Autor: Sebastian Sossalla

Disclaimer: Das sind Lösungsvorschläge, keine Musterlösungen.

Aufgabe 1: Allgemeines (22 Punkte)

1. Welche der nachfolgenden Informationen werden i.a. im Condition-Code-Register gespeichert? (4 Punkte; Punktabzug bei falschen Antworten!)

gespeichert nicht gespeichert
<X> < > Carry-Flag
< > <X> Relative-Flag
<X> < > Zero-Flag
< > <X> Address-Flag
< > <X> Index
<X> < > Overflow-Flag
< > <X> Exception-Flag
<X> < > Negative-Flag

2. Welche Adressierungsarten kennen Sie? (4 Punkte; Punktabzug bei falschen
Antworten!)

Register
Direct Adress
Register Indirect
immediate Operand
Register Indirect with displacement
register indirect with Index
Register Indirect with displacement and Index
Memory Indirect
Register Indirect with Pre-/Post-Decrement
Register Indirect with Pre-/Post-Increment

3.Welche Aussagen sind wahr, welche sind falsch? (4 Punkte; Punktabzug bei falschen Antworten!)

wahr falsch

<x> < > Bei Verwendung des Zweierkomplements ist neben der Additions- keine gesonderte Subtraktionseinheit erforderlich.

<x> < > Das Statusregister gibt u.A. Auskunft über das Vorzeichen eines Additionsergebnisses.

< > <x?> Zur Berechnung der Winkelfunktionen müssen Rechner grundsätzlich über ein Gleitpunkt-Rechenwerk verfügen.

< > <x> Die Verwendung von absoluten Sprüngen verbessert die Verschiebbarkeit des Codes im Speicher.

< > <x?> Zur Realisierung von Unterprogrammaufrufen ist zwingend ein Stack erforderlich.

< > <x> Funktionsparameter können nur über den Stack übergeben werden.

<x> < > Zur Kommunikation mit Ein-/Ausgabegeräten sind immer spezielle Assembler-Instruktionen erforderlich.

< > <x> Die Verwendung von Alignments kann Sprünge beschleunigen.

4. Welche Aussagen sind wahr, welche sind falsch? (4 Punkte; Punktabzug bei falschen Antworten!)

wahr falsch

< > <x> Die Programmlänge hängt immer nur von der Anzahl der Instruktionen ab.

<x> < > Eine Bereichsüberschreitung bei Addition löst eine Exception aus.

<x> < > Exceptions finden synchron zum Programmablauf statt.

< > <x> Bei einem Interrupt sichert die CPU den gesamten Prozessorzustand.

<x> < > Die Seitengröße beim Paging hängt in erster Linie von der Adressbusbreite ab.

<x> < > Bei Paging können Seiten als „nur lesbar" deklariert werden.

< > <x> Mehrstufige Seitentabellen sorgen für eine bessere Speicherausnutzung.

<x> < > Bei der Verwendung mehrstufiger Seitentabellen kann die Berechnung der physikalischen Adresse besonders schnell erfolgen.

5. Begründen Sie, warum Werte zwischen 1 und 255, die man im Big-Endian-Format abspeichert, zu großen Werten (> 256) werden, wenn man sie im Little-Endian-Format wieder liest. (2 Punkte)

<u>Beispiel:</u>
Big-Endian-Zahl: 00 00 00 01 ist im Big-Endian-format Wert: 1
im Little-Endian: 01 00 00 00 ist jetzt der Wert: 16^6 ~ 16 mio.

<u>Begründung:</u> wie man im Beispiel sehen kann, dreht sich die Byte-reihenfolge ohne entsprechende Vorkehrungsmaßnahmen einfach um. Dadurch dreht sich die Byte-Order plötzlich um. Und alleine mit einer Big-Endian-Zahl mit dem Wert „1“ kommt als Little-Endian-Interpretation eine zahl weit über 256 heraus.

