Kapitel 2

Computerarchitektur und Assembler

Viele Compiler erzeugen (oft im Zusammenspiel mit Assembler und Linker) ausführbaren Maschinencode für eine bestimmte Computerarchitektur. Dieses Kapitel erklärt Computerarchitektur aus Sicht des Compilers bzw. eines User-level-Programms in Maschinensprache.

2.1 Speicher

Daten jedweder Art liegen im Speicher. Der Speicher besteht aus einer Menge von 8-bit-Bytes; jedes dieser Bytes hat eine eigene Adresse (eine 32-bit- oder 64-bit-Zahl), über die auf genau dieses Byte zugegriffen werden kann. Die Bytes des Speichers liegen nebeneinander zusammenhängend im Addressraum (siehe Abb. 2.1).

Der Prozessor kann auf einzelne Bytes zugreifen, aber auch auf Gruppen von zwei, vier, oder acht Bytes. Allerdings muss die Addresse dabei durch 2, 4, bzw. 8 teilbar sein (Ausrichtung, alignment), da der Zugriff sonst eine Exception auslöst oder zumindest viel langsamer durchgeführt wird als ein ausgerichteter Zugriff (je nach Architektur).

Was der Inhalt des Speichers bedeutet, wird nicht mitgespeichert, sondern ist allein eine Frage der Interpretation durch das Programm; häufige (und von vielen Architekturen direkt unterstützte) Interpretationen sind:

Bytes	Interpretation
1	Zeichen, ganze Zahl im Bereich $-128...127$, natürliche Zahl < 256, 1 Flag, 8 Flags, Teil eines IA32-Maschinenbefehls
2	UTF-16-Zeichen, ganze Zahl im Bereich $-2^{15}...2^{15}-1$, natürliche Zahl $< 2^{16}$
4	ganze Zahl im Bereich $-2^{31}...2^{31}-1$, natürliche Zahl $< 2^{32}$, 32-bit-Adresse, RISC-Maschinenbefehl, IEEE single-precision Gleitkommazahl
8	ganze Zahl im Bereich $-2^{63}...2^{63}-1$, natürliche Zahl $< 2^{63}$, 64-bit-Adresse, IEEE double-precision Gleitkommazahl

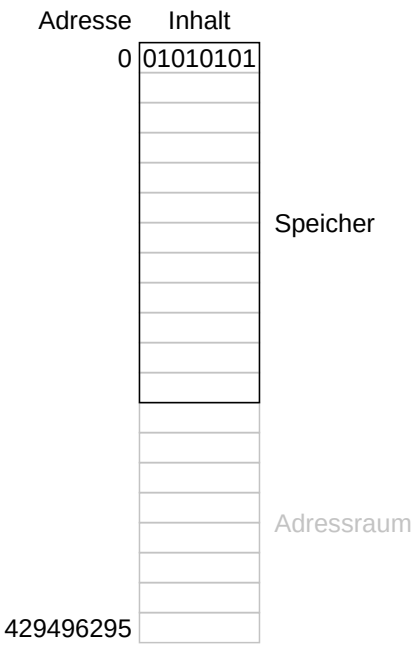

Abbildung 2.1: Adressraum und Speicher auf einer 32-bit-Maschine

So kann man zum Beispiel das Byte 11011111 als Zeichen "ß" (im Zeichensatz ISO-8859-1), als ganze Zahl -33 (in der Zweierkomplementdarstellung) oder als natürliche Zahl 223 interpretieren; andere Interpretationen durch die Software sind auch möglich, oder das Byte könnte Teil eines Datums mit mehr als einem Byte sein.

2.2 Datenstrukturen

Bei Strukturen aus mehreren Feldern liegen die einzelnen Felder direkt hintereinander im Speicher (siehe Abb. 2.2); allerdings ist oft ein Zwischenraum (padding) nötig (in unserem Beispiel nach Feld c), damit das nächste Feld (in unserem Beispiel m) ausgerichtet ist. Die Adressen der einzelnen Felder berechnen sich durch addieren eines Offsets zur Anfangsadresse der Struktur.

Bei Arrays liegen die einzelnen Elemente hintereinander im Speicher (siehe Abb. 2.2). Die Adresse eines bestimmten Elements ist $a + s * i$, wobei a die Startadresse des Arrays ist, s die Größe eines Elements, und i der Index des Elements (angefangen mit 0).

