AMD64-Assembler-Handbuch

Fabian Schmied und M. Anton Ertl, Institut für Computersprachen

Basierend auf dem Alpha-Assembler-Handbuch von Andreas Krall und dem *AMD64 Architecture Programmer's Manual*

1 Allgemeines

Die AMD64-Architektur ist die auf PCs, Laptops, und Servern dominierende Architektur. Es ist eine 64-Bit-Architektur, d.h. die Adressen, die General-Purpose-Register, und der Befehlszeiger sind 64 Bit breit. Mit den in der Übungen vorhandenen Erweiterungen bis inklusive AVX-512 stehen 16 64-Bit General-Purpose-Register, 32 512-bit-SIMD-Register, 8 Maskenregister und 8 kombinierte 80-Bit-Gleitkomma/64-bit SIMD-Register zur Verfügung. Die meisten Befehle existieren in mehreren Versionen, so dass z.B. auf den General-Purpose-Registern sowohl mit allen 64 Bits, als auch mit den unteren 8, 16 oder 32 Bits gearbeitet werden kann bzw. bei den SIMD-Registern mit 512 Bits, 256 Bits, 128 Bits, und teilweise auch nur mit einem Wert (z.B. einer 32-bit FP-Zahl) statt einem Vektor von Werten. Neben dem 64-Bit-Modus unterstützt die Architektur auch Kompatibilitäts- und Legacy-Modi zum ausführen von Programmen im IA-32-Befehlssatz oder noch aeltere Befehlssätze.

Für die Übersetzerbauübungen werden weder Gleitkommaprogramme, noch ältere Befehlssätze verwendet. Dieses Handbuch geht daher nur auf den Ganzzahlanteil des 64-Bit-Modus der AMD64-Architektur ein. Weiterführende Informationen finden Sie im im Handbuch des GNU-Assemblers[1] und im Internet (z.B. das *Linux Assembly HOWTO*[2], `http://www.x86-64.org` und das *AMD64 Architecture Programmer's Manual*[3]).

2 Assemblersyntax

Dieses Handbuch benutzt die Syntax des GNU-Assemblers GAS (aufrufbar mit `as`, Dokumentation über `info as`). GAS erlaubt Kommentare in C-Syntax, zusätzlich kann ein Zeilenkommentar mit # beginnen. Namen bestehen aus Buchstaben, Ziffern, '.', '$' und '_'. Das erste Zeichen eines Namens darf keine Ziffer sein, Zahlen und Zeichenketten entsprechen der C-Konvention.

Jede Zeile der Eingabedatei enthält einen oder mehrere durch ';' getrennte *Statements*, das letzte Statement einer Datei muss durch einen Zeilenumbruch abgeschlossen sein. Statements können auch leer sein, in diesem Fall werden sie ignoriert. Jedes Statement beginnt mit optionalen Labels (ein von ':' gefolgter Name, dazwischen darf kein Whitespace-Zeichen stehen), dann kommt der Befehl. Beginnt der Befehl mit '.', so handelt es sich um eine Assembler-Direktive (siehe Abschnitt 8), beginnt der Befehl mit einem Buchstaben, so ist es ein Maschinenbefehl (siehe Abschnitt 6 und 7). Je nach Befehl folgen etwaige Operanden bzw. Argumente.

[1] `info as`

[2] `http://www.ibiblio.org/pub/Linux/docs/HOWTO/Assembly-HOWTO`

[3] `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf`

Die Operanden eines Befehls sind Register, Adressen, Offsets und Werte. Ein Offset ist eine ganze Zahl (siehe unten, *Offsetting*), Werte und Adressen werden durch Ausdrücke aus Zahlen und Symbolen mit Operatoren wie in C dargestellt. Folgende Operatoren werden unterstützt:

Unäre Operatoren: - (Zweierkomplementnegation), ~(bitweises NOT)

Binäre Operatoren: (Operatoren mit gleichem Vorrang werden im Code von links nach rechts ausgewertet)

- **Höchster Vorrang:** * (Multiplikation), / (Division), % (Restbildung), << (Linksschieben), >> (Rechtsschieben)
- **Mittlerer Vorrang:** | (bitweises inklusives OR), & (bitweises AND), ^ (bitweises exclusives OR), ! (bitweises OR NOT)
- **Niedriger Vorrang:** + (Addition), - (Subtraktion), == (Gleichheitsprüfung), <> (Ungleichheitsprüfung), <, <=, >, >= (übrige Vergleichsoperatoren für vorzeichenbehaftete Werte – die Vergleichsoperatoren liefern -1 für erfüllte und 0 für nicht erfüllte Vergleiche)
- **Niedrigster Vorrang:** && (Logisches AND), || (Logisches OR, hat einen leicht niedrigeren Vorrang als &&); diese Operatoren liefern 1 für einen erfüllten und 0 für einen nicht erfüllten Ausdruck

Bei der Befehlssyntax unterstützt GAS zwei Varianten: die AT&T-Syntax (default) und die Intel-Syntax. Obwohl die meisten Spezifikationen im Umfeld der AMD64-Architektur (z.B. das AMD64 Architecture Programmer's Manual) die Intel-Syntax verwenden, benutzen wir in diesem Dokument die unter Linux gebräuchlichere AT&T-Syntax, die z.B. auch von *GCC* generiert wird und sich von der Intel-Syntax in einigen Punkten unterscheidet. In GAS-Programmen kann die Intel-Syntax mit der Direktive `.intel_syntax` aktiviert werden.

Operandenpräfixe: Register werden in der AT&T-Syntax mit % gekennzeichnet, Immediate-Werte mit $. Beispiele: `%eax`, `$4`. Wenn das $-Präfix weggelassen wird, wird der Wert stattdessen als Speicheradresse interpretiert.

Operandenreihenfolge: Zuerst kommt der Quelloperand, dann das Ziel. Z.B. bedeutet `mov %eax, %ebx` eine MOV-Operation von `eax` nach `ebx` (in der Intel-Syntax ist es genau umgekehrt!). Im Zweifelsfall findet sich eine gute Zusammenfassung des Intel-Befehlssatz in AT&T-Syntax unter `http://docs.sun.com/app/docs/doc/802-1948`.

Befehlssuffixe: GAS erkennt die Operandengröße am Suffix des benutzten Befehls, hier muss b (8-Bit-Byte), w (16-Bit-Word), l (32-Bit-Long), d (32-Bit Doubleword), oder q (64-Bit-Quadword) stehen. Strenggenommen wäre die Syntax für eine MOV-Operation von `eax` nach `ebx` also `movl %eax, %ebx`). Wenn GAS die Operandengröße allerdings auf Grund der Registeroperanden erkennen kann, kann das Suffix meist weggelassen werden.

Offsetting: Um eine Speicherstelle zu indizieren oder indirekt auf einen Wert zuzugreifen wird das Indexregister oder die Speicheradresse in Klammern hinter dem Offset angegeben. `movl 17(%ebp), %eax` kopiert also einen Wert von einer Speicherstelle nach `eax`. Die Quellspeicherstelle befindet sich dabei 17 Bytes hinter der Adresse, die in `ebp` steht. Siehe dazu auch die Erklärung von ModR/M-Adressierung in Abschnitt 4.

64 Bit	32 Bit	16 Bit	obere 8 Bit des 16-Bit-Teils	untere 8 Bit
`rax`	`eax`	`ax`	`ah`	`al`
`rbx`	`ebx`	`bx`	`bh`	`bl`
`rcx`	`ecx`	`cx`	`ch`	`cl`
`rdx`	`edx`	`dx`	`dh`	`dl`
`rsi`	`esi`	`si`	–	`sil`
`rdi`	`edi`	`di`	–	`dil`
`rbp`	`ebp`	`bp`	–	`bpl`
`rsp`	`esp`	`sp`	–	`spl`
`r8`	`r8d`	`r8w`	–	`r8b`
`r9`	`r9d`	`r9w`	–	`r9b`
...	...	...	–	...
`r15`	`r15d`	`r15w`	–	`r15b`

Abbildung 1: General-Purpose-Register

Relative Jumps: Bei Sprungbefehlen wird ohne besondere Kennzeichnung der Operand als Zieladresse des Sprunges angenommen. Soll ein indirekter Sprung durchgeführt werden, muss die Adresse mit einem '*' als Präfix versehen werden.

Beispielcode finden Sie später in diesem Handbuch im Abschnitt 5.3 auf Seite 9.

3 Register

3.1 General-Purpose-Register

Abbildung 1 enthält eine Übersicht über die General-Purpose-Register der AMD64-Architektur. Alle Register sind über verschiedene Namen in verschiedenen Größen ansprechbar. Die Register `ah`, `bh`, `ch` und `dh` enthalten die oberen 8 Bit des entsprechenden 16-Bit-Registers `ax`, `bx`, `cx` und `dx`.