6. Welche der nachfolgenden Informationen werden i.a. beim Speicherzugriff in der Paging-Einheit zur Erlaubnisprüfung genutzt? (4 Punkte; Punktabzug bei falschen Antworten!)

genutzt	nicht genutzt
< >	<x> Segment-Nummer
<x>	< > Read-only-Bit in Page-Tabelle
<x>	< > aktueller Privilege-Level
< >	<x> Zero-Bit
< >	<x> Adressierungsart
< >	<x> virtuelle Adresse
< >	<x> physikalische Adresse
<x>	< > Read- oder Write-Zugriff

Aufgabe 2: Speicher (5 Punkte)

1. In einer Struktur werden die Elemente x = -2, y = 32 und z = 12345 gespeichert. Ein Speicherauszug sieht folgendermaßen aus (alle Bytes im 16er-System): 20 40 fe ff 39 30 00 00
Wie sieht die Struktur aus? Welche Byte-Order hat der verwendete Rechner? Begründen Sie Ihre Antworten! (3 Punkte)

Lösung:
x = -2 (-1 = FFF....FFF, => -2 ist: FFF....FFE)
y = 32 (32 = 0x20, 2*16^1 + 0*16^0)
z = 12345

20 40 fe ff 39 30 00 00
y = 32 = 0x20 muss irgendwo auftauchen. Also ist der 1. Wert: y!
X muss irgendwo auftauchen: ...FFE. Das steckt im 3. und 4. Byte. Daraus kann man erkennen, dass es sich um ein Little-Endian-Format handeln muss: fe ff = ff fe. Dann das 5., 6., 7. und 8. Byte die Zahl „12345“ sein.
00 00 30 39 = 3*16^3 + 3*16 + 9 = 12345
das 3. und 4. Byte, muss also ein Dummy sein:

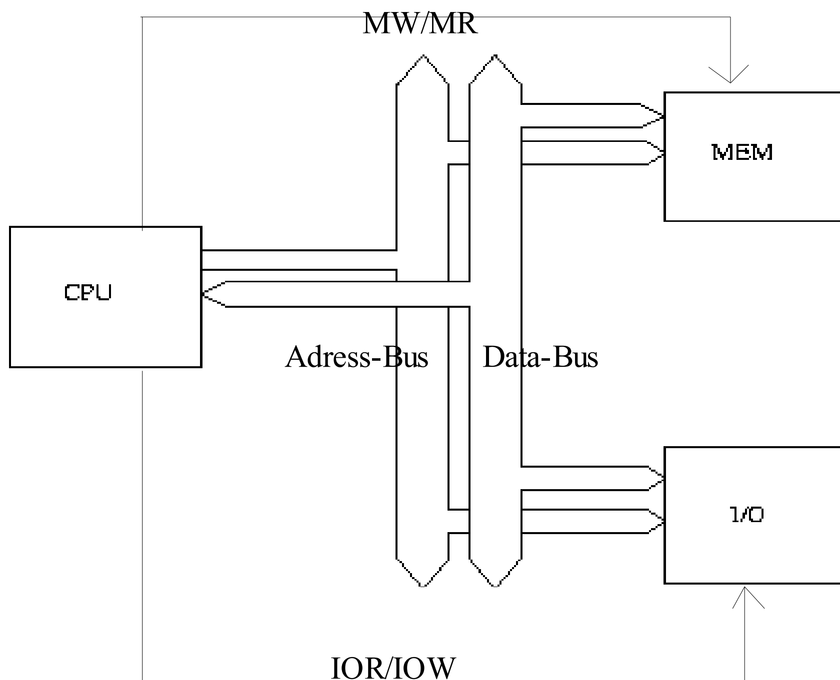
```
struct {
   char y;  /* 1 Byte*/
   short x;  /* 2 Byte */
   long int z; /* 4 Byte, kann aber auch word sein!!!*/
}
```

2. Unter welchen Voraussetzungen kann ein Hochsprachen-Programmierer die Byte-Reihenfolge seiner Daten aus seinem Programm heraus feststellen?
(2 Punkte)

Hier bin ich mir nicht sicher, was die Lösung ist.