Man kann diese Datenstrukturen miteinander verschachteln. Bei der Adressberechnung wird dabei zuerst die äussere Datenstruktur aufgelöst und dann sukzessive die inneren. In unserem Beispiel berechnen wir die Adresse von a[2].m, indem wir zuerst die Adresse von a[2] berechnen (Ergebnis: a+12), und dann den Offset 2 für m dazuzählen; Gesamtergebnis: a+14 (die Anfangsadresse a hängt von der konkreten

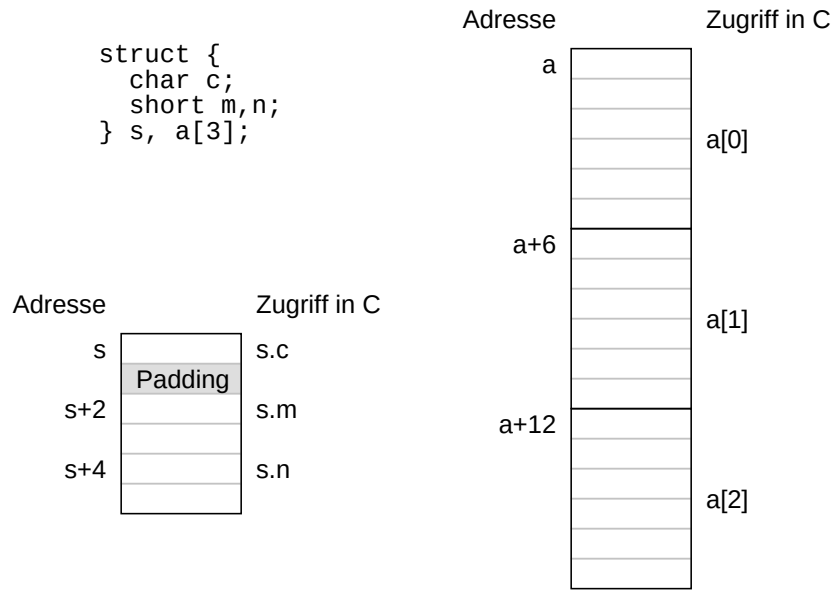

Abbildung 2.2: Repräsentation einer Struktur und eines Arrays (von Strukturen) im Speicher

Plattform (Maschine, Compiler, Betriebssystem) ab, daher geben wir sie hier nur symbolisch an.

Diese direkte Art der Verschachtelung funktioniert aber nur, wenn die Felder bzw. Elemente nicht größer sein können als der vorgesehene Platz. In den anderen Fällen (z.B. bei Objekten; Instanzen von Unterklassen können beliebig groß werden) legt man die eigentlichen Daten des Feldes/Elements woanders ab, und reserviert in der Datenstruktur nur Platz für die Adresse dieser Daten (alias Zeiger auf diese Daten). In C muß der Programmierer das explizit angeben, in vielen anderen Sprachen wird das aber automatisch gemacht.

2.3 Register

Neben dem Speicher kann der Prozessor auch noch auf Register zugreifen. Z.B. hat die Alpha-Architektur 32 Integerregister (`$0`–`$31`) und 32 Gleitkommaregister (`$f0`–`$f31`), jeweils 64 bit breit; Allerdings liefert `$31` immer 0 und `$f31` immer 0.0, und der Versuch, in diese Register zu schreiben, wird ignoriert.

Register dienen dazu, Operanden von Befehlen zwischenzuspeichern, denn viele Befehle arbeiten nur mit Registern, oder nicht in jeder Form mit Operanden im Speicher, und Registerzugriffe sind jedenfalls schneller als Speicherzugriffe. Eine weitere Verwendung für Register ist daher auch, häufig verwendete Daten zu speichern, um schnelle Zugriffe zu erlauben (siehe Abschnitt 9.3).

Ein wichtiger Unterschied zwischen Registern und dem Speicher ist, dass auf Register keine indirekten Zugriffe möglich sind.

2.4 Maschinencode und Assemblercode

Ausführbare Befehle der Architektur sind als bestimmte Bitmuster codiert; z.b. repräsentiert die Byte-Sequenz 00001101 00000100 00101101 01000000 einen bestimmten Befehl der Alpha-Architektur. Diese binäre Maschinensprache ist schwer zu lesen und zu schreiben; daher gibt es Disassembler und Assmbler, die zwischen Maschinensprache und einer lesbaren Repräsentation (Assemblersprache) hin- und her-übersetzen. Die Assembler-Repräsentation des obigen Befehls ist `addq $1, $13, $13`.

Assemblerbefehle bestehen aus einem Mnemonic (`addq`), das die Art der Operation angibt (addiere quadwords), und darauffolgenden Parametern (im Beispiel die Register `$1, $13, $13`). Im allgemeinem entspricht ein Assemblerbefehl einem Maschinenbefehl.

Zusätzlich zu den Befehlen verstehen Assembler auch noch Assembleranweisungen, um z.B. Daten in Binärform abzulegen oder für diverse Verwaltungsaufgaben, die die Programmierung erleichtern, die Wartbarkeit erhöhen.

