Systemprogrammierung - Zusammenfassung

Patrick Pletscher

21. September 2004

1 Computerarchitektur

1.1 Ressourcen und Operationen

Register

Schnellste Speicherzellen, nur sehr wenige (8-32), beschränkte Grösse, viele Operationen. Können normalerweise 32 Bit speichern.

Speicher[zellen] (aka Memory)

Viele Zellen, aber meistens langsam. Die meisten Speicherzellen können 8 Bit speichern.

Speichereinheiten

Speichereinheiten enthalten Bits. Um sich den Inhalt besser vorstellen zu können, verwandelt man sie ins Hexadezimalsystem.

Operations

Die Operationen interpretieren den Inhalt der Speichereinheiten. Eine Speichereinheit kann z.B. Integers, Floats, Characters oder andere Typen enthalten. Die Operationen entscheiden wie man diese Werte benutzt.

1.2 Core Architektur

Das Memory ist über Bus(es) mit der Computation Unit verbunden, welche wiederum über Bus(es) mit den Registern verbunden ist.

Prozessor (oder CPU - central processing unit)

Enthält die Computation engine (= ALU - arithmetic and logical unit), die Register, das Memory interface und andere spezielle Register.

Instruktionen

Eine Operation kontrolliert die Ausführung einer Operation auf diesem Prozessor. Die Instruktion spezifiziert den Opcode (welche Operation? z.B. `add mul, load, mov`) und welches die Operanden sind. Diese Operationen sind natürlich auch binär. Um nicht binär programmieren zu müssen gibt es Assemblersprachen.

Assemblerinstruktionen

Assembler bieten meisten mehr Funktionalität als nur Übersetzung, z.B. symbolische Namensauflösung und das Definieren von Datensegmentnen (`A: .byte 32`).

Instruktionsspeicherung

Auf dem SPARC haben alle Instruktionen die selben Grösse von 32 Bit. Die Instruktionen sind im Memory gespeichert.

Instruktionssequenzierung

Grundlegende Abfolge:

- Hole eine Instruktion
- Dekodiere sie
- Lese die Operanden
- Führe die Operation aus
- Speichere die Resultate

Dafür wird eine Ressource benötigt, die die Adresse der aktuellen Instruktion speichert: Der PC (=program counter) welcher als Teil der "Grundlegenden Abfolge" inkrementiert wird.

Wir brauchen auch Instruktionen um den PC zu modifizieren (z.B. für If-then-else) sogenannte *Kontrollflussinstruktionen*.

Arbeiten mit Adressen

Wir brauchen symbolische Namen um Adressen mit einer symbolischen Adresse zu versehen. Dabei hilft der Assembler.

Zusammenfassung

Drei Arten von Instruktionen:

- Berechnungen - z.B. `add, sub, mul, or, xor`
- Datenbewegung - z.B. `load, store`
- Kontrollfluss - z.B. `goto, jump, branch, call, return`

1.3 Wie werden Programme ausgeführt

Unterteilung in "trusted code" (das Betriebssystem) und "untrusted code" (die Anwenderprogramme). Verschiedene Prozesse führen entweder OS oder Anwenderprogramme aus.

Ausführen des Programmes

- Zu einer bestimmten Zeit entscheidet das OS ein Programm auszuführen.
- Ein Prozess nimmt die Instruktionen, reserviert/ erstellt Platz (für Daten) und führt eine Instruktion nach der anderen aus. Dieses Programm "besitzt" nun die CPU.
- Diese Aktivität endet wenn
  - die letzte Instruktion dieses Programmes zum OS zurückkehrt (`return`).
  - das Programm eine illegale Operation ausführt.
  - ein Hardwareproblem detektiert wird.
  - das OS die CPU zurück möchte.

2 C Programmierung

2.1 Überblick

C Kompilierungs Modell

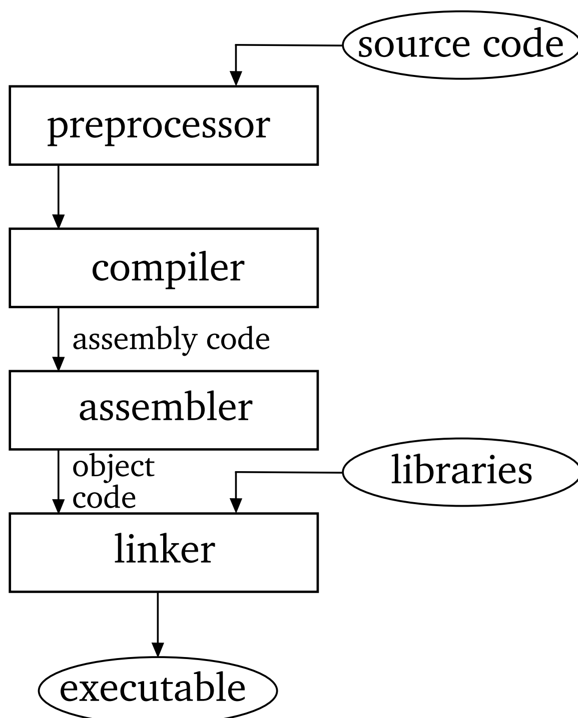

Abbildung 1: *Ablauf der C Kompilierung*

Struktur einen C Programms

Ein C Programm enthält die folgenden Elemente:

- Preprocessor Kommandos
- Typendefinitionen
- Funktionsprototypen (Deklaration von Funktionstypen und -argumenten)
- Variablen
- Funktionen

```
#include <stdio.h>
#define TIMES 10

double myfunction(float);

int main(void) {
 double wert;
 double pi = 3.14;
 printf("Multiply by 10\n");

 wert = myfunction(pi);

 printf("%d * %f = %f\n",
         TIMES, pi, wert);
}

double myfunction(double zahl){
 double count = 0;

 count = TIMES * zahl;

 return count;
}
```

Alle Programme müssen eine einzige `main()` Funktion enthalten. Alle Funktionen (auch `main()`) haben folgendes Format:

```
type function_name (parameters){
       local variables
       C Statements
}
```

Datentypen

C besitzt die folgenden grundlegenden Datentypen

C Type	Size (bytes)	Lower bound	Upper bound	Use
char	1	-	-	characters
unsigned char	1	0	255	small numbers
short int	2	-32768	+32767	integers
unsigned short int	2	0	65536	positive int
(long) int	4	-2^{31}	$-2^{31}-1$	large int
float	4	$-3.2\cdot 10^{\pm 38}$	$+3.2\cdot 10^{\pm 38}$	real numbers
double	8	$-1.7\cdot 10^{\pm 308}$	$+1.7\cdot 10^{\pm 308}$	large reals
void	0	-	-	no return value

- Die Grösse der Datentypen ist nicht standardisiert, kommt auf die Implementation an.
- Der Typ `void` wird für Funktionen benutzt, die keinen Wert oder einen Null-Pointer zurückliefern.
- Mit `#define`, kann man symbolische Konstanten einführen (z.B. `#define LIMIT 10`).

2.2 Sprachkonstrukte

Variablen und Konstanten

Spezielle Konstanten für die Benutzung mit Strings sind:

- `\n` Neue Linie
- `\t` Tabulator
- `\r` Wagenrücklauf (carriage return)
- `\b` Backspace
- `\"` Escape double quote
- `\0` end string

`sizeof()` ist eine Funktion die die Grösse einer gegebenen Variable in Bytes zurückliefert.

Variablen sollten vor der Benützung initialisiert werden, da sie sonst einen zufälligen Wert enthalten.

Typenumwandlung (casts)

Die Umwandlung kann explizit durch den Programmierer geschehen:

```
var_new_type = (new_type) var_old_type

// Example
int a;
float x = (float) a;
```

Die Umwandlung kann aber auch implizit durch den Compiler geschehen:

- `char` und `short` können zu `int` konvertiert werden
- `float` kann zu `double` konvertiert werden
- in einem Ausdruck, wenn ein Argument `double` ist, werden alle anderen zu `double` gecastet.
- in einem Ausdruck, wenn ein Argument `long` ist, werden alle anderen zu `long` gecastet.
- in einem Ausdruck, wenn ein Argument `unsigned` ist, werden alle anderen zu `unsigned` gecastet.

```
int a = 3;
float b = 5.0;
float c = a + b; // a wird in float umgewandelt
```

Scope

- Globale Variablen werden normalerweise vor der main Funktion deklariert, sie sind überall sichtbar, sollten aber vermieden werden.
- Lokale Variablen sind gültig und sichtbar in einem einzigen Block (ein Block ist eine Menge von C Statements umgeben von geschweiften Klammern {}). Wenn der Kontrollfluss ausserhalb eines Blockes ist, so existiert die Variable nicht mehr länger.
- Lokale Variablen werden erstellt (Platz im Memory wird alloziert) wenn der Kontrollfluss den Block erreicht in dem sie deklariert sind und zerstört (dealloziert) wenn der Kontrollfluss den Block verlässt.
- Das Erstellen und Zerstören von Variablen kann kontrolliert werden:
  - `extern`: Die Variable ist in einem anderen Modul definiert
  - `static`: Für lokale Variablen: Bleiben für das ganze Programm; für globale Variablen: beschränkt den Scope auf das aktuelle Modul
  - `register`: Versuche ein CPU Register für die Variable zu benutzen.
  - `auto`: Vorgabe für lokale Variablen.