Wird ein 8- oder 16-Bit-Teil eines Registers überschrieben, so werden die übrigen Bits des Registers nicht verändert. Wird jedoch der 32-Bit-Teil des Registers manipuliert, werden automatisch die restlichen 32 Bit des Registers auf 0 gesetzt.

3.2 Spezialregister

`Rip` ist der Befehlszeiger (instruction pointer), der die Adresse des nächsten auszuführenden Maschinenbefehls enthält, das Register `rsp` wird üblicherweise als Stack-Pointer und `rbp` als Frame-Pointer (Zeiger in den Activation Record) benutzt. Einige der anderen Register haben ebenfalls Bedeutungen im Rahmen der unter Linux benutzten Aufrufkonventionen; diese werden in Abschnitt 5.1 beschrieben.

Das `rflags`-Register enthält in den untersten 16 Bit Operations-Flags wie *Carry*, *Parity*, *Zero* oder *Sign*, in den nächsten 16 Bit System-Flags, die nur von Systemsoftware aus zugreifbar sind. Die oberen 32 Bit des Registers sind reserviert und liefern beim Lesen immer 0. Abbildung 2 enthält eine Übersicht über die verfügbaren Operations-Flags und ihre Bedeutung.

Flag	Name	Bedeutung
CF	Carry	Die letzte Integer-Addition, -Subtraktion, oder Compare-Operation ergab einen Übertrag (*Carry* oder *Borrow*). Inkrement- und Dekrement-Befehle beeinflussen das Flag nicht, Shift- und Rotate-Befehle schieben hinausgeschobene Bits in das Carry-Flag, logische Operationen löschen das Flag.
PF	Parity	Das letzte Resultat bestimmter Operationen hatte eine gerade Anzahl von gesetzten Bits.
AF	Auxiliary Carry	Die letzte Binary-Coded-Decimal-Operation ergab ein Carry in Bit 3. Auf BCD-Operationen wird in diesem Handbuch nicht eingegangen.
ZF	Zero	Das letzte Resultat einer arithmetischen Operation war 0. Dieses Flag wird auch von den Vergleichs-Befehle gesetzt und kann benutzt werden, um zwei Werte auf Gleichheit zu prüfen.
SF	Sign	Das letzte Resultat einer arithmetischen Operation war negativ. (Das SF ist auf Grund der benutzten Zweierkomplementdarstellung von Ganzzahlen immer gleich dem höchstwertigen Bit des Resultats.)
DF	Direction	Bestimmt die Verarbeitungsrichtung für String-Befehle. Auf String-Befehle wird in diesem Handbuch nicht eingegangen.
OF	Overflow	Das Resultat der letzten Signed-Integer-Operation war zu groß, um in den Datentyp zu passen. Dieses Flag ist nach einer DIV-Befehl und nach Shifts um mehr als ein Bit undefiniert. Logische Operationen löschen das Flag.

Abbildung 2: Operations-Flags

Neben diesen beiden gibt es noch eine Reihe weiterer Spezialregister, die in dieser LVA aber keine Rolle spielen.

3.3 SIMD-Register und Maskenregister

Als Erweiterung zur general-purpose Architektur, die mit einzelnen ganzen Zahlen und Adressen arbeitet, gibt es in der AMD64-Architektur eine Reihe von aufeinander aufbauenden SIMD-Erweiterungen (Single Instruction, Multiple Data), von SSE bis AVX-512.[4]. Im Rahmen dieser Erweiterungen gibt es auch Befehle für einzelne („skalare") Gleitkommazahlen, mit denen wir uns hier aber nicht weiter beschäftigen, genausowenig mit der älteren 80387-Gleitkomma-Erweiterung.

Die SIMD-Befehle sind vor allem für Anwendungen aus dem Multimedia- und Wissenschaftsbereich gedacht sind, bei denen viele Einzelwerte aus großen Datenmengen unabhängig voneinander verarbeitet werden müssen. Die meisten SIMD-Befehle führen die selbe Operation parallel auf den Einzelwerten durch; dabei wird eine Operation auf Einzelwerten eine Spur (engl. *lane*) genannt, entsprechend dem Bild einer Autobahn, wo sich auch mehrere Autos parallel auf mehreren Spuren bewegen und damit den Durchsatz erhöhen.

[4] Daneben gibt es noch die ältere SIMD-Erweiterung MMX, die aber durch SSE ersetzt wurde, keine Rolle mehr spielt und daher hier nicht weiter erwähnt wird

Die Operanden von der AVX-512-Befehle sind 512/256/128 Bit große Vektoren, die Einzelwerte darin können Ganzzahlen (von 8-Bit-Bytes bis zu 64-Bit-Quadwords) oder Gleitkommawerte sein. AVX-512 bietet 32 512-bit breite Register `zmm0`...`zmm31`, von denen man aber auch nur die unteren 256 bits (`ymm0`...`ymm31`) oder 128 bits (`xmm0`...`xmm31`) ansprechen kann.[5] Wir werden uns in diesem Handbuch auf die Ganzzahlverarbeitung konzentrieren.

Neben den `zmm`-Datenregistern gibt es noch 8 64 Bit breite Maskenregister `k0`...`k7`, mit denen man für jede Spur bestimmen kann, ob die Operation ausgeführt wird, oder ob das Ergebnis der Spur ein Default-Wert ist (0 oder ein Operand). Dabei spielt `k0` eine Sonderrolle: Bei Verwendung als Maske für einen Befehl auf SIMD-Registern bedeutet `k0`, dass die Operation auf allen Lanes durchgeführt wird (unmaskiert), dafür kann man `k0` also nicht verwenden. Man kann es aber verwenden, um bei Operationen auf Maskenregistern Zwischenresultate zu speichern, die dann nicht als Masken in SIMD-Befehlen verwendet werden.

4 Speichermodell und Adressierung

Die AMD64-Architektur benutzt im 64-Bit-Modus ein flaches Segmentierungsmodell für den virtuellen Speicher. Der gesamte 64-Bit-Speicherraum wird dabei als ein einziger, flacher Adressraum betrachtet. Befehle und Daten werden darin im Little-Endian-Format[6] abgelegt und über verschiedene Adressierungsmodi angesprochen:

Bei *absoluter Adressierung* werden die Adressen direkt als Werte angegeben. Beispiel: `movl 0x1234, %eax`. Seit einiger Zeit produziert der Linker bei Verwendung absoluter Adressen fuer Zugriffe auf Daten Fehlermeldungen. Verwenden Sie stattdessen RIP-relative Addressierung.

Bei *RIP-relativer Adressierung* werden Adressen als Offsets zum Befehlszeiger (`rip`) angegeben. Dies ist sinnvoll für relative Sprünge, aber auch, um positionsunabhängigen Code (PIC/PIE) zu erzeugen: Wird auf Symbole RIP-relativ zugegriffen, kann der Linker den Code in einen beliebigen Speicherbereich verschieben, ohne die Adressen anpassen zu müssen. Beispiel: `movl xvar(%rip), %eax` (der Compiler codiert für `xvar` automatisch einen relativen Offset statt einer absoluten Adresse). Sprünge werden automatisch relativ codisert, Sie brauchen (und dürfen) kein `(%rip)` angeben.

ModR/M-Adressierung dient dem indirekten Zugriff auf Speicherbereiche, deren Adressen zur Laufzeit berechnet werden. Dabei wird die effektive Adresse aus einer *Basisadresse* und einem *Index*, die aus General-Purpose-Registern ausgelesen werden, sowie einem *Skalierungsfaktor* (1, 2, 4 oder 8) und einem *Displacement*, die direkt im Code angegeben sind, berechnet. Die Formel für die Adressberechnung lautet: $Base + Index * Scale + Displacement$, das Ergebnis wird wie eine absolute Adresse behandelt.

In der AT&T-Assembler-Syntax werden derartige Adressen in der Form `DISP(BASE, INDEX, SCALE)` angegeben, wobei alle vier Werte optional sind – der Default-Wert für SCALE ist 1, für alle anderen Werte ist er 0. Wenn innerhalb der Klammern nur ein Wert mit voranstehendem Komma angegeben wird, wird er als SCALE-Wert interpretiert. Abbildung 3 zeigt einige Beispiele von ModR/M-Adressen.

[5] Die `xmm` und `ymm`-Register kommen auch in den diversen SSE und AVX-Erweiterungen vor, allerdings je nach Erweiterung mit anderen Einschränkungen; in diesem Handbuch beschreiben wir nur AVX-512.

[6] Die Bytes eines Datenwerts werden von rechts nach links angeordnet, so dass das höchstwertige Byte „rechts", d.h. an der höchsten Adresse, das niederwertige Byte „links", d.h. an der niedrigsten Adresse steht. Die hexadezimalen 16-Bit-Zahl 0xABCD würde im Speicher also als Bytefolge CD AB abgelegt.

