Laboratorium programowania niskopoziomowego

LAB 6 – Podprogramy i wykorzystanie stosu.

Podprogramy O P I S

Podprogram w asemblerze to wydzielony fragment programu, który wykonuje określone zadania i może być wywoływany wielokrotnie podczas działania programu. Podprogramy poprawiają czytelność kodu i skracają go. Korzystają z parametrów dostarczonych w rejestrach lub na stosie. Sposób wywołania podprogramów, użycie parametrów, zmiennych lokalnych, rejestrów i sposób zwracania wyniku definiuje Interfejs binarny aplikacji, ABI (ang. application binary interface). Poniżej znajduje się kompletny program działający na macierzy i wektorze. Zadanie polega na podniesieniu do kwadratu każdego elementu macierzy i wektora, następnie odpowiednie wartości macierzy i wektora należy dodać i wyciągnąć pierwiastek z zapisem do macierzy w odpowiednie miejsce. Kod C++ obrazuje konieczne operacje. Jednakże pierwiastkowanie w asemblerze należało wykonać w osobnym podprogramie.

```
#include <iostream>
#include <windows.h>

using namespace std;

extern "C" void dlugosc32(INT32**, INT32*, INT32, INT32);

int main()
{
	const int m = 5, n = 7;
	int **mm = new int*[m];
	int *v = new int[n];

	for (int i = 0; i < n; ++i) mm[i] = new int[n];

	for (int i = 0; i < m; ++i)
		for (int j = 0; j < n; ++j)
			mm[i][j] = i + j + 5;

	for (int i = 0; i < n; ++i) v[i] = i + 5;

	cout << "Wektor wejsciowy v:\n";
	for (int i = 0; i < n; ++i) cout << v[i] << " ";

	cout << "\n\nMacierz wejsciowa mm:\n";
	for (int i = 0; i < m; ++i) {
		for (int j = 0; j < n; ++j)
			cout << mm[i][j] << " ";
		cout << endl;
	}

	/* Wersja C++
	for (int i = 0; i < m; ++i)
		for (int j = 0; j < n; ++j)
			mm[i][j] = sqrt(pow(mm[i][j], 2) + pow(v[j], 2));
	//*/

	dlugosc32(mm, v, m, n); // z wykorzystaniem podprogramu pierwiastkujacego

	cout << "\n\nMacierz wyjsciowa mm:\n";
	for (int i = 0; i < m; ++i) {
		for (int j = 0; j < n; ++j)
			cout << mm[i][j] << " ";
		cout << endl;
	}

	system("PAUSE");
    return 0;
}

.code
```

```
pierwiastek32 proc
        push  rcx;
        push  rbx;
        mov   ebx, eax;                 *number* to find sqrt of
        mov   ecx, 32768;               32BIT: bitmask(starting with b15 bit set) => 0x8000 = decimal 32768
        xor   rax, rax;                 result <-0
sqrt_loop:
        xor   rax, rcx;                 set bit in eax
        push  rax;                              store result(will be destroyed by mul)
        mul   eax;                              edx:eax <-eax*eax(ignoring edx next)
        cmp   eax, ebx;                 compare with *number*
        pop   rax;                              restore result
        jbe   keep_bit;                 res ^ 2 <= *number*->bit stays set
        xor   eax, ecx;                 unset bit in eax
        keep_bit:
        shr   ecx, 1;                   next bit
        jnz   sqrt_loop;                loop till all bits are tried
        pop   rbx;
        pop   rcx;
        ret;
pierwiastek32 endp

pierwiastek64 proc
        push  rcx;
        push  rbx;
        mov   rbx, rax;                 *number* to find sqrt of
        mov   rcx, 2147483648;          64bit: bitmask(starting with b31 bit set) => 0x80000000 = decimal 2147483648
        xor   rax, rax;                 result <-0
sqrt_loop:
        xor   rax, rcx;                 set bit in eax
        push  rax;                              store result(will be destroyed by mul)
        mul   rax;                              rdx:rax <-rax*rax(ignoring rdx next)
        cmp   rax, rbx;                 compare with *number*
        pop   rax;                              restore result
        jbe   keep_bit;                 res ^ 2 <= *number*->bit stays set
        xor   rax, rcx;                 unset bit in rax
        keep_bit:
        shr   rcx, 1;                   next bit
        jnz   sqrt_loop;                loop till all bits are tried
        pop   rbx;
        pop   rcx;
        ret;
pierwiastek64 endp

dlugosc32 proc ; //RCX = **tab, RDX = *v, R8 = m, R9 = n;
        push rsi
        push rdi
        push rbx
        mov rdi, rcx
        mov rbx, rdx
p1:     mov rsi, [rdi + 8*r8 - 8]
        mov r10, r9
p2:     movsxd rax, dword ptr[rsi + 4*r10 - 4]
        imul rax, rax
        movsxd rcx, dword ptr[rbx + 4*r10 - 4]
        imul rcx, rcx
        add rax, rcx
        call pierwiastek64;
        mov dword ptr[rsi + 4*r10 - 4], eax
        dec r10
        jnz p2
        dec r8
        jnz p1
        pop rbx
        pop rdi
        pop rsi
        ret
dlugosc32 endp

end
```

