Autor: Sebastian Sossalla

Disclaimer: Das sind Lösungsvorschläge, keine Musterlösungen.

## Aufgabe 1: Allgemeines (7 Punkte)

**1a) Welche Aussagen sind wahr? (3 Punkte; Punktabzug bei falschen Antworten!)**

x CPU-Exceptions sind synchron zu dem laufenden Programm.

❒ Bei einer CPU-Exception schaltet der Prozessor automatisch in den User Modus.
(falsch, er geht in den privilegierten Modus)

❒ Jede CPU sorgt dafür, dass die Register eines Programmes nicht durch einen plötzlich auftretenden Interrupt verändert werden.
(nicht sicher, denke eher nicht!)

x Ein Interrupt Handler darf den Befehl `cli` ausführen.
(er ist im privilegierten Modus)

x Bei einem Interrupt wird das Flags-Register gespeichert.
( das eflags-Register wird gespeichert)

x Die Interrupt Vektor Tabelle muss im Hauptspeicher abgelegt werden.

x Die System Call Vektor Tabelle muss initialisiert sein, bevor ein Programm einen System Call ausführen kann.
(müsste aber vom Betriebssystem automatisch zu begin initialisiert werden!)

❒ Ein System Call Handler muss alle zu verändernden Register sichern und vor dem Ausführen von `iret` wieder restaurieren
(das is nicht nötig, würde ich sagen)

**1b) Warum ist es an gewissen Stellen im Betriebsssytem sinnvoll, die Interrupts kurzfristig abzuschalten? (1 Punkt)**

Damit in diesem Moment ausgeführte Befehle nicht unterbrochen werden. Bei der Synchronisation, um kritische Bereiche zu sichern. Nebenläufigkeit in kritischen Abschnitten, zeitkritische Bereiche.

Das ist bei Zeit- und Nebenläufigkeits-kritischen Bereichen sinnvoll, wenn diese Code-fragmente nicht durch Interrupts gestört wreden dürfen. Beispiel: In einem Interrupt-Handler darf kein weiterer Interrupt auftreten. Oder: Es wird im Hauptprogramm aus dem Puffer ausgelesen, und ein Interrupt-handler möchte in den Puffer schreiben => lese-routine ist kritischer Abschnitte

**1c) Beschreiben Sie, wofür die folgenden Prozessor-Flags beispielsweise Verwendung finden (3 Punkte):**

**i) Carry-Flag**

Übertrag aus dem höchstwertigsten Bit bei Addition oder Subtraktion (Borrow) => sequentielle Addition und Subtraktion mit größerer Wortbreite ist möglich.

**ii) Zero-Flag, Negative-Flag**

Zero-Flag zeigt an, ob das Ergebnis der letzten Operation gleich 0 war
(bedingte Programmverzweigungen, Zählschleifen)
Negative-Flag: berechnetes Ergebnis ist negativ!

**iii) Interrupt-Flag**

Zeigt an, ob Interrupts erlaubt sind, oder nicht! Interrupt-Enable-Bit!

**iv) User-Mode-Flag: damit man feststellen kann, ob Prozessor**

User-code oder System-code ausfuehrt.

## Aufgabe 2: Speicher (10 Punkte)

**2a) In einer Stuktur werden die Elemente i=-1,c=64 und n=4711 gespeichert. Ein Speicherauszug sieht folgendermaßen aus (alle Bytes hexadezimal):**
**`40 64 ff ff 67 12 00 00`**
**Wie sieht die Struktur aus? Welche Byte-Order hat der verwendete Rechner? (3 Punkte)**

64 = 2 * 32 = 4 * 16 = 0x40
i = -1 = 0xff ff
0x 00 00 12 67 = 16^3 + 2*16^2 + 6 * 16 + 7 =

Die Byte-Order ist: little-Endian:
0x4711 = 00 00 12 67 => das niederwertigste Bit an höchster Stelle.

Struktur:

```
struct {
        char c;
        short int i;
        int i;
}
```

**2b) Warum sollte ein Assembler-Programmierer die Byte-Order (Little-Endian bzw. Big-Endian) seines Rechners kennen, während die Bit-Order des Rechners im allgemeinen keine Rolle spielt? (2 Punkte)**

Die Bit-Order ist deshalb ohne Bedeutung, da die kleinste Einheit, auf die man zugreifen kann ein Byte ist. Um alles darunter kümmert sich die Hardware. Es spielt für den Programmierer also sehr wohl eine Rolle, in welcher Reihenfolge er die Bytes im Speicher vorfindet. Die Bit-Order ist egal, darum kümmert sich die Hardware.

