Einführung in die Assemblerprogrammierung mit dem MIPS-Simulator SPIM

Reinhard Nitzsche

15.09.97

Inhaltsverzeichnis

Vorwort

Dieses Tutorial ist entstanden, weil im Sommersemester 1996 nur drei statt vier Tutorien für die Vorlesung Rechnerorganisation stattfanden. Professor Schweppe trat mit dem Vorschlag statt des nicht stattfindenden Tutoriums ein Tutorial zur Assemblerprogrammierung zu schreiben an mich heran. Die weitere Ausarbeitung fand dann in Abstimmung mit Professor Rojas statt. Das Ergebnis ist das vorliegende Dokument, das die Vorlesung Rechnerorganisation um die Grundlagen der Assemblerprogrammierung entlasten soll. Tatsächlich bildet der behandelte Stoff nur eine Teilmenge der Assemblerprogrammierung und der Vorlesung Rechnerorganisation.

Im Sommersemester 1997 wurde dieses Tutorial zum ersten Mal in der Vorlesung von Professor Rojas eingesetzt. Die mir zugegangenen Reaktionen der Leserinnen und Leser waren überwiegend positiv. Wertvolle Anregungen aus dem Leserkreis wurden in dieser zweiten Version des Tutorials aufgenommen.

Ich danke allen, die mir mit Korrekturlesungen und Fehlermeldungen geholfen haben.

Berlin, im September 1997

Reinhard Nitzsche

1 SPIM

1.1 Einleitung

Die Assemblerprogrammierung ist auf Simulatoren einfacher zu erlernen als direkt auf einem Rechner. Ein Simulator kann bei Programmierfehlern leicht zurückgesetzt werden, bei einem abgestürzten Rechner dauert dies ungleich länger. Der Simulator erlaubt meist eine bessere Einsicht in die wesentlichen Teile des Rechners als dies bei professionellen Assemblern der Fall ist.

Wir werden den Simulator SPIM verwenden, der den Prozessor **MIPS** R2000 simuliert. Der **MIPS** R2000 ist ein sehr klar strukturierter RISC-Prozessor, der im Gegensatz zu modernen RISC-Maschinen leicht zu verstehen ist. In Abschnitt 2 auf Seite 7 werden wir ein bisschen mehr über diesen Prozessor und die RISC-Technologie erfahren.

Ein weiterer Vorteil der Verwendung von Simulatoren liegt in der Möglichkeit der Simulation von fremden Prozessoren. Unser SPIM, den wir verwenden werden, kann z.B. auf allen UNIX-Rechnern, ab Windows 3.11 und auf Macintosh-Rechnern verwendet werden.

Wir werden uns nun Schritt für Schritt mit der Assemblerprogrammierung vertraut machen. Ich habe mich für eine synthetische Herangehensweise an die Materie entschieden. Wir werden also nicht mit einem Beispielprogramm beginnen, sondern ganz von „unten". Dadurch gerät der Anfang etwas trocken, dieser Nachteil wird aber –so hoffe ich– durch eine bessere Verständlichkeit und gute Nachschlagemöglichkeiten wieder wettgemacht.

Zunächst widmen wir uns in diesem Abschnitt noch dem Simulator, im Abschnitt 2 werden wir uns dann einen groben Überblick über die Technik des Originals, den SPIM R2000, verschaffen. (Wem jetzt der eine oder andere Begriff nicht bekannt ist, muss nicht verzweifeln, sie werden später alle noch erläutert!) Einige Grundregeln der Assemblerprogrammierung werden wir im Abschnitt 3 kennen lernen. Nachdem wir uns in den Abschnitten 4, 5 und 6 mit der Datenhaltung, den Transferbefehlen und den arithmetischen Befehlen vertraut gemacht haben, können wir schon kleinere Programme schreiben. Einige logische Befehle (Abschnitt 8) ebnen uns den Weg zu den Kontrollstrukturen in Abschnitt 9. Es folgt die Hohe Schule der Datenhaltung, nämlich die Haltung von Verbunden und Feldern in Abschnitt 10. Erst danach lernen wir eine weitere Kontrollstruktur kennen, die mehrfache Fallunterscheidung (Abschnitt 11). Schließlich wenden wir uns im Abschnitt 12 dem Kellerspeicher (Stack) zu. Auf gewissenhafte Programmierung kommt es schließlich im Abschnitt 13 an, in dem wir uns mit Konventionen zum Prozeduraufruf beschäftigen werden. Die beiden folgenden Abschnitte 14 und 15 bieten uns einen Ausblick auf weitere Aspekte der Assemblerprogrammierung, nämlich die Unterbrechungsbehandlung und mathematische Koprozessoren.

