OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
INF FAKULTÄT FÜR INFORMATIK

Embedded Smart Systems
Jun.-Prof. Dr. Sebastian Zug

# Technische Informatik II - Theoretisches Aufgabenblatt 1

Abgabetermin: ab 5.05.2015

1. Was beschreibt die Von-Neumann Architektur und wie unterscheidet sie sich von der Harvard-Architektur?

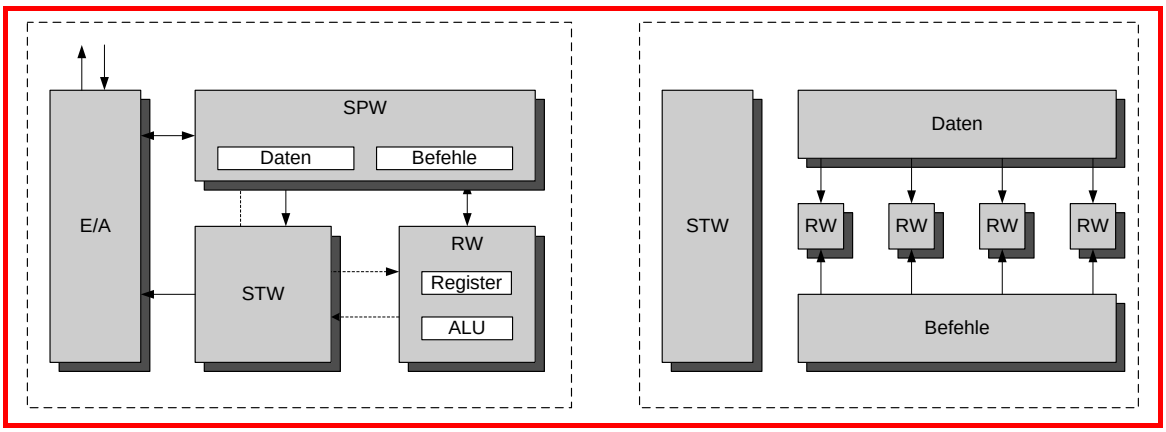



Abbildung 1: links Von-Neumann-Architektur, rechts Harvard-Architektur
STW=Steuerwerk, RW=Rechenwerk, SPW=Speicherwerk

Die **Von-Neumann-Architektur** benannt nach John von Neumann ist ein Schaltungskonzept zur Realisierung universeller Rechner. Sie ermöglicht es, Änderungen an Programmen und dem Betriebssystem sehr schnell durchzuführen und in kurzer Folge verschiedene Programme ablaufen zu lassen, ohne Veränderungen an der Hardware vornehmen zu müssen. Die Von-Neumann-Architektur enthält die folgenden Komponenten:

- Eingabe-/Ausgabewerk zur Steuerung der Ein- und Ausgabe von Daten
- Rechenwerk, das Rechenoperationen und logische Verknüpfungen ausführt (*ALU und Register*)
- Speicherwerk, das sowohl Programme als auch Daten speichert und dem Rechenwerk zugänglich macht
- Steuerwerk, das die Anweisungen eines Programms interpretiert und die Ausführung der Befehle steuert

- Bus der die einzelnen Komponenten verbindet

Eine der wichtigsten Modifikationen der Von-Neumann-Architektur ist die Aufteilung von Befehls- und Datenspeicher gemäß der Harvard-Architektur. Eine klarere Trennung von Befehlen und Daten erhöht die Betriebssicherheit deutlich. Insbesondere Pufferüberläufe werden bei einer starken Trennung von Befehlen und Daten besser beherrschbar.

Ein weiterer Vorteil der Trennung ist, dass die Datenwortbreite und Befehlswortbreite unabhängig voneinander festgelegt werden kann. Das verbessert die Effizienz des Programmspeicherbedarfs, da sie nicht direkt von den Datenbusbreiten abhängt, sondern ausschließlich vom Befehlssatz (*eingebetteten Systemen oder kleinen Microcontroller-Systemen*).