**2c) Schreiben Sie ein Unterprogramm `is_big_endian`, das testet, ob der verwendete Rechner ein Big-Endian-Rechner ist! Das Test-Ergebnis soll als Boolscher Wert an das aufrufende Programm zurückgegeben werden. (5 Punkte)**

```
is_big_endian:
    subl $4, %esp
    movl $0x00 00 00 01,(%esp)
    xorl %eax, %eax
    movb (%esp), %al // %al - linkestes Bit holen!!!
    addl $4, %esp
    ret

oder:

int is_big_endian(void) {
  int i = 0x01000000;
```

```
  return (int) *((char *) &i);
}
```

## Aufgabe 3: Arithmetik (17 Punkte)

3a) Welche Zahl steht nach Ausführung der folgenden Befehle im Register %eax? Welche Bedeutung hat diese Zahl? (2 Punkte)

```
movl $0, %eax
notl %eax
shrl $1, %eax
```

```
eax = 0;
eax = 1111 1111 1111 1111 1111 1111 1111 1111
eax = 0111 1111 1111 1111 1111 1111 1111 1111 oder:
eax = 0x7FFFFFFF
```

```
Das ist die letzte positive signierte Zahl im 2-er Komplement(bzw.
Vorzeichen-/Betragsdarstellung) die man mit 32 Bit darstellen
kann.
```

3b) Compilieren Sie die folgende Hochsprachen-Funktion in Assembler-Code! Benutzen Sie keine Multiplikations- bzw. Divisions- oder Modulo-Befehle! Optimieren Sie Ihr Programm! (7 Punkte)

Hinweis 1: der '%'-Operator ist in C, C++ und Java der Modulo-Operator.
Hinweis 2: die optimale Lösung für Intel-Prozessoren benötigt 3 Maschinen-Instruktionen.

```
unsigned int odd(unsigned int x)
{
     if (x % 3 == 0) {
          return 0;
     } else {
          return 1;
     }
}
```

```
odd:
     enter $4
     movl 8(%ebp), %ebx
     shrl $1, %ebx
     shll $1, %ebx
     movl 8(%ebp), %eax
     subl %ebx, %eax
     cmpl $0, %eax
     jne _not
     movl $0, %eax
_not:
     movl $1, %eax
     leave
     ret
```

```
odd:
 movl 4(%esp),%eax
 andl $1, %eax
 ret
```

**3c) Konvertieren Sie folgende Hochsprachen-Funktion in semantisch äquivalenten Hochsprachen-Code, so dass sich dieser möglichst leicht in Assembler-Code für eine Ein-Address-Maschineumwandeln lässt! (8 Punkte)**

**Hinweis: Ein-Adress-Maschine bedeutet, dass bei Berechnungen nur Ausdrücke der Form Akku = Akku *op* Operand erlaubt sind (*op* ist Element von {+, -, *, /}, Operand kann eine Variable oder eine Konstante sein.**

```
int test(int n)
{
     int i;
     for (i = 0; i < 2 * n; i++) {
          if (func(n + i * i)) {
               return i;
     }
}
     return -1;
}

int test(int n) {
     int i;
     int akku;

     i = 0;

_test: akku = n;
       akku +=akku;
       if (i < akku) goto loop;
       goto _end;

_loop:
          akku = i;
          akku += akku;
          akku += n;
          akku = func(akku)
          if (akku) {
               return i;
          }
         goto _test;
_end:     return;
}
```

## Aufgabe 4: Unterprogramme (18 Punkte)

Gegeben sei folgendes Hochsprachen-Unterprogramm:

```
unsigned int isqrt(unsigned int x)
{
     unsigned int i;
     i = 0;

     while (i * i < x) {
          i++;
     }
     return i;
}
```

**Ein Compiler habe daraus den folgenden (lückenhaften) Assembler-Code generiert:**

```
.align 4
isqrt: .globl isqrt
     movl 4(%esp),%ecx
     xorl %edx,%edx        /*edx = i*/
     cmpl %ecx,%edx       /*if ( i >= x)*/
     jae .L4
.align 4
.L5:
     incl %edx
     movl %edx,%eax
     imull %eax,%eax
     cmpl %ecx, %eax
     jb .L5
.L4:
     movl %edx,%eax
     ret
```

**4a) Vervollständigen Sie den Assembler-Code! (5 Punkte; Punktabzug bei sinnlosen Einfügungen)**

schon passiert - siehe oben!

**4b) Was könnten die beiden Pseudo-Befehle `.align 4` bedeuten? Wofür sind sie sinnvoll? Warum stehen sie genau an diesen Stellen? (2 Punkte)**

Der Befehl bewirkt eine Anordnung des des nachfolgenden Befehls auf einer durch 4 teilbaren Adresse.

