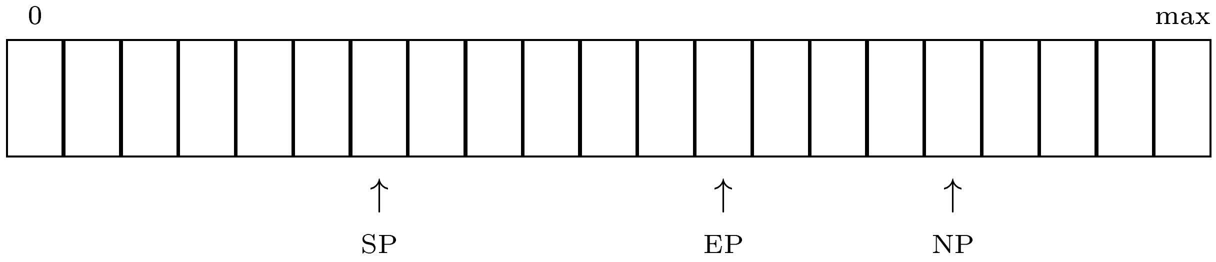
Architektur der C-Maschine

Datenspeicher S



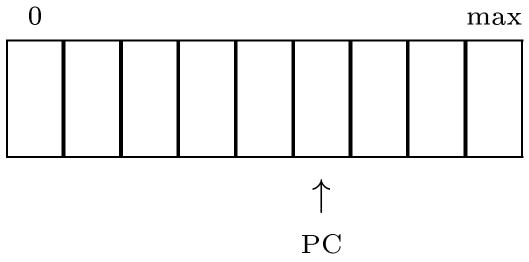
stack pointer SP

new pointer NP

extreme stack pointer EP

frame pointer FP

Befehlsspeicher C



program counter PC

Maschinenzyklus

- Lade Instruktion aus `C[PC]`.
- `PC = PC + 1`.
- Führe die Instruktion aus.

Die Befehle der C-Maschine

Arithmetik

mul	`S[SP-1] = S[SP-1] * S[SP];` `SP = SP - 1;`
add	`S[SP-1] = S[SP-1] + S[SP];` `SP = SP - 1;`
sub	`S[SP-1] = S[SP-1] - S[SP];` `SP = SP - 1;`
div	`S[SP-1] = S[SP-1] / S[SP];` `SP = SP - 1;`
mod	`S[SP-1] = S[SP-1] % S[SP];` `SP = SP - 1;`
neg	`S[SP] = -S[SP];`

Boolesche Arithmetik

and	`S[SP-1] = (S[SP-1] && S[SP]);` `SP = SP - 1;`
or	`S[SP-1] = (S[SP-1] || S[SP]);` `SP = SP - 1;`
not	`S[SP] = !S[SP];`

Vergleichsoperationen

eq	`S[SP-1] = (S[SP-1] == S[SP]);` `SP = SP - 1;`
geq	`S[SP-1] = (S[SP-1] >= S[SP]);` `SP = SP - 1;`
leq	`S[SP-1] = (S[SP-1] <= S[SP]);` `SP = SP - 1;`
gt	`S[SP-1] = (S[SP-1] > S[SP]);` `SP = SP - 1;`
lt	`S[SP-1] = (S[SP-1] < S[SP]);` `SP = SP - 1;`
neq	`S[SP-1] = (S[SP-1] != S[SP]);` `SP = SP - 1;`