Befehle und zugehörige Daten können in einem einzigen Taktzyklus in das entsprechende Rechenwerk geladen werden. Bei einer klassischen Von-Neumann-Architektur sind hierzu mindestens zwei aufeinander folgende Taktzyklen notwendig.

***Frage an Studenten:***

*Was kann Von-Neumann, was Harvard nicht kann?*

***Antwort:***

*selbst-modifizierender Code, insbesondere Laden von Programmen, siehe AVR: In - System Programming vs. Bootloader*

***Harvard Erweiterung**: Super-Harvard mit speziellen Befehlen zur Verbindung von Daten- und Programmspeicher*

2. Welche Merkmale unterscheiden RISC- und CISC-Rechner? Was war die Intention zu deren Entwicklung? Wo liegen Vor- und Nachteile?

CISC Complex Instruction Set Computing

- Mikroprogrammierung
  - änderbar ↪ Bugfixes möglich
- Mehrzyklenbefehle
- Komplexe Befehle und Addressierungsarten
- Speicher-Register-Architektur
- weniger Register, häufig mit Spezialfunktionen

RISC Reduced Instruction Set Computing

- keine Mikroprogrammierung – fest verdrahtete Hardwarestruktur (schneller)
- keine Mehrzyklenbefehle – Programmausführung deterministisch
- keine komplexen Befehle – Nachbildung in Software
- orthogonale Adressierung
- mehr general-purpose Register

↪ vorgesehen für optimierte Compiler einer Hochsprache

Pentium-Prozessoren besitzen eine vorgeschaltete Funktionseinheit zur Transformation von CISC-Befehlen für eine RISC-Architektur.

3. Auf Seite 31 der Vorlesungsunterlagen wird ein kleines Programmbeispiel für die Anwendung von Pointern vorgestellt. Kompilieren, assemblieren und disassemblieren Sie das Beispiel und erläutern Sie den Code.

```c
#include <stdio.h>
int main(){
    // 0x0804841d <+0>:     push   %ebp
    // 0x0804841e <+1>:     mov    %esp,%ebp
    // 0x08048420 <+3>:     and    $0xfffffff0,%esp
    // 0x08048423 <+6>:     sub    $0x20,%esp

     char a;
     a=5;
    //   0x08048426 <+9>:     movb   $0x5,0x1b(%esp)

     char *p = &a;
     // 0x0804842b <+14>:    lea    0x1b(%esp),%eax
    // 0x0804842f <+18>:     mov    %eax,0x1c(%esp)

     printf("p address %p\n", p);
    // 0x08048433 <+22>:     mov    0x1c(%esp),%eax
    // 0x08048437 <+26>:     mov    %eax,0x4(%esp)
    // 0x0804843b <+30>:     movl   $0x8048530,(%esp)
    // 0x08048442 <+37>:     call   0x80482f0 <printf@plt>

     printf("p content %d\n", *p);
    // 0x08048447 <+42>:     mov    0x1c(%esp),%eax
    // 0x0804844b <+46>:     movzbl (%eax),%eax
    // 0x0804844e <+49>:     movsbl %al,%eax
    // 0x08048451 <+52>:     mov    %eax,0x4(%esp)
    // 0x08048455 <+56>:     movl   $0x804853e,(%esp)
    // 0x0804845c <+63>:     call   0x80482f0 <printf@plt>

    *p=22;
    // 0x08048461 <+68>:     mov    0x1c(%esp),%eax
    // 0x08048465 <+72>:     movb   $0x16,(%eax)

     printf("a address %p\n", &a);
    // 0x08048468 <+75>:     lea    0x1b(%esp),%eax
    // 0x0804846c <+79>:     mov    %eax,0x4(%esp)
    // 0x08048470 <+83>:     movl   $0x804854c,(%esp)
    // 0x08048477 <+90>:     call   0x80482f0 <printf@plt>

     printf("a content %d\n", a);
    // 0x0804847c <+95>:     movzbl 0x1b(%esp),%eax
    // 0x08048481 <+100>:    movsbl %al,%eax
    // 0x08048484 <+103>:    mov    %eax,0x4(%esp)
    // 0x08048488 <+107>:    movl   $0x804855a,(%esp)
    // 0x0804848f <+114>:    call   0x80482f0 <printf@plt>

}
// 0x08048494 <+119>:    leave
// 0x08048495 <+120>:    ret
```