1.2 Beschaffungsmöglichkeiten

Der SPIM unterliegt lediglich der GNU-Lizenz, kann also frei benutzt und vervielfältigt werden. Im Internet ist er unter `ftp://ftp.cs.wisc.edu/pub/spim/` zu finden. In diesem Verzeichnis befinden sich u.a. folgende Dateien:

- `spim.tar.z` und `spim.tar.gz`: UNIX-Versionen inklusive Quelldateien
- `SPIM.sit.bin` und `SPIM.sit.Hqx.txt`: Macintosh-Programme
- `spim.zip`: Windows-Programme

Die Installation sollte nicht auf allzu große Probleme stoßen. Die UNIX-Versionen müssen zunächst kompiliert werden. Die Windows-Version ist für Windows 95 geschrieben worden. Auf Windows 3.11 muss zunächst noch ein 32-Bit-Treiber installiert werden, der aber mitgeliefert wird. Der Simulator läuft auch unter Windows 3.11 einigermaßen stabil.

1.3 Oberflächen

Abbildung 1: Die x-Oberfläche von SPIM.

Γ Δ Θ Λ Ξ

Π Σ Υ Γ Δ

Θ Λ Ξ Π Σ

Υ Φ Γ Δ Θ

Λ Ξ Π Σ Υ

Φ Ψ Γ Δ Θ

Die Oberflächen der UNIX- und Windows-Versionen unterscheiden sich geringfügig. Einen Überblick verschaffen die Abbildungen 1 bis 2. Die x-Oberfläche besteht im Gegensatz zu den Windows-Oberflächen aus einem einzigen Fenster, in dem Register, Programm- und Datenbereich und Ausgaben des Simulators angezeigt werden. Den Bildschirm des simulierten Rechners kann man über die Taste „Terminal“ erhalten.

Bei den Windows-Oberflächen sind Register, Programm- und Datenbereich sowie Ausgaben des Simulators („Session“) über einzelne Fenster zugänglich (Menü „Windows“). Im gleichen Menü befindet sich der SPIM-Bildschirm.

Abbildung 2: Die Windows 95-Oberfläche von SPIM.

Γ Δ Θ Λ Ξ

Π Γ Δ Θ Λ

Ξ Π Γ Δ Θ

Λ Ξ Π Σ Υ

Wer mit „seinem" Betriebssystem vertraut ist, sollte keine Probleme mit der Benutzung der jeweiligen Oberfläche haben. Zur Einarbeitung seien die jeweiligen Hilfefunktionen und die mitgelieferten Dokumentationen bzw. [Pat, S. A-36 bis A-45] empfohlen.

Zu beachten sind jedoch noch folgende Hinweise:

- Auch die letzte Programmzeile muss mit der Eingabetaste beendet werden. Andernfalls wird für die letzte Zeile eine Fehlermeldung angezeigt.
- Bei den Windows-Versionen darf die über `File/Load` geladene Datei nur in dem Verzeichnis `C:\SPIM` stehen, da andernfalls bei `File/Clear` das Programm *beendet* wird.
- Die Windows-Versionen erlauben nicht das Memory-Mapped-IO[1].

[1] s. Abschnitt 15 auf Seite 66

2 Maschinenmodell des SPIM

SPIM simuliert den RISC-Prozessor **MIPS** R2000. Dieser zeichnet sich durch eine klare Architektur und einen übersichtlichen Befehlssatz aus.

RISC

Das *RISC*-Konzept sieht vor, statt Prozessoren mit einer großen Zahl von Befehlen auszustatten, nur vergleichweise wenig Befehle zu implementieren. RISC steht für **R**educed **I**nstruction **S**et **C**omputer, das Pendant heißt CISC (**C**omplex **I**nstruction **S**et **C**omputer). Die wenigen implementierten RISC-Befehle können sehr viel schneller ausgeführt werden. Für viele Arbeiten müssen allerdings mehr RISC-Befehle verwendet werden als bei CISC-Rechnern.

