



Rechnerorganisation

SS 2005

Musterlösung

Übungsblatt Nr. 3



Prof. Dr.-Ing. Robert Tolksdorf
Freie Universität Berlin

Ausgabe am 20.05.2004 — Abgabe spätestens 03.06.2004, 12:00 Uhr

Bitte bei der Abgabe alle Namen/Matr.Nr. der Gruppe, NUMMER DER ÜBUNG/TEILAUFGABE und DATUM auf den Lösungsblättern **nicht vergessen!** Darauf achten, dass die Lösungen beim richtigen Tutor/der richtigen Tutorin abgegeben werden.

Zu spät abgegebene Lösungen werden nicht mehr berücksichtigt!

1. Aufgabe: Gauß-Summe (10 Punkte)

Schreiben Sie drei IA-32-Assembler-Routinen zur Berechnung der Gauß-Summe einer Zahl. Eine Routine, `gs_iter`, soll dabei die Gauß-Summe iterativ, die zweite, `gs_rec`, rekursiv berechnen. Die dritte schließlich, `gs_ohne`, soll ohne Iteration oder Rekursion auskommen. Vergleichen Sie für Ihre Lösungen jeweils den Aufwand. Geben Sie dazu die Verhältnisse der benötigten Assembler-Befehle der drei Lösungen in Abhängigkeit des Eingabeparameters n an ($y=gs(n)$ soll berechnet werden, $gs(n) = 1+2+\dots+n, n>0$).

2 Punkte für Kommentare

2 Punkte

// iterative Variante der Gaußsumme, $summe(n) = n + (n-1) + \dots + 1$

```
int gs_iter(int n)
{
    __asm
    {
        mov ecx, n;           // lade n in Schleifenzähler
        mov eax, 0;          // setze Summe auf 0
        cmp ecx, 0;          // Prüfe Sonderfall: Gaußsumme von 0
        je ende;             // Dann ist 0 das Ergebnis.

label:
        add eax, ecx;        // Schleife. Addiere Laufvariable zur Summe hinzu
        loop label;         // Falls wir auf 0 runtergezählt haben, fertig.

ende :
    }
}
```

2 Punkte

// Gaußsumme rekursiv, $summe(0) = 0, summe(n) = summe(n-1) + n$

```
int gs_rec(int n)
{
    __asm
    {
        mov eax, n;          // lade n nach eax
        cmp eax, 0;          // Prüfe, ob n = 0
        je ende;             // Dann ist der Rückgabewert 0 (Rekursionsanker)
        dec eax;             // eax = n-1
        push eax;            // Als Parameter auf den Stack
        call gs_rec;         // rekursiver Aufruf
        pop ebx;             // wieder vom Stapel nehmen: n-1 nach ebx
        inc ebx;             // ebx = n
        add eax, ebx;        // eax = gauß (n-1) + n

ende :
    }
}
```

2 Punkte

```
// Gaußsumme nach der Gaußschen Summenformel, summe(n) = (n * (n + 1)) / 2
int gs_ohne(int n)
{
    __asm
    {
        mov eax, n;           // Lade n in eax
        inc eax;             // eax = n + 1
        imul eax, n;        // eax = n * (n+1)
        shr eax, 1;         // eax = (n*(n+1)) / 2
    }
}
```

2 Punkte

1. Gauß iterativ:

- Für $n=0$: kein Schleifendurchlauf, also 4 Befehle
- Für $n=1$: ein Schleifendurchlauf, also $4 + 2 = 6$ Befehle
- Für $n>1$: n Schleifendurchläufe mit je 2 Befehlen
- Allgemein für $n>=0$: Anzahl Befehle bei Eingabe $n = 4 + 2*n$

2. Gauß rekursiv:

- Für $n=0$: kein rekursiver Aufruf, also 3 Befehle
- Für $n=1$: ein rekursiver Aufruf, also $9 + 3 = 12$ Befehle
- Für $n>1$: n rekursive Aufrufe mit je 9 Befehlen
- Allgemein für $n>=0$: Anzahl Befehle bei Eingabe $n = 3 + 9*n$

3. Gauß mit Gaußscher Summenformel:

- Für jedes n konstant 4 Befehle.

2. Aufgabe: Lucas-Zahlen (10 Punkte)

Schreiben Sie eine IA-32-Assembler-Routine zur Berechnung der Lucas-Zahlen (nach Edouard Lucas, 1842-1891). Diese sind rekursiv für natürliche Zahlen n wie folgt definiert: $f(0)=2$, $f(1)=1$, $f(n+2)=f(n+1)+f(n)$. Der Aufruf der Routine `lucas(n)` soll entsprechend den Wert der Lucas-Zahl zurückliefern (`lucas(10)` hat z.B. den Wert 123). Programmieren Sie rekursiv unter Verwendung eines Stacks. (Ganz nette Infos dazu:

<http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/lucasNbs.html> bzw. für die nahen Verwandten:

<http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html> bzw.