Ausdruck	DISP	BASE	INDEX	SCALE	Bedeutung
`-4(%ebp)`	-4	`ebp`	Default	Default	`ebp`-4
`foo(,%eax,4)`	foo	Default	`eax`	4	foo + 4*`eax`
`foo(,1)`	foo	Default	Default	1	foo (siehe Text)
`-4(%ebp, %eax, 4)`	-4	`ebp`	`eax`	4	`ebp` + 4*`eax` - 4

Abbildung 3: ModR/M-Adressbeispiele

Daten auf dem *Stack* werden über den Stack-Pointer `rsp` adressiert, der von Befehlen wie POP, PUSH, CALL, RET und INT implizit verändert wird.

5 Aufrufkonventionen

Um sicherzustellen, dass getrennt kompilierte Programmteile problemlos verlinkt werden können, gibt es für die AMD64-Architektur eine *System V Application Binary Interface*-Spezifikation[7], die festlegt, welche Konventionen beim Aufruf von Funktionen eingehalten werden sollten. Diese beinhaltet Einschränkungen für die Register, die in den Abschnitten 3.1 und 3.3 beschrieben wurden, und Konventionen für den Stack.

5.1 Register

Die Register `rbp`, `rbx` und `r12` bis `r15` gehören dem aufrufenden Code (*Caller*). Wenn die aufgerufene Funktion (*Callee*) die Register verändert, dann muss sie diese zuvor auf dem Stack sichern (z.B. mit dem PUSH-Befehl) und ihre Inhalte vor der Rückkehr zum Caller wiederherstellen (POP), man bezeichnet sie daher als *callee-saved Registers*. Alle anderen Register gehören dem Callee; der Caller muss also selbst für die Sicherung sorgen, wenn er ihren Inhalt über einen Aufruf hinweg benötigt (*caller-saved Registers*). Speziell gesichert wird das Register `rsp`, das üblicherweise beim Funktionsaufruf bzw. im Prolog einer Funktion verändert und nach `rbp` kopiert und im Epilog bzw. beim Rücksprung wiederhergestellt wird (siehe auch Abschnitt 5.2).

Beim Aufruf einer Funktion werden ganzzahlige Argumente der Reihe nach in `rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9` übergeben. Floating-Point-Werte und SIMD-Werte werden über die Register `xmm0`/`ymm0`/`zmm0` bis `xmm7`/`ymm7`/`zmm7` übergeben. Masken werden über die General-Purpose-Register übergeben. Wenn mehr Argumente übergeben werden sollen als Register zur Verfügung stehen, werden die Argumente von rechts nach links (d.h. letztes Argument an die höchste Adresse[8]) auf den Stack geschrieben.

Wenn eine Funktion ganzzahlige Ergebniswerte liefert, werden diese über die Register `rax` und `rdx` an den aufrufenden Code zurückgegeben. SSE-Ergebnisse werden in die Register `xmm0` und `xmm1` geschrieben.

Abbildung 4 fasst die üblichen Verwendungen der Register zusammen.

[7] `http://www.x86-64.org/documentation/abi.pdf`

[8] Der Stack wächst von oben (hohe Adressen) nach unten (niedrige Adressen), die Spitze des Stacks befindet sich daher an der niedrigsten Adresse. Wenn ein Argument im Stack an einer höheren Adresse steht als ein anderes, befindet es sich also „tiefer“ im Stack.

Register	Verwendungszweck	Sicherung
`rax`	Temporäres Register, erstes Rückgaberegister	Caller
`rbx`	Callee-gesichertes Register	Callee
`rcx`	Argumentregister für das vierte Ganzzahlargument	Caller
`rdx`	Argumentregister für das dritte Ganzzahlargument, zweites Rückgaberegister	Caller
`rsp`	Stack-Pointer	Callee (implizit)
`rbp`	Callee-gesichertes Register, wird als Frame-Pointer benutzt	Callee
`rsi`	Argumentregister für das zweite Ganzzahlargument	Caller
`rdi`	Argumentregister für das erste Argument	Caller
`r8`	Argumentregister für das fünfte Argument	Caller
`r9`	Argumentregister für das sechste Argument	Caller
`r10`	Temporäres Register, wird benutzt, um den Static-Chain-Pointer einer Funktion zu übergeben (in diesem Handbuch nicht weiter behandelt)	Caller
`r11`	Temporäres Register	Caller
`r12-r15`	Callee-gesicherte Register	Callee
`zmm0`	Argument- und Rückgaberegister für SIMD-Werte	Caller
`zmm1-zmm7`	Argumentregister für SIMD-Werte	Caller
`zmm8-xmm31`	Temporäre Register	Caller
`k0-k7`	Temporäre Register	Caller

Abbildung 4: System V ABI Aufrufkonventionen für AMD64

5.2 Stack

Unmittelbar vor dem Aufruf einer Funktion C durch die Funktion B (die ihrerseits von A aufgerufen wurde) enthält der Stack normalerweise folgende Daten in einem Activation Record:

1. Argumente für B (sofern diese nicht per Register übergeben werden). Sie wurden von A auf den Stack gelegt.
2. Die Rücksprungadresse. Sie wird automatisch vom CALL-Befehl auf den Stack gelegt.
3. Ein Feld von Activation-Record-Zeigern, die es ermöglichen, auf lokale Variablen von umgebenden Funktionen zuzugreifen (falls die aufgerufene Funktion eine statisch geschachtelte Funktion ist, darauf wird allerdings in diesem Handbuch nicht näher eingegangen).
4. Lokale Variablen der Funktion. Sie werden von der Funktion selbst initialisiert.
5. Eventuell Padding (ungenutzer Platz), das dafür sorgt, dass der Stack Pointer auf 16 Bytes ausgerichtet ist, sobald die Argumente für C auf den Stack gelegt wurden. Da das Padding vom Platz für die Argumente abhängt, kann es für jeden Aufruf anders sein.
6. Argumente für C, die im Speicher übergeben werden. Sie werden von B auf den Stack gelegt, aber man zählt sie schon zum Activation Record von C.

Der Stack ist an dieser Stelle auf eine 16-Byte-Grenze ausgerichtet (aligned). Abbildung 5 zeigt den Stack der Funktion

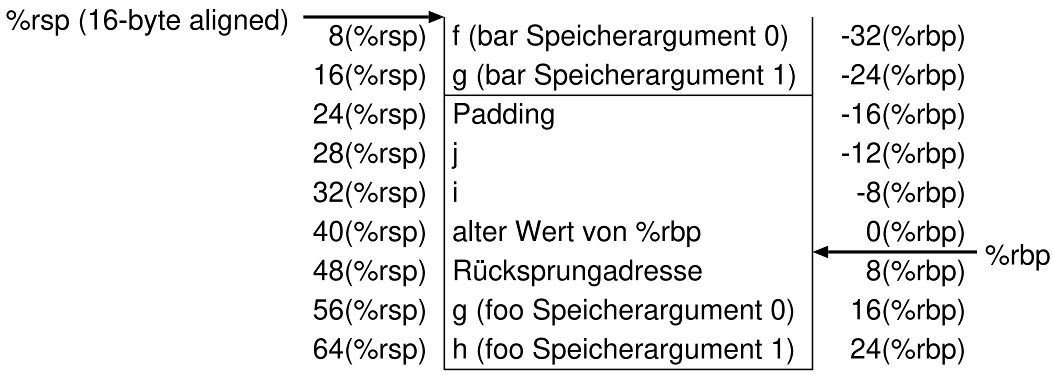

Abbildung 5: Der Stack mit einem Activation Record

```
int foo(char a, int b, long c, int d, int e, int f, int g, char h)
{
  char i=a+h;
  int j=b+g;
  bar(i,j,b,c,d,e,f,g);
  return i+j;
}
```

unmittelbar vor dem Aufruf von bar (also nachdem die Argumente bereitgelegt wurden), wobei `foo` einen Frame Pointer (siehe unten) verwendet.

Das `rbp`-Register wird häufig als Frame-Pointer benutzt: Es zeigt als Basisadresse auf jenen Bereich auf dem Stack, in dem der aktuelle Code seine lokalen Variablen ablegt (Activation Record). Über positive Offsets zum Frame-Pointer gelangt man an die Argumente, über negative Offsets an die lokalen Variablen. Üblicherweise enthält jede Funktion zur Initialisierung des Frame-Pointers und gleichzeitigen Sicherung von `rsp` einen Prolog, der wie folgt aussieht:

```
push %rbp        # rbp sichern
mov %rsp, %rbp   # neuen rbp setzen und rsp sichern
sub ..., %rsp    # Platz für lokale Variablen am Stack reservieren
```

(Manchmal wird stattdessen auch der ENTER-Befehl benutzt, der eine äquivalente Funktionalität zu diesem Codeblock bietet.)