Arithmetische Ausdrücke

Die Priorität der Operatoren ist mathematisch korrekt.

```
// Addition
x = 3 + 4;
// Subtraktion
x = 10 - 3;
// Multiplikation
x = 3 * 4;
// Division
x = 3 / 4;
 /* x = 9, if int x */
x = 73.0 /8.0;
 /* x=9.125, if float x */
// Modulo
x = 73 % 8;
```

Short-hand Operatoren

- Wenn `++` oder `--` als Prefix verwendet wird, so wird die Variable verändert, bevor sie für die Auswertung des Ausdruckes benutzt wird.

  ```
  a = 3;
  b = ++a + 3;
  /* b = 4 + 3 = 7 und a = 4 Seiteneffekt */
  ```

- Wenn `++` oder `--` als Postfix verwendet wird, so wird die Variable zuerst für die Auswertung des Ausdruckes benutzt und wird danach verändert.

  ```
  a = 3;
  b = a++ + 3;
  /* b = 3 + 3 = 6 und a = 4 Seiteneffekt */
  ```

- Beinahe alle Operatoren können mit = kombiniert werden

  ```
  a += b;  /* a = a + b */
  a *= b;  /* a = a * b */
  ```

Bit Operatoren

Die Bit-Operatoren `&` (AND), `^` (XOR) und `|` (OR) verändern die Bits nach der Standard Zwei-Werte Logik.

- Mit `&` kann man Bits auf 0 setzen. `a &= 0x00`
- Mit `^` kann man Bits umkehren. `a ^= 0xFF`
- Mit `|` kann man Bits auf 1 setzen. `a |= 0xFF`

Die Beispiele für `&` und `|` machen natürlich eigentlich keinen Sinn.

Shift Operatoren

- `<<` und `>>` werden benutzt um die Position von Bits in einem Byte oder Word zu manipulieren.
- Mit `a>>b` werden die Bits in Variable a um b Positionen nach rechts verschoben (die neuen Bits werden mit 0 gefüllt).
- Mit `a<<b` werden die Bits in Variable a um b Positionen nach links verschoben (die neuen Bits werden mit 0 gefüllt).

Vergleich und logische Operatoren

Die Vergleichsoperatoren liefern 1 oder 0 abhängig vom Resultat des Vergleiches.

- `<` kleiner als
- `>` grösser als
- `<=` kleinergleich als
- `>=` grössergleich als
- `==` gleich wie
- `!=` nicht gleich wie

`&&` und `||` sind die logischen AND und OR Operatoren

```
(a != b) && (c > d)
(a < b) || (c > d)
```

If Anweisung

```
if (expression)
  statement

if (expression)
  statement-if
else
  statement-else
```

Switch Anweisung

```
switch (selector) {
  case int_value1: statement1; break;
  case int_value2: statement2; break;
  case int_value3: statement3; break;
  default: statement4;
}
```

Falls `break` nach einem Statement weggelassen wird, so wird danach das nächste case ausgeführt.

Switch kann auch für `char` benutzt werden, da das ja eigentlich auch ein Integer ist.

for Anweisung

```
for(initialization; termination; increment) {
  statement
}
```

- Alle Elemente in der for Schleife sind optinal: `for(;;);`
- `break` kann benutzt werden um die Schleife zu beenden, ohne darauf zu warten, dass die terminierende Bedingung zu true ausgewertet wird.
- `continue` kann benutzt werden um die Ausführung des Inhalts der Schleife zu überspringen und die terminierende Bedingung neu zu überprüfen.

while Anweisung

```
while(expression){
  statement
}
```

`break` und `continue` können wie bei der for Schleife benutzt werden.

Do-while Anweisung

Die Do-while Schleife ist ähnlich wie die while Schleife, ausser dass die Schleife immer einmal durchlaufen wird und dass die Bedingung am Ende jeder Iteration überprüft wird.

```
do {
  statement
} while (expression)
```

`break` und `continue` können wie bei der for Schleife benutzt werden.

Ungerade Enden

`exit()` beendet die Ausführung des Programmes und übergibt dem OS einen Integer Fehlercode. Dabei steht z.B. 0 für keinen Fehler, 99 für crash.

2.3 Arrays

Die Nummerierung für die Elemente beginnt immer bei 0. Wenn das Array N Elemente besitzt, so ist das ist das letzte Element an Position $N-1$. Achtung: Der C-Compiler überprüft die Array-Grenzen nicht.

```
// Variablendeklaration
int data[5];

// Variablendeklaration mit Initialisierung
int data[5] = {1,2,3,4,5};

// Zugriff auf Elemente
data[3] = 32;
```

Mehrdimensionale Arrays

```
int a[2][2];

/* im Memory
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| a[0][0] | a[0][1] | a[1][0] | a[1][1] |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*/
```

Arrays und Strings

Strings sind Arrays von Buchstaben, terminiert durch den Null Buchstaben '\0'.

```
char str[6] = {'h','a','l','l','o','\0'};
char str[6] = "hallo";
```

Arrays und Pointers

Arrays und Pointers sind für den Compiler dasselbe. Der Array-Name ist ein Pointer zum Anfang des Arrays.

2.4 Pointers

Eine Variable hat einen Namen, eine Adresse, einen Typ und einen Wert:

- Der Name identifiziert die Variable für den Programmierer
- Die Adresse gibt an, wo im Hauptspeicher die Variable zu finden ist, d.h. dem Anfang des Speicherbereichs der für die Variable reserviert wurde.
- Der Typ gibt an, wie man die Daten im Speicher zu interpretieren hat und wie gross die Variable ist.
- Der Wert ergibt sich aus den aktuellen Daten der Variablen interpretiert nach dem Typ.

Pointers sind Sprachkonstrukte, die es dem Programmierer erlauben die Adresse von Variablen direkt zu manipulieren.

```
int* px;  /* px = pointer to an integer */

int x;

px = &x;  /* px bekommt die Adresse von x */

x = *px;  /* x bekommt den Inhalt davon */
          /* worauf px zeigt */

/* Der Nullpointer wird so definiert */
int* n_ptr = NULL;
```

Um einem Addressbereich der in einem Pointer `int* nummer_ptr` gespeichert z.B. einen Wert zuzuweisen benutzt man:

```
*nummer_ptr = 7;
```

Mehrere Pointer aufeinander:

```
int* *p_ptr_ptr;
p_ptr_ptr = &nummer_ptr;
/* p_ptr_ptr zeigt nun auf den Speicherbereich
   von nummer_ptr dort ist wiederum ein
   anderer Speicherbereich gespeichert wo dann
   ein wirklicher Wert steht
*/

*(*p_ptr_ptr) = 5;
```

Pointers, Arrays und Strings

Ein Array is in Wirklichkeit ein Pointer:

```
int a[10],y;
int* px;
px = a;     /* px points to a[0] */
px++;       /* px points to a[1] */
px = &a[4]; /* px points to a[4] */

px = a;
y = *(px+3); /* y gets the value */
             /* in a[3] */
```

Die Pointerarithmetik von C garantiert, dass wenn ein Pointer inkrementiert oder dekrementiert wird, er gemäss seinem Typ verändert wird. Zum Beispiel, falls `px` auf ein Array zeigt, so wird `px++` immer zum nächsten Element zeigen, unabhängig was der Typ des Arrays ist.

2.5 Structures

Structures erlauben es dem Programmierer komplexe Datentypen zu definieren.

```
struct Id_card {
  char name[100];
  char address[100];
  short int geburtsjahr;
  int telefon;
  short int semester;
} ethz, uniz;
```

Structures vom selben Typ können mit dem Operator `=` kopiert werden, sollten aber nicht mit `==` verglichen werden.

Um Zugriff auf die Elemente eines struct zu erhalten, geht man wie folgt vor:

```
ethz.name = "Hans Muster";
ethz.telefon = 1234;
```

Pointers können auch auf structs zeigen, man kann darauf mit `->` oder `*` zugreifen.

```
struct Id_card *pid;
pid = &ethz_student;
pid->name = "Hans Muster";
/* oder aber */
(*pid).name = "Hans Muster";
```

2.6 Funktionen

```
returntype function_name(def of parameters) {
  localvariables
  functioncode
}
```

main() ist auch eine Funktion

Für das Einlesen von Kommandozeilenargumenten kann man wie folgt vorgehen:

```
int main(int argc, char **argv){
  code
}
```

- `argc` enthält die Anzahl übergebener Argumente.
- `argv` Array der übergebenen Argumente.
- `argc` ist immer mindestens 1, da `argv[0]` der Programmname ist.

2.7 Dynamic Memory allocation

Die Funktionen für Memoryallozierung sind in `stdlib.h` zu finden. Typische Funktionen sind `malloc` und `free`.

```
typedef struct node {
  int x,z;
  struct node *next;
} NODE;

NODE *nptr;
if ((nptr = ((NODE)*)malloc(sizeof(NODE)))
 == NULL) {
  printf("No memory - bye bye"); exit(99);
}
```

`malloc` gibt einen Pointer zum allozierten Memory zurück. Der Pointer ist generisch (`void *`), darum ist es eine gute Idee den Pointer zum gewünschten Typ zu casten um Fehler zu vermeiden.

2.8 String library

```
#include <stdio.h>
#include <string.h>

void main() {
  char name1[12], name2[12], mixed[25];
  char title[20];

  strcpy(name1, "Rosalinda");
  strcpy(name2, "Zeke");
  strcpy(title, "This is the title");

  printf("  %s\n\n", title);
  printf("Name 1 is %s\n", name1);
  printf("Name 2 is %s\n", name2);

  if (strcmp(name1,name2) > 0)
    /* returns 1 if name1 > name 2*/
    strcpy(mixed, name1);
  else
    strcpy(mixed, name2);


  printf("The biggest name alphabetically is
   %s\n", mixed);

  strcpy(mixed, name1);
  strcpy(mixed, " ");
  strcpy(mixed, name2);
  printf("Both names are %s\n", mixed);
}

/*

will generate:

  This is the title

Name1 is Rosalinda
Name2 is Zeke
The biggest name alphabetically is Zeke
Both names are Rosalinda Zeke

*/
```

3 Computer Arithmetik und Informationsspeicherung

3.1 Umwandlung dezimal - binär

Um eine Zahl von dezimal d nach binär b umzuwandeln geht man wie folgt vor:

1. b := ""
2. Finde k so dass $2^{k+1} > d \geq 2^k$.
3. Dividiere d ganzzahlig durch 2^k und hänge die resultierende 1 oder 0 hinten an b heran.