Aufgabe 3: Hardware (14 Punkte)

Gegeben sei das folgende unvollständige Modell eines Systembusses:



1. Zeichnen Sie die fehlenden Signalleitungen ein und vervollständigen Sie die Beschriftungen. Achten Sie auch auf die Kommunikationsrichtungen. (4 Punkte)

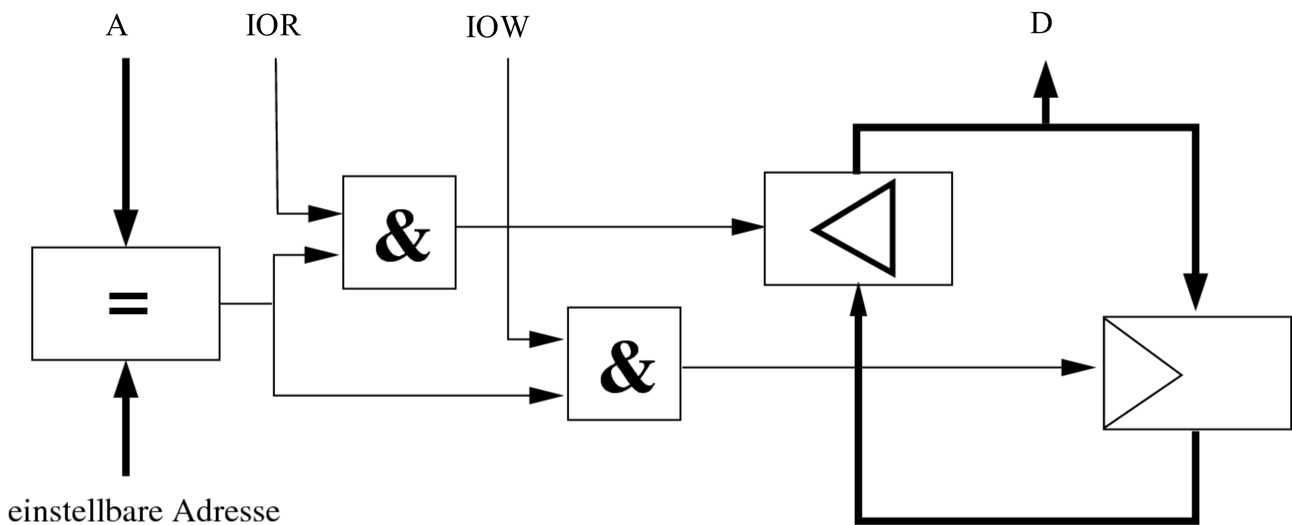
2. Welche Signale könnten eingespart bzw. zusammengelegt werden? Erläutern. Sie kurz, wie das bewerkstelligt werden müsste. (4 Punkte)

TODO:
man könnte IOR/IOW MR/MW zusammenlegen auf einen einzigen Steuerbus. Dann müsste man die Signale dementsprechend Codieren mit 3 Bit: Beispiel: 000 = nix! 001 = IOR, 010 = IOW, 011 = MR 100 MW. Jenachdem, welches Bit-folge durch den Steuerbus gejagt wird, könnte dementsprechend reagiert werden.

Alternative: Man könnte MR und IOR zusammenlegen. Anhand der Adresse kann unterschieden werden, welches Device angesprochen wird. Dasselbe für MW und IOW!

Aufgabe 3: Hardware (Fortsetzung)

3. Zeichnen Sie den Schaltplan eines möglichen Aufbaus eines I/O-Gerätes, das einen 1-Bit-Sensor auslesen und einen 1-Bit-Aktor steuern kann! (6 Punkte)



Dicke Linnien sind Mehrbit-Leitungen!

Aufgabe 4: Ausdrücke (15 Punkte)

Schreiben Sie eine Assembler-Funktion, die zwei Fließkomma-Zahlen miteinander multipliziert. (Überläufe dürfen ignoriert werden, beide Operande nicht 0.) Gehen Sie davon aus, dass der verwendete Prozessor Integer-Additions-, -Subtraktions-, -Multiplikations-, und -Divisions-Operationen sowie Shift- und Logische Operationen für 32-Bit-Arithmetik bietet.