Quelltext 1: Quellcode mit Assembler

4. Welches Problem wird durch den Begriff *Semantisch Lücke* beschrieben und wie kann es gelöst werden?

   Die Semantische Lücke beschreibt den semantischen, also bedeutungsbezogenen Unterschied zwischen zwei Beschreibungen eines Objekts, der dadurch entsteht, dass verschiedene Repräsentationsformen gewählt werden. Dieser in der Informatik verwendete Begriff wird im Allgemeinen dort deutlich, wo ein Abbild des realen Lebens in eine formale, maschinell verarbeitbare Repräsentation übertragen werden muss.

   Präziser bezeichnet der Begriff den Unterschied zwischen Formulierung von Kontextwissen in einer mächtigen Sprache (zB. Natürliche Sprache, bzw. Problem orientierte Sprache) und dessen formaler und automatisiert reproduzierbaren Repräsentation in einer weniger mächtigen formalen Sprache (zB. Programmiersprache bzw. Digitale Logik).

Das Problem der semantischen Lücke kann gelöst werden, indem die mächtigere Sprache sukzessiv eingeschränkt und damit an die darunter liegende Schicht (*weniger mächtige Sprache*) angepasst wird. Gleichzeitig werden Werkzeuge die eine Umwandlung zwischen den Schichten erlauben zur Verfügung gestellt (*Compiler, Assembler, Linker*).

5. Benennen und erläutern Sie mit eigenen Worten die notwendigen Schritte auf dem Weg von der Hochsprache zur Logikebene.

   - Teile und Herrsche – Aufteilung auf mehrere Ebenen (virtuelle Maschinen) mit spezifischen Fähigkeiten
   - Transformation der Programminhalte zwischen den Maschinen

   - **Hochsprache**
   - **Assemblersprache:** Basierend auf mnemonisch codierten Befehlen werden CPUs-pezifische Programmlösungen umgesetzt. Entsprechend bestehen Programme aus Maschinenbefehlen, symbolischen Operanden und Adressierungsarten
   - **Maschinenprogrammebene:** Legt die Betriebsarten des Rechners fest, verwaltet Ressourcen und überwacht insbesondere die Ausführung von Programmen. Programme umfassen Systemaufrufe (OS) und Maschinenbefehle (ISA)
   - **ISA: Befehlssatzeben:** Implementiert das Programmiermodell des Rechners (sichtbare Register) und den Befehlssatz RISC, CISC
   - **Mikroarchitektur:** Beschreibt den Aufbau des Steuerwerks und des Datenpfades im Rechner, Programme setzen sich aus den Konstrukten einer Hardwarebeschreibungssprache (z.B. VHDL) zusammen
   - **Logikebene**

6. Vergleiche *Betriebssysteme* und *Virtuelle Maschinen* worin bestehen Gemeinsamkeiten und was sind Unterschiede?

   Sowohl *Betriebssysteme* als auch *Virtuelle Maschinen* abstrahieren das darunterliegende System und stellen dem Anwender eine definierte Schnittstelle zur Verfügung. Der Unterschied besteht darin, dass *Betriebssysteme* ein Softwareschnittstelle und *Virtuelle Maschinen* eine Hardwareschnittstelle zur Verfügung stellen. Ein weiterer Unterschied ist die Möglichkeit das *Virtuelle Maschinen* in anderen *Virtuelle Maschinen* ausgeführt werden können. Dies ist bei *Betriebssystemen* nicht möglich.