4. Setze d gleich dem Rest der Division.
5. Setze $k = k - 1$ und falls $k \geq 0$ gehe zu 2.

3.2 Darstellung von Buchstaben

ASCII character set stellt Buchstaben mit 7 Bits dar, ursprünglich keine Unterstützung für spezielle Symbole wie z.B. ü. Heute im Normalfall 8 Bits. In Assembler können Buchstaben wie folgt benutzt werden:

```
mov   0140, %r1  ! as octal
mov   'a', %r1
mov   "a", %r1
```

3.3 Darstellung von Integer

- *unsigned*: Jedes Bit wird mit positivem Gewicht interpretiert.
- *two's complement für signed*: Erstes Bit negatives Gewicht, der Rest mit positivem Gewicht.
- binary to unsigned

$$b2u(x_{n-1}, \ldots, x_0) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

- binary to two's complement

$$b2t(x_{n-1}, \ldots, x_0) = -x_{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

- Es treten natürlich Probleme auf, wenn man eine unsigned Zahl als signed interpretiert.

C spezifiziert keine Darstellung für signed, die meisten Computer benutzen two's complement. Signed ist die Vorgabe `int x; /* signed */`.

3.4 Modulus Arithmetik

Computer benutzen of Modulus Arithmetik. Modulus Arithmetik berücksichtigt nur Zahlen in einem Bereich $[0, M)$. Falls wir in der Modulus Arithmetik die grösste mögliche Nummer $M - 1$ erreichen, gehen wir rundherum und starten wieder bei 0.

Wenn wir eine Operation von zwei Zahlen ausführen und das Resultat überschreitet den Modulus, so sagen wir ein *Overflow* sei aufgetreten.

Complement's

Complement's sind wichtig, da man eine Subtraktion in eine Addition umwandeln kann.

$$a - b = a + (r^n - 1 - b) + 1$$

Dabei ist $r^n - 1 - b + 1$ das Radixkomplement für Zahlen mit n Ziffern, welch eine System mit Basis r benutzen.

Für binäre Nummern kann man das two's complement ganz einfach finden, in dem man alle Bits umkehrt und 1 dazu addiert.

Status flags

Für signed:

- N (negativ) bit: Gesetzt falls MSB 1 ist.
- Z (zero) bit: Gesetzt falls alle Bits 0 sind.
- V (overflow) bit: Gesetzt falls
  - in $c = a - b$ a und b unterschiedliche Vorzeichen haben und c und b die gleichen haben.
  - in $c = a + b$ a und b das gleiche Vorzeichen haben, aber unterschiedlich von c.

Für unsigned: Overflow wird durch ein C (carry) bit detektiert, welches gesetzt wird, wenn Übertrag vom MSB auftritt. Bei der Subtraktion hingegen ist dieses Bit gesetzt falls das Resultat positiv ist und sonst 0.
C Bit wird gesetzt, falls Carry auftritt bei MSB bei Addition und wenn es nicht auftritt bei Subtraktion, also wenn Carry-Bit nicht gesetzt bei Subtraktion, so ist das Resultat positiv.

4 SPARC Assembler

4.1 Übersicht

Struktur eines Assembler Programmes

Assembler Programme sind linienorientiert. Der Assembler unterscheidet 4 verschiedene Typen von Linien:

- leere Linie
- Labeldefinitions Linie. Ein Identifier gefolgt von einem Kolon (":").
- Direktiven Linie. Eine optionale Labeldefinition gefolgt von einer Assemblerdirektive, gefolgt von den Argumenten der Direktive.
- Instruktions Linie. Eine optionale Labeldefinition gefolgt von einem Operatornamen, gefolgt von den Operanden.

Segmente und Anweisungen

Ein Assembler Programm enthält drei Segmente:

- data: Konstanten und Platz für Daten.
- text: Die Instruktionen.
- BSS (Block Storage Segment): Platz für dynamische Daten (aka "heap") und nicht initialisierte globale Variablen.

Eine Anweisung:

```
label: instruction
! instruction ist eine Maschineninstruktion
! oder eine synthetische Operation

label: directive
```

In Assembler kann man mit verschiedenen Systemen arbeiten:

- Hexadezimal `0x...`
- Oktal `0...`
- Dezimal

Beispiel

```
        .data
a:      .word 0x11

        .text
start:  ta 0
```

SPARC Register

SPARC ist ein load/store architecture". Also nur die load und store Anweisungen greifen auf das Memory zu, alle anderen Operanden müssen in Registern sein.

SPARC hat 32 Integer Register (jedes Register enthält 32 Bit), die `%r0 ... %r31` benannt sind.

Es gibt aber auch alternative Namen:

- Globale Register (`%g0 ... %g7`) – `%r0 ... %r7`
- Ausgabe Register (`%o0 ... %o7`) – `%r8 ... %r15`
- Lokale Register (`%l0 ... %l7`) – `%r16 ... %r23`
- Eingabe Register (`%i0 ... %i7`) – `%r24 ... %r31`

Spezielle Register:

- `%g0` (`%r0`) ist immer `0x0`.
- `%o6` (`%sp,%r14`) ist der Stackpointer.
- `%o7` (`%r15`) ist die Rücksprungadresse von der aufgerufenen Subroutine.
- `%i6` (`%fp,%r30`) ist der Framepointer.
- `%i7` (`%r31`) ist die Subroutine Rücksprungadresse.

Direktiven

- `.ascii string1, ..., stringn`
  z.Bsp.: `.ascii "Hello world\n"`
- `.global` Macht ein Label global
- `.byte val1, ..., valn`
  z.Bsp.: `.byte 0xf1`
- `.word val1, ..., valn`
- `.halfword val1, ..., valn`
- `.include "filename"`

4.2 Grundlegende Instruktionen

Programanfang

Der Anfang eines Programmes wird durch die Direktive `.text` angezeigt.

```
.text
start: ...
```

Programmende

Die Ausführung sollte man mit der Instruktion `ta` beenden. Dies ist eine Trapinstruktion die das OS aufruft mit einer Aufforderung ins Register `%g1`.

```
end:  ta 0
```

set

Laden von Konstanten in ein Register.

```
set  0x42, %r2
set  x, %r3
```

clear

Die `clr` Operation setzt einen bestimmten Speicherbereich auf 0. Diese Operation ist synthetisch.

```
clr  [%r3]
clr  a  /* a declared in .data segment */
```

load

Holt ein word aus dem Memory in ein Register.

```
ld  [%r2], %r3
```

store

Kopiert den Wert eines Registers (word) in den Speicher.

```
st  %r3, [%r2]
```

move

Kopiert den Inhalt eines Registers oder eine kleine Konstante in ein anderes Register. Diese Instruktion ist synthetisch.

```
mov  %r1, %r2
mov  1, %r2
```

add, sub

Brauchen drei Argumente: zwei Operanden und als drittes Argument das Zielregister für das Resultat. Die Operanden können entweder 2 Register sein oder ein Register und eine signed small constant (13 bit).

```
add %r3, %r4, %r5  ! %r5 = %r3 + %r4
sub %r3, 1, %r3    ! %r3 = %r3 - 1
```

signed und unsigned Multiplikation

Multiplikation von zwei 32-bit Werten, welche ein 64-bit Resultat liefern. Die most significant 32-bit werden im %y Register gespeichert und die restlichen 32-bit im Zielregister. Das zweite Argument kann eine kleine Konstante sein.

```
smul  %r1, %r2, %r3
umul  %r1, 10, %r3
```

null Operation

Die nop Operation überspringt einen Zyklus, in dem sie nichts tut.

```
nop
```

signed und unsigned Division

Die Ganzzahldivision teilt ein 64-bit Wert durch ein 32-bit Wert und speichert das Resultat im Zielregister. Das %y Register enthält die most significant 32-bit und das erste Register die restlichen 32-bit.

```
sdiv  %r1, %r2, %3  ! %r3 = {%y,%r1}/%r2
udiv  %r1, 10, %3   ! %r3 = {%y,%r1}/10
```

4.3 Instruktionspipelining

In einer Pipeline Architektur, wird jede Instruktion in verschiedene Teile aufgeteilt und seperat ausgeführt. Diese Teile sind: Fetch, Decode, Operand fetch, Execute, Store. Zu jedem Zeitpunkt werden gleichzeitig mehrere Instruktionen ausgeführt. 2 Instruktionen, die ausgeführt werden sind sichtbar: %pc, %npc.

Probleme mit Pipelines

Hazard: Wenn es einem Instruktionsstadium in der Pipeline unmöglich ist etwas während des Zyklus auszuführen. Hazards können in mehreren Situationen auftreten:

- Datenabhängigkeiten: Benötigte Daten sind noch nicht bereit.
- geteilte Ressourcen: Die Funktionseinheit die benötigt wird, wird im Moment benutzt.
- control branches: Wir wissen nicht welche Instruktion als nächstes ausgeführt wird.

Branching hazards

Die Pipeline wird automatisch mit der nächsten Instruktion gefüllt, welche sowieso ausgeführt wird (%pc,%npc). Aber wenn wir springen, ist die Instruktion die ausgeführt wurde ungültig. Die einfachste Lösung dafür ist es nach der branch Instruktion ein nop einzufügen.
Der branch delay slot kann benutzt werden um eine Instruktion auszuführen.