Das Grundmodell des SPIM wurde 1984 von John Hennesy, dem Ko-Autor von [Pat] an der Universität Stanford entwickelt. Die MIPS-Prozessoren werden u.a. von DEC verwendet. Vier Jahre zuvor entwickelte David A. Patterson, der andere Autor von [Pat] an der Berkeley-Universität den Begriff RISC und den RISC-Prozessor RISC I, der für die SPARC-Prozessoren der Firma Sun Pate stand und der mit ganzen drei (!) Befehlen auskam [Tan, S. 435].

Die gängigen Prozessoren der Personal-Computer wie z.B. INTEL 80x86 und Pentium, aber auch die Prozessoren der Motorola 68xxx-er Familie sind CISC-Rechner.

2.1 Koprozessoren

Die **MIPS** R2000-CPU verfügt über die Möglichkeit mehrere Koprozessoren anzusteuern. Dieses Koprozessoren sind für von Gleitkommaberechnungen oder Betriebssystemnahe Aufgaben zuständig. Mit diesen Möglichlichkeiten werden wir uns aber erst in Abschnitt 14 auf Seite 60 und Abschnitt 15 auf Seite 66 beschäftigen.

2.2 Register

32 Register

general purpose register

Der MIPS verfügt über *32 Register*. Jedes Register enthält ein 32-Bit-Wort und kann -theoretisch- für jeden Zweck verwendet werden. Man nennt solche Register auch *general purpose register*. Eine Ausnahme stellt das Register `$zero` dar, dessen Wert Null nicht verändert werden kann.

Zusätzlich zu diesen 32 Registern verfügt der SPIM noch über zwei weitere Register mit den Namen `lo` für „low“ und `hi` für „high“, die bei der Multiplikation und der Division verwendet werden. Im eingangs erwähnten Koprozessor 0 finden sich vier weitere Spezialregister, die für uns erst einmal nicht von Bedeutung sind.

Die Adressierung eines Registers kann durch ein Dollarzeichen (`$`) und die Registernummer `0-31` erfolgen. Die Register `lo` und `hi` sind nur mit Spezialbefehlen lesbar und schreibbar. Von der Adressierung über die Registernummern ist aber abzuraten, da dies zu einem sehr unübersichtlichen Code führt. Besser ist die Verwendung der Register*namen*, die der Assembler (siehe Abschnitt 3.1 auf Seite 10) dann in die zugehörigen Register*nummern* umsetzt. Die Registernummern sind aber bei der Fehlersuche hilfreich, da SPIM bei geladenen Programmen die Register nur durch ihre Registernummern angibt.

Bezüglich der Verwendung der Register gibt es eine Konvention.[2] Sie ist unbedingt einzuhalten, damit Programmteile verschiedener Autoren austauschbar sind. Die Nichtbeachtung einzelner Konventionen kann zu nicht vorhersagbarem Laufzeitverhalten des Programmes führen.

[2] [Pat] verwendet leider nicht konventionsgemäße Register*nummern*

Abbildung 3: MIPS-Register und ihre Verwendungszwecke nach Registernummern

Register- -name	-nr.	vereinbarte Nutzung	Bemerkungen
`$zero`	0	Enthält den Wert 0	kann nicht verändert werden.
`$at`	1	temporäres Assemblerregister	Nutzung ausschließlich durch den Assembler!
`$v0` `$v1`	2 3	Funktionsergebnisse 1 und 2	auch für Zwischenergebnisse
`$a0` `$a1` `$a2` `$a3`	4 5 6 7	Argumente 1 bis 4 für den Prozeduraufruf	
`$t0` `$t1` `$t2` `$t3` `$t4` `$t5` `$t6` `$t7`	8 9 10 11 12 13 14 15	temporäre Variablen 1-8	Können von aufgerufenen Prozeduren verändert werden.
`$s0` `$s1` `$s2` `$s3` `$s4` `$s5` `$s6` `$s7`	16 17 18 19 20 21 22 23	langlebige Variablen 1-8	Dürfen von aufgerufenen Prozeduren nicht verändert werden.
`$t8` `$t9`	24 25	temporäre Variablen 9 und 10	Können von aufgerufenen Prozeduren verändert werden.
`$k0` `$k1`	26 27	Kernel-Register 1 und 2	Reserviert für Betriebssystem, wird bei Unterbrechungen verwendet.
`$gp`	28	Zeiger auf Datensegment	
`$sp`	29	Stackpointer	Zeigt auf das erste *freie* Element des Stacks.
`$fp`	30	Framepointer	Zeiger auf den Prozedurrahmen
`$ra`	31	Return address	Enthält nach Aufruf u.a. des Befehls `jal` die Rücksprungadresse.

