Hinweise zur Bearbeitung und Abgabe

Die Lösung der Hausaufgabe muss **eigenständig** erstellt werden. Abgaben, die identisch oder auffällig ähnlich zu anderen Abgaben sind, werden als **Plagiat** gewertet! **Plagiate sind Täuschungsversuche und führen zur Bewertung „nicht bestanden" für die gesamte Modulprüfung.**

- Bitte nutzen Sie MARS zum Simulieren Ihrer Lösung. Stellen Sie sicher, dass Ihre Abgabe in MARS ausgeführt werden kann. Die Installation und das Ausführen von MARS wird im **5. Übungsblatt** beschrieben.
- Sie erhalten für jede Aufgabe eine separate Datei, die aus einem Vorgabe- und Lösungsabschnitt besteht. Ergänzen Sie bitte Ihren Namen und Ihre Matrikelnummer an der vorgegebenen Stelle. Bearbeiten Sie zur Lösung der Aufgabe nur den Lösungsteil unterhalb der Markierung:
  ```
  #+ Loesungsabschnitt
  #+ ------------------
  ```
- Ihre Lösung muss auch mit anderen Eingabewerten als den vorgegebenen funktionieren. Um Ihren Code mit anderen Eingaben zu testen, können Sie die Beispieldaten im Lösungsteil verändern.
- Bitte nehmen Sie keine Modifikationen am Vorgabeabschnitt vor und lassen Sie die vorgegebenen Markierungen (Zeilen beginnend mit `#+`) unverändert.
- Eine korrekte Lösung muss die bekannten **Registerkonventionen** einhalten. Häufig können trotz nicht eingehaltener Registerkonventionen korrekte Ergebnisse geliefert werden. In diesem Fall werden trotzdem Punkte abgezogen.
- Falls Sie in Ihrer Lösung zusätzliche Speicherbereiche für Daten nutzen möchten, verwenden Sie dafür bitte ausschließlich den **Stack** und keine statischen Daten in den Datensektionen (`.data`). Die Nutzung des Stacks ist gegebenenfalls notwendig, um die Registerkonventionen einzuhalten.
- Die zu implementierenden Funktionen müssen als Eingaben die Werte in den **Argument-Registern** (`$a0`–`$a3`) nutzen. Daten in den Datensektionen der Assemblerdatei dürfen nicht direkt mit deren Labels referenziert werden.
- Bitte gestalten Sie Ihren Assemblercode nachvollziehbar und verwenden Sie detaillierte Kommentare, um die Funktionsweise Ihres Assemblercodes darzulegen.
- Die Abgabe erfolgt über ISIS. Laden Sie die zwei Abgabedateien separat hoch.

Aufgabe 1: Collatz-Sequenz (10 Punkte)

Aufgabe: Implementieren Sie die Funktion `collatz`, welche eine Zahlensequenz nach einem bestimmen Muster in dem Array `buf` speichert und die Länge dieser Sequenz zurückgibt. Der numerische Anfangswert `n` bestimmt dabei, mit welchem Zahlenwert die Sequenz beginnen soll. Die Sequenz lässt sich dabei wie folgt schrittweise bestimmen:

- Ist `n` gerade, so nimm als nächstes `n`/2.
- Ist `n` ungerade, so nimm als nächstes 3`n`+1.
- Wiederhole die Vorgehensweise mit der erhaltenen Zahl als neues `n`.
- Beende die Sequenz, wenn `n` gleich 1 ist.

Die C-Signatur der zu implementierenden Funktion lautet:

`int`	`collatz(`	`int* buf,`	`int n`	`);`
`$v0`		`$a0`	`$a1`	

Bei `buf` handelt es sich um einen Pointer zum ersten Element eines leeren Integer-Arrays. Jedes Element ist dabei vier Bytes groß. Die Funktion `collatz` soll das Array dabei mit den berechneten Werten der Sequenz befüllen. Der Rückgabewert der Funktion bestimmt dabei, wie viele Elemente in der Sequenz vorhanden sind.

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dazu den Anfangswert `test_n`. Folgende Tabelle enthält Beispieleingaben und die erwarteten Rückgabewerte, welche zum Testen verwendet werden können:

Anfangswert `n`	**Erwarteter Rückgabewert**	**Erwartete Sequenz in `buf`**
13	10	13 40 20 10 5 16 8 4 2 1
9	20	9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
96	13	96 48 24 12 6 3 10 5 16 8 4 2 1
42	9	42 21 64 32 16 8 4 2 1
21	8	21 64 32 16 8 4 2 1

Hintergrund: Die Collatz-Vermutung[1] lautet, dass egal mit welchem positiven natürlichen Anfangswert `n` die Sequenz gebildet wird, die Zahlenfolge immer in einem Zyklus aus 4, 2, 1 endet. Für die Collatz-Vermutung gibt es noch keinen formellen Beweis, aber für Anfwangswerte bis in die Milliarden wurde die Vermutung bereits bestätigt.