<http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html>)

```
int lucas(int n)
{
    __asm
    {
        mov eax, n;           // lade n in eax
        cmp eax, 1;          // vergleiche mit 1
        je ende;             // falls n = 1, fertig
        jl null;             // Falls n<1, springe zu null
        dec eax;             // eax = n-1
        push eax;            // Als Parameter auf den Stack
        call lucas;          // rekursiver Aufruf lucas(n-1)
        pop ebx;             // Parameter vom Stack. ebx = n-1
        dec ebx;             // ebx = n-2
        push eax;            // lucas(n-1) auf dem Stack sichern
        push ebx;            // n-2 als Parameter auf den Stack
        call lucas;          // rekursiver Aufruf: lucas(n-2)
        pop ebx;             // n-2 runterpoppen. Wird weggeschmissen.
        pop ebx;             // lucas(n-1) runterpoppen, nach ebx
        add eax, ebx;        // Rückgabewert ist lucas(n) = lucas(n-1) + lucas(n-2)
        jmp ende;
    }
    null:
        add eax, 2;
    ende:
}
}
```

3. Aufgabe: Vektoren (10 Punkte)

Schreiben Sie eine IA-32-Assembler-Routine zur Berechnung der Differenz zweier Vektoren und zur Bestimmung der Orthogonalität zweier Vektoren. Die Differenz zweier Vektoren ist ein Vektor, dessen Komponenten jeweils die Differenz der entsprechenden Komponenten der beiden Eingabevektoren ist. Vektoren sind dann orthogonal, wenn ihr inneres Produkt gleich Null ist. Die entsprechenden Funktionen `vorth(a, b, n)` bzw. `vdiff(a, b, r, n)` sollen als Eingabe Vektoren a, b beliebiger Länge und die Länge n haben. Als Ergebnis gibt `vorth` den Wert 0 zurück, falls die Vektoren orthogonal sind, ansonsten -1. `vdiff` hat zusätzlich den Ergebnisvektor r als Eingabe und schreibt direkt in diesen das Ergebnis der Subtraktion.

Vektoren seien (beispielhaft) wie folgt definiert:

```
const LEN = 5;
int vector [LEN] = {1,2,3,-4,2};
int *vector_ptr = vector;
```

Damit sollen die Funktionen wie folgt aufrufbar sein:

```
int vorth (int *v_a, int *v_b, int len);
void vdiff (int *v_a, int *v_b, int *res, int len);
```

Das folgende Beispielprogramm zum Testen Ihrer Routinen sollte als Ausgabe:

```
vorth: 0
vdiff: 6 -2 6 -2 3
liefern.
```

```
#include "stdafx.h"

const LEN = 5;
int vec_a [LEN] = {1,2,3,-4,2};
int *vec_a_ptr = vec_a;
int vec_b [LEN] = {-5, 4, -3, -2, -1};
int *vec_b_ptr = vec_b;
int ergebnis [LEN] = {0,0,0,0,0};
int *ergebnis_ptr = ergebnis;

int vorth (int *v_a, int *v_b, int len);
void vdiff (int *v_a, int *v_b, int *res, int len);

int vorth (int *v_a, int *v_b, int len) {
// hier Ihre Assembler-Routine
};

void vdiff (int *v_a, int *v_b, int *res, int len) {
// hier Ihre Assembler-Routine
};

int _tmain(int argc, _TCHAR* argv[]) {
    printf(" vorth: %d \n", vorth(vec_a_ptr, vec_b_ptr, LEN));

    vdiff(vec_a_ptr, vec_b_ptr, ergebnis_ptr, LEN);

    printf(" vdiff: ");
    for (int i=0; i<LEN; i++)
        printf(" %d ", ergebnis_ptr[i]);

    while (getchar()==EOF);
    return 0;
}
```

```

// berechnet, ob zwei Vektoren orthogonal zueinander sind, Rückgabewert 0, falls Orthogonalität
//tät vorliegt, -1 andernfalls.
int vorth (int *v_a, int *v_b, int len)
{
    __asm
    {
        mov edx, v_a;           // lade Adresse von Vektor a
        mov ebx, v_b;           // lade Adresse von Vektor b
        mov ecx, len;           // lade Länge der Vektoren
        mov eax, 0;             // initialisiere Skalarprodukt mit 0

        cmp ecx, 0;             // Vergleiche Länge mit 0
        je ende;                // Falls Länge 0, dann sind die Vektoren orthogonal

label:
        mov esi, [edx + 4*ecx - 4] // lade aktuelles Element von Vektor a nach esi
        mov edi, [ebx + 4*ecx - 4] // lade aktuelles Element von Vektor b nach edi
        imul esi, edi;           // berechne das Produkt
        add eax, esi;            // addiere das Produkt zur Summe
        loop label;             // weiterer Durchlauf mit ecx:=ecx-1
        cmp eax, 0;             // Vergleiche Skalarprodukt mit 0
        je ende;                // Falls das Sakalarprodukt == 0, dann Rückgabewert 0
        mov eax, -1;            // Andernfalls sind die Vektoren nicht orthogonal,
                                // und der Rückgabewert ist -1

ende:
    }
}

// berechnet die Differenz zweier Vektoren
void vdiff (int *v_a, int *v_b, int *res, int len)
{
    __asm
    {
        mov edx, v_a;           // lade Adresse von Vektor a
        mov ebx, v_b;           // lade Adresse von Vektor b
        mov ecx, len;           // lade Länge der Vektoren
        mov eax, res;           // lade Adresse vom Ergebnisvektor

        cmp ecx, 0;             // Vergleiche Länge mit 0
        je ende;                // Falls Länge 0, dann braucht nichts subtrahiert werden.

label:
        mov esi, [edx + 4*ecx - 4] // Lade aktuellen Eintrag von Vektor a nach esi
        sub esi, [ebx + 4*ecx - 4] // Addiere aktuellen Eintrag von Vektor b hinzu
        mov [eax + 4*ecx - 4], esi // Schreibe Summe in aktuellen Eintrag des Ergebnis-
vektors
        loop label;             // Schon fertig oder weiterer Durchlauf mit ecx:=exc-1

ende:
    }
}

```