`$zero`: Enthält den Wert Null. Dieses Register ist auf Schaltkreisebene realisiert und kann *nicht* verändert werden.

`$at`, `$k0` und `$k1`: Reserviert für Berechnungen des Assemblers. Die `$k`-(Kernel)-Register werden bei der Unterbrechungsbehandlung verwendet. Obwohl sich das `$at`-Register lesen und schreiben lässt, sollte es keinesfalls verwendet werden! Wir werden uns in Abschnitt 15 auf Seite 66 eingehender mit diesen Registern auseinander setzen.

`$a0` bis `$a3`: Prozedurargumente, weitere Argumente müssen über den Stack übergeben werden, vergleiche Abschnitt 13 auf Seite 49.

`$v0` und `$v1`: Funktionsergebnisse, gegebenenfalls können diese Register auch bei der Auswertung von Ausdrücken benutzt werden, vergleiche Abschnitt 13.

`$t0` bis `$t9`: Diese Register sind für die Haltung kurzlebiger (temporärer) Variablen bestimmt. Sie können nach einem Prozeduraufruf von der aufgerufenen Prozedur verändert werden. Nach einem Prozeduraufruf kann also *nicht* davon ausgegangen werden, dass die Werte in den `$t`-Registern unverändert sind, vergleiche Abschnitt 13.

`$s0` bis `$s7`: Die `$s`-Register dienen der Haltung langlebiger Variablen. Sollen sie in einem Unterprogramm geändert werden, so müssen sie zuvor gesichert werden, vergleiche Abschnitt 13.

`$gp`: Globaler Zeiger auf die Mitte eines 64K großen Speicherblocks im statischen Datensegment, das für alle Dateien sichtbar ist.

`$fp`: Framepointer, vergleiche Abschnitt 13.

`$sp`: Stackpointer, vergleiche Abschnitt 12 auf Seite 45.

`$ra`: Returnaddress, Rücksprungadresse nach dem Aufruf von `jal` und anderen Befehlen.

3 Grundprinzipien der Assemblerprogrammierung

3.1 Assembler, Assemblersprache und Maschinensprache

Prozessoren können lediglich numerische Instruktionen ausführen. Beispielsweise bedeutet die SPIM-Anweisung 0000 0010 0011 0010 0100 0000 0010 0000: *„Addiere den Inhalt der Register 17 und 18 und lege das Ergebnis im Register 8 ab.“*. Solche vom Prozessor ausführbaren Befehle nennt man *Maschinenbefehle*.

Maschinenbefehl

Die Gesamtheit aller Maschinenbefehle die ein Rechner ausführen kann nennt man *Maschinensprache*. Hinter der oben genannten scheinbar chaotischen Zahl steckt zwar ein System (die ersten sechs Nullen und die letzten sechs Ziffern stehen beispielsweise für „Addition“), aber schon die ersten Programmierer waren aus verständlichen Gründen doch sehr an einer Visualisierung dieser numerischen Befehle interessiert.