4.4 Branching in Assembler

Spezielles condition code (cc) Register, mit dem man bestimmte Charakteristiken des Resultats einer bestimmten Anweisung testen kann. Dieses spezielle Register hat 4 bits:

- Z (Zero): 1 wenn Resultat 0 war.
- N (Negative): 1 wenn Resultat negativ war.
- C (Carry): carry bit des MSB vom Resultat
- V (oVerflow): Zeigt an, dass ein Resultat zu gross war um in ein Register zu passen.

Nicht alle Operationen setzen diese Bits, spezielle Operationen werden benötigt:

```
addcc, subcc
smulcc, sdivcc
```

Im Folgenden ist Target ein label. Der Branch wird nur gemacht, wenn die Bedingung true ist, sonst geht die Ausführung nach der branch Instruktion weiter.

Operation	Assembler syntax	Branch condition
Branch always	ba target	1 (always)
Branch never	bn target	0 (never)
Branch not equal	bne target	not Z
Branch equal	be target	Z
Branch greater	bg target	not(Z or(N xor V))
Branch less or equal	ble target	Z or (N xor V)
Branch greater or equal	bge target	not(N xor V)
Branch less	bl target	N xor V
Branch greater, unsigned	bgu target	not (C or Z)
Branch less or equal, unsigned	bleu target	C or Z
Branch carry clear	bcc target	not C
Branch carry set	bcs target	C
Branch positive	bpos target	not N
Branch negeative	bneg target	N
Branch overflow clear	bvc target	not V
Branch overflow set	bvs target	V

4.5 While Schleife

Bedingungen können mit cmp geprüft werden. Dies ist eine synthetische Instruktion, und wird durch eine Subtraktion mit Übertrag bewerkstelligt.

```
cmp  register1, register2
cmp  register2, const.
```

Die folgende while Schlaufe in C:

```
main() {
  int a = 0;
  int b = 3;
  while (a <= 17) {
    a = a + b;
  }
}
```

Kann in Assembler z.Bsp. so ausssehen:

```
        .data
a:      .word 8
b:      .word 3
        .global _main

        .text
```

```
_main:  set     a, %r1
        ld      [%r1], %r2
        set     b, %r1
        ld      [%r1], %r3

loop:   cmp     %r3, 17
        ble, a  loop
        add     %r2, %r3, %r2

 store: set     a, %r1
        st      %r2, [%r1]

 end:   ta      0
```

`ble, a` ist dabei ein sogenannter annuled branch delay slot, diese Instruktion wird also ausgeführt, falls gesprungen wird und sonst nicht.

Annuled Branches

Die Anweisung nach einem annuled branch wird als annuliert markiert, falls der Branch nicht gemacht wird und dann wird mit der Instruktion in `%npc` fortgefahren.

4.6 do while Schlaufen

Gleich wie while Schlaufe, nur dass dort vor dem Loop noch einmal alle Instruktionen im Loop ausgeführt werden.

4.7 If-then else

```
if ((a+b) >= c) {
  a += b;
  c++;
else {
  a -= b;
  c--;
}
c += 10;
```

```
             ! a -> %r2, b -> %r3, c -> %r4
             add    %r2, %r3, %r6
             cmp    %r6, %r4
             bl,a   else             ! if
             sub    %r2, %r3, %r2    ! 1. in else
             add    %r2, %r3, %r2    ! 1. in if
             inc    %r4
             ba     store
             add    %r4, 10, %r4
else:        dec    %r4
             add    %r4, 10, %r4
store:       set    a, %r1
             st     %r2, [%r1]
end:         ta     0
```

4.8 switch

```
switch(i) {
  case 1:   i += 1;
            break;
  case 2:   i += 2;
            break;
  case 15:  i += 15;
  case 3:   i += 3;
            break;
  case 4:   i += 4;
  case 6:   i += 6;
            break;
  case 5:   i += 5;
            break;
  default   i--;
}
```

```
           .data
           .align  4
table:     .word   L1,L2,L3,L4,L5,L6,L7,L8,L9
           .word   L10,L11,L12,L13,L14,L15

           .text
           .align  4
start:     set     i, %r1
           ld      [%r1], %l0
           ld      [%r1], %o0     ! %o0 = i
           cmp     %o0, 1
           blu     default        ! too small
           cmp     %o0, 15        ! too large
           nop
           set     table, %o1     ! jump table
           sll     %o0, 2, %o0    ! %o0 x 4
           add     %o1, %o0, %o0
                    ! %o0 points to case in
                    ! table
           jmpl    %o0, %g0
           nop
L1:        ba      end
           add     %l0,1,%l0      ! i++
L2:        ba      end
           add     %l0,2,%l0      ! i+=2
L15:       add     %l0,15,%l0     !i+=15, no break
L3:        ba      end
           add     %l0,3,%l0      !i+=3
L4:        add     %l0,4,%l0      !i+=4, no break
L6:        ba      end
           add     %l0,6,%l0      !i+=6
L5:        ba      end
           add     %l0,5,%l0      !i+=5
L7:
L8:
L9:
L10:
L11:
L12:
L13:
L14:
default:   sub     %l0,1,%l0      !i--
end:       ta      0
```

5 Caching

5.1 Cache Architekturen

Direct-mapped cache : Die Memoryadresse bestimmt eindeutig die zugehörige Cache Linie.

Fully associative : Eine Memoryadresse kann irgendwo im Cache gespeichert werden. Es gibt keine im Konflikt stehenden Adressen. Sehr teuer, da man den ganzen Cache durchsuchen muss um eine Memoryadresse zu finden.

Set associative cache : Cache Linien unterteilt in Mengen, Memoryadressen zeigen auf die Mengen (mittlere Bits, nicht die untersten Bits). In einer Menge ist die Zuweisung völlig assoziativ. Zwei Memoryadressen stehen im Konflikt, wenn sie auf die gleiche Menge zeigen. Aber in der gleichen Menge gibt es verschiedene Positionen wo die Adressen hin können. Typische Mengengrössen sind 4.

6 Arithmetische Operationen in Assembler

6.1 Boolsche Operationen

Für Boolsche Ausdrucke gibt es folgende Ausdrücke:

```
! Benutzung wie folgt:
!b_op    r1, r2, dest_reg

and      r1 AND r2
andn     r1 AND ~r2
or       r1 OR r2
xor      r1 XOR r2
xnor     r1 XOR ~r2
orn      r1 OR ~r2
```

NOT kann wie folgt bewerkstelligt werden:

```
not      source_reg, dest_reg
xnor     source_reg, %g0, dest_reg
```

6.2 Test Operationen

```
! a -> %l1
! Verschiedene Moeglichkeiten fuer das
! Testen nach a > 0

cmp     %l1, 0
ble     next

! cmp ist synthetisch und wird
! uebersetzt zu

subcc   %l1, %g0, %g0
ble     next

! Alternativ kann man auch tst benutzen

tst     %l1
ble     next

! tst ist synthetisch und wird
! uebersetzt zu

orcc    %l1, %g0, %g0
```

6.3 Addition von Binärzahlen

Die Summe zweier Zahlen in binär Darstellung ist gleich einer XOR-Verknüpfung und der Übertrag kann mit einem AND berechnet werden.

```
add (int a, int b) {

  sum = a^b;

  while (carry = (a&b) << 1){
    a = sum;
    b = carry;
    sum = a^b;
  }

  return(sum);
}
```

6.4 Multiplikation von Binärzahlen

$$\underbrace{23}_{\text{Multiplicand}} \times \underbrace{32}_{\text{Multiplier}}$$

Grobe Abfolge (genaueres siehe Abbildungen im Skript):

1. Falls hinterstes Bit von Muliplier 0 so kopiere 0 zu Temp, sonst kopiere Multiplicand nach Temp.
2. Addition von Temp und bisherigem Produkt nach Produkt.
3. $\{Produkt, Multiplier\}$ um 1 nach rechts shiften.
4. Gehe wieder zu 1 falls noch nicht ganz Multiplier durchgeshiftet, mit jetztigen Werten in Produkt und Multiplier.

Bei der unsigned Multiplikation wird am Ende das two's complement vom Multiplicand gebildet und dann wie folgt vorgegangen: $\{Produkt, Multiplier\} + \{TWC, 0 \ldots 0\}$ wobei TWC das two's complement vom Multiplicand ist.

Division

siehe Skript

7 Stack und Datenstrukturen

7.1 Memory

32 Bit SPARC Architektur hat 4 Gigabytes Addressraum.

Memory aus Sicht von Assembler

Speicherdatentypen in Assembler sind byte, halfword, word und doubleword (in C meist: char, short, int, long). Manipulation dieser Positionen im Speicher durch load und store. Die Speicheradressen sind nach Standardlängen aligniert. Die Adressen sind also in der Form $n \cdot 2$ für half-words, $n \cdot 4$ für words und $n \cdot 8$ für doubles.

7.2 Stack

In der SPARC Architektur werden Anwendungsprogramme irgendwo oberhalb von der Position 0x2000 im Speicher geladen. Das Runtime-System stellt zusätzlichen Speicherbereich für die sogenannten automatischen Variablen (z.Bsp. lokale Variablen einer Funktion) zur Verfügung. Dieser Speicherbereich ist im oberen Bereich der Speicheradressen und ist als FILO organisiert → Stack. Der Stack dient auch zur Zwischenspeicherung von Variablen die sich bei einem Subroutinen-Aufruf in Registern befinden.
Der Stack wächst im Memory von oben nach unten, die Adresse der letzten benutzen Speicherposition ist im `%sp` (=`%o6`) gespeichert. Der Speicher oberhalb des `%sp` wird benutzt.
Der Stackpointer ist immer doubleword aligniert (teilbar durch 8).
Zusätzlicher Speicher auf dem Stack kann alloziert werden indem man den stack pointer zu einer tieferen Position im Speicher verschiebt. Der Platz der so alloziert wird, ist der Platz zwischen vorheriger Position und jetztiger Position.

```
sub %sp, 64, %sp
! provides additional 64 bytes of memory on stack.
```

7.3 Alignment

Alignment wird bewerkstelligt in dem man die 3 letzten Bits einer two's complement Binärnummer auf 0 setzt.

