x86 Assembler

Praktische Einführung

Michael Müller, Sebastian Lackner

3. Juni 2013

Dieses Handout fasst die wichtigsten Informationen des x86 Assembler Befehlssatzes zusammen und kann während der praktischen Einführung als Referenz für nützliche Befehle/Konstrukte verwendet werden. Die Aufgaben im Anschluss dienen der Vertiefung des erlernten Wissens.

1 Assembler Syntax

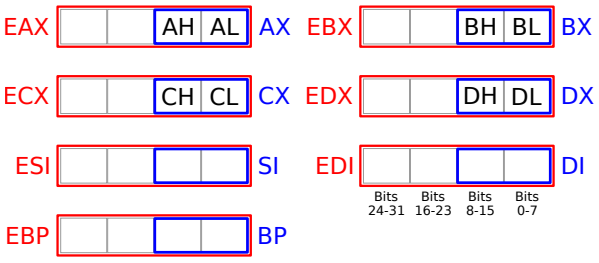
Befehl:	`BEFEHL OPERAND1, OPERAND2`
Kommentar:	`; ein Kommentar`
Label:	`sprungmarke: COMMAND`
Speicher:	`BEFEHL size[ADDRESS]`

Ein Operand kann in Assembler entweder eine *Konstante*, ein *Register* oder eine *Speicheradresse* sein.

Datentypen Als Größenangabe / Datentypen sind die folgenden Angaben möglich:

Assembler	C	Größe
`byte`	`char`	1 Byte / 8 Bit
`word`	`short`	2 Byte / 16 Bit
`dword`	`int`	4 Byte / 32 Bit

Register Die nachfolgende Abbildung zeigt die verfügbaren Register. Bei CDECL müssen bestimmte Register wiederhergestellt werden!



Speicher Speicheradressen können Ausdrücke der Form

$$(\text{Konstante}) + (\text{Register1}) + (f \cdot \text{Register2})$$
$$f \in \{1, 2, 4, 8\}$$

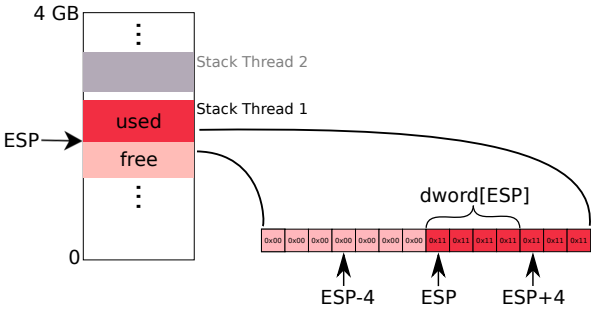
sein, wobei nicht alle Ausdrücke in Klammern angegeben werden müssen.

2 Stack

Der Stapelspeicher / Stack dient für die folgenden Zwecke:

- Zwischenspeicher / lokale Variablen
- Speicher der Rücksprungadresse
- Parameterübergabe bei Funktionen

Die aktuelle Position des Stapelspeichers steht immer in dem Register `ESP`. Zum Reservieren muss `ESP` erniedrigt werden, zum Freigeben muss `ESP` erhöht werden.



3 Tipps um Fehler zu vermeiden

- Es ist nicht möglich zwei Speicheradressen als Operanden anzugeben
- Bei Speicherzugriffen sollte immer die Größe der zu lesenden/schreibenden Daten angegeben werden
- Werte die auf den Stack gePUSHt wurden müssen in der umgekehrten Reihenfolge gePOPt werden
- Es ist hilfreich als Kommentar zu dokumentieren wofür Register verwendet werden

4 Assembler Befehle

4.1 NOP

`NOP`
Tut gar nichts.

4.2 MOV

`MOV DESTINATION, SOURCE`
Kopiert den Inhalt von Source nach Destination. Beide Operanden müssen die selbe Größe haben. Als Destination kann ein Register oder eine Speicheradresse angegeben werden. Source kann zusätzlich eine Konstante sein.

4.3 XCHG

`XCHG OPERAND1, OPERAND2`
Vertauscht den Inhalt der beiden angegebenen Operanden, welche die selbe Größe haben müssen. Es können Register oder Speicheradressen angegeben werden, jedoch nicht zwei Speicheradressen gleichzeitig.

4.4 INC / DEC

`INC COUNTER`
`DEC COUNTER`
Inkrementiert bzw. dekrementiert den Inhalt von Counter. Counter kann ein Register oder eine Speicheradresse sein.