Maschinensprache

Assembler
symbolische Bezeichner

Einfachste *Assembler* sind Programme, die aus Buchstaben bestehenden *symbolischen Bezeichnern* einen Maschinenbefehl zuordnen. Tatsächlich muss der Assembler auch noch einige weitere Entscheidungen treffen, etwa welche Register verwendet werden und wie diese zu codieren sind.

symbolische Adressen
Marke

Aber die meisten Assembler können mehr als diese recht simple 1:1-Umsetzung von symbolischen Befehlen in Maschinenbefehle. Zur Grundausstattung eines Assemblers gehört z.B. die Verwaltung *symbolischer Adressen*. Das sind Wörter oder *Marken*, die der Programmierer im Assemblerprogramm definieren kann und die dann vom Assembler so verwaltet werden, dass von anderen Stellen im Programm her auf diese Marken Bezug genommen werden kann. Dies ist äußerst nützlich bei Sprunganweisungen, bei denen der Programmierer andernfalls erst ermitteln müsste, an welche Stelle (oder über wie viele Speicherworte) gesprungen werden soll. Auch Variablen werden durch solche Marken realisiert. Die meisten Assembler erlauben auch die Bezugnahme auf Marken, die gar nicht in der Datei selber definiert wurden. Dies ermöglicht die Nutzung bereits vorhandener Bibliotheken.

Assemblersprache

Den Wortschatz eines Assemblers nennt man die *Assemblersprache*, oft aber auch ebenfalls Assembler.

3.2 Befehlsarten

Im einfachsten Fall besteht eine Assemblersprache wie gesagt aus einer 1:1-Umsetzung von symbolischen Befehlen in Maschinensprache. Assembler verfügen jedoch auch über weitere Befehlsarten:

Pseudobefehl

(1) Ein *Pseudobefehl* ist eine Erweiterungen der Maschinensprache. Pseudobefehle werden vom Assembler in eine Folge von Maschinenbefehlen umgesetzt. Sie werden vom Assembler zur Verfügung gestellt und können vom Benutzer nicht neu definiert werden. Auch bestimmte Adressierungsarten, die der Prozessor gar nicht beherrscht, fallen in diese Kategorie. Für den Assemblerprogrammierer sind Pseudobefehle (fast) wie Maschinenbefehle zu benutzen, sie unterscheiden sich rein äußerlich nicht von Maschinenbefehlen.

Assembleranweisung
Direktive

(2) Eine *Assembleranweisung oder Direktive* ist eine Anweisung an den Assembler, etwas zur Assemblierzeit zu tun, z.B. Platz für Variablen zu lassen o.ä. Eine Direktive führt *nicht* zur Erzeugung von Maschinenbefehlen! Die Direktiven des SPIM beginnen alle mit einem Punkt, z.B. `.data`.

Makro

Es gibt auch Assembler, die benutzerdefinierte Pseudobefehle oder *Makros* erlauben. Der SPIM beherrscht diese Technik jedoch nicht.

3.3 Aufbau einer Assembler-Befehlszeile

Grundsätzlich kann in jede Zeile des Assembler-Programms nur ein einziger Assembler- oder Pseudobefehl geschrieben werden:

$\langle$*Marke*$\rangle$: $\langle$*Befehl*$\rangle$ $\langle$*Arg 1*$\rangle$ $\langle$*Arg 2*$\rangle$ $\langle$*Arg 3*$\rangle$ #$\langle$*Kommentar*$\rangle$
$\langle$*Marke*$\rangle$: $\langle$*Befehl*$\rangle$ $\langle$*Arg 1*$\rangle$, $\langle$*Arg 2*$\rangle$, $\langle$*Arg 3*$\rangle$ #$\langle$*Kommentar*$\rangle$

Die Argumente eines Befehls können nach Belieben durch Kommata oder Lücken (auch Tabulatoren) getrennt werden. $\langle$*Marke*$\rangle$ ist natürlich optional. Alle Zeichen vom Kommentarzeichen # an werden als Kommentar gewertet und überlesen. Kommentar

Abgesehen von einigen Assembleranweisungen haben fast alle SPIM-Befehle zwischen einem und drei Argumenten.[3] Zum Beispiel haben fast alle arithmetischen Befehle drei Argumente, zwei zur Angabe der Quelle und eines zur Angabe des Ziels. Die Befehle zur Übertragung von Daten zwischen Prozessor und Hauptspeicher haben zwei Argumente.