In Assembler wird dies mit einer AND Operation gemacht:

```
add  %sp, -94 & 0xfffffff8, %sp
! or
add  %sp, -94 & -8, %sp ! provides 96 bytes
```

7.4 Framepointer

Typische Subroutinen allozieren automatische Variablen wenn sie aufgeruft werden und darum wird der Stackpointer beim Aufruf der Subroutine erhöht. Wenn das passiert, dann wird der alte Wert im Framepointer (`%fp` oder `%i6`) gespeichert.

Eine spezielle Instruktion wird benutzt um den Stackpointer zu erhöhen und den alten Wert in den Framepointer zu speichern, dadurch wird Platz für die automatischen Variablen in den Registern und im Hauptspeicher reserviert.

```
save %sp, -64 - bytes_of_local_storage, %sp
! 64 bytes for register variables/contents
```

Beispiel für das Reservieren von 5 Variablen (word size):

```
save %sp, (-64 - (5*4)) & -8, %sp
```

Der Assembler wertet die arithmetischen Ausdrücke aus.

7.5 Zugriff auf den Stack (load)

Das Alignment muss berücksichtigt werden. Adressen werden relativ zum Stackpointer `%sp` oder dem Framepointer `%fp` berechnet. Double words werden in ein Paar von Registern geladen oder davon gespeichert.

Load Instruktionen sind:

`ldsb`	load signed byte (propagte sign to the left)
`ldub`	load unsigned byte
`ldsh`	load signed halfword (propagte sign to the left)
`lduh`	load unsigned halfword
`ld`	load word
`ldd`	load double, register number must be even and the load occurs on register n and n+1

Eine load Instruktion braucht zwei Zyklen um die Daten vom Memory zu holen. Das System verzögert automatisch die Pipeline, wenn die nächste Instruktion diese Daten benötigt.

7.6 Zugriff auf den Stack (store)

Store Instruktionen sind:

`stb`	store low byte of register (bits 0-7)
`sth`	store low two bytes of register
`st`	store register
`std`	store double, register number must be even, first four bytes from register n, the other 4 from register n+1

7.7 Lokale Variablen auf dem Stack

Variablen Offset und Alignment

Man benutzt meistens Symbole um die Offset-Konstanten zu definieren.

```
int a,b;
char ch;
short c;
int d;

! wird zu:

a_s = -4
b_s = -8
ch_s = -9
c_s = -12
d_s = -16

! [..]
ld [%fp + a_s], %l1
```

Die Offsets müssen den Alignment-Vorlagen genügen und sind immer relativ zum `%fp` adressiert, die Offsets für lokale Variablen sind folglich negativ.

lokale Variablen Werden relativ zum `%fp` adressiert.

Padding Zwischen den lokalen Variablen und den Funktionsparametern gibt es möglicherweise 4 Bytes Padding, da der `%sp` 8 Bytes aligniert sein muss.

Funktionsparameter Immer 4 Byte Bereiche, auch für `short` oder `char`.

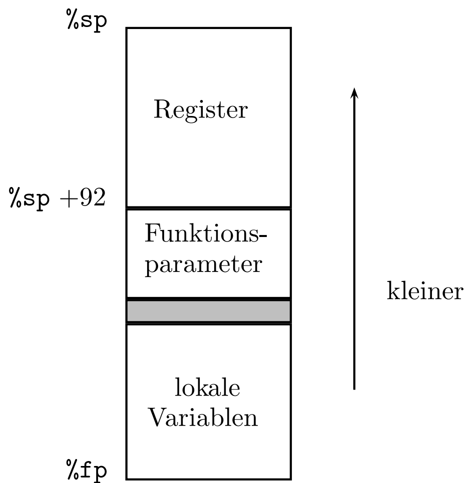

Abbildung 2: *Position von lokalen Variablen im Stackframe*

7.8 Structs

Structs werden als zusammenhängender Block gespeichert. Members sind auf ihre Grösse aligniert. Das Struct selbst ist auf die Grösse des grössten Members aligniert.

Ein Beispiel wäre zum Beispiel folgendes struct.

```
struct FOO {
   int a;
   float b;
   char c;
   double d;
   char i;
   short s;
}
```

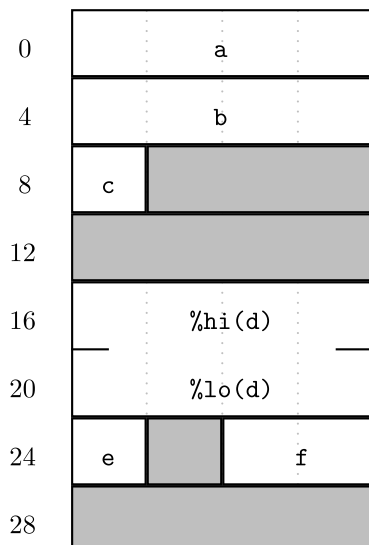

Abbildung 3: *Beispiel eines Structs im Speicher*

Die Grösse des struct ist folglich 40 Byte und sein Alignment 8 Byte. Für die Offsets erbigt sich:

```
st = -40

st_a = 0
st_b = 4
st_c = 8
st_d = 16
st_e = 24
st_f = 26
```

Wobei angenommen wird, dass das struct die erste lokale Variable ist (somit ergibt sich ein `st=-40`).

Auf Element `e` kann nun wie folgt zugegriffen werden:

```
ld [%fp + st + st_e], %o0
```

Structs Offsets sind positive Ganzzahlen, während Stackvariablen Offsets negativ sind. Die Structelemente liegen also in umgekehrter Reihenfolge auf dem Stack.

Erläuterungen

1. Struct wird als zusammenhängender Block gespeichert.
2. Elemente sind immer auf ihre Grösse aligniert, deshalb ergeben sich teilweise Paddings.
3. Alignment des Struct ist die Grösse seines grössten Members; falls das Struct eine Grösse hat, welche modulo seines Alignments ungleich Null ist, so wird am Ende des Struct ein Padding eingefügt, so dass es seinen Alignmentvorgaben entspricht.
4. Bei der Übersetzung eines C Programms in Assember werden Elemente in der Reihenfolge des C Codes immer an der kleinst möglichen Stelle gespeichert, welche den Alignmentvorgaben entspricht. Es wird aber immer an höherer Stelle als das vorhergehende gespeichert.

7.9 Arrays

Die Adresse eines Elementes eines Arrays wird durch einen Pointer auf die Basisadresse des Arrays plus ein Offset berechnet. Um in einem Array von words das Element `a[2]` zu laden benutzt man:

```
mov  2, %l0
sll  %l0, 2, %o0 ! multiply *4
add  %fp, %o0, %o0
ld   [%o0 + a_base], %o0
```

Hier ein Beispiel für ein Integer Array.

```
int array[5];
```

8 Subroutinen

Werden durch jump Instruktionen aufgerufen. Der Inhalt von Registern muss für einen Subroutinen Aufruf auf dem Stack gespeichert werden. Auch die Adresse der Aufrufinstruktion wird gespeichert, sie wird return address genannt. Diese

arr + 0	a[0]
arr + 4	a[1]
arr + 8	a[2]
arr + 12	a[3]
arr + 16	a[4]

Abbildung 4: *Beispiel eines Arrays im Speicher*

Adresse wird in %o7 bzw. %i7 in der Subroutine gespeichert. Aufrufe von Subroutinen sind verzögert, daher ist die aktuelle Rücksprungadresse %i7+8 (zwei 32bit Instruktionen nach dem Aufruf), hier in Subroutine.

8.1 Auruf von Subroutinen

Wenn der Name der Subroutine gegeben ist:

```
call <name>
```

Speichert den %pc in %o7 und hat einen delay slot.

Wenn die Adresse der Subroutine zur Laufzeit berechnet wird, dann wird der Aufruf wiefolgt gemacht:

```
jmpl <source_reg>, <dest_reg>
```

jmpl ruft die Funktion an der Adresse die im Register <source_reg> gespeichert ist auf. Der %pc wird im <dest_reg> gespeichert. Der return aus einer Subroutine wird auch mit dem jump & link Mechanismus gemacht:

```
jmpl %i7 + 8, %g0
! discards the %pc
```

Was äquivalent ist zu:

```
ret
```

8.2 Subroutinen und der Stack

```
save %sp, -64, %sp
```

- Speichert die Register %l0-%l7, %i0-%i7 auf dem Stack (in die zu Verfügung stehenden 64 Bytes).
- Kopiert die Register %o0-%o7 nach %i0-%i7.
- Führt eine Addition aus (meistens dafür gebraucht um mehr Platz auf dem Stack zu erhalten).

```
restore
```

- Kopiert die Register %o0-%o7 von %i0-%i7.
- Lädt den Inhalt von den Registern %l0-%l7, %i0-%i7 vom Stack.
- Führt eine Addition aus.

8.3 Typischer Subroutinen Aufruf

```
main:
...
call mysubr
nop             ! or something meaningful
...
mysubr:
save %sp, -112, %sp
...
ret
restore         ! fills delay slot of ret
```

8.4 Argumentübergabe

Die SPARC Architektur benutzt die Register %o0,..,%o5 zur Argumentübergabe an Subroutinen. Wenn mehr als sechs Argumente benötigt werden oder für grössere Datenstrukturen wird der Stack benutzt.