Die Fließkomma-Zahlen seien wie folgt kodiert:

- Bit 31: Vorzeichen
- Bit 30-16: Charakteristik
- Bit 15-0: Mantisse (einschließlich der führenden „1”)

Die zwei zu multiplizierenden Zahlen werden auf dem Stack übergeben und das Ergebnis wird im Register %eax zurückerwartet.

eine Lösungsmöglichkeit:

Prinzipielles Vorgehen ist wie gesagt: Mantissen multiplizieren, Exponenten addieren. Die Vorzeichen kann man verXORen, denn:

```
V1 V2 | V1 * V2
------+--------
+  +  |   +
+  -  |   -
-  +  |   -
-  -  |   +
```

Das entspricht genau der XOR-Operation, wenn das Vorzeichenbit für positive Zahlen gleich 0 ist und für negative gleich 1.

Man muss also die drei Komponenten getrennt voneinander behandeln und am Ende durch Veroderungen wieder zusammenfügen, nach dem Schema:

```
   0000xxxx
OR 0yyy0000
OR z0000000
-----------
=  zyyyxxxx
```

Damit das funktioniert, müssen bei allen Komponenten vorher die Bereiche, die bei der Veroderung jeweils nicht berücksichtigt werden dürfen, ausgenullt werden. Das geschieht dadurch, dass man mit der richtigen Bitmaske verundet. `0x0000ffff` heißt z.B., dass die oberen 16 Bits ausgenullt werden; bei `0x7fff0000` werden das oberste Bit und die unteren 16 Bits entfernt.

```
Multiply: .globl Multiply

/* Mantisse der ersten Zahl laden (%eax) */
        movl 4(%esp), %eax

 /* Mantisse der zweiten Zahl laden (%ebx) */
        movl 8(%esp), %ebx

/* Mantissen miteinander multiplizieren (%ax) */
        imulw %bx, %ax
        andl $0x0000ffff, %eax

 /* Exponent der ersten Zahl laden, Vorzeichenbit entfernen (%ebx) */
        movl 4(%esp), %ebx
        andl $0x7fff0000, %ebx

 /* Exponent der zweiten Zahl laden, Vorzeichenbit entfernen (%ecx) */
        movl 8(%esp), %ecx
        andl $0x7fff0000, %ecx
```

```
/* Exponenten miteinander addieren, in Ergebnis einfuegen (%eax) */
        addl %ecx, %ebx
        orl %ebx, %eax

/* Vorzeichen per XOR miteinander verknuepfen (%ebx) */
        movl 4(%esp), %ebx
        xorl 8(%esp), %ebx

/* Neues Vorzeichen in Ergebnis einfuegen (%eax) */
        andl $0x800000, %ebx
        orl %ebx, %eax

/* Ruecksprung */
        ret
}
```

Aufgabe 5: System-Calls (8 Punkte)

1. Warum stellen viele Hardware-Plattformen einen Mechanismus für Systemaufrufe zur Verfügung? (Hinweis: gehen Sie in Ihrer Begründung auf Gemeinsamkeiten und Unterschiede von Systemcalls und Funktionsaufrufen ein!) (5 Punkte)

Hintergrund:
Bei modernen Plattformen läuft der Kernel-prozess im privilegierten Bereich. Ein Benutzer-prozess hingegen läuft nur im unprivilegierten Zustand, wo ihnen weniger Befehler zur Verfügung stehen, und bestimmte Arbeiten nicht direkt erledigt werden können. Muss der Benutzer-Prozess eine privilegirte Aufgabe erfüllen, wie z.B. Zugriff auf die Festplatte, so wird bewusst eine Exception ausgelöst, und der Exception-Handler des Betriebssystems ruft so eine System-Funktion auf. Der User-Prozess gibt solange, wie das Betriebssystem diese Aufgabe bearbeitet, seine CPU-Kontrolle ab.

Unterschiede: Ein normaler Funktiosaufruf im Vergleich jedoch, wird weiterhin im unprivilegierten Modus ausgeführt, und gehört sozusagen zum User-Prozess. Außerdem benötigt man bei einem normalen Funktionsaufruf eine feste Adresse! Diese ändert sich aber, nach Updates oder anderen Modifikationen des Systems. Also wird bewusst eine Exception ausgelöst und lässt den Exception-Handler die System-routine erledigen.

