Autor: Sebastian Sossalla

Disclaimer: Das sind Lösungsvorschläge, keine Musterlösungen.

Aufgabe 1: Allgemeines (10 Punkte)

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

<x> Mit add können Signed- und Unsigned-Integer-Zahlen addiert werden.
< > Mit add können Integer- und Fließkommazahlen addiert werden.
< > Ein Rechner mit Big-Endian-Format braucht mehr Speicher als einer mit Little-Endian-Format.
< > Für die Berechnung von cos und sin braucht man spezielle Hardware.
(Hinweis: n+1-terKoeffizient kann aus dem n-ten ausgerechnet werden.)
<x> Der Stack ist ein Teil des Hauptspeichers.
<x> Alle Instruktionen, alle Variablen und alle Konstanten von Maschinen-Programmen haben eindeutige Adressen.
< > Auf long-Werte im Speicher kann man nur mit Instruktionen mit l-Suffix zugreifen (z.B. addl, movl); nicht jedoch mit Byte- oder 16bit-Instruktionen (z.B. subb, movw).
(Hinweis: man kann auch einen Long-wert mit 2 w-suffix-befehlen nacheinander zugreifen...)
<x> Alle Kontrollstrukturen können auf if-then-goto-Befehle abgebildet werden.
< > Mit Hilfe der Adressierungsarten können Daten vor unerlaubtem Zugriff geschützt werden.
<x> Mit Hilfe der MMU können Daten vor unerlaubtem Zugriff geschützt werden.
< > Die MMU rechnet physikalische Adressen in Segmente um.
(Hinweis: Die MMU rechnet die virtuellel in eine physikalische Adresse um.)
< > Segmentierung und Paging schließen einander aus.

2. Welche Werte können in einer Page-Tabelle sinnvollerweise gespeichert sein? (3 Punkte; Punktabzug bei falschen Antworten!)
<x> physikalische Adresse
< > virtuelle Adresse
< > Segmentnummer
<x> ,,Present"-Bit
<x> ,,Read-Only"-Bit

< > Größe der Seite

3. Was ist der Unterschied zwischen relativen und absoluten Sprungbefehlen? (2 Punkte)

Relative Sprungbefehle benutzen als Bezugspunkt ihre eigene Adresse und relativ zu dieser wird ein Offset angegeben. Also sozusagen die Distanz von der eigenen Adresse weg zu der hingesprungen werden soll. Neue Adresse = aktuelle Adresse + Offset. Absolute Sprungbefehle springen zu absoluten Adressen. Also Labels im Programm, die eine absolute Adresse besitzen. Neue Adresse = Label-Adresse.

4. Wann braucht ein Programm besonders viel Stack-Space? (2 Punkte)
Wenn ein Programm rekursive Algorithmen/Routinen besitzt, mit einer hohen Rekursions-tiefe, oder allgemein eine hohe Unterprogramm-tiefe besitzt, dann brauchen solche Routinen besonders viel Stack-Space, da immer wieder innerhalb eine Routine erneut eine Rouine aufgerufen werden muss und Parameter auf den Stack geschoben werden müssen.

Aufgabe 2: Main-Bus (6 Punkte)

1. Nennen Sie die Signale eines Ihnen bekannten Main-Busses! (2 Punkte)

IO-Read, IO-Write, Memory-Read, Memory-Write, Interrupt-Signale

2. Beschreiben Sie die Aufgabe der einzelnen Signale dieses Busses! (4 Punkte)

IO-Read: Die CPU schickt dieses Signal an das I/O-Device, wenn Daten von einem Ein-Ausgabe-Gerät gelesen werden sollen und schreibt die Adresse von der gelesen werden soll auf den Adressbus. Das I/O-Device bekommt das Signal, holt sich die Adresse vom Bus, ,bearbeitet die Anfrage, und stellt die angeforderten Daten auf die Daten-leitung.