Zadanie do wykonania:

1. Sprawdzić czy kod działa prawidłowo dla funkcji pierwiastek32 i pierwiastek64
2. Przejść krokowo i prze analizować kod
3. Usunąć *push rcx* i *push rbx* w pierwiastku i powtórnie przeanalizować kod – czy działa prawidłowo ? Jeżeli nie to co się stało i dlaczego ?
4. Wykonać samemu program główny obsługujący dynamiczną macierz trójwymiarową tab*** zawierającą zmienne typu int(32bitowe), w którym zostanie wywołany podprogram podnoszący każdą wartość tej tablicy do 3 potęgi.
5. Wykonać samemu program główny obsługujący dynamiczną macierz trójwymiarową tab*** zawierającą zmienne typu INT64 (64bitowe), w którym zostanie wywołany podprogram podnoszący każdą wartość tej tablicy do potęgi N.

Obsługa stosu X64 O P I S

Częściowa obsługa stosu została już omówiona przy okazji poprzedniego laboratorium i specyficznego nagłówka (w ramach przypomnienia podajemy go jeszcze raz):

```
sumaMeUpV proc uses rbx rsi rdi, mm:qword, uu:ptr, vv:ptr, m:qword, n:qword
;RCX = **mm, RDX = **uu, R8 = **vv, R9 = m, Stos = n;
```

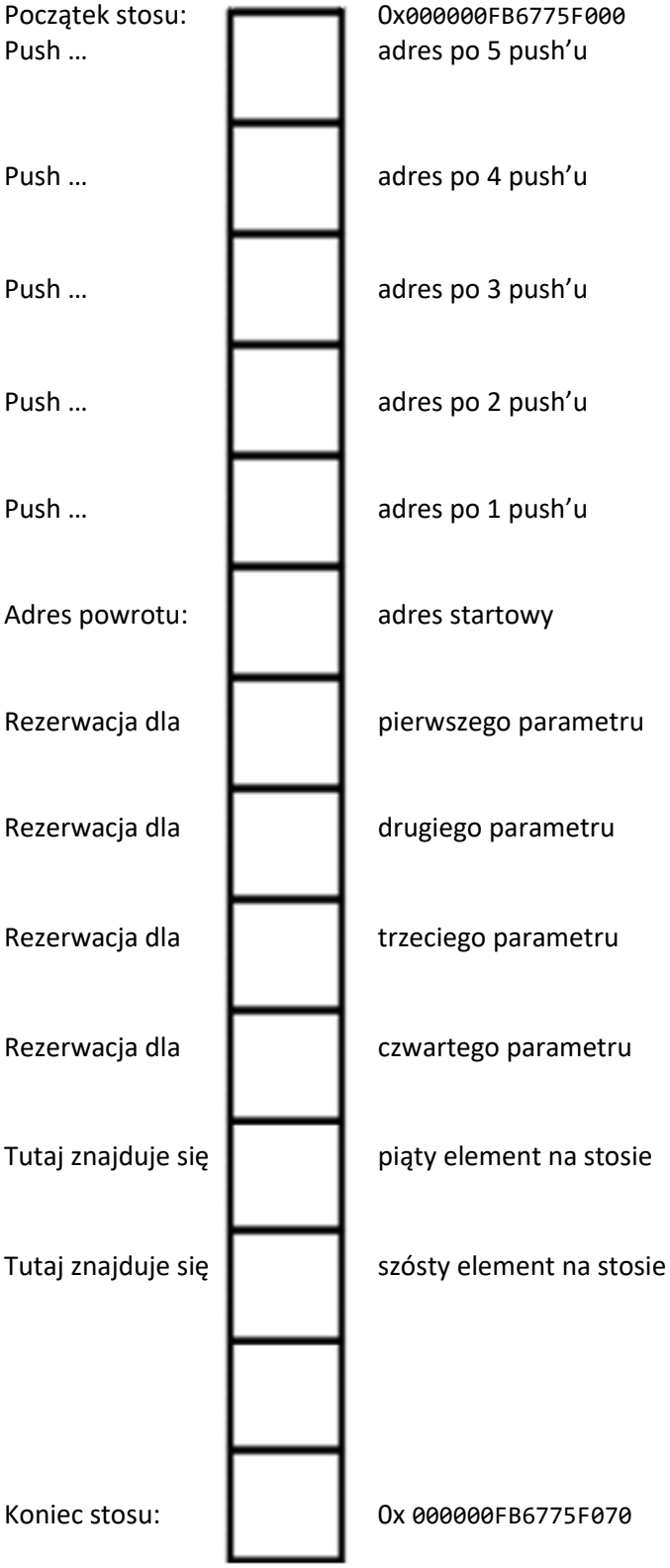
Piąty parametr (w tym przykładzie to ostatni parametr) oraz potencjalnie wszystkie następne parametry są umieszczane na stosie. Dla pierwszych czterech parametrów jest również zarezerwowana pamięć na stosie, ale ich wartości na stosie są nie zapisane, są wpisane bezpośrednio do rejestrów (RCX, RDX, R8, R9).