Laden einer Konstanten

loadc q

```
S[SP+1] = q;
SP = SP + 1;
```

Laden mit absoluter Adressierung

loada q

```
S[SP+1] = S[q];
SP = SP + 1;
```

Laden mit indirekter Adressierung

load

```
S[SP] = S[S[SP]];
```

Speichern

store

```
S[S[SP]] = S[SP-1];
SP = SP - 1;
```

Unbedingter Sprung

jump q

```
PC = q;
```

Bedingter Sprung bei 0

jumpz q

```
if (S[SP] == 0) PC = q;
SP = SP - 1;
```

Indizierter Sprung

jumpi l

```
PC = l + S[SP];
SP = SP - 1;
```

Stackpointer dekrementieren

pop

```
SP = SP - 1;
```

Kopieren

copy
```
S[SP+1] = S[SP];
SP = SP + 1;
```

Speicherplatz anlegen

new
```
if (NP - S[SP] <= EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

Retten von FP und EP

mark
```
S[SP+2] = FP;
S[SP+3] = EP;
SP = SP + 4;
```

Funktionsaufruf

call n
```
FP = SP - n - 1;
S[FP] = PC;
PC = S[SP];
SP = SP - 1;
```

Erhöhen von EP

enter p
```
EP = SP + p;
if (EP >= NP) error("Stack overflow");
```

Erhöhen von SP

alloc q
```
SP = SP + q;
```

Verlassen einer Funktion

return
```
SP = FP - 3;
PC = S[FP];
EP = S[FP-1];
if (EP >= NP) error("Stack overflow");
FP = S[SP+1];
```

Parameterübergabe

move n

```
S[SP+n-1] = S[S[SP]+n-1];
S[SP+n-2] = S[S[SP]+n-2];
             ⋮
S[SP+1]   = S[S[SP]+1];
S[SP]     = S[S[SP]];
SP = SP + n - 1;
```

Zugriff auf lokale Variablen und Parameter

loadrc j

```
SP = SP + 1;
S[SP] = FP + j;
```

loadr j

```
SP = SP + 1;
S[SP] = S[FP + j];
```

Keine Operation

nop

```
SP = SP;
```

Beendigung des Maschinenprogramms

halt

Übersetzung von C in Code der C-Maschine

Zuweisung:

code $(x = e)\ \varphi =$

$\quad$ $\text{code}_R\ e\ \varphi$
$\quad$ $\text{code}_L\ x\ \varphi$
$\quad$ `store`

Anweisung:

code $(e;)\ \varphi =$

$\quad$ $\text{code}_R\ e\ \varphi$
$\quad$ `pop`

Leere Anweisung:

code $(\ )\ \varphi =$

$\quad$ `nop`

Anweisungsfolge:

code $(s\ \ ss)\ \varphi =$

$\quad$ code $s\ \varphi$
$\quad$ code $ss\ \varphi$

Einseitige bedingte Anweisung:

code (**if** $(e)\ \ s')\ \varphi =$

$\quad$ $\text{code}_R\ e$
$\quad$ `jumpz` l
$\quad$ code $s'\ \varphi$
l:

Zweiseitige bedingte Anweisung:

code (**if** $(e)\ \ s_1$ **else** $s_2)\ \varphi =$

$\quad$ $\text{code}_R\ e\ \varphi$
$\quad$ `jumpz` l_1
$\quad$ code $s_1\ \varphi$
$\quad$ `jump` l_2
l_1: code $s_2\ \varphi$
l_2: ...

for-Schleife:

code (**for** $(e_1; e_2; e_3)$ s) φ =

```
        code_R e1 φ
        pop
l1:     code_R e2 φ
        jumpz l2
        code s φ
        code_R e3 φ
        pop
        jump l1
l2:
```

while-Schleife:

code (**while** (e) s) φ =

```
        code_R e φ
        jumpz l1
        code s φ
l1:
```

do-Schleife:

code (**do** s **while** (e)) φ =

