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.
- 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: Postleitzahl finden (10 Punkte)

Aufgabe: Implementieren Sie die Funktion `plz`, welche die Zeichenkette `address` nach einer Postleitzahl (PLZ) durchsucht. Fünf aufeinanderfolgende Ziffern (0-9) sind als Postleitzahl zu verstehen. Falls eine Postleitzahl gefunden wurde, soll diese als Zahlenwert zurückgegeben werden. Falls keine Postleitzahl gefunden wurde, soll 0 zurückgegeben werden. Die C-Signatur der zu implementierenden Funktion ist:

`int plz( char *address);`
`$v0` `$a0`

Bei `address` handelt es sich um einen Pointer zum ersten Zeichen einer Zeichenkette. Diese Zeichenkette endet mit einem Nullterminator. Jedes Zeichen ist ein Byte groß und kann anhand des ASCII-Zahlenwerts untersucht werden. Eine ASCII-Tabelle finden Sie zum Beispiel auf der MIPS Green Card.

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

Eingabe-Zeichenkette	Erwartete Ausgabe
`TU Berlin, 10623 Berlin`	10623
`Panoramastraße 1A in 1017 Berlin`	0
`Die 4 Bremer (28195) Stadtmusikanten`	28195
`78462 Konstanz Hauptbahnhof`	78462

Hinweise:

- Falls mehr als 5 Ziffern aufeinanderfolgen, sollen die ersten fünf aufeinanderfolgenden Ziffern als PLZ gewertet werden (Beispiel `"Test 12345678"`). Falls die Eingabe mehr als eine PLZ enthält, soll nur die erste PLZ gewertet werden.
- Um eine ASCII-Ziffer in einen Zahlenwert umzurechnen, ziehen Sie den Wert 48 (ASCII `'0'`) ab. Falls das Ergebnis kleiner als 0 oder größer als 9 ist, handelte es sich bei dem Ausgangswert nicht um eine ASCII-Ziffer.
- Um eine Dezimalziffer z an eine Zahl a anzuhängen, rechnen Sie: $a := 10 \cdot a + z$.
- Für die Multiplikation mit 10 können Sie folgenden Trick nutzen: $10 \cdot x = (8 \cdot x) + x + x$. Die Multiplikation mit 8 kann durch eine logische Schiebeoperation umgesetzt werden.

Aufgabe 2: Teilzeichenfolge zählen (10 Punkte)

Aufgabe: Die zu implementierende Funktion `count` soll zählen, wie häufig die Teilzeichenfolge `part` in der Zeichenfolge `text` vorkommt, und die Häufigkeit der Teilzeichenfolge zurückgeben. `part` und `text` sind nullterminierte Zeichenketten. Um die Aufgabe zu lösen, muss die vorgegebene Hilfsfunktion `rollhash` verwendet werden. Die C-Signatur der zu implementierenden Funktion lautet:

`int count( char *text, char *part);`		
`$v0`	`$a0`	`$a1`

Hilfsfunktion: Eine Hashfunktion liefert für eine Zeichenkette einen Hashwert. Der Hashwert ist eine Art Fingerabdruck der Zeichenkette. Die Hashwerte von identischen Zeichenketten sind ebenfalls identisch[1]. Die Besonderheit einer *rollenden Hashfunktion* ist, dass zu der betrachteten Zeichenkette dynamisch Zeichen hinzugefügt und entfernt werden können. Die gehashte Zeichenkette bezeichnen wir als *Hashfenster*.

Die Hilfsfunktion `rollhash` implementiert eine solche rollende Hashfunktion[2]: Das Zeichen `in` wird hinten zum Hashfenster hinzugefügt (angehängt); das Zeichen `out` vorn aus dem Hashfenster entfernt. Falls kein Zeichen hinzugefügt bzw. entfernt werden soll, kann als `in` bzw. `out` der Wert 0 übergeben werden. Der übergebene Hashwert (Funktionsargument `hash`) wird durch das Hinzufügen und/oder Entfernen von Zeichen verändert, wobei `rollhash` den neuen Hashwert zurückgibt. Die C-Signatur von `rollhash` ist:

`int rollhash( int hash, char in, char out);`			
`$v0`	`$a0`	`$a1`	`$a2`

[1] Es ist auch möglich, dass zwei *unterschiedliche* Zeichenketten den gleichen Hashwert liefern (Kollision). Dies ignorieren wir in der Aufgabe.

[2] Hintergrund: `rollhash` verwendet als Hashwerte Permutationen der Länge 8. In hexadezimaler Darstellung enthält dabei ein Hashwert jede Ziffer zwischen 0 und 7 genau einmal. Diese rollende Hashfunktion nutzt Eigenschaften der symmetrischen Gruppe aus (https://de.wikipedia.org/wiki/Symmetrische_Gruppe). Zum Lösen der Aufgabe ist dieser Hintergrund nicht wichtig.

