Übung 5

Armin Kleinert und Ruth Höner zu Siederdissen

Aufgabe 1: Stack

1. *Was ist ein Stack im Allgemeinen? Was ist ein durch Hardware unterstützter Stack? Beschreiben Sie die grundlegende Funktionsweise und Möglichkeiten.*

Der Stack ist eine abstrakte Datenstruktur, die einen Stapel simulieren soll.
In der Hardware ist der Stack ein Teil des Speichers, meist im Hauptspeicher, der anders organisiert ist als der Rest des Speichers. Er ist nach dem LIFO- Prinzip organisiert, bei dem die Daten, die als letztes gespeichert wurden, als erstes wieder aus dem Speicher geholt werden müssen. Aus diesem Grund wird der Speicher mit einem Stapel verglichen. Ein Stack ist besonders praktisch, um Werte, die man später noch benötigt abzulegen, wenn man weiß das der Wert im Register in den nächsten Berechnungen verändert werden könnte.

Da diese Funktion sehr häufig verwendet wird, gibt es inzwischen meist ein spezielles Register, den Stack Pointer, das nur dafür gedacht ist die Adresse abzuspeichern, an der die letzten Daten, die im Stack gespeichert wurden, stehen. Die Befehle für den Stack wie push() und pop() nutzen diesen Stack Pointer für die Ausführung und der Stackpointer aktualisiert sich danach von selbst, um wieder auf den nächsten Eintrag zu zeigen. Dies wird als Hardwareunterstützung bezeichnet.

Der Stack wächst meist von höheren zu niedrigeren Adressen. Wenn Daten mit dem Befehl push() aus einem Register auf den Stack geschoben werden, verringert der Stack Pointer die Adresse um 1 und an diese Adresse werden die Daten im Stack gespeichert. Nun zeigt der Stack Pointer auf diese Daten, die quasi “oben” auf dem Stapel liegen und beim nächsten pop()-Befehl werden die Daten an dieser Adresse ausgelesen und in das gegebene Register geladen. Danach erhöht der Stack Pointer die Adresse wieder um 1, um auf die nächsten Daten im Stapel zu zeigen.

2. *Warum wird bei einem Unterfunktionsaufruf (call) der Inhalt des Program Counter auf dem Stack gesichert, bei einem Sprung (jmp) aber nicht?*

Bei einem Unterfunktionsaufruf hält das Programm an einer bestimmten Stelle an, ruft eine Funktion auf und wenn diese Funktion fertig durchlaufen ist, springt das Programm wieder an die Stelle, an der vorher angehalten wurde. Da innerhalb dieser Unterfunktion der Program Counter natürlich auf die Adressen im Unterprogramm zeigt und auch die Werte in den Statusregistern wahrscheinlich verändert werden, muss der Zustand wie er vor dem Aufruf war, gespeichert werden. So kann nach Beendigung der Unterfunktion der Inhalt wieder in den Program Counter geladen werden und dieser weiß, wo die Stelle war, ab der im Hauptprogramm weitergemacht werden soll.
Bei einem Jump wird keine eigene Funktion aufgerufen, sondern mehr der nächste Abschnitt im Programm. Es wird nicht benötigt, wieder an die Stelle zurückspringen zu können, wo der Jump-Befehl aufgerufen wurde. Aus diesem Grund muss der Inhalt des Program Counters nicht gespeichert werden.

3. *Globale Variablen erhalten vom C-Compiler feste Adressen im Hauptspeicher, sogenannte statische Speicherallokation. Funktionslokale Variablen werden auf dem Stack angelegt, dies wird automatische Speicherallokation genannt. Warum wird das so gemacht?*

Globale Variablen müssen meist während der kompletten Laufzeit des Programms gespeichert werden und erreichbar sein. Die lokalen Variablen einer Funktion werden meist nur so lang benötigt, wie die Funktion selbst läuft. Der Speicherplatz im Stack wird also beim Eintritt in die Funktion reserviert, die Variablen werden darin gespeichert und wenn das Ende der Funktion erreicht wurde, wird der Speicherplatz im Stack auch automatisch wieder freigemacht, indem der Stack Pointer verschoben wird. Dadurch, dass nur der Stack Pointer verschoben werden muss, statt alle Bytes auf 0 zu setzen, ist das "Freigeben" der lokalen Variablen sehr schnell. Zusätzlich hat die automatische Speicherallokation den Vorteil, dass die Reservierung und das Freigeben des Speichers nicht Aufgabe des Programmierers sind, denn Speicherverwaltung ist eine häufige Fehlerquelle bei der Programmierung.