IO-Write: Es sollen Daten auf ein I/O-Gerät geschrieben werden. Dazu werden die zu schreibenden Daten auf den Daten-bus und die Ziel-Adresse auf den Adress-Bus gelegt und die CPU schickt das Signal I/O-Write los. Das I/O-Gerät erhält das Signal, holt sich die ziel-adresse und die Daten vom Bus, und schreibt die Daten in die Adresse.

MEM-Read: Selbes vorgehen wie I/O-Read, nur das signal MR wird an den Speicher geschickt, um Daten vom Speicher zu holen.

MEM-Write: Selbes vorgehen wie IO-Write, nur das Signal MW wird an den Speicher geschickt, um Daten in den Speicher zu schreiben.

Interrupt-Signale: Kurzfristige asynchrone unterbrechung des Programms für eine von der CPU abzuarbeitende Befehls-sequenz. Sinn: schnell auf I/O-Geräte oder Zeitgeber zu reagieren.

Aufgabe 3: Speicher (10 Punkte)

Gegeben sei folgende Struktur:

```
struct {
        char name[8];
        unsigned char fachsemester;
        char dummy;
        unsigned short punkte;
        unsigned short note;
};
```

Die Struktur werde auf einem 16bit-Rechner mit korrektem Alignment gespeichert.

1. Wieviel Speicherplatz verbraucht eine solche Struktur? Begründen Sie Ihre Antwort! (2 Punkte)
16 bit = 2 Byte

12	34	56	78	fachs dummy	punke	note	

Das Alignment passt nicht => 8 Byte für name + 1 Byte für fachsemester + 1 Byte für dummy + 2 byte für punkte + 2 byte für note= 14 Byte

2. Welcher Eintrag ist vermutlich im folgenden Speicherdump sichtbar?

An welcher Stelle beginnt, an welcher Stelle endet der Eintrag? (5 Punkte)
Hinweis: alle Zahlen im Hexadezimal-System

... 02 00 41 44 41 4d 00 00 00 00 02 03 5a 00 01 00 4a 4b ...
Hinweis: 41 = ,,A", 42 = ,,B", 43 = ,,C", ...
... 02 00 A D A M 00 00 00 00 02

Der Eintrag endet bei:
Beginnt bei: 3. Byte von links: 41 = "A" und endet beim 3. Byte von rechts:
neuer Eintrag beginnt bei Bye 4a, 2. Byte von rechts

3. Arbeitet der Rechner vermutlich im Little- oder Big-Endian-Format? Begr
ünden Sie Ihre Antwort! (3 Punkte)
Er arbeitet im little-Endian Format.
Die Daten des Eintrag sind:

name = ADAM
Semester = 2
Punkte = 5a = 5 * 16 + 10 = 90
Note = 1

Der name ist normalerweise immer rechtsbündig gespeichert:
also "   ADAM", Die punkte 90, im Big-Endian wäre die Puntzahl zu groß. Und die Note muss eins sein, also muss Big-Endian vorliegen, sonst wäre die Note: 0x10 = 16

Aufgabe 4: Ausdrücke / Kontrollstrukturen (12 Punkte)

Konvertieren Sie die Hochsprachen-Funktion in semantisch äquivalenten Hochsprachen-Code, so dass sich dieser möglichst leicht in Assembler-Code für eine Ein-Adress-Maschine umwandeln lässt!

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 wert[1000];

int test(int x, unsigned int n)
{
int ret;
unsigned int i;
ret = 0;

for (i = 0; i < n; i++) {
        unsigned int diff2;
        diff2 = (wert[i] - x) * (wert[i] - x);

        if (100 < diff2) {
        ret = 1;
          break;
        }
    }
  return ret;
}
```

eine recht bäuerliche Lösung:

```
int  wert[1000];

int test(int x, unsigned int n) {
int ret;
unsigned int i;
ret = 0;

i = 0;

_test:  if (i < n) goto _loop;
        goto _end;

_loop:  unsigned int diff2;

        akku = wert[i];
        akku = akku - x;
        akku = akku * akku;
        diff2 = akku;

        if (100 < diff2) {
                ret = 1;
                goto _end;
        }
```

```
        i++;
        goto _test;