Rysunek po prawej stronie przedstawia uproszczony schemat stosu (stos może być dłuższy). Istotny jest sposób umieszczania elementów na stosie po instrukcji PUSH (są one umieszczane na stosie w kierunku malejącego adresu, czyli każde odesłanie na stos spowoduje, że adres w RSP będzie się zmniejszał o 8 bajtów). Natomiast elementy przekazane jako parametry funkcji są umieszczone na stosie przed wywołaniem funkcji. Aby prawidłowo się do nich odwoływać należy:

1. Odesłać na stos rejestr bazowy: push RBP

2. Do RBP przypisać RSP

3. Poprzez stały adres w RBP można odwoływać się do poszczególnych elementów np: mov rax, [rbp + 6*8]



Zadania do wykonania:

1. Zmodernizuj zadanie z poprzedniego laboratorium – tak aby działało bez aliasów i bez automatycznego zabezpieczania rejestrów:

```
extern "C" void sumaMeUpV(INT64**, INT64**, INT64**, INT64, INT64);
```

Wpisz funkcję sumy w pliku .asm:

```
.code

sumaMeUpV proc uses rbx rsi rdi, mm:qword, uu:ptr, vv:ptr, m:qword, n:qword
;RCX = **mm, RDX = **uu, R8 = **vv, R9 = m, Stos = n;
      mov   r11,n

petlaM:
      mov rdi, [rcx + 8*r9 - 8]
      mov rsi, [rdx + 8*r9 - 8]
      mov rbx, [r8 + 8*r9 - 8]
      mov r10, r11

petlaN:
      mov rax, [rdi + 8*r10 - 8]
      add rax, [rsi + 8*r10 - 8]
      mov [rbx + 8*r10 - 8], rax
      dec r10
      jnz petlaN
      dec r9
      jnz petlaM

      ret
sumaMeUpV endp
```

2. Dodaj ręcznie (push/pop) zabezpieczanie rejestrów *rbx rdi* przed odesłaniem na stos *rbp* sprawdź czy kod działa? Jeżeli nie to dlaczego? Czy da się to poprawić ?
3. Wykonaj sumę 8 zmiennych przekazanych do funkcji zewnętrznej np.: w = a + b + c + d + e + f + g + h;
4. Wykonaj kod w asemblerze odpowiadający podanemu poniżej kodowi C++ (ilość elementów wszystkich wektorów to N). Schemat przekazywanych parametrów ma być zgodny z tym co zostało napisane poniżej:

*for (int i=0;i<N;++i) Out0[i] = (In1[i]+ In2[i]+ In3[i]+ In4[i]+ In5[i]+ In6[i]+ In7[i]) * x / y;*

```
extern "C" void sumaV (INT64 n, INT64 x, INT64 y INT64* Out0, INT64* In1, INT64* In2,
INT64* In3, INT64* In4, INT64* In5, INT64* In6, INT64* In7);
```

Obsługa stosu X86 O P I S

W architekturze X86 przyjmuje się, że wszystkie parametry są na stosie. Natomiast wynik zwracamy w EAX. Poniżej przykład z dezasemblowanego kodu, w którym przekazywano 9 parametrów typu int. Proszę zwrócić uwagę na kolejność instrukcji i parametrów – są przesyłane od ostatniego, po to aby na samej górze stosu był potem pierwszy parametr.

```
00B623ED  mov         eax,dword ptr [h]
00B623F0  push        eax
00B623F1  mov         ecx,dword ptr [g]
00B623F4  push        ecx
00B623F5  mov         edx,dword ptr [f]
00B623F8  push        edx
00B623F9  mov         eax,dword ptr [e]
00B623FC  push        eax
00B623FD  mov         ecx,dword ptr [d]
00B62400  push        ecx
00B62401  mov         edx,dword ptr [c]
00B62404  push        edx
00B62405  mov         eax,dword ptr [b]
00B62408  push        eax
00B62409  mov         ecx,dword ptr [a]
00B6240C  push        ecx
00B6240D  mov         edx,dword ptr [nn]
00B62410  push        edx
00B62411  call        _Example (0B612D5h)
00B62416  add         esp,24h
00B62419  mov         dword ptr [w],eax
```