Warum: zu keiner Zeit verlässt der User-Prozess den unprivilegierten Ring und kann weder andere Prozesse, noch den Kernel-Prozess selbst gefährden, da vertrauenswürdiger Code aus dem Kernel im privilegierten Modus ausgeführt

wird.

Systemcalls sind Betriebssystem-routinen (privilegierter Modus)
normale Funktionen hingegen laufen immer unprivilegiert ab.

2. Welche zwei mit Systemcalls verwandten Konzepte gibt es noch? Grenzen Sie die drei Konzepte voneinander ab! (3 Punkte)

Interrupts:
Sozusagen eine Unterbrechung von außen: Ein Bestimmtes Ereignis wird signalisiert worauf ein Interrupt-handler ausgeführt wird. Interrupts finden völlig asynchron zur Programmausführung statt.
Interrupt-handler haben den Aufbau:

- normale Unterprogramme
- kein Parameter
- kein Rückgabewert

und dürfen keine Register verändern.

Exceptions:
Eine Exception wird ausgelöst, wenn ein Fehler zur Programmlaufzeit auftritt. Beispiel: Division durch Null. Exceptio finden synchron (reproduzierbar) zur Programmausführung statt.
Exception-Handler haben den Aufbau:

- normale Unterprogramme
- z.T. mit Parametern
- ohne Rückgabewerte
- aber meist mit nebeneffekten

und dürfen Register-und Speicherwerte verändern!

Aufgabe 6: Unterprogramme (16 Punkte)

Ein Compiler hat aus einem Hochsprachen-Unterprogramm den folgenden Assembler-Code generiert:

```
func:
        subl $4, %esp
        movl $0, (%esp)
.L2:
        movl (%esp), %eax
        imull (%esp), %eax
        cmpl 8(%esp), %eax
        jb .L4
```

```
        jmp .L3
.L4:
        incl (%esp)
        jmp .L2
.L3:
        movl (%esp), %eax
        addl $4, %esp
ret
```

1. Wie könnte der zugrundeliegende C-Code ausgesehen haben? (6 Punkte)
Basti's Lösung:

```
unsigned int func ( unsigned int y) {
    unsigned int x = 0;

    do {
        x= x*x;
    } while( x++ < y);
        return x;
}
```

Aufgabe 6: Unterprogramme (Fortsetzung)

2. Schätzen Sie den Speicherplatz in Byte ab, den der Assembler-Code belegt. Begründen Sie Ihre Antwort! (4 Punkte)

func: => Adressen brauchen keinen Speicher!
subl $4, %esp => 8 Bit Schätzung auf 256 verschiedene Befehle: Codiert in 8 Bit
+2 Bit: verschiedene Datentypen: long, byte, word!
+ 3 Bit: 8 Adressierungsarten für konstanten
+ 32 Bit: Konstanten könnten bis zu 32 Bit lang sein
+ 3 Bit: "%" wieder ungefähr 8 Adressierungsarten für Register
+ 3 Bit: Anzahl der Register muss codiert werden!

= 51 Bit => aufrunden auf 56 => ca. 7 Byte

Dasselbe mit den anderen Befehlen:

```
        movl $0, (%esp)      laut Rechnung oben wieder 7 Byte
.L2:    => kein Speicher
        movl (%esp), %eax  => 8 + 2 + 3 + 3 + 3 + 3= 22 Bit => ca. 3 Byte
        imull (%esp), %eax => ca. 3 Byte
        cmpl 8(%esp), %eax => ca. 3 Byte
        jb .L4             => 8  + offset-codierung ca. 8 bit = 16 bit ~ 2 byte
        jmp .L3            => 8 + offset-codierung ca. 8 bit = 16 bit ~ 2 byte
.L4:    => kein Speicher
        incl (%esp)        => 8 + 2 + 3 + 3 ~ 2 byte
        jmp .L2            => 3 byte
.L3:
        movl (%esp), %eax  => 3 byte
        addl $4, %esp      => 7 byte
ret     => 8 bit => ca. 1 byte
```

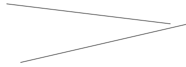
also überschlagen: 7 + 6 * 3 + 2 + 7 + 1 = 33 byte