Am Ende einer Funktion steht normalerweise ein Epilog, der die vom Prolog vorgenommenen Änderungen rückgängig macht und die Kontrolle an die aufrufende Funktion zurückgibt. Der LEAVE-Befehl wird benutzt, um den Stack aufzuräumen (d.h. `rsp` wieder auf den Wert des Frame-Pointers zu setzen und `rbp` vom Stack zu holen), mit RET wird dann zur Rücksprungadresse gesprungen (und die Adresse vom Stack entfernt).

```
leave            # rsp auf rbp setzen, rbp wiederherstellen
ret              # Rücksprung
```

(LEAVE ist äquivalent zu `mov %rbp, %rsp` und `pop %rbp`.)

Variante ohne Frame-Pointer Um Befehle zu sparen kann der Stack-Pointer auch direkt benutzt werden, wenn auf den Activation Record zugegriffen wird. Dabei wird `rbp` als normales (Callee-gesichertes) Register benutzt, und die Funktion muss manuell sicherstellen, dass `rsp` beim Rücksprung aus der Funktion wieder den ursprünglichen Wert enthält.

5.3 Beispielprogramm

Das folgende C-Beispielprogramm zeigt die Anwendung der Aufrufkonventionen:

```
long xvar;
extern long callee(long,long,long);

long caller() {
  long i = xvar;
  return i + callee(-1, -2, -3);
}
```

Hier liest eine Funktion *caller* einen globalen Wert *xvar* in eine lokale Variable *i* ein, ruft eine zweite Funktion *callee* mit drei Argumenten auf, addiert den Wert von *i* zum Ergebnis von *callee* und retourniert die Summe als ihren eigenen Rückgabewert.

Hier ein Auszug aus dem mittels `gcc -S` generierten Assembler-Code mit Kommentaren:

```
        .text                       # Programmbereich aktivieren
.globl caller                       # caller als externes Symbol definieren
        .type   caller, @function   # caller als Funktion deklarieren
caller:                             # Aufrufstelle für caller

# Funktions-Prolog
.LFB3:
        pushq   %rbp                # Frame Pointer auf dem Stack sichern
.LCFI0:
        movq    %rsp, %rbp          # neuen Frame Pointer setzen
.LCFI1:
        subq    $16, %rsp           # Platz für lokale Variablen reservieren

# Eigentliche Funktion
.LCFI2:
        movq    xvar(%rip), %rax    # Inhalt von xvar rip-relativ in rax
                                    # kopieren
        movq    %rax, -8(%rbp)      # rax als lokale Variable i speichern
        movq    $-3, %rdx           # drittes Argument (3) in edx laden
        movq    $-2, %rsi           # zweites Argument (2) in esi laden
        movq    $-1, %rdi           # erstes Argument (1) in edi laden
        call    callee              # callee aufrufen
        addq    -8(%rbp), %rax      # Inhalt von lokaler Variablen i zu rax
                                    # (Rückgabewert von callee) dazuzählen,
```

```
                                        # Resultat in rax speichern

# Funktions-Epilog
        leave                           # Activation Record der Funktion entfernen
        ret                             # Rückkehr von call
```

6 Befehlssatz

In diesem Abschnitt werden einige Befehle aus dem Befehlssatz der AMD64-Architektur vorgestellt. Der GNU-Assembler benutzt in den meisten Fällen dieselben Mnemonics für die Befehle wie die AMD64-Architekturspezifikation. Bitte beachten Sie aber, dass die Befehle – wie in Abschnitt 2 beschrieben – eventuell durch Suffixe ergänzt werden müssen, um die Größe der Operanden zu definieren, sofern der Assembler sie nicht erkennen kann.

Alle Befehle der AMD64-Architektur finden Sie in den Architekturhandbüchern der Hersteller[9]).

6.1 Datentransferoperationen

Diese Befehle kopieren Daten zwischen Registern und dem Arbeitsspeicher.

6.1.1 Move

Move-Operationen kopieren Byte-, Word-, Long- und Quadword-Werte von Registern, Speicheradressen oder im Code angegebenen Werten zu Registern oder Speicheradressen.

```
mov source, dest
movsx source, dest   # Move mit Sign Extension
movzx source, dest   # Move mit Zero Extension
```

Source und *destination* müssen für MOV die gleiche Größe haben, MOVSX und MOVZX können kleinere Werte zu größeren Werten kopieren und füllen die restlichen Bits entweder mit Nullen auf (*Zero Extension*) oder stellen sicher, dass das Vorzeichen erhalten bleibt (*Sign Extension*). Im Gegensatz zu anderen Operationen benötigen MOVSX und MOVZX *zwei* Größenangaben, wenn der Assembler die Operandengrößen nicht selbst eruieren kann. In der Syntax lässt man dann das X weg und hängt zwei Suffixes an: das erste gibt die Größe des Quelloperanden, das zweite die des Zieloperanden an. (MOVSBW bedeutet beispielsweise eine Move-Operation mit Sign-Extension von einem 8-Bit-Operanden zu einem 16-Bit-Ziel.)

Source und *destination* können bei MOV-Operationen (was für die meisten Befehle gilt) nicht beide gleichzeitig Speicheradressen sein.

Anstelle von `mov $0, reg` wird häufig die Anweisung `xor reg, reg` benutzt, da dies in manchen Fällen effizienter sein kann.

[9] `http://www.complang.tuwien.ac.at/ubvl/#Unterlagen`

Flag-Kürzel	Flag	Bedeutung
o	OF = 1	Overflow
no	OF = 0	No overflow
b, c, nae	CF = 1	Below, carry, not above or equal (unsigned)
ae, nb, nc	CF = 0	Above or equal, not below, no carry (unsigned)
e, z	ZF = 1	Equal, zero
ne, nz	ZF = 0	Not equal, not zero
be, na	CF = 1 oder ZF = 1	Below or equal, not above (unsigned)
a, nbe	CF = 0 und ZF = 0	Above, not below or equal (unsigned)
s	SF = 1	Sign
ns	SF = 0	No sign
p, pe	PF = 1	Parity, parity even
np, po	PF = 0	No parity, parity odd
l, nge	SF <> OF	Less, not greater or equal (signed)
ge, nl	SF = OF	Greater or equal, not less (signed)
le, ng	ZF = 1 oder SF <> OF	Less or equal, not greater (signed)
g, nle	ZF = 0 und SF = OF	Greater, not less or equal (signed)

Abbildung 6: CMOV-Befehle

6.1.2 Conditional Move

Conditional-Move-Operationen sind äquivalent zu normalen Move-Operationen, werden aber nur ausgeführt, wenn ein bestimmtes Bit des `rflags`-Registers gesetzt ist. In vielen Fällen ist diese Art von Move-Operation effizienter als eine äquivalente Formulierung des Programms mit Hilfe von Jumps.

```
cmovCC source, dest
```

Die entsprechenden Flags werden z.B. über Vergleichs- und Test-Befehle gesetzt (siehe Abschnitt 6.5). Abbildung 6 listet die möglichen Flag-Kürzel für `CC` auf.

6.1.3 Stack-Operationen

Üblicherweise existiert für jede Funktion oder Prozedur ein Activation Record in einem dafür reservierten Speicherbereich – dem Stack. Die Funktion legt darin lokale Variablen ab, kann mit seiner Hilfe Registerinhalte sichern und Parameter für aufgerufene Funktionen übergeben (siehe Abschnitt 5.2). Die Stack-Befehle dienen der einfacheren Manipulation des Stacks, auf dessen aktuelle Spitze immer der Stack-Pointer `rsp` zeigt.

Die PUSH-Operation legt einen Byte-, Word-, Long- oder Quadword-Wert aus einem Register, von einer Speicheradresse oder aus dem Code auf den Stack, POP liest einen Wert vom Stack in ein Register oder einen Speicherbereich aus. ENTER erzeugt einen neuen Stack-Frame für eine Prozedur oder Funktion, LEAVE entfernt den Activation Record einer Prozedur, wie in Abschnitt 5.2 beschrieben.

```
push source
```

```
pop dest
enter size, depth
leave
```

PUSH und POP passen `rsp` entsprechend um 2, 4, oder 8 Bytes an, so dass der Stack-Pointer nach der Operation auf die neue Spitze des Stacks zeigt.

Der `depth`-Parameter (ein Wert aus dem Intervall 0-31) für ENTER gibt an, wieviele Activation-Record-Zeiger von der aufrufenden Prozedur kopiert werden sollen, um eine geschachtelte Funktion zu realisieren, der Wert `size` bestimmt, wieviele Bytes (z.B. für lokale Variablen der Prozedur) auf dem Stack allokiert werden. ENTER mit einer Schachtelungstiefe von Null entspricht der Codesequenz `push rbp; mov %rsp, %rbp; sub ..., %rsp` aus Abschnitt 5.2. LEAVE ist äquivalent zur dort angegebenen Codesequenz `mov %rbp, %rsp; pop %rbp`.