_end;   return ret;
}
```

Aufgabe 5: Compiler (12 Punkte)

Wie könnte der Hochsprachen-Code zu dieser Funktion ausgesehen haben? (7 Punkte)

```
unknown:
        subl $8, %esp
        movl 12(%esp), %ecx
        movl 16(%esp), %edx
        cmpl %edx, %ecx
        jb .L7
        cmpl %ecx, %edx
        movl %ecx, %eax
        jb .L8
.L1:
        addl $8, %esp
        ret
.L8:
        movl %edx, 4(%esp)
        subl %edx, %ecx
.L6:
        movl %ecx, (%esp)
        call unknown
        jmp .L1
.L7:
        subl %ecx, %edx
        movl %edx, 4(%esp)
jmp .L6
```

Lösung:
Das ist der schöne ggt-Algorithmus: (nach einige Male Optimierung...)

```
/*x liegt in %edx*/
/*y liegt in %ecx*/
/**/

unknown(unsigned int x, unsigned int y) {
        int erg;

        erg = y;
```

```
        if (y < x) {
                x = x - y;
                erg = unknown(x, y);
                return erg;
                /* L7*/
        } else {
           erg = y;
           if (x < y) {
                erg = unknown(x, y-x);
                return erg;
           }

        }   /*L1*/
        return erg;

}
```

Ein besserer Compiler macht folgenden Assembler-Code aus der Hochsprachen-Funktion:

```
unknown:
        movl 4(%esp), %eax
        movl 8(%esp), %edx
.L7:
        cmpl %edx, %eax
        jae .L2
        subl %eax, %edx
        jmp .L7
.L2:
        cmpl %eax, %edx
        jae .L5
        subl %edx, %eax
        jmp .L7
.L5:
ret
```

Was hat der Compiler hier zusätzlich optimiert? Was ist an diesem Code besser als am vorhergehenden? (5 Punkte)

Stupide übersetzt sieht der Code jetzt so aus:

```
x /* in %edx*/
Y /* in %eax*/

unknown(int x, int y) {
```

```
L7:     if (y >= x) {
                goto L2;
        } else {
                x = x - y;
                goto L7;
        }
L2:     if (x >= y) {
                goto L5;
        } else {
                 y = y - x;
                goto L5;
        }
```

- Es wurden die beiden Überflüssigen temporären Variablen entfernt: der Stack wird dadurch entlastet. Also weniger speicherverbauch und viel weniger Befehle, da der Stack nicht mehr initialisiert/aufgeräumt werden muss.
- Die sprungmarken L6 und L8 wurden entfernt. Weniger Sprünge im Programm Ausserdem sind die Sprungmarken besser angeordnet und nicht so durcheinander wie oben. Dadurch sind die sprungdistanzen kürzer: schnellerer Programmablauf!
- Die rekursion wurde komplett in eine Iteration umgewandelt. Dadurch viel schnellerer Programmablauf und weniger Speicher, da komplett auf den Stack verzichtet werden kann.

Aufgabe 6: Stack (10 Punkte)

1. Die Programmiersprachen C (alter Standard) und Java erlauben es nicht, Inhalte ganzer Arrays bzw. Strukturen als Parameter an Unterprogramme zu übergeben. Ebenso sind Inhalte von Arrays bzw. Strukturen nicht als Rückgabewerte von Funktionen erlaubt.
Was könnte der Grund sein? (4 Punkte)

Wenn es erlaubt wäre, ganze Inhalte von Arrays/Strukturen zu übergeben, dann müssten für die ganzen Werte auch auf den Stack gepushed werden. Das wäre viel zu Ineffizient und würde die Programmlaufzeit und den Speicherverbrauch erheblich erhöhen. Das ist unnötig, denn es ist vollkommen ausreichend, nur Zeiger auf diese Objekte zu übergeben bzw. zurückzugeben.

2. Gegeben sei folgende Funktion:

