Rechnerstrukturen

8. Assembler

Für Hartgesottene

- Maschinensprache
  - Opcode, Operanden, Adressierung
  - Für Menschen schwer verdauliche Folgen von 0 und 1
- Assembler
  - Symbolische Äquivalente für Maschineninstruktionen
  - 1:1-Umsetzung
- Assemblerprogrammierung ist selten geworden
  - Eingebettete Systeme, Hardwaresteuerungen
  - Compilerbauer
  - Systemsoftware
  - Optimierung (mittlerweile schwer schlagbar)

Assembler vs. Hochsprache

- Maschinensprache
  - Instruktion = Bitkombination
  - Für Menschen ungeeignet
- Assembler
  - Symbolische Instruktionen (Mnemonic)
  - 1:1-Abbildung auf Maschinensprache (Assembler)
- Eigenschaften
  - Einfache Instruktionen
  - Einfache Datentypen
    - Bit, Byte, Wort, Langwort, ...
    - Integer, Float, Boolean
    - Zeiger
  - Unmittelbarer Zugriff auf Hardware
- Hochsprachen
  - Symbolisch
  - Komplexe Sprachelemente
  - Übersetzung (Compiler)
- Eigenschaften
  - Vielfältige Sprachelemente
  - Einfache Datentypen
    - Integer, Float, Boolean, ...
    - Zeiger (sprachabhängig)
  - Zusammengesetzte Typen
    - Record, Feld, ...
  - Eigene Datentypen (z.B. OO)
  - Höhere Abstraktionen verbergen Hardware
    - nicht alle Instruktionen verwendbar
    - eingeschränkter HW-Zugriff

Grundlagen

- Assemblerprogrammierung HW-spezifisch
  - Tiefe Kenntnisse des gewählten Instruktionssatzes
  - Detaillierte Architektur der Zielhardware
  - Funktionalität der zugreifbaren Systemsoftware
- Speicherlayout festlegen
  - Wo befindet sich der Code?
  - Wo befinden sich die Daten?

Pseudoinstruktionen

- Steuerung des Assemblers
  - Speicherlayout
  - Definition symbolischer Namen
  - Definition und Initialisierung globaler Datenstrukturen
  - Makros

Beispiel

```
INCLUDELIB LIBCD
INCLUDELIB OLDNAMES

PUBLIC  ?cpp_fib@@YAJJ@Z                ; cpp_fib
; Function compile flags: /Odt /ZI
; File d:\peter\teach\computer architecture\08_assembler\fibbonacci assembler\fib.cpp
;   COMDAT ?cpp_fib@@YAJJ@Z
_TEXT   SEGMENT
_f$ = -4
_n$ = 8
?cpp_fib@@YAJJ@Z PROC NEAR              ; cpp_fib, COMDAT

; 4    : long cpp_fib ( long n ) {

  00000 55       push    ebp
  00001 8b ec        mov     ebp, esp
  00003 83 ec 44     sub     esp, 68             ; 00000044H
  00006 53       push    ebx
  00007 56       push    esi
  00008 57       push    edi

; 5    :    long f;
; 6    :
; 7    :    if (n<2) f=1; else f=cpp_fib(n-1)+cpp_fib(n-2);

  00009 83 7d 08 02  cmp     DWORD PTR _n$[ebp], 2
  0000d 7d 09        jge     SHORT $L960
  0000f c7 45 fc 01 00
    00 00        mov     DWORD PTR _f$[ebp], 1
  00016 eb 25        jmp     SHORT $L961
$L960:
  00018 8b 45 08     mov     eax, DWORD PTR _n$[ebp]
  0001b 83 e8 01     sub     eax, 1
```

Datentypen

- Assembler-relevante Eigenschaften
  - Größe: Anzahl der belegten Bytes
  - Zugriff: Adressierungsarten
- Einfache Datentypen
  - Boolean, Integer, Float bzw. Double, String, …
- Benutzerdefinierte Typen
  - Strukturen
  - Felder
  - Zeiger bzw. Referenz

