KSP Aufgabe 1

Konstruieren Sie die Ninja-VM Version 1, die die in VM Instruktionen gelisteten Instruktionen ausführen kann. Jede Instruktion eines Programms wird in einem `unsigned int` gespeichert, wobei der Opcode (die Zahl, die in der Tabelle hinter jeder Instruktion steht) in den obersten 8 Bits des `unsigned int` abgelegt wird. Die restlichen 24 Bits dienen zur Aufnahme eines Immediate-Wertes. Der wird im Augenblick nur bei der Instruktion `pushc` benötigt, wo er den auf den Stack zu legenden Wert darstellt.

Table 1. VM Instruktionen

Instruktion	Opcode	Stack Layout
`halt`	`0`	`... -> ...`
`pushc <const>`	`1`	`... -> ... value`
`add`	`2`	`... n1 n2 -> ... n1+n2`
`sub`	`3`	`... n1 n2 -> ... n1-n2`
`mul`	`4`	`... n1 n2 -> ... n1*n2`
`div`	`5`	`... n1 n2 -> ... n1/n2`
`mod`	`6`	`... n1 n2 -> ... n1%n2`
`rdint`	`7`	`... -> ... value`
`wrint`	`8`	`... value -> ...`
`rdchr`	`9`	`... -> ... value`
`wrchr`	`10`	`... value -> ...`

Den Effekt einer jeden Instruktion auf den Stack listet die Spalte Stack Layout in der VM Instruktionen. Punkte `·····` in der Tabelle stehen jeweils für den durch die Instruktion nicht veränderten Stackinhalt; der `Top-of-Stack` befindet sich jeweils rechts. Die Operanden für arithmetische Operationen sind immer ganze Zahlen, genauso wie die bei `rdint` und `rdchr` eingelesenen bzw. bei `wrint` und `wrchr` ausgegebenen Werte.

Testprogramme

Die drei hier gelisteten Programme sollen fest in die VM einprogrammiert werden. Der Benutzer kann durch Eingabe eines Kommandozeilenargumentes beim Aufruf der VM eines der Programme auswählen, das dann in den Programmspeicher der VM kopiert wird. Nach dem Anlaufen der VM wird das Programm erst in einer lesbaren Form aufgelistet; eine mögliche Darstellung können Sie der Referenzimplementierung entnehmen. Anschließend wird es ausgeführt. Beachten Sie bitte, dass "Auflisten" und "Ausführen" zwei getrennte Aktionen auf dem Programm sind, die strikt hintereinander passieren sollen.

Programm 1

Ninja Programmfragment	Instruktionssequenz

`writeInteger((3 + 4) * (10 - 6));`
`writeCharacter('\n');`
	`pushc 3`
	`pushc 4`
	`add`
	`pushc 10`
	`pushc 6`
	`sub`
	`mul`
	`wrint`
	`pushc 10`
	`wrchr`
	`halt`

Programm 2

Ninja Programmfragment	Instruktionssequenz

`writeInteger(-2 * readInteger() + 3);`
`writeCharacter('\n');`
	`pushc -2`
	`rdint`
	`mul`
	`pushc 3`
	`add`
	`wrint`
	`pushc '\n'`
	`wrchr`
	`halt`

Programm 3

Ninja Programmfragment	Instruktionssequenz

`writeInteger(readCharacter());`
`writeCharacter('\n');`
	`rdchr`
	`wrint`
	`pushc '\n'`
	`wrchr`
	`halt`

Hier ist wieder die Referenzimplementierung: njvm

Hinweise Aufgabe 1

1. Die Compilerschalter sind wie in Aufgabe 0 auf `-g -Wall -std=c99 -pedantic` zu setzen.
2. Ihre VM benötigt einen Programmspeicher und einen Stack. Beide können Sie global definieren, damit Sie von überall darauf zugreifen können. Wählen Sie die Basis-Datentypen mit Bedacht, insbesondere im Hinblick auf vorzeichenbehaftete (`signed`) vs. vorzeichenlose (`unsigned`) Größen!
3. Ihre VM wird zwei Interpreter für den Programmcode beinhalten: einen für das **Listen des Programms** und einen für die eigentliche **Ausführung**. Beide sind strukturell gleich aufgebaut: eine Schleife (*worüber?*) und darin eine Mehrfachverzweigung (*wozu?*). Beginnen Sie mit dem Programmlister - wenn Sie den haben, ist der Interpretierer für die Ausführung nicht mehr schwer. Beachten Sie aber die zwei unterschiedlichen Abbruchkriterien in den beiden Interpretierern (*welche genau?, und warum sind die eigentlich unterschiedlich?*).
4. Ihr Programm darf unter keinen Umständen abstürzen! Natürlich gibt es Fehlersituationen, in denen Sie das Programm nicht vernünftig weiterlaufen lassen können - dann geben Sie eine verständliche Fehlermeldung aus und beenden das Programm. Eine solche Situation betrifft beispielsweise das Teilen durch 0, eine andere den Stack-Überlauf bzw. -Unterlauf (*wann können diese beiden Fehler auftreten?*).
5. Sie müssen die drei Programme *manuell*, also *per Hand* assemblieren, solange es noch keinen Assembler gibt.
   a. Wie macht man das?
   Nehmen wir als Beispiel `pushc 3` → Der **Opcode** für `pushc` ist `1` (er kommt in die höchsten 8 Bits); die **Immediate-Konstante** ist `3` (sie kommt in die untersten 24 Bits). Also ist das Bitmuster des Befehls (4 Bytes) `00000001 00000000 00000000 00000011`, oder hexadezimal `0x01000003`. Auf gar keinen Fall arbeitet man hier mit Dezimalzahlen! Es geht aber viel eleganter: verwenden Sie den C-Präprozessor, um eine lesbare Codierung der drei kleinen Programme zu ermöglichen! Beispielsweise könnte der Anfang des ersten Programms im C-Quelltext lauten:

```
unsigned int code1[] = {
(PUSHC << 24) | IMMEDIATE(3),
(PUSHC << 24) | IMMEDIATE(4),
(ADD << 24),
...
}
```

mit den entsprechenden Definitionen für die Opcodes, z.B.

```
#define PUSHC 1
#define ADD 2
```

und für das Kodieren des Immediate-Wertes (*warum braucht man das?*)

```
#define IMMEDIATE(x) ((x) & 0x00FFFFFF)
```

1. Der Immediate-Wert kann auch **negativ** sein. Vielleicht haben Sie Verwendung für dieses Makro:

```
#define SIGN_EXTEND(i) ((i) & 0x00800000 ? (i) | 0xFF000000 : (i))
```

Was macht das Makro eigentlich? Und wie genau funktioniert es?

CAUTION	ACHTUNG: Es ist unbedingt erforderlich, die Makros aus 5. und 6. genau zu verstehen! Machen Sie sich die Wirkungsweise an Beispielen klar und rechnen Sie damit, im Praktikum dazu befragt zu werden!