```
unsigned int fac(unsigned int n)
{
        if (n == 0)
                return 1;
        else
                return n * fac(n - 1);
}
```

Wieviel Stack-Space wird ein Programm vermutlich brauchen, das fac mit dem Parameter n = 100 aufruft?
Hinweise:
- die Parameterübergabe erfolge über den Stack
- die Resultatsrückgabe erfolge über ein Register
- ,,unsigned int"-Variablen seien 32bit groß
- Adressen seien 32bit groß
- die Funktion sei ohne Frame-Pointer compiliert
- der Compiler habe nicht optimiert

Begründen Sie Ihre Antwort! (6 Punkte)

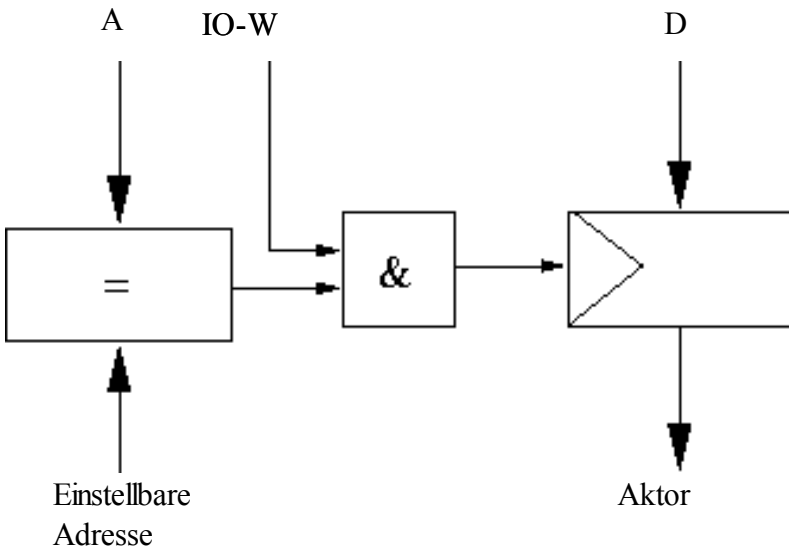
für n = 100: für jeden Call wird der Parameter n-1 auf den Stack gelegt + ret-Adresse. Ich nehme an, es werden keine lokalen variablen benutzt. pro Programmaufruf sind das: 100 * 4 Byte + 100 *4 Byte = 800 Byte

Aufgabe 7: I/O (10 Punkte)

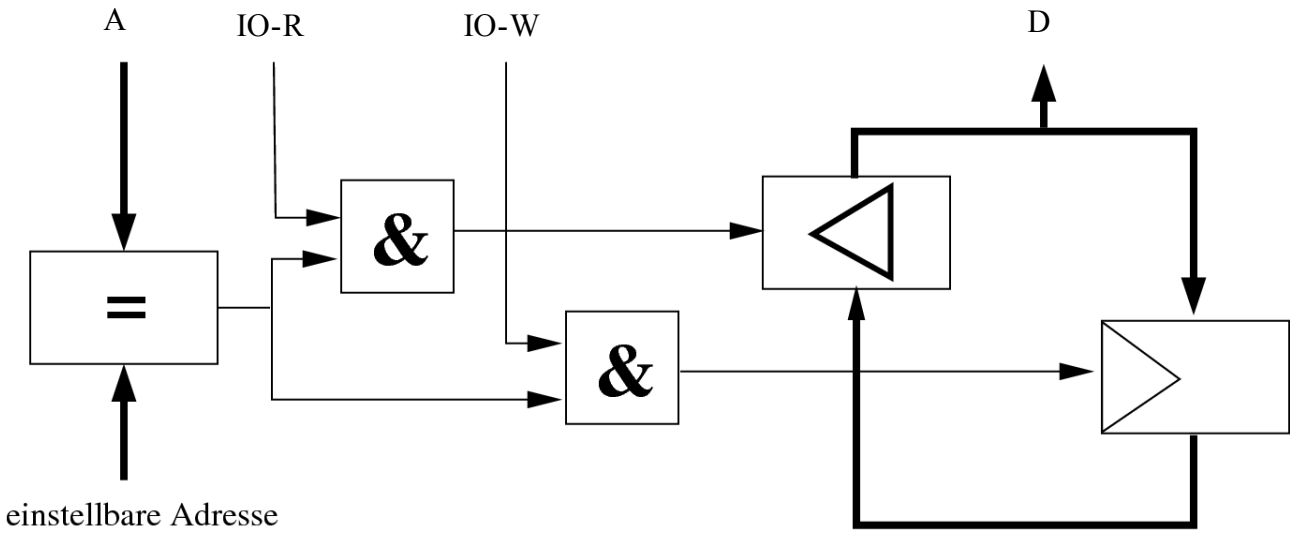
1. Beschriften Sie die Signale des unten stehenden Schaltplans eines Output-Ports! (3 Punkte)

2. Erweitern Sie den Schaltplan so, dass auf dem Port mit outb ausgegebene Werte mit inb wieder gelesen werden können! (7 Punkte)

1.)



2.)



Aufgabe 8: MMU (10 Punkte)

Ein Programm brauche folgende virtuellen Speicherbereiche:

0x08000000 – 0x080ff0ea Textsegment
0x20000000 – 0x20100123 Datensegment
0x40000000 – 0x400ff9af Library
0xbffff120 – 0xbfffffff Stack

Der Rechner bietet 4kByte große Pages.
Hinweis: $4096 = 2_{12}$, $1024 = 2_{10}$

1. Wieviele virtuelle Pages benötigt das Programm für das Text-, Daten-, Library- und Stack-Segment? Begründen Sie Ihre Antwort! (3 Punkte)