Vorgehen: Es müssen zwei Hashwerte gespeichert werden: ein Hashwert p für `part` und ein Hashwert t für `text`. Zum Durchlaufen der Zeichenkette `part` muss eine Position gespeichert werden. Zum Durchlaufen der Zeichenkette `text` müssen **zwei** Positionen gespeichert werden.

1. *Initialisierungsphase:* Beide Hashwerte müssen zunächst auf 0x76543210 (Hashwert der leeren Zeichenkette) gesetzt werden.
   Sei n die Länge von `part`. Alle n Zeichen aus `part` müssen nun der Reihe nach durch Aufruf von `rollhash` zu p hinzugefügt werden. Außerdem müssen die ersten n Zeichen von `text` durch Aufruf von `rollhash` zu t hinzugefügt werden.
   Nach der Initialisierungsphase wird p nicht mehr verändert.
2. *Suchphase:* In dieser Phase wird nur noch `text` durchlaufen.
   Falls zu Beginn eines Schritts der Suchphase $t = p$ ist, wurde `part` in `text` gefunden und die Häufigkeit muss um 1 erhöht werden.
   Dann wird durch Aufruf von `rollhash` ein Zeichen zu t hinzugefügt und ein Zeichen entfernt. Das hinzuzufügende Zeichen steht immer n Zeichen hinter dem zu entfernenden Zeichen.
   In der Suchphase wird `text` durchlaufen, bis der Nullterminator erreicht wurde und kein Zeichen mehr zu t hinzugefügt werden kann.

Bei diesem Vorgehen handelt es sich um den Rabin-Karp-Algorithmus[3].

Beispiel: Die folgende Tabelle zeigt den Ablauf für `text` = `"AxyzTESTxyz1xyxyz"`, `part` = `"xyz"`. Grün hinterlegte Zeichen werden in dem jeweiligen Schritt zum Hashfenster hinzugefügt, rot hinterlegte Zeichen aus dem Hashfenster entfernt. In der Initialisierungsphase werden in diesem Beispiel `part` und `text` zugleich durchlaufen. Dadurch muss n, die Länge von `part`, in der Programmlogik nicht explizit bestimmt werden.

Hashfenster `text`	**Hashwert `text`**	**Hashfenster `part`**	**Hashwert `part`**
Initialisierungsphase:			
	0x76543210		0x76543210
A	0x36124057	x	0x42163705
Ax	0x32076514	xy	0x07431265
Axy	0x17623540	xyz	**0x13207564**
Suchphase:			
Axyz	**0x13207564**		
xyzT	0x15630742		
yzTE	0x56732140		
zTES	0x25014673		
TEST	0x63720154		
ESTx	0x30267415		
STxy	0x52670314		
Txyz	**0x13207564**		
xyz1	0x06714532		
yz1x	0x17432560		
z1xy	0x32504167		
1xyx	0x54630721		
xyxy	0x21456037		
yxyz	**0x13207564**		

[3] Weitere Hintergrundinformationen: https://de.wikipedia.org/wiki/Rabin-Karp-Algorithmus

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

Eingabe `text`	`part`	**Rückgabewert**
`AxyzTESTxyz1xyxyzxyz1xyz1xyz1xyzHALLOxyz1xyzxyzxyz`	`xyz`	11
`AxyzTESTxyz1xyxyz`	`xyz`	3
`AxyzTESTxyz1xyxyzxyz1xyz1xyz1xyzHALLOxyz1xyzxyzxyz`	`xyz1xyz`	4
`AxyzTESTxyz1xyxyzxyz1xyz1xyz1xyzHALLOxyz1xyzxyzxyz`	`abc`	0
`Zeiger durchlaufen Zeichenketten in kurzer Zeit.`	`Zei`	3

Hinweise:

- Sie können davon ausgehen, dass `text` mindestens gleich lang oder länger als `part` ist. `part` ist außerdem mindestens 1 Zeichen lang.
- Stellen Sie sicher, dass Übereinstimmungen auch ganz am Anfang und ganz am Ende von `text` erkannt und gezählt werden.
- Auch überlappende Übereinstimmungen werden gezählt. Nach diesem Prinzip kommt beispielsweise `"xyz1xyz"` 3 Mal in `"xyz1xyz1xyz1xyz"` vor (1. xyz1xyz1xyz1xyz, 2. xyz1xyz1xyz1xyz, 3. xyz1xyz1xyz1xyz).