Die Argumente treten *immer* in folgender Reihenfolge auf:

1.) Register des Hauptprozessors, zuerst das Zielregister,

2.) Register des Koprozessors,

3.) Adressen, Werte oder Marken

Marken können übrigens auch in einer Zeile für sich definiert werden. Sie gelten dann für den folgenden Befehl. Dies ist bei längeren Markennamen, etwa bei Prozedurnamen, praktisch. In Markennamen sind auch Ziffern erlaubt, allerdings nicht als erstes Zeichen. Auch der Unterstrich (_) ist zugelassen. Marken dürfen nicht genauso heißen wie ein Befehl! Die Folge sind sehr schwer erklärliche Fehlermeldungen. In Abschnitt A.2 auf Seite 76 befindet sich eine alphabetische Übersicht der Befehlsnamen.

3.4 Trennung von Programm und Daten

Eines der Grundprinzipien des von-Neumann-Rechners ([Pat, S. 32], [Tan, S. 17-18], [Coy, S. 19-23]) ist ein gemeinsamer Speicher für Daten und Programme. Dieser gemeinsame Speicher führt aber auch zu Problemen. Der Programmierer muss verhindern, dass der Rechner versucht, die abgelegten Daten auszuführen oder Programmteile als Daten zu lesen oder zu überschreiben.

Ein Möglichkeit besteht beispielsweise darin, alle Daten an einem Ende des Programmes unterzubringen und die Ausführung des Programmes entweder vorher zu stoppen oder über den Datenbereich hinweg zu springen.

Eleganter ist das Konzept des SPIM. Der SPIM teilt den Hauptspeicher in mehrere sogenannte *Segmente: Daten-, Text- und Stacksegment* ein. So gibt es ein *Datensegment* für die Daten und ein *Textsegment* für das Programm. Ferner gibt es noch jeweils ein Text- und Datensegment für das Betriebssystem und ein *Stacksegment* für den Stack. Segment

Die Segmente haben jeweils eine feste Größe und einen bestimmten Verwendungszweck. So gibt es u.a. ein Segment für das Benutzerprogramm und eines für die Benutzerdaten. Der Simulator erlaubt die Festlegung der Größe der Segmente mit einem Parameter beim Aufruf des Simulators.

Mehr über das Hauptspeicherkonzept des SPIM R2000 werden wir in Abschnitt 12.1 auf Seite 45 und besonders in Abbildung 8 auf Seite 46 erfahren.

Mit den Direktiven `.text` und `.data` können wir festlegen, in welches Segment die folgenden Befehle geschrieben werden sollen.

[3] Die einzigen Befehle ohne Argumente sind: `syscall`, `nop`, `bc`$\langle z\rangle$`t`, `bc`$\langle z\rangle$`f` und `rfe`

.data-Direktive

Alle auf die `.data`-*Direktive* folgenden Befehle (fast immer Direktiven) werden in das Datensegment eingetragen. Theoretisch können hier auch Befehle stehen, die zwar übersetzt, aber nie ausgeführt werden.

.text-Direktive

Alle auf die `.text`-*Direktive* folgenden Befehle werden in das Textsegment eingetragen. Sie können ausgeführt werden.

Die `.data`-Direktive dient also zur Ablage von Daten im Datensegment, die `.text`-Direktive zur Ablage von Befehlen im Textsegment.

3.5 Tips zur übersichtlichen Gestaltung von Assemblerprogrammen

Assemblerprogramme werden noch schneller als Hochsprachenprogramme unübersichtlich. Daher sollten folgende Regeln unbedingt eingehalten werden:

(1) Zu jedem Assemblerprogramm gehört eine Beschreibung seines Verhaltens. Dies kann z.B. in Form eines Hochsprachenalgorithmus zu Beginn des Programmes erfolgen. An diesen Hochsprachenalgorithmus sollte man sich dann aber auch peinlich genau halten.

(2) Das Programm muss umfangreich kommentiert werden, andernfalls wird eine spätere Fehlersuche oder Veränderung des Programmes sehr zeitaufwendig. Es ist jedoch keinesfalls sinnvoll jeden Befehl in seiner Wirkung zu beschreiben, z.B.:

```
add    $t1, $t2, $t1      # $t1 := $t2 + $t1
```

Diese Information ist redundant, da sie unmittelbar aus der Programmzeile entnommen werden kann. Zweckmäßiger ist eine Information über den Sinn und den Zusammenhang der Zeile, z.B.:

```
add    $t1, $t2, $t1      # summe := messung + summe;
```

Hier bietet sich auch die Möglichkeit das eingangs erwähnte Hochsprachenprogramm zu kopieren und dieses dann Schritt für Schritt zu übersetzen.