4.5 ADD / SUB

`ADD COUNTER, VALUE`
`SUB COUNTER, VALUE`
Addiert bzw. subtrahiert den Inhalt von Value zu Counter. Counter kann ein Register oder eine Speicheradresse sein. Value kann zusätzlich noch eine Konstante sein.

4.6 PUSH

`PUSH SOURCE`
Speichert Source auf dem Stack. Source kann ein Register, eine Speicheradresse oder eine Konstante sein. Bei Konstanten und Speicheradressen muss die Größe angegeben werden.

4.7 POP

`POP DESTINATION`
Liest einen Wert vom Stack und speichert ihn in Destination. Destination kann ein Register oder eine Speicheradresse sein. Bei Speicheradressen muss die Größe angegeben werden.

4.8 JMP

`JMP SPRUNGMARKE`
Springt zur der angegeben Sprungmarke. Diese kann entweder eine konstante Speicheradresse, der Wert an einer Speicheradresse oder ein Register sein.

4.9 CMP

`CMP OPERAND1, OPERAND2`
Vergleicht beide Operanden und setzt dabei die Prozessorflags. Operand1 und Operand2 können jeweils Konstanten, Speicheradressen oder Register sein. Es können jedoch nicht zwei Konstanten oder zwei Speicheradressen angegeben werden.

4.10 Bedingte Sprungbefehle

`J.. SPRUNGMARKE`
Springt, wenn bestimmte Prozessorflags gesetzt sind (durch `CMP`). Hierbei muss beachtet werden, ob die Zahl als `signed` oder `unsigned` betrachtet werden soll. Folgende Sprünge sind möglich:

Signed	Unsigned	Bedeutung
JE	JE	$\text{Op1} = \text{Op2}$
JNE	JNE	$\text{Op1} \neq \text{Op2}$
JL	JB	$\text{Op1} < \text{Op2}$
JLE	JBE	$\text{Op1} \leq \text{Op2}$
JG	JA	$\text{Op1} > \text{Op2}$
JGE	JAE	$\text{Op1} \geq \text{Op2}$

4.11 CALL

`CALL SPRUNGMARKE`
Speichert die Rücksprungadresse auf dem Stack und springt zu der angegeben Sprungmarke. Auf diese Weise lässt sich ein Funktionsaufruf realisieren.

4.12 RET

`RET`
Liest die Rücksprungadresse vom Stack und springt zu dieser. Dieser Befehl wird am Ende von Funktionen verwendet.

5 Assembler-Konstrukte

5.1 Funktionsrumpf (CDECL)

Eine CDECL-Funktion der folgenden Form

```
int befehl(int a, int b, int c){
    return 1;
}
```

wäre in Assembler folgendermaßen möglich:

```
befehl:
  ; dword[esp]     Ruecksprungadr.
  ; dword[esp+4]   Parameter a
  ; dword[esp+8]   Parameter b
  ; dword[esp+12]  Parameter c
  MOV eax, 1 ; Setze Ergebnis
  RET        ; Ruecksprung
```

Achtung: EAX, ECX und EDX dürfen verändert werden, andere Register müssen mit PUSH gesichert und später mit POP geladen werden.

5.2 Funktionsaufruf (CDECL)

Ein einfacher Funktionsaufruf

```
befehl(a, b, c)
```

erfolgt in Assembler folgendermaßen:

```
PUSH dword c ; Parameter pushen
PUSH dword b ; in umgekehrter
PUSH dword a ; Reihenfolge!
CALL befehl  ; Aufruf der Fkt.
; Parameter vom Stack entfernen
ADD esp, 12 ; = 4*3
```

5.3 If-Abfrage

Möchte man eine If-Abfrage wie z.B.

```
if( a == b ){
    // Befehle fuer True
}else{
    // Befehle fuer False
}
```

realisieren, dann geht dies mit:

```
CMP eax, ebx ; eax=a, ebx=b
JNE .else
  ; Befehle fuer True
  JMP .endif
.else:
  ; Befehle fuer False
.endif:
```

5.4 While-Schleife

Ein C-Code der folgenden Form

```
while( a < b ){
    // Inhalt der Schleife
}
```

kann folgendermaßen umgesetzt werden:

```
; eax=a, ebx=b
.loop:
  CMP eax, ebx  ; Schleife verlassen
  JAE .end      ; falls a >= b
  ; Inhalt der Schleife
  JMP .loop
.end:
```

5.5 Do-While-Schleife

Eine Do-While-Schleife, d.h.

```
do{
    // Inhalt der Schleife
}while( a < b )
```

kann in Assembler geschrieben werden als:

```
; eax=a, ebx=b
.loop:
  ; Inhalt der Schleife
  CMP eax, ebx  ; Schleife fortsetzen
  JB .loop      ; falls a < b
.end:
```