Benutzerdefinierte Typen

- Struktur
  - Größe = Σ Einzelgrößen
  - Zugriff
    - Register oder Speicherzelle enthält Anfangsadresse
    - Offset relativ dazu definiert Element der Struktur
- Feld
  - Größe = Elementgröße $\cdot$ Elementanzahl
  - Zugriff
    - Register oder Speicherzelle enthält Feldanfang
    - Indizierter Zugriff (effaddr= base + i $\cdot$ Größe)
    - Geeignete Zugriffsbefehle für Größe = 1,2 und 4

Kontrollstrukturen

- Anweisungsfolge
  - Umsetzung Anweisung für Anweisung
- Bedingte Anweisung
- Schleifen
- Unterprogramme (Methoden)

Bedingte Anweisung

- Hochsprache

```
S1;

if (B) {
    S_if;
}
else {
    S_else;
}

S2;
```

- Assembler

```
          S1;

          Teste(B);
          jcc if_else

          S_if;
          goto if_end
if_else:  S_else

if_end:   S2;
```

while-Schleife

- Hochsprache

```
S1;

while (B) {
    S_while;
}

S2;
```

- Assembler

```
            S1;

loop:       Teste(B);
            jcc while_end

            S_while;

            goto loop

while_end:  S_else

            S2;
```

Unterprogramme

- Aufrufer
  - Legt Argumente auf den Keller
  - Ruft Unterprogramm auf
  - Korrigiert Stack
- Unterprogramm
  - Reserviert Platz für Auto-Variablen
  - Sichert Inhalte benutzer Register
  - …
  - Restauriert Inhalte benutzter Register
  - Gibt Platz für Auto-Variablen frei
  - (Funktionsergebnis kommt in ausgezeichnetes Register)
  - Rücksprung

Assembler Live

Alte Lösung (Tanenbaum)

```
0: LOCO 0     // Stack initialisieren
1: SWAP
2: LODD 100   // an Adresse 100 steht n
3: PSHI       // n kommt auf den Stack
4: CALL 8
5: STOD 101   // Fib(n) an Adresse 101 speichern
6: INSP 1     // Argument n vom Keller nehmen
7:            // Programmende

8: DESP 1     // Speicher für lokale Variable f anlegen
9: LOCO -2
10: ADDL 2    // AC := n-2
11: JPOS 15   // if n>1 then goto
12: LOCO 1
13: STOL 0    // f := 1
14: JUMP 25   // Rücksprung vorbereiten
15: LOCO -1
16: ADDL 2    // AC := n-1
17: CALL 8    // AC := Fib(n-1)
18: INSP 1
19: STOL 0    // f := AC
20: LOCO -2
21: ADDL 2    // AC := n-2
22: CALL 8    // AC := Fib(n-2)
23: INSP 1
24: ADDL 0    // AC := Fib(n-2) + Fib(n-1)
25: LODL 0    // Ergebnis in AC laden
26: INSP 1    // Lokale Variable f freigeben
27: RETN
```

IA-32 Assembler

```
long call_asm_fib ( long n ) {
    long f;
    __asm {
        push n; // Push argument n on stack
        call asm_fib;
        jmp asm_end;
asm_fib:
        sub esp,4; // reserve space for auto variable
        push ebp;
        mov ebp,esp;
        cmp [ebp+0xC],2; // n<2?
        jge asm_fib_rec;
        mov eax,1; // (n<2) is true; return 1
        pop ebp;
        add esp,4; // cleanup auto variable f
        ret
asm_fib_rec:
        mov eax,[ebp+0xC]; // (n<2) is false
        dec eax; // eax=n-1
        push eax;
        call asm_fib;
        add esp,4;
        mov [ebp+0x4],eax; // f=fib(n-1)
        mov eax,[ebp+0xC];
        sub eax,2; // eax=n-2
        push eax;
        call asm_fib;
        add esp,4;
        add eax,[ebp+0x4]; // eax=fib(n-1)+fib(n-2);
        pop ebp;
        add esp,4;
        ret; // return eax
asm_end:
        add esp,4; // cleanup stack
        mov f,eax;
    }
    return f;
}
```