```
l1:     code s φ
        code_R e φ
        jumpz l2
        jump l1
l2:
```

switch-Anweisung:

$$
\begin{array}{l}
\text{code} \ (\ \textbf{switch}\ (e)\ \{ \\
\qquad \begin{array}{lll}
\textbf{case}\ 0: & ss_o & \textbf{break;} \\
\textbf{case}\ 1: & ss_1 & \textbf{break;} \\
\qquad \vdots & & \\
\textbf{case}\ k-1: & ss_{k-1} & \textbf{break;} \\
\textbf{default}: & ss_k & \textbf{break;} \\
\}\ )\ \varphi = & &
\end{array}
\end{array}
$$

$$
\begin{array}{rl}
 & \text{code}_R\ e\ \varphi \\
 & \texttt{copy} \\
 & \texttt{loadc}\ k \\
 & \texttt{leq} \\
 & \texttt{jumpz}\ l_k \\
 & \texttt{copy} \\
 & \texttt{loadc}\ 0 \\
 & \texttt{geq} \\
 & \texttt{jumpz}\ l_k \\
 & \texttt{jumpi}\ l \\
l_0: & \text{code}\ ss_0\ \varphi \\
 & \texttt{jump}\ l' \\
l_1: & \text{code}\ ss_1\ \varphi \\
 & \texttt{jump}\ l' \\
 & \vdots \\
l_{k-1}: & \text{code}\ ss_{k-1}\ \varphi \\
 & \texttt{jump}\ l' \\
l_k: & \texttt{pop} \\
 & \text{code}\ ss_k\ \varphi \\
 & \texttt{jump}\ l' \\
l: & \texttt{jump}\ l_0 \\
 & \texttt{jump}\ l_1 \\
 & \vdots \\
 & \texttt{jump}\ l_k \\
l': &
\end{array}
$$

Zugriffe auf Objekte:

$\text{code}_L\ (x)\ \varphi =$ // für Namen x

`loadc` $\varphi(x)$

$\text{code}_R\ (x)\ \varphi =$ // falls x einfach oder Zeiger

`loada` $\varphi(x)$

$\text{code}_R\ (x)\ \varphi =$ // sonst

$\text{code}_L\ x\ \varphi$
`load`

$\text{code}_R\ (e)\ \varphi =$ // falls e Feld

$\text{code}_L\ e\ \varphi$

$\text{code}_L\ (*e)\ \varphi =$

$\text{code}_L\ e\ \varphi$
`load`

$\text{code}_R\ (\&e)\ \varphi =$

$\text{code}_L\ e\ \varphi$

$\text{code}_L\ (e_1[e_2])\ \varphi =$

$\text{code}_L\ e_1\ \varphi$
$\text{code}_R\ e_2\ \varphi$
`loadc` $|t|$
`mul`
`add`

$\text{code}_L\ (e.a)\ \varphi =$

$\text{code}_L\ e\ \varphi$
`loadc` $\varphi(a)$
`add`

$\text{code}_L\ ((*e).a)\ \varphi =$

$\text{code}_L\ e\ \varphi$
`load`
`loadc` $\varphi(a)$
`add`

$\text{code}_L\ (e \to a)\ \varphi =$

$\text{code}_R\ e\ \varphi$
`loadc` $\varphi(a)$
`add`

Dynamischer Speicherplatz:

code (**malloc**(e)) φ =

    code$_R$ e φ
    `new`

code (**free**(e)) φ =

    code e φ

Funktionsadresse:

code$_R$ (f) φ =   // Anfangsadresse von f

    f_{adr}

Funktionsdefinition:

code (t f (parms) { V_defs ss }) φ =

`enter` $f_{\max}$	// maximale Größe des lokalen Stacks
`alloc` s_{local}	// Platz für lokale Variablen
code ss φ'	// Code in neuer Adressumgebung
`return`	// Rücksprung

Zugriff auf Parameter und Variablen:

Extern ($\varphi(x) = (E, j)$):

code$_L$ (x) φ =

    `loadc` j

code$_R$ (x) φ =

    `loada` j

Automatisch ($\varphi(x) = (A, j)$):

code$_L$ (x) φ =

    `loadrc` j

code$_R$ (x) φ =

    `loadr` j

Funktionsaufruf:

$e_1, \ldots, e_k$: Ausdrücke, die Wert eines einfachen Typs liefern
$e_{k+1}, \ldots, e_m$: struct-Parameter
t_i: Typ des i-ten Parameters mit Größe $n_i = |t_i|$
n: $n = \sum_{i=1}^{m} |t_i|$

code_R ($g(e_1, \ldots, e_k, e_{k+1}, \ldots, e_m)$) φ =

```
mark
codeR e1 φ
⋮
codeR ek φ
codeL ek+1 φ
move nk+1 φ
⋮
codeL em φ
move nm φ
codeR g φ
call n
```

C-Programm:

V_defs: globale Definitionen
F_def_1: Funktion 1
F_def_n: Funktion n (`main()`)

code(V_defs F_def_1 F_def_2 … F_def_n) $\emptyset$ =

```
     enter |V_defs|+4   //
     alloc |V_defs|     // Platz für Variablen in V_defs
     mark               //
     codeR main φ       // φ  ist die externe Adressumgebung
     call 0             // Starte Hauptprogramm
     halt               //
l1   code F_def1 φ      //
     ⋮                  //
ln   code F_defn φ      //
```

Initialisierung: SP = 1, FP = 0, EP = 0, PC = 0, NP = max;

Beispiel

Wir geben die Übersetzung der folgenden rekursiven C-Funktion an.