Hinweise:

- Nehmen Sie an, dass `n` immer positiv und größer als 0 ist.
- Für die Multiplikation mit 3 können Sie folgenden mathematischen Kniff nutzen: $3 \cdot x = x + x + x$.
- Für die Division durch 2 können Sie eine logische Schiebeoperation verwenden.
- Erreicht die Sequenz den Zahlenwert 1, womit die Sequenz beendet wird, müssen die nicht beschriebenen Elemente von `buf` nicht weiter betrachtet werden.
- Das Integer-Array `buf` ist angelegt für maximal 1000 Sequenzelemente. Damit sind theoretisch Collatz-Sequenzen mit einem Anfangswert `n` von bis zu 1 Milliarde berechenbar. Allerdings kommt es bei deren Berechnung zu arithmetischen Überläufen, weswegen nur Anfangswerte $\texttt{n} < 1000$ verwendet werden sollten.

[1] Weitere Hintergrundinformationen: https://de.wikipedia.org/wiki/Collatz-Problem

Aufgabe 2: Fleißnersche Schablone (10 Punkte)

Aufgabe: Die zu implementierende Funktion `fleissner` soll einen Text `text` mittels der Maske `mask` verschlüsseln. Der verschlüsselte Text soll in `buf` gespeichert werden. `text` ist eine nullterminierte Zeichenkette. `mask` ist ein binäres Byte-Array. Bei `buf` handelt es sich um einen Pointer zum ersten Element einer leeren Zeichenkette. Die C-Signatur der zu implementierenden Funktion lautet:

`void`	`fleissner(`	`char* buf,`	`char* text,`	`int8_t* mask`	`);`
		`$a0`	`$a1`	`$a2`	

Verschlüsselung: Die Verschlüsselung basiert auf einer Maske, die mehrfach im Uhrzeigersinn rotiert wird. Die Elemente, in denen die Maske mit einer 1 beschrieben ist, werden dabei im Ausgabetext durch Zeichen aus dem Eingabetext befüllt.

Hilfsfunktion: Die Hilfsfunktion `rotate_mask` implementiert die Rotation einer Maske um 90 Grad. Die Maske wird dabei als eine 6×6 Matrix interpretiert. Die übergebene Maske `mask` wird direkt überarbeitet. Die C-Signatur von `rotate_mask` ist:

`void`	`rotate_mask(`	`int8_t* mask`	`);`
		`$a0`	

Vorgehen: Der Ausgangstext, welcher in `buf` geschrieben wird, muss abhängig von `mask` befüllt werden. Die Maske muss dabei mithilfe der Hilfsfunktion `rotate_mask` angepasst werden, sodass alle Elemente des Eingabetextes verwendet werden.

1. *Befüllungsphase:* Interpretieren Sie den Ausgabetext `buf` als eine 6×6 Matrix. Befüllen Sie den Ausgabetext mit fortfolgenden Zeichen aus dem Eingabetext, allerdings nur an den Stellen, an denen die Maske auf 1 gesetzt ist.
2. *Rotationsphase:* Rotieren Sie die Maske mithilfe der Hilfsfunktion `rotate_mask`.
3. Wiederholen Sie die beiden vorherigen Schritte, bis alle Zeichen im Eingabetext verwendet wurden.

Beispiel: Mit den folgenden Werten für `text` und `mask` wird die Vorgehensweise der Verschlüsselung anhand der Beispieleingaben veranschaulicht.

$$text = \text{“ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789“} \qquad mask = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

In der folgenden Tabelle wird das Beispiel schrittweise dargestellt. Die Maske wird zwischen den jeweiligen Schritten um 90 Grad rotiert. Der Ausgabetext wird schrittweise in Matrixform aufgebaut. Dabei werden einzelne Zeichen aus dem Text in den Buffer kopiert, je nachdem wo eine 1 in der Maske steht.