Stackframe

Im Stackframe wird gespeichert:

- lokale Variablen der Prozedur
- Übergabeparameter für callee's
- Struct pointer für callee's welche structs zurückgeben
- Register Window
- evtl. Padding

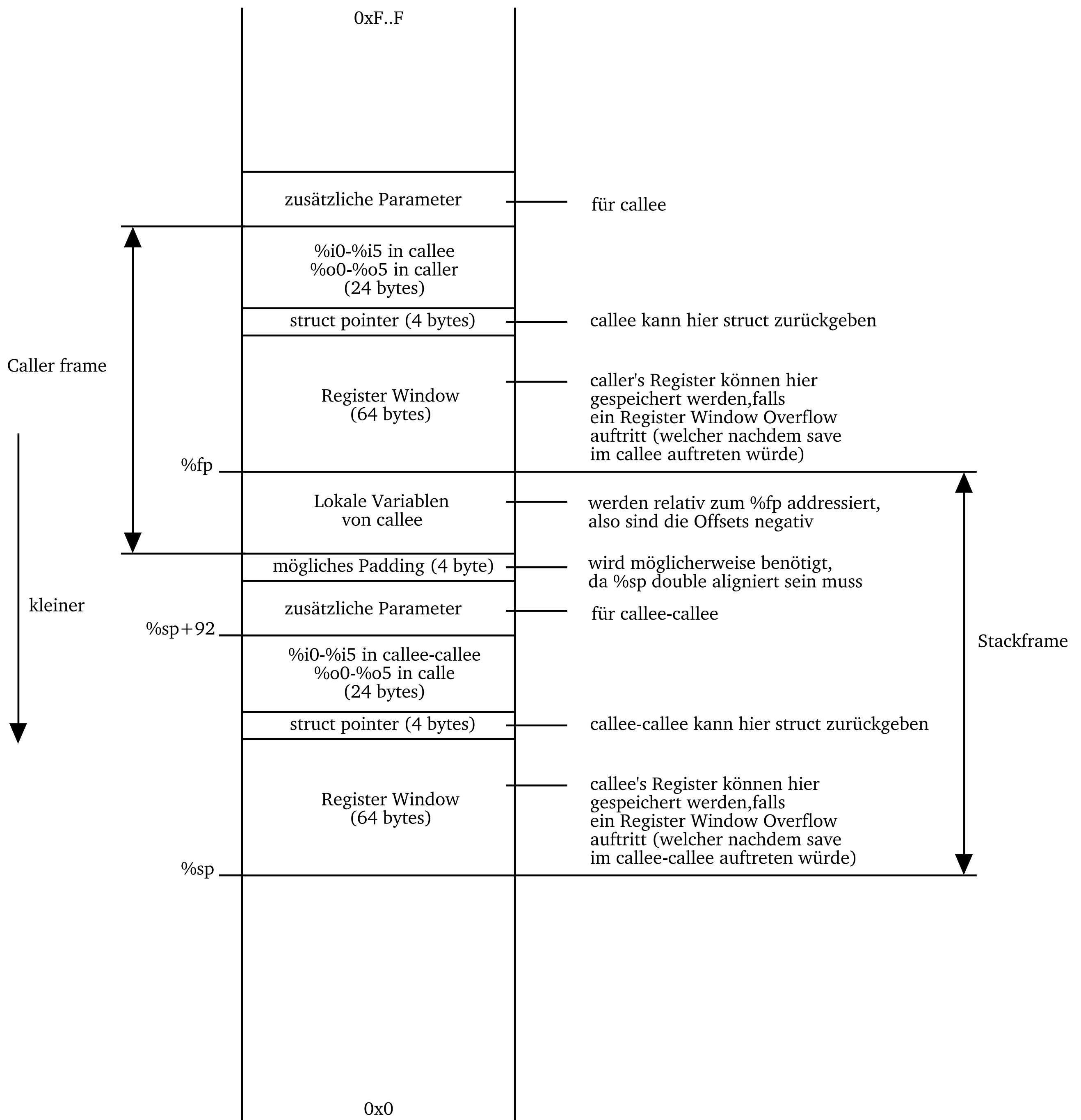

Abbildung 5: *Aufbau des Stackframe*

Es gilt zu beachten, dass Structs by value in lokalen Variablen gespeichert werden und eine Reference darauf übergeben wird.

Calling Conventions

- Parameter werden in den Registern `%o0-%o5` übergeben
- Falls Prozedur mehr als 6 Parameter hat, werden diese auf dem Stack (ab Adresse `%sp+92`) übergeben, ein Wort pro Parameter und jeweils so, dass sie den oberen Teil des Wortes ausfüllen.
- Spezialfälle
  - structs by value
  - long long int, double
  - Rückgabe von structs

structs by value

Problem: struct passt evtl. nicht in ein Register. Lösung: Intern werden structs immer by-reference übergeben, also mit einem Pointer auf die struct. Wenn also eine Prozedur einen value struct Parameter hat, wird im Stack eine lokale Kopie abgelegt, und ein Pointer auf diese übergeben.

long long int, double

8 Byte lange Basistypen werden in einen High- und einen Lowteil aufgeteilt, und nacheinander übergeben.

Beispiel:

```
void foo(long long int x){ .. }
```

Die hohen 32 Bits von x landen in `%o0`, die tiefen in `%o1`. Auch möglich, dass high bits in `%o5`, und low bits an der Adresse `%sp+92` landen, also zur Hälfte in einem Register, zur Hälfte auf dem Stack.

Beispiel für Stackframe und Parameterübergabe

```
struct s {
  int key;
  long long int data;
  char c;
}

void foo(i1,...,i6){
  short k;
  struct s st;
  .
  .
  bar(i1,...,i6,k,&st);
  .
  .
}
```

8.5 Rückgabe von Parametern

Parameter werden in `%o0` zurückgegeben, im Callerwindow, also in `%i0` im Calleewindow. Wenn eine Struct zurückgegeben werden soll, dann wird der Structure Pointer im Callerframe benutzt, dieser liegt an der Stelle `%fp+64`. Structure Pointer müssen vom Caller initialisiert werden, so dass er auf einen gültigen Speicherbereich zeigt.

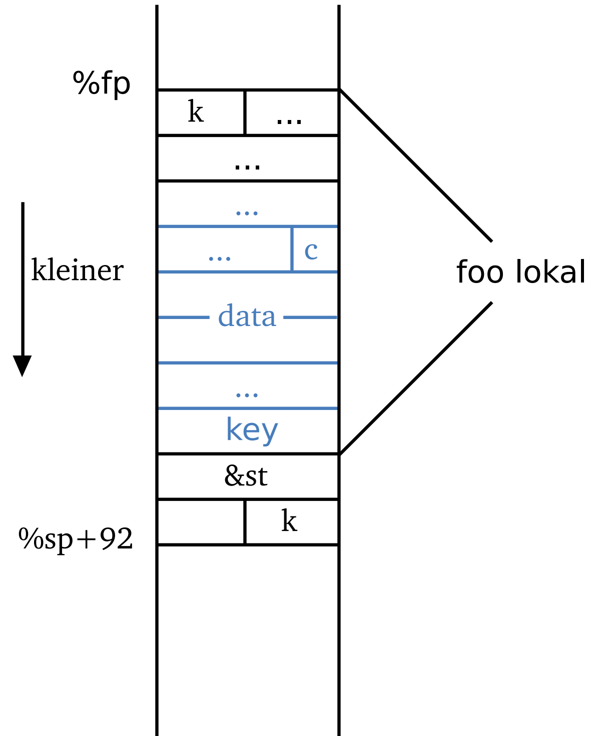

Abbildung 6: *Stack nach dem Aufruf von save in bar*

restore

Die `restore` Anweisung unterstützt wie gesagt eine zusätzliche Addition, also falls die Summe von zwei Werten in `%o0` bzw. `%o1` (im Callee) zurückgegeben werden soll, dann kann man wie folgt vorgehen:

```
restore %o0, %o1, %o0
```

Zu bemerken ist, dass die Destination `%o0` ist und nicht `%i0`, dies da dies schon wieder im Caller ist.

unimp

Falls ein struct zurückgegeben werden soll, so legt der Caller als eine weitere Besonderheit nach dem Delayslot eine `unimp` Instruktion an, deren Argument die Grösse des für die struct bestimmten Speicherbereichs enthält. Also muss der Callee sowohl über den Delayslot als auch die `unimp` Instruktion springen.

```
jmpl %i7+12, %g0  ! jump past unimp
```

8.6 Leaf Subroutinen

Leaf-Prozeduren sind Prozeduren welche ...

- keine Funktionsaufrufe tätigen.
- kein eigenes Register Window anfordern, also keine `save`/`restore` Instruktionen
- Nur auf `%o0-%o5` und `%g1` arbeiten (da sie kein eigenes Window haben).

Der Rücksprung ist anders, da `%pc` in `%o7`, nicht in `%i7`

```
jmpl  %o7+8, %g0

! oder Synonym fuer obiges
retl
```

9 Register Speicherung und Traps

Dass nicht jedes Mal wenn eine Subroutine aufgerufen wird, die Register in den Hauptspeicher kopiert werden müssen, wird ein Pool von Registern benutzt und ein Fenster (CWP bits im PSR (=processor status register)) das auf die aktuellen Register zeigt. Wenn eine Subroutine eine neue Menge von Registern (in, local, out) benötigt so wird einfach der CWP (Current Window Pointer) verschoben:

- Die Ausgabe von caller A wird der Input vom callee B.
- `%o6` ist der `%sp` von A und wird abgebildet auf `%i6` für B, wo B seinen `%fp` findet.
- `%o7` ist die Adresse der aufrufenden Instruktion in A und wird abgebildet auf `%i7`, wo B nach der Rücksprungadresse sucht, wenn er fertig ist.
- `%o0-%o5` sind die Übergabeargumente von A an B, B findet sie in `%i0-%i5`.
- save: `cwp--`
- restore: `cwp++`

Somit muss nur der CWP verschoben werden, was sehr schnell gemacht werden kann. Die Additionsoperationen von `save` und `restore` haben eine Besonderheit: das Destinationsregister ist im neuen Set von Registern.

```
save %sp, -64, %sp    ! die zwei %sp sind
                      ! unterschiedlich
```

9.1 Register overflow/underflow

Es gibt nur eine beschränkte Anzahl von Registern, in der SPARC Architektur z.B. 7 (also 6 Calls). Was passiert wenn wir sie alle benutzt haben:

- Der CWP erreicht den WIM (welcher das Ende der Registersets anzeigt) und ein Hardware Trap tritt ein.
- Der Trap resultiert in einem Programm, das aufgerufen wird und die Register auf den Stack kopiert, so dass Platz für Register der aufgerufenen Subroutine entstehen.
- Dafür wird der Platz verwendet, der vorher mit der `save` Operation reserviert wurde benutzt.

9.2 Register windows

`CWP` ist ein Counter

- geht herum, ähnlich Modulus
- save: `CWP--`
- restore: `CWP++`

`WIM` ist eine Maske von Bits

- geht herum, ähnlich Modulus
- Bit geht nach rechts bei einem Overflow. Z.Bsp. 0x08, 0x04, 0x02, 0x01, 0x08, 0x04, 0x02
- Bit geht nach links bei einem Underflow. Z.Bsp. 0x01, 0x02, 0x04, 0x08, 0x01, 0x02, 0x04

Es hat immer ein Windows übrig für Traps.

9.3 Ablauf eines Overflow Traps

Es wird versucht einen `save` auszuführen, aber `WIM[--CWP] == 1`, es tritt also ein Overflow auf, somit wird der Overflow trap handler aufgerufen, mit dem Registerset CWP (nach Subtraktion). Der Trap handler geht wiefolgt vor:

- `CWP--` (nocheinmal), so dass der `CWP` jetzt auf dem Registerset ist, das etwas enthält, was gespeichert werden muss.
- Speichere nun `%lx` und `%ix` in den Speicher an `mem[%sp]`.
- Rotiere das `WIM`, so dass es jetzt auf das aktuelle `CWP` zeigt.
- $\texttt{CWP} + 2$
- Der `CWP` ist wieder auf dem ursprünglichen Registerset, `CWP--` ist frei und somit hat die `save` Instruktion dieses Mal Erfolg.

9.4 Ablauf eines Underflow Traps

Es wird versucht ein `restore` auszuführen, aber `WIM[++CWP] == 1`, es tritt also ein Underflow auf, der Trap handler wird gestartet:

- $\texttt{CWP} - 2 \mod \texttt{NWINDOWS}$
- Den `WIM` rotieren, $\texttt{WIM}[\texttt{CWP} + 3 \mod \texttt{NWINDOWS}] = 1$
- $\texttt{CWP} + 2$
- Laden von `%lx,%ix` vom Speicher `mem[%sp]`
- Das Register ist jetzt restored
- `CWP--`
- `CWP` nun wieder bei ursprünglichem Registerset vor `restore`

9.5 Trap Maschinerie

Wenn wir zu einem Traphandler springen, so arbeiten wir manchmal für das Programm, das gerade ausgeführt wird (z.Bsp. Register window overflow) oder wir arbeiten für ein anderes Programm (z.Bsp. Mausbewegungen oder Disk I/O). Die Maschine muss irgendwie zwischen diesen beiden Situationen unterscheiden. Die Lösung ist, dass die Hardware verschiedene Modes unterstützt, einen User mode und einen System mode.

9.6 Definitionen

Sind nicht standardisiert!

Interrupt: Asynchrone Unterbrechung des Instruktionsfluss. Wird durch ein externes Ereignis verursacht. Kann eine Programmausführung zwischen 2 Instruktionen treffen.

Exception: Unerwartetes Ereignis während der Ausführung des Programmes (Division durch Null, Page fault). Trifft während einer Instruktion ein.

Trap: Softwaregenerierter Interrupt. Wird durch eine Exception oder durch eine expliziete trap Instruktion (z.Bsp. ein Systemcall) ausgelöst.

9.7 Trapinstruktionen txx

Springe zu OS code um gewisse Funktionen auszuführen. Die Programmausführung wird vom Usercode zum Systemcode transferiert. Der Zustand der Maschine (CPU) wird gespeichert und wird wiederhergestellt beim Zurückkehren zum Usercode (return von Trap: rett). `txx` ist nicht verzögert!

9.8 Status Register des Prozessors

Y-Register: für Multiplikation und Division: lesen und schreibe mit `rdy,wry` im Usermode.

PC und nPC: lesen und schreiben ist implizit durch `jmpl, ret, ...` im Usermode.

PSR Processor State Register: lesen und schreiben nur im Superusermodues (`%psr, rdpsr, wrpsr`)

WIM Window Invalid Mask Register: lesen und schreiben nur im Superusermodus (`%wim, rdwim, wrwim`).

TBR Trap Base Register: lesen und schreiben nur im Superusermodus (`%tbr, rdtbr, wrtbr`)

Processor State Register (PSR)

besteht aus verschiedenen Bits, unter anderem:

- `ET`: aktiviere Trap: `ET == 1`, während einem Trap: `ET == 0`.
- `CWP`

Window Invalid Mask (WIM)

k Flags um die k Registersets zu implementieren. ($2 \leq k \leq 32$).

Trap Base Register (TBR)

- TBA: trap base address, oberer Teil der Basisadresse der Traptabelle. Bits 31-12.
- tt: trap type, 256 mögliche Traps, Offset in die Traptabelle. Bits 11-4
- zero: letzen 4 bits sind immer 0.

Der TBR setzt die bits TBA und tt zusammen in eine Zieladresse für den Call. Die zero-bits machen dass aufeinanderfolgende Trap-Einstiegspunkte 16 Bytes auseinander sind. Die Tabelle kann die ersten 4 Instruktionen des Traphandlers beinhalten.

```
handler_vect:
    set    handler, %l3
    jmpl   %l3, %r3
    nop
    .
    .
handler: ...
```

9.9 Priorität von Traps- und Interrupts

0x80-0xFF: Software-Traps
0x00-0x7F: Hardware-Traps, nur teilweise benutzt in heutigen Implementationen.

Wenn `ET==1`, dann werden Traps nach Priorität ausgeführt (siehe Tab. Skript), sonst wenn `ET==0` werden alle Interrupts ignoriert und jeder weitere Trap resetet die Maschine. Interrupts haben kleinere Priorität als Exceptions, haben also höhere Trapnummern.

9.10 Schritte die ausgeführt werden, wenn ein Trap auftritt

Wenn `ET==1`:

- Setze `ET = 0` – deaktiviere Traps
- Setze PS = S – speichere aktuellen Ausführungsmodus
- Setze S = 1 – wechsle in den Superusermodus
- Setze `CWP` = `CWP` − 1 mod `NWINDOWS` – rücke Registerwindow vor ohne Test ob Window overflow
- `%l1 = PC; %l2 = nPC` – speichere getrappte Programmcounters
- Setze tt=trap_type – schreibe tt Feld
- Setze PC = TBR; nPC = TBR+4 – Transferiere Kontrolle an Traptabelle
- (Reset Trap: PC=0;nPC=4)

Optional:

- `%l0 = %ps` – speichere den PSR wenn der Traphandler ihn verändert und wiederherstelle ihn am Ende
- Wenn die Register `%l3-%l7` nicht genügend sind und `WIM[CWP] == 1`, dann muss das Registerwindow explizit gespeichert werden.
- Wenn der Trap ein Interrupt ist: PSR muss gespeichert werden, PIL setzen, ET=1 und das Window muss auf jeden Fall gespeichert werden.

Wenn ET==0:

- Interrupts werden ignoriert.
- Weitere Traps/expceptions führen zu einem Reset der Maschine.

9.11 Rückkehr vom Traphandler

`rett address`

- Setze CWP = CWP + 1 mod NWINDOWS, rücke also das Registerwindow vor.
- Setze nPC = address
- Initiiere verzögerten Transfer zur Rücksprungadresse von der Trapinstruktion
- Setze S = PS
- Wiederherstelle den vorherigen Ausführungsmodus
- Setze ET=1

Bemerkungen:

- Möglicherweise wiederherstellen von PSR, PIL, Register Window
- Instruktion vor `rett` muss `jmpl` sein
- `jmpl` setzt PC, `rett` setzt nPC

Optionen für die Rückkehr vom Traphandler

Repetiere Instruktion, die den Trap auslöste:

```
jmpl    %l1, %g0  !old PC
rett    %l2       !old nPC

    pc -> jmpl     npc -> rett
    pc -> rett     npc -> old PC
    pc -> old PC   npc -> old nPC
```

Kehre zur Instruktion zurück, welche nach der kommt, welche den Trap auslöste:

```
jmpl    %l2, %g0  !old nPC as new PC
rett    %l2+4     !old nPC+4

    pc -> jmpl     npc -> rett
    pc -> rett     npc -> old nPC
    pc -> old nPC  npc -> old nPC+4
```

10 Input-Output

10.1 Übersicht

Memory mapped I/O

- Geräteregister sind Teil des Adressraums.
- Statusregister um den Zustand des Gerätes bekannt zu machen.

I/O Ports

- Jeder Controller bekommt eine spezielle Adresse, die Port genannt wird (welche nicht Memoryadressen sind, aber spezielle Adressen)

Interrupt Request Lines (IRQ)

- Physikalische Eingabe zum Interrupt controller chip
- Signale wann das Gerät bereit ist gehen direkt zum interrupt controller.

I/0 als ein Problem

Meist langsamer als CPU, somit müsste CPU auf I/O warten, oder schon weiter arbeiten, aber so möglicherweise die entsprechenden Daten nicht. Die Lösung ist die Benutzung von Interrupts (Traps), welche die CPU benachrichtigen, dass das Gerät bereit ist. Die CPU sendet dann den nächsten Teil des Jobs und macht danach wieder etwas anderes bis zum nächsten Interrupt.

Für Geräte, welche gleich schnell oder schneller als die CPU sind, sagt die CPU den Geräten nur, wo sie die Daten aufnehmen sollen. Das Gerät löst dann nur einen Interrupt aus, falls es alle Operationen gemacht hat.

I/O in SysProg

In den meisten Programmiersprachen wird I/O durch den Aufruf von Programmen in spezillen Libraries bewerkstelligt. Diese Libraries stellen zuerst fest ob die Anfrage korrekt ist und rufen danach das OS auf um die I/O Operation auszuführen. Diese Interaktion zwischen dem Userprogramm und dem OS findet durch Traps statt.

10.2 Memory mapped I/O

In der SPARC Architektur wird mit I/0 Geräten über das Memory kommuniziert.

Eine bestimmte Region des Memory (0xfff00000-0xffffe000) ist reserviert und words in dieser Region werden als Geräteregister benutzt. Typischerweise sind für jedes Gerät zwei Geräteregister:

- I/O-Register für das Gerät: schreiben und lesen in diesem Register ist äquivalent zu Schreiben und Lesen vom Gerät.
- Statusregister für das Gerät: Indiziert den Status des Geräts als eine Serie von Flags (R=ready, E=error, I=interrupt ...)

Somit wird nur `ld` und `st` gebraucht.

10.3 Character Geräte

Um auf einem Drucker im Register 0xffff0000 einen Buchstaben auszugeben:

```
mov     "a", %o0
set     0xffff0000, %o1
stb     %o0, [%o1]
```

Oder um von einem Keyboard in Register 0xffff0008 ein ASCII Buchstaben zu lesen:

```
set     0xffff0008, %o1
ldub    [%o1], %o0
```

10.4 Interrupt getriebenes I/0

```
crt = 0xffff0000   ! fictious crt device register

                   ! crt registers
data = 0           ! data register
status = 4         ! status register

                   ! status register bits
crt_ready = 0x80   ! ready
crt_error = 0x40   ! error
crt_intr = 0x20    ! interrupt
crt_reset = 0x10   ! reset device

                   ! registers assignments
crt_r = l2         ! %l2 crt base register
ptr_r = l3         ! %l3 pointer to string
ptr_adr_r = l4     ! %l4 address of pointer
data_r = l5        ! %l5 data
status_r = l6      ! %l6 status

.data
outtxt:   .asciz   "Hello world...\n"
ptr_m:    .word    outtxt !pointer to string

.text

start:                        !code to start trans
  set  ptr_m, %ptr_adr_r      !addr string pointer
  ld   [%ptr_adr_r], %ptr_r   !pointer to string
  set  crt, %crt_r            !addr dev register
  mov  crt_reset, %status_r   !clear er/int status
  stb  %status_r, [%crt_r+status] !+set ready bit
  mov  crt_intr, %status_r    !enable interrupts
  stb  %status_r, [%crt_r+status]
  ldub [%ptr_r], %data_r      !output first char
  stb  %data_r, [%crt_r+data]
  inc  %ptr_r                 !increment pointer
  st   %ptr_r, [%ptr_adr_r]
  ret                         !return


next:                         !interrupt code
  set  ptr_m, %ptr_adr_r      !pointer address
  ld   [%ptr_adr_r], %ptr_r   !pointer to string
  ldub [%ptr_r], %data_r      !load byte of data
  tst  %data_r                !test if end string
  be   done
  set  crt, %crt_r            !address of dev reg
  stb  %data_r, [%crt_r+data]!output next char
  inc %ptr_r                  !increment pointer
  st  %prt_r, [%ptr_adr_r]
  jmpl %r18, %r0              !return from
  rett %r18+4                 !interrupt

done:
  clr  %ptr_r                 !just clear pointer
  st   %ptr_r, [%ptr_adr_r]
  mov  crt_reset, %status_r  !and reset device
  stb  %status_r, [%crt_r+status] !clear err/int
  jmpl %r18, %r0              !return from
  rett %r18+4                 !interrupt
```

10.5 Handling von Interrupts

Eine typische CPU, die Interrupts behandeln soll, hat einen weiteren Schritt am Ende des Instruktionszyklus (Fetch, Decode, Operand fetch, execute, Store), welcher überprüft ob ein Interrupt eingetroffen ist. Die CPU hat ein Statusbit (IF), welches anzeigt, ob Interrupts behandelt werden sollen.

10.6 I/O in UNIX

In UNIX, werden alle Geräte als Files gesehen. Somit sehen alle I/O Operationen nach Dateioperationen aus (create, open, read, write, close). Um diese Operationen auszuführen muss das Programm den zugehörigen Trap ausführen. Das OS übersetzt dann die Dateioperationen in Operationen auf dem entsprechenden Gerät.

11 Byte Ordnung

11.1 Allgemeines

```
int i; /* 4 bytes */

/* &i zeigt auf die Adresse vom Byte
mit der kleinsten Adresse. */
```

Die Frage ist nun: wie setzen wir die Bytes (von `i`) in das Memory. Dafür gibt es zwei Möglichkeiten.

Big endian

Das MSB wird bei der kleinsten Adresse gespeichert, das LSB bei der grössten.

```
            +-+-+-+-+-+
0x104       |   MSB   |
            +-+-+-+-+-+
            +-+-+-+-+-+
0x105       |         |
            +-+-+-+-+-+
            +-+-+-+-+-+
0x106       |         |
            +-+-+-+-+-+
            +-+-+-+-+-+
0x107       |   LSB   |
            +-+-+-+-+-+
```

Falls a == 0x01234567

So würde a im Memory so aussehen: 67 45 23 01

Big endian wird von SPARC verwendet.

Little endian

Das MSB wird bei der grössten Adresse gespeichert, das LSB bei der kleinsten.

```
                +-+-+-+-+-+
0x104           |   LSB   |
                +-+-+-+-+-+
                +-+-+-+-+-+
0x105           |         |
                +-+-+-+-+-+
                +-+-+-+-+-+
0x106           |         |
                +-+-+-+-+-+
                +-+-+-+-+-+
0x107           |   MSB   |
                +-+-+-+-+-+
```

Falls a == 0x01234567
So würde a im Memory so aussehen: 01 23 45 67

Little endian wird von x86 verwendet.

12 Linking

12.1 ld: Der UNIX Linker

Ein Linker kombiniert verschiedene Dateien: Sourcen, Libraries und löst Referenzen auf. Ohne einen Linker müsste man alles in einer einzigen Datei haben.

12.2 Statisches Linking

Verschiedene einzelne Dateien, die seperat compiliert werden und danach zu einem einzigen Executable gelinkt werden.

12.3 ELF Object File Format

- ELF header: Magische Nummber, Typ (.o, exec, .so), Maschine, Byte ordering, etc.
- Programm header Tabelle: Page size, virtual address memory segments (sections), segment sizes.
- .text: Code
- .data: initialisierte (statische) Daten
- .bss: uninitialisierte (statische) Daten, "Block storage start"
- .symtab: Symboltabelle, Prozedur und statische Variablennamen, Sektionnamen und Lokationen
- .rel.text: Relocation info für .text Sektion, Adressen von Instruktionen welche im Executable modifiziert werden müssen, Instruktionen für das Modifizieren.
- .rel.data: Relocation Info für .data Sektion, Adressen von Pointerdaten welche im zusammengefügten Executable modifiziert werden müssen.
- .debug: Info für Debugging (gcc -g).

12.4 C Regeln

extern: Name wird an den Linker exportiert. static: Name wird nicht an den Linker exportiert.

Interpretation von top-level Deklarationen:

`int x`	Referenz
`int x = 0`	Definition
`extern int x`	Referenz
`extern int x = 0`	Definition

Programmsymbole sind entweder strong oder weak:

- strong: Prozeduren und initialisierte Globale.
- weak: uninitialisierte Globale.

12.5 Linker's Symbol Regeln

1. Ein strong Symbol kann nur ein Mal auftreten.
2. Ein weak Symbol kann durch ein strong Symbol vom selben Namen überschrieben werden. Referenzen zum weak Symbolen lösen sich zum strong Symbol auf.
3. Wenn mehrere weak Symbole vorhanden sind, kann der Linker irgendeines auswählen.

12.6 Static libraries

Verbessere den Linker so, dass er versucht nicht auflösbare externe Referenzen durch Nachschauen nach Symbolen in Archiven aufzulösen. Das Executable enthält dann nur den Code vom Archiv den es auch wirklich benötigt.

12.7 Startup Code

Das .text Segment beginnt mit einem speziellen Label (`__start`) und einer Sequenz von Instruktionen:

- Library startup code
- `call _init`, weiterer Systemstartup code
- `call atexit`, setzt den cleanup Code, der ausgeführt wird wenn `_exit` aufgerufen wird.
- `call main`, Userprogramm
- `call _exit`

Während dem Laden übergibt das OS `__start` die Kontrolle.

12.8 Shared libraries und dynamisches Linking

Shared libraries, deren Members werden zur Laufzeit in das Memory geladen und in die Applikation verlinkt. Shared libraries Routines können von mehreren Prozessen geteilt werden.