(3) Assemblerprogramme werden etwas übersichtlicher, wenn sie in Spaltenform geschrieben werden. In der ersten Spalte (bis zum ersten Tabulatorstop) stehen alle Marken. Dadurch sind Sprungziele und Variablen schnell zu finden. In der zweiten Spalte (bis zum zweiten Tabulatorstop) stehen alle Befehle. Vom dritten Tabulatorstop an folgen die Argumente, die Kommentare stehen hinter dem fünften oder sechsten Tabulatorstop.

(4) Die Namen von Marken sollten für sich sprechen, z.B. `for:` am Anfang einer Zählschleife. Einfache Nummerierungen wie `a:`, `b:` sind sinnlos.

(5) Die Zuordnung der Variablen des Hochsprachenprogramms zu den Registern sollte kommentiert werden, da dies eine häufige Fehlerquelle ist. Die Konventionen zur Verwendung der Register (siehe Abschnitt 2 auf Seite 7) sollten unbedingt eingehalten werden.

3.6 Aufbau eines Assemblerprogramms

Den prinzipiellen Aufbau eines Assemblerprogramms können wir dem folgenden Beispiel entnehmen:

Beispiel 1: Ein erstes Beispielprogramm ohne besonderen praktischen Wert

```
# FIRST.S
# berechnet den Umfang des Dreiecks mit den Kanten x, y, z
.data
x:      .word   12
y:      .word   14
z:      .word   5
U:      .word   0
.text
main:   lw      $t0, x          # $t0 := x
        lw      $t1, y          # $t1 := y
        lw      $t2, z          # $t2 := z
        add     $t0, $t0, $t1   # $t0 := x+y
        add     $t0, $t0, $t2   # $t0 := x+y+z
        sw      $t0, U          # U := x+y+z
        li      $v0, 10         # EXIT
        syscall
```

Zu beachten ist, dass die Marke `main:` immer vorhanden sein muss, da diese vom SPIM-Betriebssystem angesprungen wird. Es ist empfehlenswert, jedes Programm wie in dem Beispiel angegeben zu beenden. Den Befehl `syscall` werden wir in Abschnitt 7 auf Seite 26 kennen lernen.

Anzumerken ist noch, dass es egal ist, ob man zunächst das Datensegment und dann das Textsegment verwendet oder umgekehrt. Die Einträge in die beiden Segmente können auch gemischt erfolgen. Wesentlich ist nur, dass die Reihenfolge der Einträge ins Textsegment korrekt ist.

3.7 Notation der Befehle

Wenn wir neue Befehle kennen lernen, so führen wir sie mit einer Tabelle wie der folgenden ein:

Befehl	Argumente	Wirkung	Erläuterung
`div`	`Rd, Rs1, Rs2`	Rd=INT(Rs1/Rs2)	divide
`li`℗	`Rd, Imm`	Rd=Imm	load immediate
	:=	`ori Rd, $zero, Imm`	

In der Spalte „Befehl“ steht der Name (der symbolische Bezeichner) des Befehls. Pseudo-Befehle sind mit dem Zeichen ℗ markiert. In der nachfolgenden Zeile steht dann die Befehlsfolge, zu der der Pseudobefehl aufgelöst wird.

In der Spalte „Argumente“ stehen die Argumente, die der Assembler erwartet.

In der Spalte „Wirkung“ wird die Wirkung des Befehls, meistens algebraisch, beschrieben. Die Spalte „Erläuterung“ enthält den ausgeschriebenen Namen des Befehls auf englisch.

Die angegebenen Argumente geben auch Aufschluss über die Art der Argumente, Rd und Rs stehen beispielsweise für Register. Eine komplette Übersicht liefert Abbildung 4 auf der folgenden Seite.

Die Tabelle wird leichter verständlich, wenn wir die Adressierungsarten kennen gelernt haben. Die angehängten Kleinbuchstaben `s` oder `d` stehen für „source“ (Quelle) bzw. „destination“ (Ziel). Viele Befehle haben zwei Quellargumente, diese werden durch eine nachstehende `1` bzw. `2` gekennzeichnet.