Der Grund, warum die Direktiven nun ausgerechnet an diesen Stellen stehen, ist der, dass zum einen der Beginn des Unterprogramms an einer gut anzuspringenden Adresse liegen soll (und die einzelnen Befehle sind mit Operanden auch meist 2 oder 4 Byte lang und liegen somit ebenfalls an "guten" Adressen ) und zum Anderen, bestimmt die zweite .align-Direktive, dass die Adresse, an der die Schleife, die häufig bzw. öfter angesprungen wird, ebenfalls an einer ordentlichen Adresse liegt.

Im gegebenen Programmstück werden also 2 Dummy-Byte (NOP) zwischen jae und incl eingefügt. (Kann man erkennen, wenn man den Assemblercode zunächst zu einer Object-Datei kompiliert und dann mit objdump -d diese wieder disassembliert.)

**4c) Schätzen Sie ab, wieviel Byte Speicherplatz das Assembler-Code-Stück belegt! Begründen Sie**
Ihre Antwort! (5 Punkte)

```
.align 4
isqrt: .globl isqrt
     movl 4(%esp),%ecx
     xorl %edx,%edx        /*edx = i*/
     cmpl %ecx,%edx       /*if ( i >= x)*/
     jae .L4
.align 4
.L5:
     incl %edx
     movl %edx,%eax
     imull %eax,%eax
     cmpl %ecx, %eax
     jb .L5
.L4:
     movl %edx,%eax
     ret
```

128 Befehle: - 7 Bit codierung,
- 3 datentypen: long, byte, word also 2 Bit
- Codierung etc: 8 Register: 3 Bit codierung,
- 8 Adressierungsarten für Register: 3 Bit codierung,
- 32 Bit für Adressen, Konstanten,

movl 4(%esp),%ecx Bits: 7 + 3 + 3 + 3 + 3 = 19 Bit ~ 3 Byte

```
     xorl %edx,%edx  Bits: 7 + 3 + 3 + 3 = 2 Byte
     cmpl...    = 2 Byte
     jae .L4 = 7 + 32 = 39 ~5 Byte
.align 4:  7 Bit befehls-codierung + 32 Bit konstante =  39 Bit =
          3 Byte
     incl %edx: 7 + 3 + 3 = 13 Bit ~2 Byte
Insgesamt: 3 + 2 + 2 + 5 + 3 + 2 + 2 + 2 + 2 + 5 + 2 + 1 ~
14 + 6 + 10 + 1 ~ 31 Bytes
```

**4d) Während der Initialisierung eines PCs wird u.a. folgender BIOS-Code (ROM) durchlaufen:**

```
...
     movw $L1, %sp
     jmp L3
L1: .word L2
L2: ...
L3: ...
```

```
ret
```

**Erklären Sie die Besonderheit dieses Code-Stückes! Was ist der Sinn dieser auf den ersten Blick umständlichen Programmierweise? (6 Punkte)**

**Das ist ein manueller Unterprogramm-Aufruf. Das bedeuted: Rücksprungadresse wird nicht auf dem Stack gespeichert, sondern es wird in das %sp-register wird eine ROM-Adresse geschoben, an der die rücksprungadresse hinterlegt ist. Weil wir im Rom sind können wir nicht einfach nen Stack anlegen oder auf selbigen schieben. Sinn machen würde deses Code-Fragment zum Beispiel bei einem ROM-Speichertest bei der Initialisierung eines Rechners.**

## Aufgabe 5: Segmentierung / Paging (11 Punkte)

### 5a) Was sind die grundsätzlichen Unterschiede zwischen Segmentierung und Paging? (3 Punkte)

Beim Paging werden logischer und physischer Adreßraum in Abschnitte gleicher Länge, die man Seiten (Pages) nett, aufgeteilt. Das Betriebssystem gibt die Größe der Seiten vor. Die Seitentabellen ordnen jeder virtuellen Adresse einer Seite eine physikalische Adresse zu.
Das Paging wird transparent, also für den Programmierer nicht sichtbar hinter die Segmentierung geschalten.

Segmente sind dabei Speicherabgrenzungen unterschiedlicher Größe. Segmente bestehen dabei wieder aus Seiten. Bei Segmentierung müssen neben den Anfangs-Adressen des Segments auch die Segmentlänge abgespeichert werden. Bei den Pages ist das nicht nötig, da sie Größenmäßig nicht variieren. Zusätzlich kann für Segmente eigene Schutzvorkehrungen getroffen werden (Read-Only, Write-Only, Execute etc..)