```
int
f(int n)
{
  if (n<=1)
      return(1);
  else
      return( n * f(n-1) );
}
```

Code der C-Maschine

l_0:	`enter 7`	// lokaler Speicher
	`alloc 0`	// keine lokalen Variablen
	`loadr 1`	// $\varphi(n) = (A, 1)$
	`loadc 1`	// Konstante 1
	`leq`	// $n <= 1$?
	`jumpz` l_1	// Falls $n > 1$, rekursiver Aufruf
	`loadc 1`	// sonst, Wert 1
	`loadrc -3`	// Adresse für Funktionswert
	`store`	// Gib Funktionswert 1 zurück
	`return`	// Sprung in aufrufende Funktion
l_1:	`loadr 1`	// Zugriff auf n (für `mul`)
	`mark`	// Vorbereitung des rekursiven Aufrufs
	`loadr 1`	// Lade aktuellen Parameter
	`loadc 1`	// Konstante 1
	`sub`	// Dekrementiere aktuellen Parameter
	`loadc` l_0	// Adresse von f
	`call 1`	// Rekursiver Aufruf
	`mul`	// Multiplikation $n \cdot f(n-1)$
	`loadrc -3`	// Adresse für Funktionswert
	`store`	// Speichern des Produkts
l_2:	`return`	// Sprung in aufrufende Funktion

Übersetzung einer for-Schleife (SUN SPARC)

C-Programm

```
s = 9;
for (i=3;i<=77;i+=4) {
    s = s + i;
}
```

Assembler-Programm

```
        mov     0x9,%o0
        st      %o0,[%fp+-0x8]
        mov     0x3,%o1
        st      %o1,[%fp+-0x4]
L16:
        ld      [%fp+-0x4],%o2
        cmp     %o2,0x4d
        bg      L15
        ld      [%fp+-0x4],%o0
        ld      [%fp+-0x8],%o1
        add     %o1,%o0,%o1
        st      %o1,[%fp+-0x8]
L14:
        ld      [%fp+-0x4],%o2
        add     %o2,0x4,%o2
        st      %o2,[%fp+-0x4]
        b       L16
L15:
```

Übersetzung einer while-Schleife (SUN SPARC)

C-Programm

```
s = 0;
i = 255;
while (i>7) {
      s = s + i;
      i  = i - 3;
}
```

Assembler-Programm

```
        st      %g0,[%fp+-0x8]
        mov     0xff,%o0
        st      %o0,[%fp+-0x4]
L14:
        ld      [%fp+-0x4],%o1
        cmp     %o1,0x7
        ble     L15
        ld      [%fp+-0x4],%o0
        ld      [%fp+-0x8],%o1
        add     %o1,%o0,%o1
        st      %o1,[%fp+-0x8]
        ld      [%fp+-0x4],%o2
        sub     %o2,0x3,%o2
        st      %o2,[%fp+-0x4]
        b       L14
L15:
```

Übersetzung der switch-Anweisung (SUN SPARC)

C-Programm

```
switch (i) {
case 1:  s = 10; break;
case 5:  s = 11; break;
case 65: s = 12; break;
default: s = 13; break;
}
```

Assembler-Programm

```
        b       L17
L18:
        mov     0xa,%o0
        st      %o0,[%fp+-0x8]
        b       L16
L19:
        mov     0xb,%o1
        st      %o1,[%fp+-0x8]
        b       L16
L20:
        mov     0xc,%o2
        st      %o2,[%fp+-0x8]
        b       L16
L21:
        mov     0xd,%o3
        st      %o3,[%fp+-0x8]
        b       L16
L17:
        ld      [%fp+-0x4],%o0
        cmp     %o0,1
        be      L18
        cmp     %o0,5
        be      L19
        cmp     %o0,65
        be      L20
        b       L21
L16:
```

Zugriff auf Variablen und Felder (SUN SPARC)

C-Programm

```
main()
{

  int i,s;
  int a[20];
  i = 12;
  s = 20;

  a[i] = s;

}
```

Assembler-Programm

```
        mov     0xc,%o0
        st      %o0,[%fp+-0x4]

        mov     0x14,%o1
        st      %o1,[%fp+-0x8]

        sub     %fp,0x58,%o2
        ld      [%fp+-0x4],%o3
        sll     %o3,0x2,%o4
        ld      [%fp+-0x8],%o5
        st      %o5,[%o2+%o4]
```