4. *Was ist ein Stackframe? Warum sind Stackframes sinnvoll?*
   Erläutern Sie in diesem Zusammenhang die x86 Befehle ENTER und LEAVE. Was muss beachtet werden, wenn sowohl ENTER und LEAVE als auch PUSH und POP benutzt werden sollen?

Der Stack wird neben der Speicherung des Program Status Words auch für die Speicherung des lokalen Zustands eines Unterprogramms verwendet. Dafür wird der Stack in sogenannte "Stack Frames" aufgeteilt. Beim Aufruf eines Unterprogramms (Funktionen) wird ein Stack Frame reserviert und in ihm werden die lokalen Variablen, Parameter und Daten gespeichert, die in dem Unterprogramm erzeugt oder verwendet werden. Die Adresse des Stack Frames wird in einem Register gespeichert und die lokalen Daten können relativ zu dieser Adresse vom Unterprogramm adressiert werden. Das Einteilen des Stacks in Stack Frames ist sinnvoll, da bei einem Aufruf eines Unterprogramms in einem anderen Unterprogramm einfach im Stack der nächste Stack Frame reserviert werden kann, um darin den lokalen Zustand des Unter-Unterprogramms zu speichern. So bleibt der lokale Zustand des noch nicht beendeten ersten Unterprogramms erhalten, bis der Programmablauf des Unter-Unterprogramms fertig ist und im ersten Unterprogramm weiter läuft.

Der Befehl ENTER erzeugt einen neuen Stack Frame für eine Funktion. Er sollte normalerweise als erster Befehl in einer Funktion aufgerufen werden, um den Stack Frame für die Funktion zu erzeugen. Der LEAVE Befehl wird am Ende einer Funktion angewendet, um den Stack Frame wieder freizugeben.

Durch ENTER kann eine ganze Stack Frame generiert und mit LEAVE ganz freigegeben werden. Mit PUSH und POP kann man die Daten nur in "Inkrementen" (8, 16, 32 oder 64 Bit) ablegen oder abheben. Wenn man die 4 Befehle gleichzeitig verwendet, kann es vorkommen, dass man versehentlich die Stack Frame mit einem unvorsichtigen Aufruf von POP bearbeitet und der Stack Pointer danach auf falsche Daten zeigt und dies als Adresse falsch interpretiert.

Aufgabe 2: Fibonacci Zahlen

Erklären Sie die Zeitunterschiede sowohl zwischen den rekursiven und iterativen Funktionen als auch zwischen den C und Assembler Funktionen. Warum wird Rekursion überhaupt benötigt?

Die iterative Version verwendet Register und verzichtet auf den Stack. Das ist schneller, da die Register direkt in der CPU liegen, also direkt angesprochen werden können. Der Stack dagegen liegt im RAM, der über einen Bus angesprochen werden muss. Für jeden Rekursiven Aufruf muss die

aktuelle Stackframe (erklärt in Aufgabe 4) gespeichert (Teil von CALL) und später gelesen werden, sobald die aufgerufene Funktion den RET Befehl nutzt.

Die C-Version des iterativen Codes läuft bei uns genauso schnell wie der Assembler-Code. Der rekursive Code dagegen ist schneller als unserer. Der C-Compiler nutzt mit den Optionen -O1, -O2 und -O3 verschiedene non-volatile Register und hat damit weniger Zugriffe auf den Stack.

Der C-Compiler benutzt nur einen CALL statt zwei und generiert mehr Abschnitte, um Sonderfälle zu behandeln (z.B., wenn Inlining möglich ist).

Quellen

- Vorlesungsfolien + Tutorium
- https://www.sciencedirect.com/topics/engineering/storage-duration
- https://de.wikipedia.org/wiki/Aufrufstapel
- https://mudongliang.github.io/x86/html/file_module_x86_id_78.html
- https://stackoverflow.com/questions/3699283/what-is-stack-frame-in-assembly
- Rust Manual (Zur Erklärung, wieso der Stack für lokale Variablen geeignet ist)