	Text	**Maske**	**Aufgebauter `buf`**
Iteration 1:			
	ABCDEFGHI	0 1 0 1 0 1	- A - B - C
	JKLMNOPQR	0 0 0 0 1 0	- - - - D -
	STUVWXYZ0	0 0 1 0 0 0	- - E - - -
	123456789	0 1 0 0 1 0	- F - - G -
		0 0 0 0 0 1	- - - - - H
		0 0 0 1 0 0	- - - I - -
Iteration 2:			
	ABCDEFGHI	0 0 0 0 0 0	- A - B - C
	JKLMNOPQR	0 0 1 0 0 1	- - J - D K
	STUVWXYZ0	0 0 0 1 0 0	- - E L - -
	123456789	1 0 0 0 0 1	M F - - G N
		0 0 1 0 1 0	- - O - P H
		0 1 0 0 0 1	- Q - I - R
Iteration 3:			
	ABCDEFGHI	0 0 1 0 0 0	- A S B - C
	JKLMNOPQR	1 0 0 0 0 0	T - J - D K
	STUVWXYZ0	0 1 0 0 1 0	- U E L V -
	123456789	0 0 0 1 0 0	M F - W G N
		0 1 0 0 0 0	- X O - P H
		1 0 1 0 1 0	Y Q Z I 0 R
Iteration 4:			
	ABCDEFGHI	1 0 0 0 1 0	1 A S B 2 C
	JKLMNOPQR	0 1 0 1 0 0	T 3 J 4 D K
	STUVWXYZ0	1 0 0 0 0 1	5 U E L V 6
	123456789	0 0 1 0 0 0	M F 7 W G N
		1 0 0 1 0 0	8 X O 9 P H
		0 0 0 0 0 0	Y Q Z I 0 R
			⇒ 1ASB2CT3J4DK5UELV6MF7WGN8XO9PHYQZI0R

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dazu die Zeichenkette `test_text` und das Byte-Array `test_mask`. Folgende Tabelle enthält Beispieleingaben und die erwarteten verschlüsselten Texte, welche zum Testen verwendet werden können:

$$mask_A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad mask_B = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Eingabe `text`	`mask`	**Erwarteter String in `buf`**
`ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`	*A*	`1ASB2CT3J4DK5UELV6MF7WGN8XO9PHYQZI0R`
`MicroprocessorW/OInterlockedPipeline`	*A*	`dMniPctieprseeosrlopilrrnoWe/ocOkceI`
`MIPS-AssemblyProgrammingIsALotOfFun!`	*A*	`LMaIoPmtmOSbfm-liFyAunsPngr!osIgseAr`
`ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`	*B*	`SJKAB12CT3LU4V5D6WEMXNFOG7PHYQRZO89I`
`MicroprocessorW/OInterlockedPipeline`	*B*	`nesMidPctiseprerllooorpWri/ocOIkenec`
`MIPS-AssemblyProgrammingIsALotOfFun!`	*B*	`ambMILoPmtlmOifSFn-ygPArsuosIgrsAn!e`

Zusatzinformation – Masken erzeugen: Diese Sektion ist nicht relevant zum Lösen der Aufgabe! Sie bietet aber die Möglichkeit neue Masken zum Testen des Assemblerprogramms zu erzeugen.
Da valide Masken jedes Element in einer 6×6 Matrix nach 3 Rotationen einmalig abdecken müssen, bedarf es eine spezielle Vorgehensweise bei ihrer Erzeugung[2]. Befüllen Sie dafür einen Quadranten einer 6×6 Matrix mit den Ziffern 1-4. Dabei ist die relative Häufigkeit der Zahlen unwichtig. Danach rotieren Sie diesen Quadranten in den jeweils nächsten Quadranten, wobei die Ziffern zyklisch verschoben werden. Dies bedeutet, dass jeder Zahlenwert folgender Gleichung folgt: $z = (z \ mod \ 4) + 1$. Dies wiederholen Sie bis alle Quadranten der Matrix befüllt sind. Um nun eine Maske zu erzeugen, werden alle Elemente mit einer beliebigen gleichen Zahl, z.B. 4, auf 1 gesetzt und alle anderen Elemente auf 0. Bei den erzeugten Masken sind damit auch immer genau neun Felder auf eine 1 gesetzt.

Hinweise:

- Die Zeichenkette `test_text` muss immer 36 Zeichen lang sein.
- Das Byte-Array `test_mask` muss immer in der Form 6×6 (6 Zeilen, 6 Spalten) bleiben. Die Elemente im Byte-Array sollen nur Werte von 0 oder 1 annehmen.
- Der Datentyp `int8_t` ist ein Integer mit nur 8 bits, anstatt der sonst üblichen 32 bits. In C kann er verwendet werden, um Bytes darzustellen.
- Ein 2-dimensionales Byte-Array wird im Speicher zeilenweise in hintereinanderfolgende Bytes abgelegt. Dadurch lässt es sich wie ein 1-dimensionales Byte-Array modifizieren und auslesen.
- Zeichenketten müssen mit einem Nullterminator abgeschlossen werden.

[2]Weitere Hintergrundinformationen: https://de.wikipedia.org/wiki/Fleißnersche_Schablone