```
 0x080ff0ea
-0x08000000
-----------
 0x000ff0ea

4096 = 2^12 = (2^4)^3 = 16^3 = 0x1000
rechne:   0xff0ea / 0x1000 = ff,0ea = 255 + 1  = 256 Page

 0x20100123
-0x20000000
-----------
 0x00100123

100123 / 0x1000 = 100,123 = 16^2 = 256 + 1 = 257 Pages

 0x400ff9af
-0x40000000
-----------
 0x000ff9af

0xff9af / 0x1000 = ff,9af = 255 + 1 = 256 Pages

 0xbfffffff
-0xbffff120
-----------
 0x00000edf

0xedf / 0x1000 = 0,edf = 1 Page
```

2. Wieviele Pages benötigt das Programm für die Page-Tabellen? Begründen Sie Ihre Antwort! (7 Punkte)

Base-Register

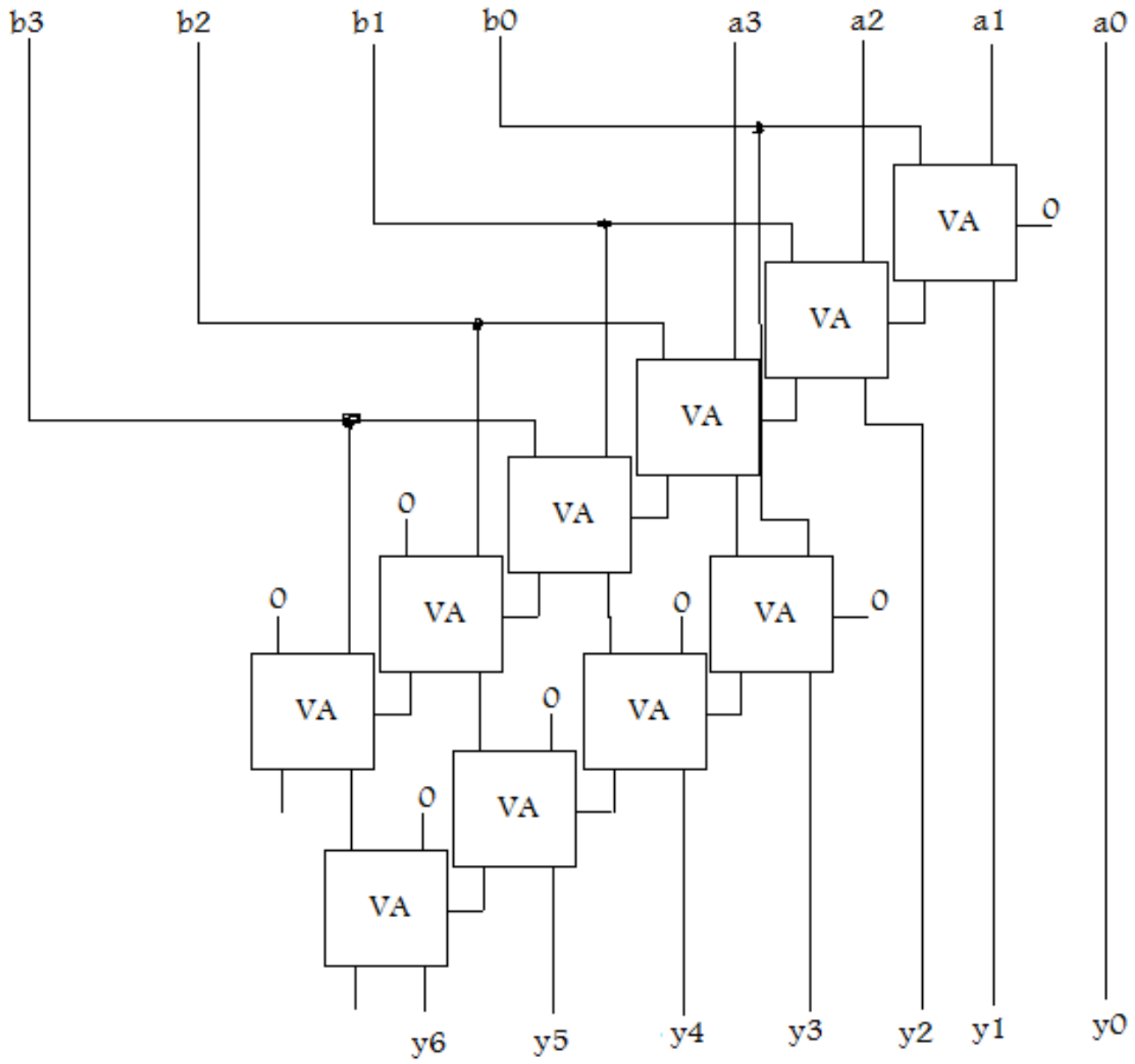
Hinweis: Die Page-Tabellen seien zweistufig. Sowohl die Page-Group-Tabelle als auch die Page-Tabelle enthalten jeweils 1024 Einträge und sind je 4kByte groß.

Aufgabe 9: Hardware (ALU) (10 Punkte)