5.6 For-Schleife

Das Äquivalent zu

```
for(int i = 0; i < N; i++){
    // Inhalt der Schleife
}
```

wäre in Assembler:

```
MOV ecx, 0 ; ecx entspricht i
MOV edx, N
.loop:
  CMP ecx, edx  ; Schleife verlassen
  JAE .end      ; falls i >= N
  ;Inhalt der Schleife
  INC ecx
  JMP .loop
.end:
```

6 Aufgabenübersicht

Die nachfolgenden Aufgaben sind nach aufsteigendem Schwierigkeitsgrad sortiert. Bei allen Aufgaben ist bereits ein Grundgerüst vorbereitet, welches lediglich noch vervollständigt werden muss. Dieses enthält ggf. auch weitere nützliche Hinweise zur Lösung der Aufgabe.

6.1 `aufgabe-formel/` (sehr einfach)

Zum Einstieg soll die Formel $4 \cdot (3+2)$ mit Assemblerbefehlen ausgerechnet und das Ergebnis mithilfe des `EAX`-Registers zurückgegeben werden. Der Ansatz kümmert sich bereits um das Speichern und Wiederherstellen aller Register, sodass Sie trotz CDECL-Aufrufkonvention sämtliche Register zur Verfügung haben. Da wir den Multiplikationsbefehl noch nicht besprochen haben, drücken Sie die Multiplikation durch mehrere Additionen aus.

6.2 `aufgabe-formel-param/` (einfach)

Um das Auslesen der an eine Funktion übergebenen Parameter zu üben, soll die Funktion `int calculation(int a, int b, int c)` zur Berechnung von $a+2\cdot b+3\cdot c$ programmiert werden. Wie bei der vorherigen Aufgabe sollen Multiplikationen mithilfe von Additionen ausgedrückt werden.

6.3 `aufgabe-cpu/` (normal)

Assembler erlaubt nicht nur sehr effizienten Code, sondern auch gewisse Befehle, für die es kein Äquivalent in C gibt. Der Assemblerbefehl `RDTSC` z.B. liest die seit Systemstart verstrichene Anzahl von Ticks des Prozessors in das Registerpaar `EAX:EDX` (eine 64-Bit Zahl) - damit kann wiederum durch eine einfache Zeitmessung die Prozessorgeschwindigkeit berechnet werden. Ergänzen Sie den Assemblercode um den Aufruf des `RDTSC`-Befehls und die Rückgabe des Ergebnisses. Aufgrund der Größe der Zahl muss hierbei eine Rückgabe über einen Pointer erfolgen, das Register `EAX` alleine ist nicht ausreichend.

6.4 `aufgabe-string/` (normal)

Ein ASCII-String wird in der Regel als Folge von Bytes im Speicher gehalten, und mit dem ASCII-Code `0x00` terminiert. Programmieren Sie eine Funktion `int count_string(char *str)` in Assembler, welche bei einem übergebenen String-Pointer `str` die Länge des Inhaltes ermittelt. Dazu muss mithilfe einer Schleife gezählt werden, an welcher Stelle das Null-Byte kommt.

6.5 `aufgabe-mutex/` (normal)

Sie haben in der Vorlesung das Konzept eines Mutex kennengelernt. In dieser Aufgabe wollen wir selbst einen Mutex in Assembler programmieren. Wie in der Vorlesung kurz angedeutet, kann dazu der atomare Assemblerbefehl `XCHG` verwendet werden. Ergänzen Sie die Implementierungen der Assemblerfunktionen `trylock` und `unlock`, sodass der wechselseitige Ausschluss gewährleistet ist. Der Sourcecode enthält hierbei noch einige zusätzliche Hinweise, wie eine mögliche Implementierung funktionieren könnte.

6.6 `aufgabe-array/` (schwer)

In dieser Aufgabe soll eine Assembler-Funktion `int sum_array(int array[], int N)` geschrieben werden, welche die Einträge eines Integer-Arrays durchläuft, und am Ende die Summe der Einträge zurückgibt.

6.7 `aufgabe-primzahl/` (schwer)

Diese Aufgabe dient zum Üben des Aufrufs von C Befehlen aus Assembler. Dazu soll exemplarisch eine Funktion `int sum_primes(unsigned int until)` programmiert werden, welche alle Primzahlen bis zu einer übergebenen Größe aufsummiert. Der Primzahltest selbst liegt bereits als C-Funktion `int isPrime(unsigned int number)` vor, und liefert `1` im Falle einer Primzahl bzw. `0` falls keine Primzahl vorliegt zurück.