2.5 Befehle

Verschiedene Architekturen haben verschiedene Befehle. Allerdings kann man die Befehle als Kombinationen von Grundoperationen ansehen, die zum Großteil für alle Architekturen gleich sind. In folgenden betrachten wir als Beispiel einige Befehle der Alpha-Architektur.

Rechenbefehle

`addq` ist ein Beispiel für die Klasse der arithmetischen/logischen Befehle. Es gibt diese Befehle in zwei Formen:

- `addq` *`reg1, reg2, reg3`*
- `addq` *`reg1, const, reg3`*

wobei das Ergebnis nach *reg3* geschrieben wird, und die anderen Operanden die zu addierenden Zahlen enthalten. Die Register müssen für `addq` und andere Integer-Befehle Integer-Register sein, *const* ist eine natürliche Zahl < 256.

Folgendes Programm enthält besteht aus arithmetischen/logischen Befehlen

```
subq $16, 1, $0  /* $0=$16-1 */
mulq $0, $16, $0 /* $0=$0*$16 */
srl  $0, 1, $0   /* $0 = $0>>1 */
```

Dieses Programm berechnet $x = y(y-1)/2$, wobei y in `$16` liegt und das Ergebnis x in `$0` gespeichert wird. Die Division wird in diesem Beispiel durch Verschieben um ein Bit nach rechts implementiert.

Speicherzugriffe

`ldq` ist ein Beispiel für einen Speicherzugriffsbefehl: er lädt ein 8-byte-Datum aus dem Speicher in ein Register. Dieser Befehl hat die Form

- `ldq` *reg1*, *const*`(`*reg2*`)`

Hierbei gibt *const*`(`*reg2*`)` die Adresse an: die Summe des Inhalts von *reg2* und *const*. Im Falle von `ldq` und anderen Ladebefehlen wird das Ergebnis nach *reg1* geschrieben, bei Schreibbefehlen wie z.B. *stq* wird der Wert in *reg1* genommen und in den Speicher geschrieben.

Beispiel:

```
mulq $17,6,$17     /* $17=$17*6   ; skaliere i */
addq $16, $17, $16 /* $16=$17+$16 ; a[i]       */
ldw  $0, 2($16)    /* $0=$16.m    ; a[i].m     */
addq $0, 1, $0     /* $0++        ; a[i].m+1   */
stw  $0, 2($16)    /* $16.m=$0    ; a[i].m=... */
```

Dieses Programm entspricht der C-Anweisung `a[i].m++` (mit den Datenstrukturdefinitionen aus Abb. 2.2), wobei die Anfangsadresse von `a` in `$16` liegt, und `i` in `$17` liegt.

Kontrollfluss

Um den Kontrollfluss steuern zu können, gibt es Verzeigungsbefehle (branches). Die unbedingte Verzweigung ist

- `br` *Ziel*

Diese Anweisung springt zum *Ziel*. Das Ziel wird in Maschinensprache als Offset vom Sprungbefehl angegeben; der Assembler nimmt einem allerdings die Berechnung dieses Offsets ab, man braucht nur einen Label angeben, zu dem gesprungen werden soll. Am Sprungziel muss man den Label natürlich definieren, mit *Ziel*`:`.

Zusätzlich zur unbedingten Verzweigung gibt es auch noch die bedingte Verzweigung, z.B.

- `beq` *reg*, *Ziel*

Diese Anweisung springt zum *Ziel*, wenn der Inhalt von *reg* gleich (`equal`) 0 ist.

Beispiel:

```
  beq  $16, ziel
  addq $0, 1, $0
ziel:
```

In diesem Beispiel wird `$0` erhöht, wenn `$16` ungleich 0 ist (entsprechend der C-Anweisung `if (a) b++;`).

Ein etwas größeres Beispiel:

```
/* C code: */
long vsum(long x[], long n)
{
  long s=0, *p=x;
  for (i=0; p<x+n; p++)
    s += p;
  return s;
}

/* Assembler-Code; x/p in $16, n in $17, s in $0, x+n in $1, temp in $2 */
  mov    0, $0        /* $0 = 0 */
  s8addq $17, $16, $1 /* $1 = $17*8+$16 */
loop:
  cmplt  $16, $1, $2  /* $2 = $16<$1 */
  beq    $2, exit     /* if ($s==0) goto exit */
  ldq    $2, 0($16)   /* $2 = *$16 */
  addq   $0, $2, $0   /* $0 = $0 + $2 */
  addq   $16, 8, $16  /* $16 = $16 + 8 */
  br loop             /* goto loop */
exit:
```

Dieses Beispiel Berechnet die Summe der Elemente des Arrays x. In diesem Beispiel sieht man folgende Unterschiede zwischen C und Assembler:

- Verwendung von Verzweigungsbefehlen statt Kontrollstrukturen mit Blockstruktur.
- In C wird bei Adressarithmetik automatisch skaliert, in Assembler muss der Programmierer das explizit machen (z.B. Übersetzung von `p++` nach `addq $16, 8, $16`).
- Verwendung von Registern statt Variablen.
- Zwischenergebnisse müssen in Assembler explizit in Registern abgelegt werden.