### 5b) Weshalb werden mehrstufige Seitentabellen verwendet? (2 Punkte)

**Kleinere Speichertabellen und wesentlich bessere Speicherausnutzung durch mehrstufige Adressierung. Teil der Seitentabellen kann ausgelagert werden.**
**Dadurch wird der Zugriff auf die gesuchte Page viel schneller möglich, als ohne. Ohne mehrstufige Seitentabellen müsste sequentiell gesucht werden.**

### 5c) Ein 32 Bit System verwendet zweistufige Seitentabellen mit jeweils 1024 Einträgen. Wie groß ist eine Seite? (6 Punkte)

1024 = 2^10: 32 bit - 10 bit - 10 bit = 12 bit: 2^12 = 2^2 * 2^10 = 4 KB

## Aufgabe 6: Interrupts (8 Punkte)

Beschreiben Sie, welche Schritte beim Auftreten eines Interrupts im User-Mode in einer CPU ablaufen, die zwischen User- und Supervisor-Mode unterscheidet! Achten Sie gegebenenfalls auf die Reihenfolge der Schritte! Welche Aktionen können gleichzeitig geschehen, welche Aktionen müssen nacheinander ausgeführt werden?

1. Interrupt-Request wird erzeugt
2. CPU prüft, ob der Interrupt gültig ist (maskiert oder nicht) und nimmt ggf. an.

(Interrupt-Anforderungen bleiben so lange bestehen, bis sie quittiert werden.!)
3. CPU schaut nach, ob Interrupt-Enable-Bit auf 0 ist: Falls "JA" CPU quittiert den Prozess
4. Abspeichern der Instruction-Pointers, einiger Register wie CC-Register, Abspeichern des Current-Privilege-Levels, Stackpointer für den Reibungslosen Rücksprung auf den Stack
5. Wechseln zu Supervisormodus.)
6. Sprung in die Interrupt-Service-Routine
7. Service-Routine sichert alle benutzen Register, damit Unterbrochene Programme nicht gestört werden und setzt das Interupt-Enable-Bit auf 0
8. Nach Ausführung der Routine: Laden einiger Register (z.B. neuer Wert für das Interrupt-Enable-Bit, Laden des neuen Current-Privilege-Levels)
9. Laden des Instruction-Pointers
10. Wiederaufnahme des Unterbrochenen Programms

## Aufgabe 7: RAM / ROM (9 Punkte)

**Fragen zum Anschluss von RAM- bzw. ROM-Bausteinen an den Main-Bus des Rechners. Begründen Sie Ihre Antworten!**

**7a) Was könnte passieren, wenn man beim Anschließen eines RAM-Bausteins an den Main-Bus eines Rechners aus Versehen Adressleitungen des RAM-Bausteins an falsche Adressleitungen des Main-Busses anschließt? Wenn man zum Beispiel Adressleitung A0 des RAM-Bausteins an A1 des Main-Busses und A1 des RAM-Bausteins an A0 des Main-Busses anschließt? (2 Punkte)**

Lösung:
Gar nix. Man schreibt zwar an die falsche Adresse. Aber man liest genau die zuletzt falsch beschriebene Adresse wieder aus.

**7b) Was passiert, wenn man Datenleitungen untereinander vertauscht? (2 Punkte)**

Lösung:
Wieder nix: Die Bit-Order ändert sich zwar, aber die falsch abgespeicherte Bit-Order wird wieder genauso falsch ausgelesen.

**7c) Was könnte passieren, wenn man versehenlich Adress- mit Datenleitungen vertauscht? (2 Punkte)**

Das würde nicht gut gehen. Sobald eine Adressleitung mit einer Datenleitung vertauscht wurde, funktioniert überhaupt nix mehr.

**7d) Was könnte passieren, wenn man bei einem ROM-Baustein Adressleitungen bzw. Datenleitungen miteinander vertauscht? (3 Punkte)**

Bei einem ROM-Baustein stehen bereits Daten fest drin. Er ist nicht beschreibbar. Wenn jetzt Adressleitungen oder Datenleitungen vertauscht wereden, werden völlig falsche Daten ausgelesen, die man eigentlich haben möchte (bei Adressleitungen) bzw. Die Daten in einer falschen Bit-Order ausgelesen.

## Aufgabe 8: Hardware (10 Punkte)

Es soll ein Inkrementierer in Hardware aufgebaut werden. Als Eingabe werden 4 Leitungen angenommen.

Als Ausgabe sollen 4 Leitungen für die Signalisierung des normalen Ergebnisses dienen. Zusätzlich ist ein Übertrag über eine fünfte Leitung zu signalisieren.
Für die Implementierung des Schaltnetzes dürfen AND-, OR-, XOR- sowie NOT-Gatter verwendet werden.

**Lösung:**

Wurde bereits gelöst in Klausur_2007.pdf