7. Gegeben seien die 32-Bit-Assembler-Quelltexte 2 und 3, welche in AT&T-Syntax gehalten sind. Welche C-Konstrukte stellen diese Ausschnitte jeweils dar?

```
1   movl    $0x0,0x18(%esp)
2   jmp     pos2
3 pos1:
4   mov     0x18(%esp),%eax
5   mov     %eax,(%esp)
6   call    func
7   addl    $0x1,0x18(%esp)
8 pos2:
9   cmpl    $0x9,0x18(%esp)
10  jle     pos1
```

Quelltext 2: Assemblerquelltext zur Analyse I

```
1   cmpl    $0x5,0x18(%esp)
2   jne     pos3
3   call    doSomething
4 pos3:
```

Quelltext 3: Assemblerquelltext zur Analyse II

Die Lösung zu 2 ist in 4 dargestellt. Es handelt sich dabei um eine FOR-Schleife.

```
1   int i;
2   for(i=0;i<10;i++){
3     func(i);
4   }
```

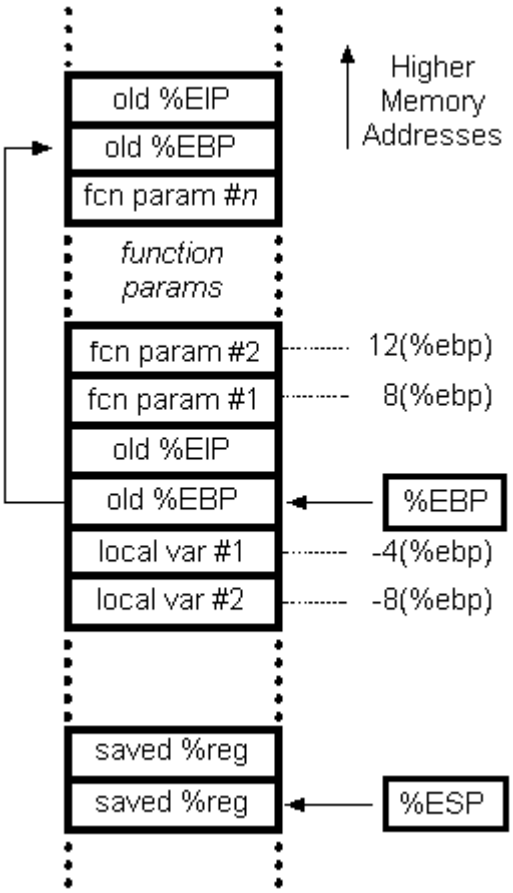
Quelltext 4: Lsg. zur Analyse I

Die Lösung zu 3 ist in 5 dargestellt. Es handelt sich dabei um eine IF-Anweisung.

```
1   if(i==5){
2     doSomething();
3   }
```

Quelltext 5: Lsg. zur Analyse II

8. Erläutern Sie die Verwendung des Framepointers bei der Addressierung im Stack.




Framepointer (%ebp) zeigt auf Anfang des aktuellen Stackframes. Dort (am Anfang) liegt der Framepointer zum vorherigen Stackframe. Man verwendet den %ebp um auf Funktionsparameter (z.B. 8(%ebp)) und lokalen Speicher (z.B. -8(%ebp)) zuzugreifen

9. Gegeben sei ein Auszug aus einem Assemblerquelltext, in dem eine Funktion aufgerufen wird.

```
1   ...
2   push %ebx
3   pushl $0x1337
4   push %eax
5   push %ecx
6   call func
7   addl  $12, %esp
8  L2:
9   ...
10 func:
11  push %ebp
12  movl %esp, %ebp
13  subl  $0x12, %esp
14 L1:
15  ...
16  leave
17  ret
18  ...
```

Quelltext 6: Assemblercode für eine Stack-Analyse

Beschreibe, wie viele *Parameter/lokale Variablen* die Funktion hat und was während des Funktionsaufrufs passiert.

**Parameter** zu erkennen am Aufruf zum Aufräumen des Stacks
`add esp, 12` -> 12 Byte bei 4 Byte pro Eintrag = 3 Einträge

**lokale Variablen** zu erkennen am Umsetzen des Stackpointers (ESP)
`sub esp, 12h` -> 12h = 18 Byte lokale Variablen

10. Welche Alternativen bestehen zur Parameterübergrabe an ein Unterprogramm mittels Stack?

- globale Variablen
- Übergabe mittels Register (AMD64 Standardkonfiguration im gcc)
- gemeinsamer Speicher (überlappung im virtuellen Adressraum)
- Kommunikationsbezogen (sockets, Interprozesskommunikation)