3. Wandeln Sie das Unterprogramm in Assembler so um, dass es einen Frame-Pointer benutzt. (6 Punkte)

```
func:
        pushl %ebp
        movl %esp, %ebp          oder einfach nur: enter $4
        subl $4, %esp
        movl $0, (%esp)
.L2:
        movl (%esp), %eax
        imull (%esp), %eax
        cmpl 8(%ebp), %eax
        jb .L4
```

```
        jmp .L3
.L4:
        incl (%esp)
        jmp .L2
.L3:
        movl (%esp), %eax
        addl $4, %esp

        movl %ebp, %esp
        popl %ebp
        ret
```

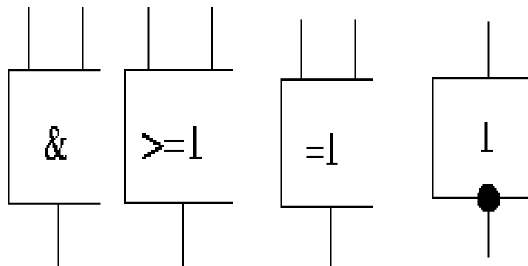


leave $4

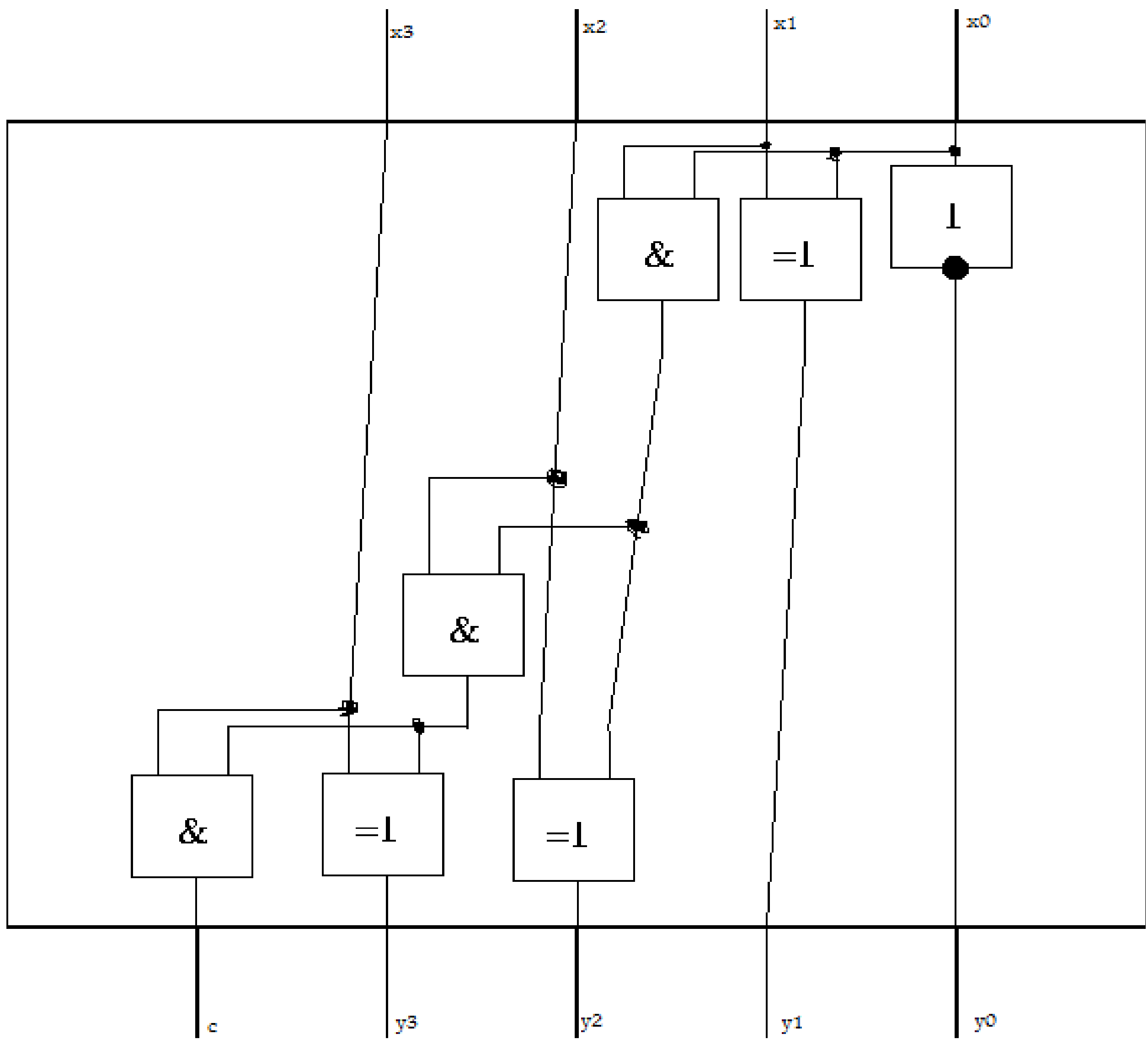
Hinweis: 8(%ebp) ist logischerweise immer der letzte übergabe-parameter. 12(%ebp) ist der vorletzte ... usw...