Entwickeln Sie eine Schaltung, die eine vorzeichenlose zweistellige BCD-Zahl in eine binär-codierte Zahl umrechnet! Sie dürfen die normalen AND-, OR- und NOT-Gatter sowie Volladierer für ihre Schaltung verwenden.

Dazu hat Manu was gutes gebastelt:

Manu's erklärung:
Die Aufgabe ist ein Hammer. Der Ansatz besteht darin, die BCD-Bits einzeln umzurechnen und dann zu addieren: x = (b3b2b1b0 a3a2a1a0)
a0 = 1, a1 = 2, a2 = 4, a3 = 8, b0 = 10, b1 = 20, b2 = 40, b3 = 80
A =0 0 0 a3 a2 a1 a0
B = b3 0 b3 0 0 0 0 + 0 b2 0 b2 0 0 0 + 0 0 b1 0 b1 0 0 + 0 0 0 b0 0 b0 0
= b3 b2 (b3 + b1) (b2 + b0) b1 b0 0 (einfach b3, b2, b1, b0, also 80, 40, 20, 10 einzeln
umrechnen)
A + B = b3 b2 (b3 + b1) (a3 + b2 + b0) (a2 + b1) (a1 + b0) a0
= b3 b2 b3 a3 a2 a1 a0 + 0 0 b1 b2 b1 b0 0 + 0 0 0 b0 0 0 0


b3
b2
b1
b0
a3
a2
a1
a0
VA
0
y6
y5
y4
y3
y2
y1
y0