Jak widać kompilator używa trzech rejestrów do przekazywania parametrów: eax, ecx, edx, a potem znowu zaczyna używać eax, ecx, edx i tak w kółko aż zakończy przekazywanie parametrów. W przypadku, gdy ilość przekazywanych parametrów będzie niepodzielna przez trzy, ostatni przekazany parametr może znajdować się w ecx, albo w eax, zamiast w edx jak ma to miejsce przy 9 parametrach. Oczywiście po każdym przekazaniu parametru wartość z rejestru jest odsyłana na STOS. Związku z tym przyjmuje się, że wszystkie wartości są na stosie i tak należy się do nich odwoływać.

Schemat odwołania się do poszczególnych parametrów na stosie jest prawie identyczny jak w architekturze X64 i wygląda następująco:

1. Odesłać na stos rejestr bazowy: push EBP

2. Do EBP przypisać ESP

3. Poprzez stały adres w EBP można odwoływać się do poszczególnych elementów np: mov eax, [ebp + 2*4]; // odwołanie do pierwszego parametru na stosie

Poniżej znajduje się przykładowy kod C++:

```cpp
#include <iostream>

extern "C" int Example(int, int, int, int, int, int, int, int, int);

int main()
{
	int nn = 2;
	int a = 21; //1
	int b = 18; //2
	int c = 39; //3
	int d = 12; //4
	int e = 57; //5
	int f = 83; //6
	int g = 91; //7
	int h = 17; //8

	int w = Example(nn, a, b, c, d, e, f, g, h); //switch(nn) -> return a..h
	std::cout << "Example: " << w << "\n";

	system("PAUSE");
    return 0;
}
```

Oraz kod asm:

```
.model flat, C
.code

Example proc
	push ebp;
	mov ebp, esp;

	mov eax, [ebp+2*4];
	cmp eax, 1;
	je case1;
	cmp eax, 2;
	je case2;
	cmp eax, 3;
	je case3;
	cmp eax, 4;
	je case4;
	cmp eax, 5;
	je case5;
	cmp eax, 6;
	je case6;
	cmp eax, 7;
	je case7;
	cmp eax, 8;
	je case8;
	mov eax, 0; default
	jmp exit;
	case1:
	mov eax, [ebp+3*4];
	jmp exit;
	case2:
	mov eax, [ebp+4*4];
	jmp exit;
	case3:
	mov eax, [ebp+5*4];
	jmp exit;
```

```
        case4:
        mov eax, [ebp+6*4];
        jmp exit;
        case5:
        mov eax, [ebp+7*4];
        jmp exit;
        case6:
        mov eax, [ebp+8*4];
        jmp exit;
        case7:
        mov eax, [ebp+9*4];
        jmp exit;
        case8:
        mov eax, [ebp+10*4];
        jmp exit;
        exit:

        pop ebp
        ret;
Example endp

End
```

Proszę zwrócić uwagę na linijkę: „`.model flat, C`” jest ona bardzo istotna do prawidłowej kompilacji programu

Zadania do wykonania:

1. Przeanalizuj kod z funkcji *Example*
2. Dodaj do funkcji asemblerowej *Example* linijkę odpowiadającą za automatyczne zabezpieczenie rejestrów EBX, ESI, EDI – opowiedz co się stało i jak teraz działa kod?
3. Przeanalizuj co się stanie jak do tej wersji dasz dodatkowo taki o to nagłówek funkcji:
   `Example proc ak:dword, bk:dword, ck:dword, dk:dword, ek:dword, fk:dword, gk:dword, hk:dword`
   Podpowiedź: zobacz zakładkę dezasemblacji – kompilator sam coś dodatkowo robi.
4. Wykonaj dodawanie macierzy dwuwymiarowych V=V+U o wymiarach MxN, dane typu int (32bit) kod wykonaj w czterech wersjach:
   a. Bez użycia żadnych elementów w nagłówku
   b. Z nagłówkiem z automatycznym zabezpieczaniem rejestrów
   c. Z nagłówkiem z aliasami
   d. Z nagłówkiem z automatycznym zabezpieczaniem rejestrów i aliasami
5. Wykonaj sumę macierzy trójwymiarowej bez użycia automatycznego zabezpieczana rejestrów i bez aliasów.