6.2 Adressladen

Der LEA-Befehl berechnet und lädt die effektive Adresse einer Speicherstelle und legt diese in ein General-Purpose-Register.

```
lea source, destination
```

LEA ist ähnlich zum MOV-Befehl, der benutzt werden kann, um Daten von einer Speicheradresse in ein Register zu kopieren, aber anstatt den *Inhalt* des angegebenen Speicherbereichs zu laden, lädt LEA die *Adresse*.

Im einfachsten Fall kann LEA durch MOV ersetzt werden, z.B. ist der Befehl `lea (%ebx), %eax` gleichbedeutend mit `mov %ebx, %eax`. Mit LEA kann jedoch jeder beliebige Adressausdruck ausgewertet werden, z.B. `lea (%edi,%ebx,1), %eax`, was nicht durch ein MOV nachgebildet werden kann.

LEA kann auch in begrenztem Maß dazu eingesetzt werden, um Multiplikationen nachzubilden, indem ModR/M-Adressierung benutzt wird (z.B. multipliziert `lea (%ebx,%ebx,8), %eax` den Wert von `ebx` mit Neun und speichert das Ergebnis in `eax`).

6.3 Arithmetische Operationen

Arithmetische Befehle werden benutzt, um Grundrechenoperationen auf Ganzzahlen durchzuführen.

6.3.1 Addition und Subtraktion

ADD addiert zwei ganzzahlige Operanden und setzt die entsprechenden Bits des `rflags`-Registers (OF, SF, ZF, AF, CF, PF). Wenn man ein Wert zu einem Operanden mit höherer Bitbreite addiert, so wird der kleinere Wert zunächst mit Sign-Extension auf die größere Bitbreite erweitert. SUB subtrahiert zwei ganzzahlige Operanden. ADC und SBB sind äquivalent zu ADD und SUB, addieren bzw. subtrahieren aber zusätzlich Eins zum/vom Resultat, wenn das Carry-Flag gesetzt ist. ADCX ist wie ADC, überschreibt aber nur das Carry-Flag, kein anderes. ADOX ist wie ADCX, verwendet aber das Overflow-Flag anstelle des Carry-Flags (sowohl als zusätzlichen Addenden als auch im Ergebnis). NEG führt eine arithmetische Negation durch (d.h. es subtrahiert den Operanden von Null, dreht also das Vorzeichen um).

```
add  source, source_dest
sub  source, source_dest
adc  source, source_dest   # Add mit Carry
adcx source, source_dest   # Add mit Carry
adox source, source_dest   # Add mit Carry
sbb  source, source_dest   # Sub mit Carry (Borrow)
neg  source_dest
```

SUB und SBB subtrahieren den ersten vom zweiten Operanden und speichen das Ergebnis im zweiten Operanden (`source_dest = source_dest - source`).

6.3.2 Multiplikation und Division

MUL und DIV führen vorzeichenlose, IMUL und IDIV vorzeichenbehaftete Multiplikation und Division von Ganzzahlen durch. Bei einer Multiplikation kann die Bitbreite des Operationsergebnisses doppelt so groß sein wie die der Quelloperanden.

```
mul factor
imul factor
imul factor, multiplicand_product
imul factor, multiplicand, product
div divisor
idiv divisor
```

MUL erwartet einen Operanden (je nach Größe) in `al`, `ax`, `eax` oder `rax` und legt das Ergebnis in `ax`, `dx:ax`, `edx:eax` oder `rdx:rax` ab.

IMUL verhält sich in der einargumentigen Form wie MUL, kann jedoch auch zwei oder drei Argumente nehmen. (Siehe dazu auch *::Machine-Dependencies::i386-Dependent::i386-Notes* in der GAS-Dokumentation.)

DIV und IDIV erwarten den Dividenden in `ah:al`, `dx:ax`, `edx:eax` oder `rdx:rax`. Der Quotient wird in `al`, `ax`, `eax` oder `rax` gespeichert, der Rest steht in `ah` oder im entsprechenden `dx`-Register. Division ist die langsamste aller Ganzzahloperationen.

6.3.3 Inkrement und Dekrement

INC und DEC erhöhen bzw. verringern den Inhalt eines Registers oder einer Speicherstelle um 1.

```
inc source_dest
dec source_dest
```

INC und DEC verhalten sich genau wie ADD und SUB mit Eins als erstem Argument, verändern aber das Carry-Flag nicht.

6.4 Rotate und Shift

Diese Befehle führen zyklische Rotation oder nichtzyklische Schiebeoperationen um eine bestimmte Anzahl (Count) von Bits durch. Der Count kann ein 8-bit-Wert oder der Inhalt des `cl`-Registers sein. (Ein Rotate oder Verschieben um N Bits entspricht einem N-maligen Rotieren oder Verschieben um ein Bit, die Werte von CF und OF sind danach jedoch nicht definiert.)

```
rcl count, source_dest   # Rotate through Carry left
rcr count, source_dest   # Rotate through Carry right
rol count, source_dest   # Rotate left
ror count, source_dest   # Rotate right

sal count, source_dest   # Shift arithmetic left (signed)
sar count, source_dest   # Shift arithmetic right (signed)
shl count, source_dest   # Shift left (unsigned)
shr count, source_dest   # Shift right (unsigned)
shld count, source, source_dest   # Shift left double (unsigned)
shrd count, source, source_dest   # Shift right double (unsigned)
```

RCL und RCR rotieren den Operanden durch das Carry, d.h. ein hinausrotiertes Bit wird ins Carry geschrieben und der Wert des Carry-Flags wird am anderen Ende des Werts hineinrotiert. Auch ROL und ROR verändern das Carry-Flag auf das zuletzt hinausrotierten Bits, rotieren das Bit aber sofort wieder hinein, d.h. der vorherige Wert des Carry-Flags geht verloren. Das Overflow-Flag zeigt nach ROL- und ROR-Operationen um ein Bit an, ob sich durch die Rotation das Vorzeichenbit des Operanden verändert hat.

SHL und SAL können als effizienter Ersatz für eine Multiplikationsoperation mit Zwei (bzw. Potenzen davon), SHR und SAR statt einer Division durch Zwei benutzt werden. (Das Ergebnis von SAR unterscheidet sich bei negativem Operanden allerdings in der Rundung vom Ergebnis von IDIV.) SAR konserviert das Vorzeichen des Operanden bei der Schiebeoperation, d.h. es werden bei negativen Werten Einsen und bei positiven Werten Nullen von links hereingeschoben, SHR schiebt immer Nullen herein. Alle Shift-Operationen schieben das hinausgeschobene Bit ins Carry-Flag.

SHLD und SHRD führen Schiebeoperationen mit doppelter Länge durch, d.h. `count` Bits des `source`-Werts werden (statt Nullen) in den `source_dest`-Wert hineingeschoben.

6.5 Vergleichen und Testen

CMP subtrahiert analog zu SUB den Wert des ersten Operanden von dem des zweiten, überschreibt diesen jedoch nicht. Der einzige Effekt der Operation ist daher das Setzen der Flags (CF, SF, ZF, AF, CF, PF). TEST führt eine logische AND-Operation mit den beiden Operanden aus (analog zum AND-Befehl), überschreibt jedoch ebenfalls keinen der beiden Operanden, sondern setzt nur die entsprechenden Flags (SF, ZF und PF; OF und CF werden gelöscht). TEST wird üblicherweise benutzt, um bestimmte oder alle Bits eines Operanden auf 0 zu überprüfen.

BT kopiert das im ersten Operanden (Wert oder Register) angegebene Bit des zweiten Operanden (Register oder Speicheradresse) in das Carry-Flag. BTC invertiert danach das gelesene Bit im Operanden, BTS setzt es auf Eins, BTR auf Null.

Die SET-Operationen setzen den Wert ihres Byte-Operanden (Register oder Speicheradresse) nach Prüfen eines Flags auf Null oder Eins.

```
cmp source, source
test source, source
bt index, source
btc index, source_dest
bts index, source_dest
btr index, source_dest
setCC dest
```

Die für `CC` einsetzbaren Flag-Kürzel sind dieselben wie bei den CMOV-Befehlen (siehe Abbildung 6).

6.6 Logische Operationen

AND, OR und XOR führen die bekannten logischen Operationen bitweise auf ihren Operanden (Werte, Register oder Speicheradresse) aus. Sie setzen dabei die entsprechenden Flags (ZF, SF und PF, CF und OF werden gelöscht) und überschreiben ihren zweiten Operanden (Register oder Speicheradresse). NOT invertiert den Wert des Operanden und überschreibt diesen, ohne Flags zu verändern.

```
and source, source_dest
or source, source_dest
xor source, source_dest
not source_dest
```

AND und OR können benutzt werden, um auf Null, Vorzeichen und Parität zu prüfen, indem beide Operanden dasselbe Register enthalten. Wenn XOR dasselbe Register in beiden Operanden erhält, wird das Register auf Null gesetzt.