Aufgabe 7: Hardware (10 Punkte)

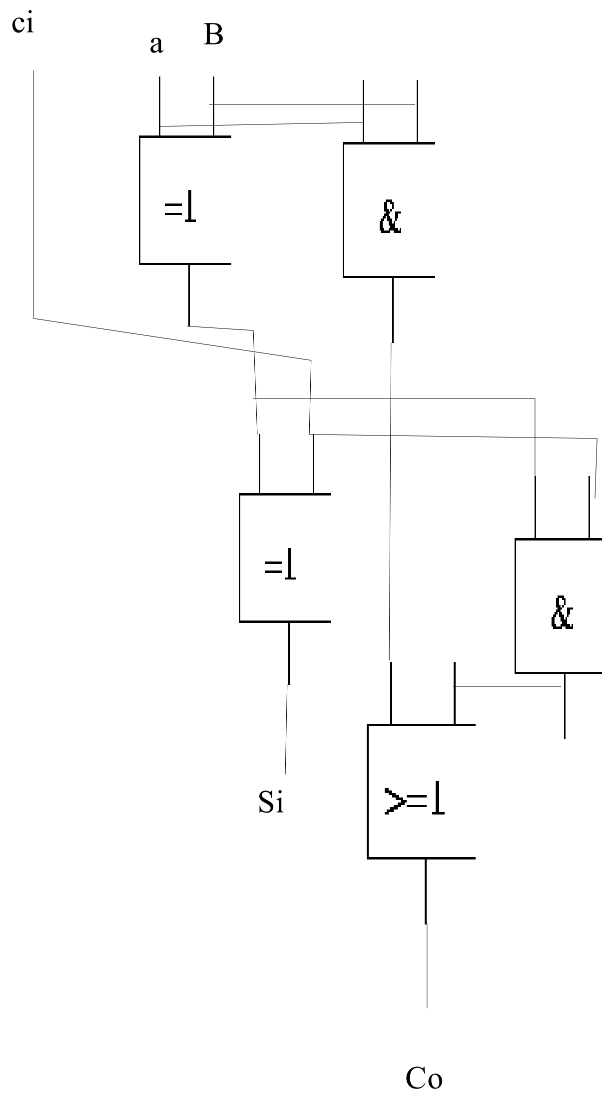
Es soll ein Inkrementierer in Hardware aufgebaut werden. Als Eingabe werden 4 Leitungen angenommen. Als Ausgabe sollen 4 Leitungen für die Signalisierung des normalen Ergebnisses dienen. Zusätzlich ist ein Übertrag über eine fünfte Leitung zu signalisieren. Für die Implementierung des Schaltnetzes dürfen die folgenden Gatter verwendet werden:



So, was für eine scheiß Aufgabe...
Hier habe ich mit Verlaub einfach die Lösung von Manu eingefügt:



Da mir diese Lösung ist mir viel zu kompliziert ist, hier mal meine:
Benutze Volladdierer-Zelle:



Den Inkrementierer mit Volladdierer-zelle aufbauen: