ax, dx ; încarcă partea de jos a deîmpărțitului 2 în AX

shr edx, 16 ; shiftare la dreapta a deîmpărțitului 2 pentru a obține partea de sus în DX

mov bx, word [divisor2] ; încarcă împărțitorul 2 în BX

div bx ; împarte DX:AX la BX, rezultatul se află în AX, restul în DX

xor ebx, ebx ; inițializează EBX cu zero

mov bx, ax ; mută rezultatul (câtul) în BX

PRINTF32 `Quotient: %hhu\n\x0`, ebx ; afișează câtul

xor ebx, ebx ; inițializează EBX cu zero

mov bx, dx ; mută restul în BX

PRINTF32 `Remainder: %hu\n\x0`, ebx ; afișează restul

mov eax, dword [dividend3] ; încarcă partea de jos a deîmpărțitului 3 în EAX

mov edx, dword [dividend3 + 4] ; încarcă partea de sus a deîmpărțitului 3 în EDX

mov ebx, dword [divisor3] ; încarcă împărțitorul 3 în EBX

div ebx ; împarte EDX:EAX la EBX, rezultatul se află în EAX, restul în EDX

PRINTF32 `Quotient: %u\nRemainder: %u\n\x0`, eax, edx ; afișează câtul și restul

leave

ret

9. Numărul de numere negative și pozitive dintr-un vector

Folosiți intrucțiunea cmp și mnemonici de salt condițional.

Intrucțiunea inc urmată de un registru incrementează cu 1 valoarea stocată în acel registru.

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 10

section .data

dword\_array dd 1392, -12544, -7992, -6992, 7202, 27187, 28789, -17897, 12988, 17992

section .text

extern printf

global main

main:

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter.

xor ebx, ebx ; Store positive number in ebx.

xor edx, edx ; Store negative number in edx.

next\_element:

mov eax, dword [dword\_array + ecx\*4 - 4] ; încarcă elementul curent al vectorului în eax

cmp eax, 0 ; compară elementul cu zero

jl add\_to\_neg ; sare la eticheta add\_to\_neg dacă elementul este mai mic decât zero

inc ebx ; incrementează numărul de numere pozitive (ebx)

jmp test\_end ; sare peste eticheta add\_to\_neg pentru a testa sfârșitul buclei

add\_to\_neg:

inc edx ; incrementează numărul de numere negative (edx)

test\_end:

loop next\_element ; decrementează ecx și continuă bucla dacă nu a ajuns la zero

PRINTF32 `Num pos is %u, num neg is %u\n\x0`, ebx, edx

ret

10. Numărul de numere pare și impare dintr-un vector

Puteți folosi instrucțiunea div pentru a împărți un număr la 2 și pentru a compara apoi restul împărțirii cu 0.

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 10

section .data

dword\_array dd 1392, 12544, 7991, 6992, 7202, 27187, 28789, 17897, 12988, 17992

section .text

extern printf

global main

main:

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter.

xor esi, esi ; Store even number in esi.

xor edi, edi ; Store odd number in edi.

next\_element:

; We need to initialize the dividend (EDX:EAX) to 0:array\_element

xor edx, edx ; Reset edx to 0

mov eax, dword [dword\_array + ecx\*4 - 4] ; Load current element of the array into eax

; Store the divisor (2) in EBX.

mov ebx, 2

div ebx ; Divide eax by ebx. Quotient is stored in eax, remainder in edx.

; EDX stores the remainder. If remainder is 0, the number is even, otherwise the number is odd.

test edx, edx ; Test if edx is 0

jne add\_to\_odd ; Jump to add\_to\_odd if remainder is not 0

inc esi ; Increment the even number count (esi)

jmp test\_end ; Jump over add\_to\_odd to test the end of the loop

add\_to\_odd:

inc edi ; Increment the odd number count (edi)

test\_end:

loop next\_element ; Decrement ecx and continue the loop if it's not zero

PRINTF32 `Num even is %u, num odd is %u\n\x0`, esi, edi

ret

Laborator 7

0. Fibonacci sum

%include "printf32.asm"

section .data

N dd 9 ; Compute the sum of the first N Fibonacci numbers

sum\_print\_format db "Sum first %d Fibonacci numbers is %d", 10, 0

section .text

extern printf

global main

main:

push ebp

mov ebp, esp

; The calling convention requires saving and restoring `ebx` if modified

push ebx

xor eax, eax ; Store the sum in eax

mov ecx, [N] ; Load the value of N into ecx

mov ebx, 0 ; Initialize the first Fibonacci number (0)

mov edx, 1 ; Initialize the second Fibonacci number (1)

calc\_fibo:

add eax, ebx ; Add the current Fibonacci number (ebx) to the sum (eax)

add ebx, edx ; Compute the next Fibonacci number by adding the two previous numbers

xchg ebx, edx ; Swap ebx and edx to update their values for the next iteration

; The `xchg` above is equivalent to the following:

; push eax

; mov eax, ebx

; mov ebx, edx

; mov edx, eax

; pop eax

loop calc\_fibo ; Decrement ecx and continue the loop if it's not zero

push eax ; Push the sum (eax) as the second argument for printf

push dword [N] ; Push the value of N as the first argument for printf

push sum\_print\_format ; Push the format string for printf

call printf ; Call the printf function to print the sum

add esp, 12 ; Adjust the stack pointer to clean up the pushed arguments

; Restore the `ebx` that was previously saved

pop ebx

xor eax, eax ; Set eax to 0 before returning

leave ; Restore the stack frame

ret ; Return from the function

Laborator 8

Operatii asupra stivei

Stiva poate fi modificata in 2 moduri:

1. Prin utilizarea instructiunilor special implementate pentru lucrul cu stiva, dintre care cele mai uzuale sunt push si pop:

%include "io.asm"

section .text

global CMAIN

CMAIN:

mov eax, 7

mov ebx, 8

add eax, ebx

push eax ; pune pe stiva continutul registrului eax

mov eax, 10 ; acum putem folosi registrul, intrucat valoarea lui este salvata pe stiva

PRINTF32 `%d \n\x0`, eax ; 10

pop eax ; recupereaza valoarea registrului eax

PRINTF32 `%d \n\x0`, eax ; 15

2. Adresand memoria, cu ajutorului registrului in care este tinut pointerul catre capul stivei (“stack pointer”) esp:

%include "io.asm"

section .text

global CMAIN

CMAIN:

mov eax, 7

mov ebx, 8

add eax, ebx

sub esp, 4 ; rezerva 4 octeti pe stiva

mov [esp], eax ; muta la noua adresa catre care pointeaza esp continutul registrului eax

mov eax, 10

PRINTF32 `%d \n\x0`, eax

mov eax, [esp] ; recupereaza valoarea de pe stiva

add esp, 4 ; restabileste valoarea registrului esp

PRINTF32 `%d \n\x0`, eax

0. Media aritmetică a elementelor dintr-un vector

Afișați partea întreagă a mediei (câtul împărțirii) cu primele 5 zecimale exacte din partea fracționară. (pentru a calcula prima cifră din partea fracționară trebuie să înmulțiti restul obținut cu 10, urmat de o împarțire cu același împărțitor).

%include "../utils/printf32.asm"

%define ARRAY\_SIZE 13

%define DECIMAL\_PLACES 5

section .data

num\_array dw 76, 12, 65, 19, 781, 671, 431, 761, 782, 12, 91, 25, 9

section .text

extern printf

global main

main:

xor eax, eax ; Clear the accumulator (eax) to store the sum

mov ecx, ARRAY\_SIZE ; Use ecx as loop counter

add\_element:

mov bx, word [num\_array + (ecx - 1) \* 2] ; Load the current element into bx

add ax, bx ; Add the current element to the sum

loop add\_element ; Decrement ecx and continue the loop if it's not zero

PRINTF32 `Sum of numbers: %d\n\x0`, eax ; Print the sum of the numbers

; Compute the mean quotient

xor edx, edx ; Clear the remainder (edx) to store the decimal part

mov bx, ARRAY\_SIZE ; Move the value of ARRAY\_SIZE into bx

div bx ; Divide the sum (eax) by ARRAY\_SIZE

PRINTF32 `Mean of numbers: %d\x0`, eax ; Print the integer part of the mean

mov ecx, DECIMAL\_PLACES ; Use ecx as loop counter for computing decimal places

compute\_decimal\_place:

mov ax, dx ; Move the remainder (decimal part) into ax

mov bx, 10 ; Set bx to 10 for multiplication

mul bx ; Multiply the remainder by 10

mov bx, ARRAY\_SIZE ; Move the value of ARRAY\_SIZE into bx

div bx ; Divide the result by ARRAY\_SIZE to get the next decimal digit

PRINTF32 `%d\x0`, eax ; Print the current decimal digit

dec ecx ; Decrement ecx (loop counter)

cmp ecx, 0 ; Compare ecx with 0 to check if there are more decimal places to compute

jg compute\_decimal\_place ; If ecx is greater than 0, continue computing decimal places

PRINTF32 `\n\x0` ; Print a new line

xor eax, eax ; Clear eax before returning

ret ; Return from the function

1. Max

Calculați maximul dintre numerele din 2 registre (eax și ebx) folosind o instrucțiune de comparație, o instrucțiune de salt și instrucțiuni push/pop.

%include "../utils/printf32.asm"

section .text

extern printf

global main

main:

; Numbers are placed in these two registers.

mov eax, 1

mov ebx, 4

cmp eax, ebx

ja print\_max

push eax

push ebx

pop eax

pop ebx

print\_max:

PRINTF32 `Max value is: %d\n\x0`, eax

ret

2. Construirea array-ului inversat

Implementați TODO-urile fără a folosi instrucțiunea mov în lucrul cu array-urile astfel încât în array-ul output la finalul programului să se afle array-ul input inversat.

Dupa o rezolvare corecta programul ar trebui sa printeze:

Reversed array:

911

845

263

242

199

184

122

%include "../utils/printf32.asm"

%define ARRAY\_LEN 7

section .data

input dd 122, 184, 199, 242, 263, 845, 911

output times ARRAY\_LEN dd 0

section .text

extern printf

global main

main:

push ARRAY\_LEN ; Se pune valoarea ARRAY\_LEN pe stivă

pop ecx ; Se extrage valoarea ARRAY\_LEN din stivă în registrul ecx

push\_elem:

push dword [input + 4 \* (ecx - 1)] ; Se pune elementul curent din input pe stivă

loop push\_elem ; Se decrementează ecx și se continuă până când ecx devine 0

push ARRAY\_LEN ; Se pune valoarea ARRAY\_LEN pe stivă

pop ecx ; Se extrage valoarea ARRAY\_LEN din stivă în registrul ecx

pop\_elem:

pop dword [output + 4 \* (ecx - 1)] ; Se extrage elementul de pe stivă și se pune în array-ul output

loop pop\_elem ; Se decrementează ecx și se continuă până când ecx devine 0

PRINTF32 `Reversed array: \n\x0` ; Se afișează mesajul "Reversed array:"

xor ecx, ecx ; Se resetează ecx la zero pentru a itera prin array-ul output

print\_array:

mov edx, [output + 4 \* ecx] ; Se încarcă elementul curent din array-ul output în edx

PRINTF32 `%d\n\x0`, edx ; Se afișează elementul curent

inc ecx ; Se incrementează ecx

cmp ecx, ARRAY\_LEN ; Se compară ecx cu ARRAY\_LEN pentru a verifica dacă toate elementele au fost afișate

jb print\_array ; Dacă ecx este mai mic decât ARRAY\_LEN, se continuă afișarea elementelor

xor eax, eax ; Se resetează eax la zero înainte de a reveni din funcție

ret ; Se revine din funcție

3. Adresarea si printarea stivei

Programul stack-addressing.asm din arhiva laboratorului alocă și inițializează două variabile locale pe stivă:

\* un array format din numerele naturale de la 1 la NUM

\* un string “Ana are mere”.

-> Înlocuiți fiecare instrucțiune push cu o secvență de instrucțiuni echivalentă.

-> Printați adresele și valorile de pe stivă din intervalul [ESP, EBP] (de la adrese mici la adrese mari) octet cu octet.

-> Printați string-ul alocat pe stivă octet cu octet și explicați cum arată acesta în memorie. Gândiți-vă de la ce adresă ar trebui să afișați și când ar trebui să vă opriți.

-> Printați vectorul alocat pe stivă element cu element. Gândiți-vă de la ce adresă ar trebui să începeți afișarea și ce dimensiune are un element.

După o implementare cu succes, programul ar trebui să afișeze ceva asemănător cu următorul output (nu fix același lucru, adresele de pe stivă pot să difere):

0xffcf071b: 65

0xffcf071c: 110

0xffcf071d: 97

0xffcf071e: 32

0xffcf071f: 97

...

0xffcf0734: 4

0xffcf0735: 0

0xffcf0736: 0

0xffcf0737: 0

0xffcf0738: 5

0xffcf0739: 0

0xffcf073a: 0

0xffcf073b: 0

Ana are mere

1 2 3 4 5

%include "../utils/printf32.asm"

%define NUM 5

section .text

extern printf

global main

main:

mov ebp, esp ; Se salvează valoarea registrului ESP în EBP pentru a menține o referință la începutul stivei

mov ecx, NUM ; Se încarcă numărul de elemente NUM în registrul ECX

push\_nums:

sub esp, 4 ; Se alocă 4 octeți pe stivă

mov dword [esp], ecx ; Se pune valoarea curentă din ECX pe stivă

loop push\_nums ; Se decrementează ECX și se continuă până când ECX devine 0

sub esp, 4 ; Se alocă 4 octeți pe stivă

mov dword [esp], 0 ; Se pune valoarea 0 pe stivă

sub esp, 4 ; Se alocă 4 octeți pe stivă

mov dword [esp], "mere" ; Se pune adresa string-ului "mere" pe stivă

sub esp, 4 ; Se alocă 4 octeți pe stivă

mov dword [esp], "are " ; Se pune adresa string-ului "are " pe stivă

sub esp, 4 ; Se alocă 4 octeți pe stivă

mov dword [esp], "Ana " ; Se pune adresa string-ului "Ana " pe stivă

lea esi, [esp] ; Se încarcă în registrul ESI adresa de început a string-ului alocat pe stivă

PRINTF32 `%s\n\x0`, esi ; Se afișează string-ul folosind formatul specificat

; Printarea stivei în formatul "address: value".

mov eax, ebp ; Se încarcă valoarea registrului EBP în EAX pentru a itera prin adresele de pe stivă

print\_stack:

PRINTF32 `0x%x: 0x%x\n\x0`, eax, [eax] ; Se afișează adresa și valoarea de pe stivă folosind formatul specificat

sub eax, 4 ; Se scade 4 din EAX pentru a trece la adresa următoare de pe stivă

cmp eax, esp ; Se compară EAX cu ESP pentru a verifica dacă s-a ajuns la capătul stivei

jge print\_stack ; Dacă EAX este mai mare sau egal cu ESP, se continuă afișarea

; Printarea string-ului.

lea esi, [esp] ; Se încarcă în registrul ESI adresa de început a string-ului alocat pe stivă

PRINTF32 `%s\n\x0`, esi ; Se afișează string-ul folosind formatul specificat

; Restaurarea valorii anterioare a registrului EBP (Base Pointer).

mov esp, ebp ; Se restabilește ESP la valoarea anterioară stocată în EBP

; Ieșirea din program fără erori.

xor eax, eax

ret

4. Variabile locale

Programul merge-arrays.asm din cadrul arhivei de laborator, îmbină două array-uri sortate crescător (array\_1 și array\_2) punând array-ul rezultat în array\_output definit în secțiunea .data.

Modificați programul astfel încat array\_output să fie alocat pe stivă. Alocarea array-ului se face cu instructiunea sub.

%include "../utils/printf32.asm"

section .data

%define ARRAY\_1\_LEN 5

%define ARRAY\_2\_LEN 7

%define ARRAY\_OUTPUT\_LEN 12

section .data

array\_1 dd 27, 46, 55, 83, 84

array\_2 dd 1, 4, 21, 26, 59, 92, 105

section .text

extern printf

global main

main:

mov ebp, esp ; Salvăm valoarea curentă a EBP (Base Pointer) pentru a o restaura mai târziu

sub esp, 4 \* ARRAY\_1\_LEN ; Alocăm spațiu pe stivă pentru array\_1

mov eax, esp ; EAX va fi utilizat pentru a parcurge array\_1 pe stivă

mov edx, 0 ; Inițializăm contorul pentru array\_1

array\_1\_on\_stack:

mov ecx, [array\_1 + 4 \* edx] ; Încărcăm un element din array\_1

mov [eax], ecx ; Stocăm elementul în array\_1 pe stivă

inc edx ; Incrementăm contorul

add eax, 4 ; Avansăm în sus pe stivă cu 4 octeți (dimensiunea unui element din array)

cmp edx, ARRAY\_1\_LEN ; Verificăm dacă am terminat de parcurs array\_1

jl array\_1\_on\_stack

mov eax, esp ; Pregătim EAX pentru a parcurge array\_1 pe stivă

sub esp, 4 \* ARRAY\_2\_LEN ; Alocăm spațiu pe stivă pentru array\_2

mov ebx, esp ; EBX va fi utilizat pentru a parcurge array\_2 pe stivă

mov edx, 0 ; Inițializăm contorul pentru array\_2

array\_2\_on\_stack:

mov ecx, [array\_2 + 4 \* edx] ; Încărcăm un element din array\_2

mov [ebx], ecx ; Stocăm elementul în array\_2 pe stivă

inc edx ; Incrementăm contorul

add ebx, 4 ; Avansăm în sus pe stivă cu 4 octeți (dimensiunea unui element din array)

cmp edx, ARRAY\_2\_LEN ; Verificăm dacă am terminat de parcurs array\_2

jl array\_2\_on\_stack

mov ebx, esp ; Pregătim EBX pentru a parcurge array\_2 pe stivă

sub esp, 4 \* ARRAY\_OUTPUT\_LEN ; Alocăm spațiu pe stivă pentru array\_output

mov ecx, esp ; ECX va fi utilizat pentru a parcurge array\_output pe stivă

merge\_arrays:

mov edx, [eax] ; Încărcăm un element din array\_1 pe stivă înregistrul EDX

cmp edx, [ebx] ; Comparam elementul din array\_1 cu elementul din array\_2

jg array\_2\_lower ; Dacă elementul din array\_2 este mai mic, saltăm la eticheta array\_2\_lower

array\_1\_lower:

mov [ecx], edx ; Elementul din array\_1 este mai mic, îl stocăm în array\_output pe stivă

add eax, 4 ; Avansăm în sus pe stivă înregistrul EAX (array\_1)

add ecx, 4 ; Avansăm în sus pe stivă înregistrul ECX (array\_output)

jmp verify\_array\_end ; Saltăm la verificarea finală a array-urilor

array\_2\_lower:

mov edx, [ebx] ; Elementul din array\_2 este mai mic, îl încărcăm înregistrul EDX

mov [ecx], edx ; Stocăm elementul din array\_2 în array\_output pe stivă

add ebx, 4 ; Avansăm în sus pe stivă înregistrul EBX (array\_2)

add ecx, 4 ; Avansăm în sus pe stivă înregistrul ECX (array\_output)

verify\_array\_end:

mov edx, ebp ; Încărcăm valoarea anterioară a EBP (Base Pointer)

cmp eax, edx ; Verificăm dacă am parcurs toate elementele din array\_1

jge copy\_array\_2 ; Dacă da, saltăm la copierea elementelor rămase din array\_2 în array\_output

sub edx, 4 \* ARRAY\_1\_LEN ; Calculăm adresa la care se termină array\_1 pe stivă

cmp ebx, edx ; Verificăm dacă am parcurs toate elementele din array\_2

jge copy\_array\_1 ; Dacă da, saltăm la copierea elementelor rămase din array\_1 în array\_output

jmp merge\_arrays ; Revenim la eticheta merge\_arrays pentru a continua îmbinarea array-urilor

copy\_array\_1:

xor edx, edx ; Resetăm înregistrul EDX la 0

mov eax, [eax] ; Încărcăm un element din array\_1 pe stivă înregistrul EAX

mov [ecx], edx ; Stocăm 0 în array\_output pe stivă (elementele rămase din array\_1)

add ecx, 4 ; Avansăm în sus pe stivă înregistrul ECX (array\_output)

add eax, 4 ; Avansăm în sus pe stivă înregistrul EAX (array\_1)

cmp eax, ebp ; Verificăm dacă am parcurs toate elementele din array\_1

jb copy\_array\_1 ; Dacă nu, continuăm copierea următorului element în array\_output

jmp print\_array ; Dacă da, trecem la afișarea array\_output

copy\_array\_2:

xor edx, edx ; Resetăm înregistrul EDX la 0

mov edx, [ebx] ; Încărcăm un element din array\_2 înregistrul EDX

mov [ecx], edx ; Stocăm elementul din array\_2 în array\_output pe stivă

add ecx, 4 ; Avansăm în sus pe stivă înregistrul ECX (array\_output)

add ebx, 4 ; Avansăm în sus pe stivă înregistrul EBX (array\_2)

mov edx, ebp ; Calculăm adresa la care se termină array\_1 pe stivă

sub edx, 4 \* ARRAY\_1\_LEN ; Calculăm adresa la care începe array\_2 pe stivă

cmp ebx, edx ; Verificăm dacă am parcurs toate elementele din array\_2

jb copy\_array\_2 ; Dacă nu, continuăm copierea următorului element în array\_output

print\_array:

PRINTF32 `Array merged:\n\x0`

xor eax, eax

xor ecx, ecx

print:

mov eax, [esp] ; Încărcăm un element din array\_output de pe stivă înregistrul EAX

PRINTF32 `%d \x0`, eax ; Afișăm elementul

add esp, 4 ; Avansăm în jos pe stivă cu 4 octeți (dimensiunea unui element din array\_output)

inc ecx ; Incrementăm contorul

cmp ecx, ARRAY\_OUTPUT\_LEN ; Verificăm dacă am afișat toate elementele din array\_output

jb print ; Dacă nu, continuăm afișarea următorului element

PRINTF32 `\n\x0` ; Trecem la o nouă linie

xor eax, eax ; Resetăm înregistrul EAX la 0

mov esp, ebp ; Restaurăm valoarea anterioară a EBP (Base Pointer)

ret ; Încheiem programul cu succes

5. GCD - Greatest Common Divisor

Codul calculează cel mai mare divizor comun dintre două numere date ca parametru prin registrele eax și edx, și pune valoarea calculată tot în registrul eax.

În cadrul label-ului print afișați rezultatul sub forma:

gcd(49,28)=7

%include "../utils/printf32.asm"

section .text

extern printf

global main

main:

; Input values (eax, edx) : cele 2 numere pentru care se calculează cmmdc-ul.

mov eax, 49 ; Se încarcă primul număr (49) în registrul eax

mov edx, 28 ; Se încarcă al doilea număr (28) în registrul edx

push eax ; Se pune primul număr pe stivă

push edx ; Se pune al doilea număr pe stivă

gcd:

neg eax ; Se negă valoarea primului număr

je L3 ; Dacă numărul este 0, se sare la eticheta L3

L1:

neg eax ; Se negă valoarea primului număr

push eax ; Se pune primul număr negat pe stivă

push edx ; Se pune al doilea număr pe stivă

pop eax ; Se scoate primul număr negat de pe stivă în registrul eax

pop edx ; Se scoate al doilea număr de pe stivă în registrul edx

L2:

sub eax,edx ; Se scade al doilea număr din primul număr

jg L2 ; Dacă primul număr este mai mare decât al doilea, se sare la eticheta L2

jne L1 ; Dacă cele două numere sunt diferite, se sare la eticheta L1

L3:

add eax,edx ; Se adună cele două numere pentru a obține cmmdc-ul

jne print ; Dacă numărul nu este 0, se sare la eticheta print

inc eax ; Altfel, se incrementează numărul pentru a obține valoarea absolută

print:

pop edx ; Se scoate al doilea număr de pe stivă în registrul edx

pop ebx ; Se scoate primul număr de pe stivă în registrul ebx

PRINTF32 `gcd(%d, %d) = %d\n\x0`, ebx, edx, eax ; Se afișează rezultatul cmmdc-ului

xor eax, eax ; Se resetează registrul eax la 0

ret ; Se încheie programul cu succes

Laborator 9

0. Șirul lui Fibonacci

%include "../utils/printf32.asm"

%define NUM\_FIBO 10

section .text

extern printf

global main

main:

mov ebp, esp ; Salvăm adresa bazei stivei în registrul ebp

push 0 ; Punem primul termen al lui Fibonacci (0) pe stivă

push 1 ; Punem al doilea termen al lui Fibonacci (1) pe stivă

mov ecx, NUM\_FIBO - 2 ; Numărul de termeni rămași de calculat în seria lui Fibonacci

calc\_fibo:

mov eax, [esp] ; Încărcăm ultimii doi termeni ai seriei în registrul eax

add eax, [esp + 4] ; Adunăm cei doi termeni pentru a obține următorul termen

push eax ; Punem următorul termen pe stivă

loop calc\_fibo ; Se repetă până când toți termenii sunt calculați

mov ecx, NUM\_FIBO ; Numărul total de termeni în seria lui Fibonacci

print:

mov eax, dword [esp + (ecx - 1) \* 4] ; Se încarcă termenul curent în registrul eax

PRINTF32 `%d \x0`, eax ; Se afișează termenul

dec ecx ; Se decrementează contorul termenilor

cmp ecx, 0 ; Se compară contorul cu 0

ja print ; Dacă contorul este mai mare decât 0, se sare la eticheta print

PRINTF32 `\n\x0` ; Se afișează o linie nouă

mov esp, ebp ; Se restabilește adresa bazei stivei

xor eax, eax ; Se resetează registrul eax la 0

ret ; Se încheie programul cu succes

1. Hello, world!

Remarcați că:

-> Programul hello-world.asm folosește apelul funcției puts (funcție externă modulului curent) pentru a efectua afișarea. Pentru aceasta pune argumentul pe stivă și apelează funcția.

-> Variabila msg din programul hello-world.asm conține octetul 10. Acesta simbolizează caracterul line-feed (\n), folosit pentru a adăuga o linie nouă pe Linux.

-> Încheierea cu \n este, în general, utilă pentru afișarea șirurilor. Funcția puts pune automat o linie nouă după șirul afișat, însă aceasta trebuie adăugată explicit în cazul folosirii funcției printf.

%include "../utils/printf32.asm"

section .data

msg db 'Hello, world!', 0

section .text

extern puts

global main

main:

push ebp ; Salvăm adresa bazei stivei pe stivă pentru a crea un cadru de stivă

mov ebp, esp ; Actualizăm adresa bazei stivei cu adresa curentă a stivei

push msg ; Punem adresa șirului pe stivă ca argument pentru funcția puts

call puts ; Apelăm funcția puts pentru a afișa șirul de caractere

add esp, 4 ; Eliberăm argumentul de pe stivă prin ajustarea adresei stivei cu 4 octeți

leave

ret

2. Afișarea unui șir

Pentru afișarea unui string putem folosi macro-ul intern PRINTF32. Sau putem folosi o funcție precum puts. În fișierul print-string.asm este implementată afișarea unui string folosind macro-ul PRINTF32.

Urmărind fișierul hello-world.asm ca exemplu, implementați afișarea șirului folosind și puts.

%include "../utils/printf32.asm"

section .data

mystring db "This is my string", 0

section .text

extern puts

extern printf

global main

main:

push ebp ; Salvăm adresa bazei stivei pe stivă pentru a crea un cadru de stivă

mov ebp, esp ; Actualizăm adresa bazei stivei cu adresa curentă a stivei

PRINTF32 `[PRINTF32]: %s\n`, mystring ; Afișăm șirul utilizând macro-ul PRINTF32

push mystring ; Punem adresa șirului pe stivă ca argument pentru funcția puts

call puts ; Apelăm funcția puts pentru a afișa șirul de caractere

add esp, 4 ; Eliberăm argumentul de pe stivă prin ajustarea adresei stivei cu 4 octeți

leave ; Restaurăm baza stivei și pointerul instrucțiunilor de revenire

ret ; Încheiem programul cu succes

3. Afișarea lungimii unui șir

Programul print-string-len.asm afișează lungimea unui șir folosind macro-ul PRINTF32. Calculul lungimii șirului mystring are loc în cadrul programului (este deja implementat).

Implementați programul pentru a face afișarea lungimii șirului folosind funcția printf.

La sfârșit veți avea afișată de două ori lungimea șirului: inițial cu apelul macro-ului PRINTF32 și apoi cu apelul funcției externe printf.

Gândiți-vă că apelul printf este de forma printf("String length is %u\n", len);. Trebuie să construiți stiva pentru acest apel.

Pașii de urmat sunt:

-> Marcarea simbolului printf ca simbol extern.

-> Definirea șirului de formatare "String length is %u", 10, 0.

-> Realizarea apelului funcției printf, adică:

-> Punerea celor două argumente pe stivă: șirul de formatarea și lungimea.

-> Apelul printf folosind call.

-> Restaurarea stivei.

Lungimea șirului se găsește în registrul ecx.

%include "../../utils/printf32.asm"

section .data

mystring db "This is my string", 0

print\_format db "String length is %d", 10, 0

section .text

extern printf

global main

main:

push ebp ; Salvăm adresa bazei stivei pe stivă pentru a crea un cadru de stivă

mov ebp, esp ; Actualizăm adresa bazei stivei cu adresa curentă a stivei

mov eax, mystring ; Pregătim registrul eax cu adresa șirului mystring

xor ecx, ecx ; Inițializăm registrul ecx (lungimea șirului) cu 0

calculate\_length:

mov bl, byte [eax] ; Încărcăm următorul byte din șir în registrul bl

test bl, bl ; Verificăm dacă byte-ul este zero (terminatorul NULL)

jz print\_result ; Dacă da, ne oprim și trecem la afișare

inc eax ; Trecem la următorul byte din șir

inc ecx ; Incrementăm lungimea șirului

jmp calculate\_length ; Continuăm să calculăm lungimea șirului

print\_result:

PRINTF32 `[PRINTF32]: %d\n`, ecx ; Afișăm lungimea șirului utilizând macro-ul PRINTF32

push ecx ; Pregătim lungimea șirului pe stivă ca argument pentru funcția printf

push print\_format ; Pregătim adresa formatului pentru funcția printf

call printf ; Apelăm funcția printf pentru a afișa lungimea șirului

add esp, 8 ; Eliberăm argumentele de pe stivă prin ajustarea adresei stivei cu 8 octeți

leave

ret

4. Afișarea șirului inversat

Funcția reverse\_string inversează un șir și are următoarea signatură: void reverse\_string(const char \*src, size\_t len, char \*dst);. Astfel ca primele len caractere și șirul src sunt inversate în șirul dst.

Realizați inversarea șirului mystring într-un nou șir și afișați acel nou șir.

Pentru a defini un nou șir, recomandăm ca, în secțiunea de date să folosiți construcția

store\_string times 64 db 0

Construcția creează un șir de 64 de octeți de zero, suficient pentru a stoca inversul șirului. Apelul echivalent în C al funcției este reverse\_string(mystring, ecx, store\_string);. În registrul ecx am presupus că este calculată lungimea șirului.

Nu puteți folosi direct valoarea ecx în forma ei curentă. După apelul funcției printf pentru afișare numărului valoarea ecx se pierde. Ca să o păstrați, aveți două opțiuni:

-> Stocați valoarea registrului ecx în prealabil pe stivă (folosind push ecx înaintea apelului printf) și apoi să o restaurați după apelul printf (folosind pop ecx).

-> Stocați valoarea registrului ecx într-o variabilă globală, pe care o definiți în secțiunea .data.

Nu puteți folosi un alt registru pentru că sunt șanse foarte mari ca și acel registru să fie modificat de apelul printf pentru afișarea lungimii șirului.

section .data

mystring db "This is my string", 0

print\_format db "String length is %d", 10, 0

store\_string times 64 db 0

section .text

extern printf

extern puts

global main

reverse\_string:

push ebp

mov ebp, esp

mov eax, dword [ebp+8] ; Adresa sursă (șirul de inversat)

mov ecx, dword [ebp+12] ; Lungimea șirului

add eax, ecx ; Pregătim adresa de sfârșit a șirului

dec eax ; Trecem la ultimul byte din șir

mov edx, dword [ebp+16] ; Adresa destinație (șirul inversat)

copy\_one\_byte:

mov bl, byte [eax] ; Încărcăm un byte din șirul sursă în registrul bl

mov byte [edx], bl ; Copiem byte-ul în șirul destinație

dec eax ; Trecem la byte-ul anterior din șirul sursă

inc edx ; Trecem la byte-ul următor din șirul destinație

loopnz copy\_one\_byte ; Continuăm copierea până la sfârșitul șirului sursă sau până când ecx devine zero

inc edx ; Adăugăm terminatorul NULL la sfârșitul șirului destinație

mov byte [edx], 0

leave ; Restaurăm baza stivei și pointerul instrucțiunilor de revenire

ret ; Încheiem funcția

main:

push ebp ; Salvăm adresa bazei stivei pe stivă pentru a crea un cadru de stivă

mov ebp, esp ; Actualizăm adresa bazei stivei cu adresa curentă a stivei

mov eax, mystring ; Pregătim registrul eax cu adresa șirului mystring

xor ecx, ecx ; Inițializăm registrul ecx (lungimea șirului) cu 0

calculate\_length:

mov bl, byte [eax] ; Încărcăm următorul byte din șir în registrul bl

test bl, bl ; Verificăm dacă byte-ul este zero (terminatorul NULL)

jz print\_result ; Dacă da, ne oprim și trecem la afișare

inc eax ; Trecem la următorul byte din șir

inc ecx ; Incrementăm lungimea șirului

jmp calculate\_length ; Continuăm să calculăm lungimea șirului

print\_result:

; Salvăm lungimea șirului (ecx) pe stivă pentru a o păstra după apelul funcției printf

push ecx

push ecx ; Pregătim lungimea șirului pe stivă ca argument pentru funcția printf

push print\_format ; Pregătim adresa formatului pentru funcția printf

call printf ; Apelăm funcția printf pentru a afișa lungimea șirului

add esp, 8 ; Eliberăm argumentele de pe stivă prin ajustarea adresei stivei cu 8 octeți

; Restaurăm lungimea șirului (ecx) de pe stivă

pop ecx

push store\_string ; Pregătim adresa șirului inversat pentru apelul funcției reverse\_string

push ecx ; Pregătim lungimea șirului ca argument pentru funcția reverse\_string

push mystring ; Pregătim adresa șirului original pentru apelul funcției reverse\_string

call reverse\_string ; Apelăm funcția reverse\_string pentru a inversa șirul

add esp, 12 ; Eliberăm argumentele de pe stivă prin ajustarea adresei stivei cu 12 octeți

push store\_string ; Pregătim adresa șirului inversat pentru apelul funcției puts

call puts ; Apelăm funcția puts pentru a afișa șirul inversat

add esp, 4 ; Eliberăm argumentul de pe stivă prin ajustarea adresei stivei cu 4 octeți

leave ; Restaurăm baza stivei și pointerul instrucțiunilor de revenire

ret ; Încheiem programul cu succes

5. Implementarea funcției toupper

Ne propunem implementarea funcției toupper care traduce literele mici în litere mari. Pentru aceasta, porniți de la fișierul toupper.asm din arhiva de exerciții a laboratorului și completați corpul funcției toupper.

Șirul folosit este mystring și presupunem că este un șir valid. Acest șir este transmis ca argument funcției toupper în momentul apelului.

Faceți înlocuirea in place, nu este nevoie de un alt șir.

Ca să traduceți o litera mică în literă mare, trebuie să scădeți 0x20 din valoare. Aceasta este diferența între litere mici și mari; de exemplu a este 0x61 iar A este 0x41. Puteți vedea în pagina de manual ascii.

a să citiți sau să scrieți octet cu octet folosiți construcția byte [reg] așa cum apare și în implementarea determinării lungimii unui șir în fișierul print-string-len.asm, unde [reg] este registrul de tip pointer în care este stocată adresa șirului în acel punct.

Vă opriți atunci când ați ajuns la valoarea 0 (NULL byte). Pentru verificare puteți folosi test așa cum se întâmplă și în implementarea determinării lungimii unui șir în fișierul print-string-len.asm.

section .data

before\_format db "before %s", 13, 10, 0

after\_format db "after %s", 13, 10, 0

mystring db "abcdefghij", 0

section .text

extern printf

global main

toupper:

push ebp

mov ebp, esp

mov eax, dword [ebp+8] ; Adresa șirului mystring

check\_one\_byte:

mov bl, byte [eax] ; Încărcăm următorul byte din șir în registrul bl

test bl, bl ; Verificăm dacă byte-ul este zero (terminatorul NULL)

je out ; Dacă da, ne oprim și trecem la încheierea funcției

sub bl, 0x20 ; Scădem 0x20 din valoarea registrului bl pentru a transforma litera mică în literă mare

mov byte [eax], bl ; Salvăm byte-ul modificat înapoi în șir

inc eax ; Trecem la următorul byte din șir

jmp check\_one\_byte ; Continuăm verificarea și modificarea literelor până la terminatorul NULL

out:

leave ; Restaurăm baza stivei și pointerul instrucțiunilor de revenire

ret ; Încheiem funcția

main:

push ebp ; Salvăm adresa bazei stivei pe stivă pentru a crea un cadru de stivă

mov ebp, esp ; Actualizăm adresa bazei stivei cu adresa curentă a stivei

push mystring

push before\_format

call printf ; Afișăm șirul original

add esp, 8 ; Eliberăm argumentele de pe stivă prin ajustarea adresei stivei cu 8 octeți

push mystring

call toupper ; Apelăm funcția toupper pentru a converti literele mici în litere mari

add esp, 4 ; Eliberăm argumentul de pe stivă prin ajustarea adresei stivei cu 4 octeți

push mystring

push after\_format

call printf ; Afișăm șirul modificat

add esp, 8 ; Eliberăm argumentele de pe stivă prin ajustarea adresei stivei cu 8 octeți

leave ; Restaurăm baza stivei și pointerul instrucțiunilor de revenire

ret ; Încheiem programul cu succes