6.7 Kontrolltransfer

6.7.1 Jumps und Loops

JMP führt einen unbedingten Sprung zur angegebenen Adresse durch. Die Adresse kann relativ zum Befehlszeiger `rip` oder indirekt in einem Register oder per Speicheradresse angegeben werden. Für relative Sprünge werden im Assemblercode normalerweise Labels als Sprungziele notiert, woraus bei der Assemblierung automatisch Offsets zu `rip` berechnet werden.

Die J-Befehle führen bedingte Sprünge auf Basis von Flag-Werten aus und sind immer relativ. JCXZ, JECXZ und JRCXZ sind bedingte Sprungbefehle, die aber nicht ein Flag prüfen, sondern zu einem Sprung führen, wenn das Register `cx`, `ecx` bzw. `rcx` den Wert 0 enthält.

Die LOOP-Befehle dekrementieren den Inhalt des `cx`-, `ecx`- oder `rcx`-Registers ohne ein Flag zu verändern und führen dann einen bedingten Sprung aus, wenn ihre Schleifenbedingung erfüllt ist. Die Bedingung für LOOP selbst ist erfüllt, wenn das Register nach dem Dekrementieren einen anderen Wert als Null enthält. Für LOOPE und LOOPZ ist die Bedingung erfüllt, wenn außerdem noch das Zero-Flag gesetzt ist, für LOOPNE und LOOPNZ muss das Zero-Flag (zusätzlich zur Bedingung von LOOP) gelöscht sein.

```
jmp label; jmp *address
jCC label
jcxz label; jecxz label; jrcxz label
loop label
loope label; loopne label
loopnz label; loopz label
```

Die für CC einsetzbaren Flag-Kürzel sind dieselben wie für die CMOV-Befehle (siehe Abbildung 6).

6.7.2 Prozeduraufrufe

Der CALL-Befehl dient zum Aufruf einer Prozedur. Er ist äquivalent zum JMP-Befehl, legt aber vor dem Sprung die Adresse des nächsten Befehls als Rücksprungadresse auf den Stack. RET holt diese Adresse später wieder vom Stack und führt den Rücksprung aus. Beide Befehle verändern dabei natürlich den Stack-Pointer `rsp`. RET kann als optionales Argument eine Bytezahl übernehmen, das angibt, wieviel nach dem Entfernen der Rücksprungadresse von `rsp` abgezogen werden soll (z.B. um übergebene Parameter vom Stack zu entfernen).

```
call label
call *address
ret
ret size
```

6.8 Flags

Zur Verwaltung des `rflags`-Registers gibt es eine ganze Reihe an Assembler-Befehlen: PUSHF, PUSHFD und PUSHFQ sowie POPF, POPFD und POPFQ dienen der Sicherung der Flags (`flags`, `eflags` bzw. `rflags`) auf dem Stack und der Wiederherstellung früher gesicherter Flags. Dabei wird natürlich der Stack-Pointer angepasst. Flags die nicht verändert werden können oder dürfen werden dabei nicht manipuliert.

CLC, CMC und STC dienen dem Löschen, Invertieren und Setzen des Carry-Flags. Dies ist z.B. nötig, bevor eine Shift- oder Rotate-Operation gestartet wird, die einen bestimmten Carry-Wert benötigt.

LAHF lädt das unterste Byte des `rflags`-Registers (dieses enthält CF, PF, AF, ZF und SF) in das `ah`-Register, SAHF setzt dieses Byte auf den Wert in `ah`.

STI und CLI setzen und löschen das *Interrupt-Flag*. Wenn dieses gelöscht ist, werden keine externen Interrupts behandelt. Darauf wird in diesem Handbuch nicht weiter eingegangen, es ist vor allem für die Systemprogrammierung relevant.

```
pushf; pushfd; pushfq
popf; popfd; popfq
```

```
clc   # clear carry
cnc   # complement carry
stc   # set carry
lahf  # load status flags into ah
sahf  # store ah into flags
sti   # set interrupt flag
cli   # clear interrupt flag
```

6.9 No Operation

Der NOP-Befehl tut nichts (außer Platz zu verbrauchen und den Befehlszeiger zu erhöhen).

```
nop
```

6.10 Befehlssatzübersicht

Die folgende Abbildung enthält eine Übersicht über die wichtigsten Ganzzahlbefehle der AMD64-Architektur, ihre Semantik und Argumente. Abkürzungen: R. . . Register, I. . . Immediate-Wert, M. . . Memory, D. . . Displacement-Wert bzw. Sprungziel. X:Y steht für einen doppeltlangen Wert, wobei X die höherwertigen und Y die niederwertigen bits stellt.

Abkürzung	**Argumente**	**Bedeutung**	
`adc` *src, src_dest*	R, R/M R/M, R I, R/M	add with carry	*CF,src_dest = src_dest + src +* ***CF***
`adcx` *src, src_dest*	R/M, R	add with carry	*CF,src_dest = src_dest + src +* ***CF***
`add` *src, src_dest*	R, R/M R/M, R I, R/M	arithmetic add	*CF,src_dest = src_dest + src*
`adox` *src, src_dest*	R/M, R	add with carry	*OF,src_dest = src_dest + src +* ***OF***
`and` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise and	*src_dest = src_dest and src*
`bt` *index, src*	R, R/M I, R/M	bit test	*CF = src[index]*
`btc` *index, src*	R, R/M I, R/M	bit test and complement	*CF = src[index]* *src[index] = not(src[index])*
`btr` *index, src*	R, R/M I, R/M	bit test and reset	*CF = src[index]* *src[index] = 0*
`bts` *index, src*	R, R/M I, R/M	bit test and set	*CF = src[index]* *src[index] = 1*
`clc`		clear carry	*CF = 0*
`cli`		clear interrupt flag	
`cmovCC` *src, dest*	R/M, R	conditional move	***if CC then*** *dest = src*
`cmp` *src1, src2*	R, R/M R/M, R I, R/M	compare	*src2 - src1 (setzt nut Flags)*
`cnc`		complement carry	*CF = not(CF)*
`dec` *src_dest*	R/M	decrement	*src_dest = src_dest - 1*
`div` *divisor*	R/M	divide	`rax` *=* `rdx:rax`*/divisor,* `rdx` *= Rest (und Varianten)*

Abkürzung	Argumente	Bedeutung	
`enter` *size, depth*	I, I	enter function, erzeugt Activation Record	
`idiv` *divisor*	R/M	signed divide	*`rax` = `rdx:rax`/divisor, `rdx` = Rest* *(und Varianten)*
`imul` *factor*	R/M	signed multiply	*`rdx:rax` = `rax` * factor* *(und Varianten)*
`imul` *factor, src_dest*	R/M, R	signed multiply	*src_dest = src_dest * factor*
`imul` *factor, src, dest*	I, R/M, R	signed multiply	*dest = src * factor*
`inc` *src_dest*	R/M	increment	*src_dest = src_dest + 1*
`jCC` *label*	D	conditional jump	***if CC then*** *jump to label*
`jcxz` *label*	D	jump if cx zero	***if `cx` == 0 then*** *jump to label*
`jecxz` *label*	D	jump if ecx zero	***if `ecx` == 0 then*** *jump to label*
`jmp` *label*	D	jump relative	*jump to label*
`jmp*` *address*	R/M	jump indirect	*jump to address*
`jrcxz` *label*	D	jump if rcx zero	***if `rcx` == 0 then*** *jump to label*
`lahf`		load status flags into `ah`	*`ah` = low_byte_of(`flags`)*
`lea` *src, dest*	R/M, R	load effective address	*dest = address_of(src)*
`leave`		leave function, entfernt einen Activation Record	
`loop` *label*	D	loop if cx (und Varianten)	*`cx` = `cx` - 1* *if `cx` != 0 then* *jump to label* *(und Varianten)*
`loope` *label*	D	loop if cx and equal (und Varianten)	*`cx` = `cx` - 1* *if `cx` != 0 and ZF then* *jump to label* *(und Varianten)*
`loopne` *label*	D	loop if cx and not equal (und Varianten)	*`cx` = `cx` - 1* *if `cx` != 0 and not(ZF) then* *jump to label* *(und Varianten)*
`loopnz` *label*	D	loop if cx and not zero (und Varianten)	*`cx` = `cx` - 1* *if `cx` != 0 and not(ZF) then* *jump to label* *(und Varianten)*
`loopz` *label*	D	loop if cx and zero (und Varianten)	*`cx` = `cx` - 1* *if `cx` != 0 and ZF then* *jump to label* *(und Varianten)*
`mov` *src, dest*	R, R/M R/M, R I, R/M	move	*dest = src*
`movsx` *src, dest*	R/M, R	move with sign extension	*dest = sign_extend(src)*
`movzx` *src, dest*	R/M, R	move with zero extension	*dest = zero_extend(src)*
`mul` *factor*	R/M	multiply	*`rdx:rax` = `rax` * factor* *(und Varianten)*
`nop`		no operation	
`not` *src_dest*	R/M	bitwise not	*src_dest = not(src_dest)*
`or` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise or	*src_dest = src_dest or src*
`pop` *dest*	R/M	pop from stack	
`popf`		pop flags from stack	*pop value of `flags` register*
`popfd`		pop flags from stack	*pop value of `eflags` register*
`popfq`		pop flags from stack	*pop value of `rflags` register*

Abkürzung	**Argumente**	**Bedeutung**	
`push` *src*	R/M	push onto stack	
`pushf`		push flags onto stack	*push value of* `flags` *register*
`pushfd`		push flags onto stack	*push value of* `eflags` *register*
`pushfq`		push flags onto stack	*push value of* `rflags` *register*
`rcl` *count, src_dest*	I, R/M `%cl`, R/M	rotate through carry left	*src_dest = rol(src_dest, count,* ***CF****)*
`rcr` *count, src_dest*	I, R/M `%cl`, R/M	rotate through carry right	*src_dest = ror(src_dest, count,* ***CF****)*
`rol` *count, src_dest*	I, R/M `%cl`, R/M	rotate left	*src_dest = rol(src_dest, count)*
`ror` *count, src_dest*	I, R/M `%cl`, R/M	rotate right	*src_dest = ror(src_dest, count)*
`sahf`		store ah into flags	*low_byte_of(*`flags`*) =* `ah`
`sal` *count, src_dest*	I, R/M `%cl`, R/M	shift arithmetic left	*src_dest = src_dest << count*
`sar` *count, src_dest*	I, R/M `%cl`, R/M	shift arithmetic right (signed)	*src_dest = src_dest >> count* *(signed)*
`sbb` *src, src_dest*	R, R/M R/M, R I, R/M	sub with borrow	*src_dest = src_dest - src* **- *CF***
`setCC` *dest*	R/M	set to flag value	*dest = CC*
`shl` *count, src_dest*	I, R/M `%cl`, R/M	shift left	*src_dest = src_dest << count*
`shld` *count, src, src_dest*	I, R, R/M `%cl`, R, R/M	shift left double	*src_dest = src_dest:src << count*
`shr` *count, src_dest*	I, R/M `%cl`, R/M	shift right (unsigned)	*src_dest = src_dest >> count* *(unsigned)*
`shrd` *count, src, src_dest*	I, R, R/M `%cl`, R, R/M	shift right double (unsigned)	*src_dest = src:src_dest >> count* *(unsigned)*
`stc`		set carry	*CF = 1*
`sti`		set interrupt flag	
`sub` *src, src_dest*	R, R/M R/M, R I, R/M	arithmetic subtract	*src_dest = src_dest - src*
`test` *src1, src2*	R, R/M R/M, R I, R/M	test	*src2 and src1 (setzt nur Flags)*
`xor` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise xor	*src_dest = src_dest xor src*

Tabelle 1: Wichtige Ganzzahlinstruktionen der AMD64-Architektur.

7 AVX-512 SIMD-Befehle

SIMD-Befehle dienen zur gleichzeitigen Bearbeitung eines ganzen Vektors von Operanden mit einem einzigen Befehl, wie es zum Beispiel für wissenschaftliche und Multimedia-Anwendungen sinnvoll sein kann. Dieses Handbuch stellt einige Ganzzahlbefehle aus diesen Befehlssatzerweiterungen kurz vor, die auf den in Abschnitt 3.3 vorgestellten Registern operieren.

Die Syntax eines AVX-512 Befehls kann man anhand folgender Beispiele demonstrieren:

Befehl	Operandengröße	Operanden	Anmerkung
`mov`*s*	64/32 Bit	$G/M \to X$ oder $X \to G/M$	Es werden nur 64/32 Bits übertragen, bei einem `xmm`-Register als Ziel werden die restlichen Bits auf 0 gesetzt.
`vmovdqu`*b*	SIMD	$M \to Z$, $Z \to M$ oder $Z \to Z$	Übertragung in voller Breite des Registers, maskiert
`kmov`*s*	8/16/32/64 Bit	$G/M \to K$, $K \to G/M$ oder $K \to K$	Übertragung von/zu einem Maskenregister

Abbildung 7: Move-Befehle für SIMD- und Maskenregister

```
vpsubd %zmm4,           %zmm5, %zmm6
vpsubd %zmm4,           %zmm5, %zmm6{%k4}
vpsubd %zmm4,           %zmm5, %zmm4{%k1}{z}
vpsubd (%rsi),          %zmm5, %zmm6{%k4}
vpsubd (%rsi){1to16}, %zmm5, %zmm6{%k4}
```

Bei all diesen Befehlen werden 16 32-bit-Werte parallel von den 16 32-Bit-Werten in `zmm5` subtrahiert. Bei den ersten drei Befehlen wird `zmm4` von `zmm5` subtrahiert, beim vorletzten die 16 32-bit-Werte, die an der Adresse in `esi` gespeichert sind, und im letzten Fall wird nur ein 32-bit-Wert von dieser Adresse gelesen, und derselbe Wert von allen 16 Werten in `zmm5` subtrahiert; je nach Breite der Einzelwerte und der SIMD-Breite muss man `1to2`, `1to4`, `1to8`, `1to16` verwenden; `1to32` und `1to64` funktionieren nicht.

Beim ersten Befehl werden alle 16 Resultatwerte in `zmm6` gespeichert, im zweiten, vierten und fünften nur die, wo das entsprechende Bit in `k4` gesetzt ist und die anderen Werte bleiben unverändert; im dritten Befehl werden ebenfalls die Werte, bei denen das entsprechende Bit in `k4` gesetzt ist, in `zmm6` gespeichert, und die anderen Werte werden auf 0 gesetzt.

In den Befehlsübersichten kommt oft der Platzhalter *s* vor, der für `b` (8 bit), `w` (16 bit), `d` (32-bit) oder `q` (64 bit), wobei nur die Suffixe für die unterstützten Breiten unterstützt werden.

G steht für ein General-Purpose-Register, *X* für `xmm`-Register, *Z* für ein `zmm/ymm/xmm`-Register, *K* für ein Maskenregister und *M* für eine Speicherstelle.

Die Reihenfolge der Operanden ist im Vergleich zur Intel-Syntax vollständig umgedreht: Zieloperand zuletzt, und z.B. im Fall einer Subtraktion $a - b$ ist b der erste Operand, a der zweite.

7.1 Datentransferbefehle

Diese Befehle transferieren Daten zwischen SIMD-Registern, Maskenregistern, General-Purpose-Registern und Speicherbereichen. Abbildung 7 beschreibt die verschiedenen Befehle.

b steht für `8`, `16`, `32`, `64` und beschreibt die Breite einer Spur in Bits (für die Maskierung; bei einem unmaskierten `vmovdqu` sind die Spuren egal, die Spurbreite muss aber trotzdem angegeben werden). Bei `KMOVB` und `KMOVW` muss aber trotzdem die 32-bit-Form des General-Purpose-Registers angegeben werden, z.B. `%eax`).

Befehl	Elementgröße	Operation
vpadd*s*	8/16/32/64 Bit	Addition (modulo-Arithmetik bei Überlauf)
vpadds*s*	8/16/32/64 Bit	Addition mit Sättigung (Overflows führen zum größten oder kleinsten darstellbaren Wert)
vpsub*s*	8/16/32/64 Bit	Subtraktion (modulo-Arithmetik)
vpsubs*s*	8/16 Bit	Subtraktion mit Sättigung
vpsubus*s*	8/16 Bit	Subtraktion unsigned mit Sättigung
vpmulhw	16 Bit	Multipliziert signed 16-Bit-Elemente miteinander und schreibt die oberen 16 Bit des 32-Bit-Ergebnisses in das Zielelement
vpmullw	16 Bit	Wie VPMULHW, schreibt aber die unteren 16 Bit
vpmulhuw	16 Bit	Wie VPMULHW, aber mit vorzeichenlosen (unsigned) Werten
vpsll*s*	16/32/64 Bit	Logisches Linksschieben
vpsrl*s*	16/32/64 Bit	Logisches Rechtsschieben
vpsra*s*	16/32 Bit	Arithmetisches Rechtsschieben, d.h. Rechtsschieben, wobei das Sign-Bit nachgeschoben wird
vpmaxsw vpminsw	16 Bit	Maximum/Minimum (signed)
vpmaxub vpminub	8 Bit	Maximum/Minimum (unsigned)

Abbildung 8: SIMD-Arithmetik- und Schiebebefehle

Bei den vmovdqu-Befehlen kann der Zieloperand eine Maske haben, und wenn ein Operand (auch der Quelloperand) eine Speicherstelle ist, wird nur auf die Speicherstellen zugegriffen, bei denen das entsprechende Maskenbit gesetzt ist. Das ist wichtig, wenn auf die Speicherstelle nicht zugegriffen werden darf und ein Zugriff eine *segmentation violation* (SIGSEGV) ergeben würde.