2.6 Funktionen

Damit der Assemblercode mit von Compilern erzeugtem Code zusammenarbeitet, ist es sinnvoll, den Assemblercode in Form von Funktionen zu schreiben, die sich an die Konvention zum Funktionsaufruf halten, an die sich auch die Compiler auf dieser Plattform halten.

Diese Konvention legt fest, welche Register von der aufgerufenen Funktion zerstört werden dürfen (der Inhalt dieser *caller-saved*-Register muss ggf. vom Aufrufer gesichert werden), und welche die aufgerufene Funktion unverändert lassen muss (bzw. abspeichern und danach wiederherstellen muss (*callee-saved*)). Sie legen auch fest, wie Argumente und Rückgabewerte übergeben werden, und ähnlich Details.

Die Aufrufkonvention ist so gestaltet, dass eine beliebige Funktion eine beliebige andere aufrufen kann, ohne viel über die andere Funktion zu wissen.

Auf der Alpha-Architektur werden z.B. die ersten sechs Integer-Argumente in den Registern `$16--$21` übergeben, das Funktionsergebnis in `$0`, und die Rücksprungadresse in `$26`. Der Rücksprung erfolgt mit dem Befehl `ret ($26)`.

Wir können also unser erstes Programmfragment durch Hinzufügen dieser Befehle und eines Einsprunglabels zu einer vollwertigen Funktion aufrüsten (die Register sind nicht ganz zufällig schon entsprechend gewählt):

```
  .globl f /* Funktionsnamen für den Linker exportieren */
f: /* Funktionsname */
  subq $16, 1, $0  /* $0=$0*$16 */
  mulq $0, $16, $0 /* $0=$0*$16 */
  srl  $0, 1, $0   /* $0 = $0>>1 */
  ret  ($26)
```

Diese Funktion kann zum Beispiel von C aus mit `n=f(10);` aufgerufen werden. Die `.globl`-Anweisung exportiert den Namen `f` so, dass der Linker ihn verwenden kann, und der Assembler-Code mit C-Code in einem anderen File zusammengebunden werden kann. Andere Namen sind nur innerhalb des Sourcefiles sichtbar, entsprechend der Speicherklasse `static` in C.

Die meisten bisher gezeigten Programmfragmente können so in vollwertige Funktionen umgewandelt werden. Weitere Feinheiten der Alpha-Aufrufkonvention werden im Übungsskriptum behandelt.

Neben `.globl` produzieren Compiler noch weitere Assembleranweisungen mit Informationen für den Debugger und andere Werkzeuge.

2.7 Was dieses Kapitel verschweigt

Dieses Kapitel kann natürlich nur eine stark vereinfachte Darstellung von Computerarchitektur geben, für eine fundierte Darstellung empfiehlt es sich, ein gutes Lehrbuch zu lesen, z.B. [HP90]. Einige der ausgelassenen Themen:

Die Architektur ist die Spezifikation für das Interface zwischen Maschinenprogramm und Prozessor. Daneben gibt es noch die *Mikroarchitektur*, die die Ausführungszeit bestimmt; hierunter fallen Konzepte wie z.B. Caches, Pipelines, und superskalare Ausführung, die für das Programm normalerweise nicht sichtbar sind. Für eine Architektur (z.B. IA32) gibt es oft viele Mikroarchitekturen (z.B. 386, 486, P5, P6, K5, K6, K7, Netburst), und von den Mikroarchitekturen oft wieder verschiedene Subvarianten (z.B. vom P6: Pentium Pro–Pentium III).

Die virtuelle Speicherverwaltung ist ebenfalls für User-Level-Programme kaum sichtbar, allerdings auf Betriebssystemebene schon; sie wird daher zur Architektur gezählt, aber trotzdem in diesem Kapitel ignoriert.

Dieses Kapiel geht auch nicht auf die Architektur im I/O-Bereich ein (z.B. Platten, Graphikkarte, etc.); I/O steht User-Level-Programmen nur über Betriebssystemaufrufe zur VerfÜgung.

Die Darstellung des Speichers ist eine Vereinfachung; neben den hier dargestellten weit verbreiteteten byte-addressierten 32- und 64-bit-Maschinen mit flacher Addressierung gibt es noch andere.