Beispiele für diese Befehle:

```
movq %xmm0, %rax
vmovdqu32 (%rsi), %zmm0{%k1}
kmovq %rax, %k1
```

7.2 Rechenbefehle

Für die Abwicklung der Grundrechenarten und Schiebeoperationen gibt es SIMD-Befehle, deren Funktionalität im Wesentlichen jener der in Abschnitt 6 beschriebenen arithmetischen und Shift-Befehle entspricht, nur dass die SIMD-Befehle auf den einzelnen Elementen zweier Vektoren arbeiten (und zwar wird immer ein Element des einen Vektors mit dem entsprechenden Element des anderen Vektors kombiniert) und die Ergebnisse wieder in die entsprechenden Elemente in einem Vektor speichern. Der erste Quell-Operand kann dabei im Speicher liegen, der zweite und das Ziel liegen immer in einem SIMD-Register. Beim Ziel kann noch eine Maske angegeben werden (optional mit {z}, um statt der ausmaskierten Ergebnisse 0 zu schreiben). Abbildung 8 listet die wichtigsten Befehle.

Befehl	Elementgröße	Operation
`vpcmp`*cs*	8/16/32/64 Bit	Vergleich von vorzeichenbehafteten Zahlen
`vpcmp`*c*`u`*s*	8/16/32/64 Bit	Vergleich von vorzeichenlosen Zahlen

Abbildung 9: SIMD-Vergleichsbefehle

Befehl	Größe	Operation
`kand`*s*	8/16/32/64 Bit	bitwise and
`kandn`*s*	8/16/32/64 Bit	bitwise and not
`knot`*s*	8/16/32/64 Bit	bitwise not
`kor`*s*	8/16/32/64 Bit	bitwise or
`kshiftl`*s*	8/16/32/64 Bit	shift by constant
`kshiftr`*s*	8/16/32/64 Bit	shift by constant
`kxnor`*s*	8/16/32/64 Bit	bitwise equivalence
`kxor`*s*	8/16/32/64 Bit	bitwise xor

Abbildung 10: Befehle für Maskenregister

7.3 Vergleichsbefehle

Die Vektorbefehle beeinflussen die in Abschnitt 3.1 vorgestellten Flags nicht. Die Vergleichsbefehle in Abbildung 9 schreiben das Resultat des Vergleichs in ein Maskenregister, und zwar ein Bit pro Spur. Eventuelle weitere Bits des Maskenregisters werden auf 0 gesetzt. c steht für die Art des Vergleichs: `EQ` ($=$), `LT` ($<$), `LE` ($\leq$), `FALSE` (immer 0), `NEQ` ($\neq$), `NLT` ($\geq$), `NLE` ($>$), `TRUE` (immer 1). Abgesehen vom anderen Zielregister verhalten sich diese Vergleichsbefehle wie Arithmetikbefehle. Beispiel eines möglichen Befehls:

```
vpcmpltud (%rsi){1to16}, %zmm2, %k1{%k2}
```

7.4 Befehle für Maskenregister

Zusätzlich gibt es noch Befehle, um Maskenregister zu verknüpfen, siehe Abbildung 10. Beispiele:

```
kandnq  %k2, %k3, %k4
knotq   %k6, %k7
kshiftlq  $3, %k3, %k4
```

8 Assemblerdirektiven

Der GNU-Assembler stellt eine Reihe von Assembleranweisungen zur Verügung, von denen einige hier beschrieben werden. Einige der Anweisungen sind nur deshalb in der Liste, weil Sie vom *GCC*-Compiler erzeugt werden. Genauere Informationen und eine vollständige Liste finden Sie in der Assembler-Dokumentation unter *::Pseudo Ops.*

`.align` *Zahl*	Sorgt dafür, dass die folgenden Befehle und Daten so angeordnet werden, dass ihre Adressen an bestimmten Speichergrenzen ausgerichtet (aligned) werden. Das Verhalten dieser Direktive ist unterschiedlich, je nachdem welches Binärformat generiert werden soll: Beim *ELF*-Format gibt das Argument das Alignment in Bytes an. `.align 8` bedeutet beispielsweise, dass die Adressen Vielfache von Acht sein und alle dazwischenliegenden unbenutzen Bytes auf Null (oder NOP-Befehle, je nach System und Speicherbereich) aufgefüllt werden sollen. Beim *a.out*-Format gibt die Zahl hingegen die Anzahl der unteren Adressbits an, die Null sein müssen. Damit wäre `.align 3` die der oberen Direktive entsprechende Anweisung.
`.ascii` *Text*	Speichert einen in doppelte Hochkommata eingeschlossenen Text ab.
`.asciiz` *Text*	Speichert einen in doppelte Hochkommata eingeschlossenen Text ab und schließt ihn mit Null ab.
`.byte` *expr [, expr]**	Speichert die auf 8 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Hinter dem Ausdruck kann noch ein durch einen Doppelpunkt getrennter Wiederholungsfaktor stehen.
`.comm` *name, expr*	Reserviert einen Speicherbereich mit mindestens *expr* Bytes unter dem Namen *name*. Der Linker legt alle Common-Blöcke mit dem selben Namen übereinander.
`.data`	Alle nachfolgenden Daten werden in der `.data`-Sektion angelegt. Die `.text`- und `.data`-Sektionen werden häufig auch als „Segmente“ bezeichnet, wobei diese nichts mit der von x86- und AMD64-Architekturen unterstützten Speichersegmentierung zu tun haben.
`.double` *expr [, expr]**	Speichert die nachfolgenden Ausdrücke als 64-Bit-Gleitkommazahlen aufeinanderfolgend ab. Siehe `.byte`.
`.endr`	Das Ende eines Repeat-Blockes. Siehe `.rpt`.
`.err`	Wird vom Übersetzer verwendet. Beendet den Assembler.
`.extern` *name [size]*	Definiert ein globales, externes Symbol mit dem Namen *name*. Der optionale Parameter *size* gibt die Größe in Bytes an.
`.file` *number string*	Wird vom Compiler verwendet. Ordnet eine Dateinummer dem Dateinamen zu.
`.float` *expr [, expr]**	Speichert die nachfolgenden Ausdrücke als 32-Bit-Gleitkommazahlen aufeinanderfolgend ab. Siehe `.byte`.
`.globl` *name*	Deklariert *name* als externes Symbol. Falls *name* anderswo im Programm definiert wird, wird das Symbol vom Linker exportiert, sonst wird es importiert.
`.ident`	GAS ignoriert diese Direktive.
`.lcomm` *name, size*	Für das Symbol *name* werden in der `bss`-Sektion *size* Bytes Speicherplatz reserviert.
`.long` *expr [, expr]**	Speichert die auf 32 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.
`.quad` *expr [, expr]**	Speichert 64 Bit große Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.
`.rept` *expr*	Wiederholt alle Befehle die zwischen `.rept` und `.endr` stehen *expr* mal. Es dürfen keine Label in diesen Befehlen vorkommen. `.rept`-Anweisungen dürfen nicht geschachtelt werden.
`.section` *name*	Der folgende Code wird in die angegebene Sektion assembliert. Sektionen

	werden in diesem Handbuch nicht genauer beschrieben.
`.size` *name, expr*	Legt die Größe des Symbols *name* in Bytes fest.
`.space` *expr*	Füllt *expr* aufeinanderfolgende Bytes mit Null.
`.string` *string*	Definiert einen String.
`.struct` *expr*	Ermöglicht die Definition von Datenstrukturen. In nachfolgenden Anweisungen wie `.word` oder `.byte` vorkommende Label erhalten einen Wert relativ zur `.struct`-Anweisung plus *expr*.
`.text`	Alle nachfolgenden Daten werden in der `.text` Sektion angelegt. Die `.text`- und `.data`-Sektionen werden häufig auch als „Segmente“ bezeichnet, wobei diese nichts mit der von x86- und AMD64-Architekturen unterstützten Speichersegmentierung zu tun haben.
`.type` *name, typedescr*	Spezifiziert den Typen eines Symbols als Funktion oder Objekt.
`.uleb128` *expr [, expr]**	Speichert Werte im kompakten *Unsigned Little Endian Base 128*-Format ab.
`.version` *string*	Definiert Versionsinformation.
`.word` *expr [, expr]**	Speichert die auf 16 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.
name `=` *expr*	Weist dem Symbol *name* den Wert des Ausdrucks *expr* zu.
name `=` *register*	Das Register *register* erhält den Namen *name*.