Auf den Seiten 70 bis 77 befinden sich hilfreiche Übersichten zum Nachschlagen.

Abbildung 4: Argumentarten

Notation	Erläuterung	Beispiel
`Rd`, `Rs1`, `Rs2`, `Rs`	Registeroperanden, d: destination (Ziel), s: source (Quelle)	`$t1`, `$s3`
`Imm`	Unmittelbar (Immediate)	`25`, `0x23`
`label`, `Adr`	Hauptspeicheradressen	`main`, `4+feld`, `feld($t1)`, `3+feld($t1)`

4 Datenhaltung I: ganze Zahlen und Zeichenketten

In längeren Programmen gibt es Variablen, die unterschiedlich lange und unterschiedlich oft benötigt werden. Einige werden nur ein einziges Mal benötigt um Zwischenergebnisse abzulegen, andere werden über lange Abschnitte immer wieder benötigt.

Wir unterscheiden daher zwischen *lang-* und *kurz*lebigen Variablen. Die langlebigen Variablen werden meist im Hauptspeicher gehalten, damit sie nicht wertvolle Register belegen. Die kurzlebigen werden während ihrer Lebensdauer meist in Registern gehalten.

lang- und kurzlebige Variablen

Unser SPIM kann Berechnungen ausschließlich auf Registern durchführen. Variablen, die im Speicher abgelegt sind, müssen also zunächst mit Ladebefehlen in Register geladen werden und nach erfolgten Berechnungen ggf. wieder zurückgespeichert werden. Eine solche Rechnerarchitektur nennt man *Load-Store-Architektur*. Im Gegensatz dazu gibt es Rechnerarchitekturen, die Berechnungen auch im Hauptspeicher zulassen, z.B. die Motorola-CPUs der 68er Serie und die Intel 80x86.

Load-Store-Architektur

Jedes Register des SPIM enthält ein 32-Bit-Wort. Man sagt auch es ist 32 Bit *breit*.

Registerbreite

4.1 Ganze Zahlen im Datensegment

4.1.1 Die `.word`-Direktive

`.word`

Im Hauptspeicher legt die Direktive .word die nachfolgenden Zahlen in je ein Doppelwort. Sinnvollerweise wird diese Direktive im Datensegment (nach der `.data`-Direktive) verwendet. Die Anwendung im Textsegment ist zwar zulässig, aber äußerst schlechter Stil. Es besteht stets die Gefahr, dass durch ungeschickte Programmierung versucht wird diese Daten als Befehle auszuführen.

Die Werte können dezimal oder hexadezimal angegeben werden. Hexadezimale Werte müssen mit „`0x`" beginnen.

Um auf die im Hauptspeicher abgelegten Variable komfortabel zugreifen zu können, sollte zudem eine Marke vor die `.word`-Direktive gesetzt werden:

```
marke: .word 256 0x100
```

Diese Direktive würde also in die beiden nächsten noch unbenutzten 32-Bit-Wörter im Datensegment jeweils den Wert 256 (=`0x100`)ablegen.

4.1.2 Weitere Ganzzahltypen

Eng verwandt mit der `.word`-Direktive sind die Direktiven

`.half`, `.byte`

- `.byte` für 8-Bit-Zahlen und
- `.half` für 16-Bit-Zahlen.

Die Initialisierung und das Anlegen mehrerer Werte geschieht wie bei der `.word`-Direktive.

4.2 Zeichenketten im Datensegment: die `.asciiz`-Direktive

ASCII-Code

Neben Zahlen bilden Zeichen, also einzelne Buchstaben oder Ziffern, den zweiten grundlegenden Datentyp. Kein Speicher oder Prozessor kann Zeichen direkt speichern oder verarbeiten. Seit Anbeginn der Textverarbeitung im weitesten Sinne behilft man sich damit die Zeichen als Zahlen zu kodieren und dann diese Zahlen zu verarbeiten. Nur bei der Ein- und Ausgabe wird dann eine lesbare Form verwendet. Es gibt eine Reihe verschiedener solcher Zuordnungen (Codes), die sich jedoch im wesentlichen gleichen. Nur die Anordnung von Sonderzeichen (etwa der deutsche Umlaut „Ä“) ist höchst unterschiedlich. Standard ist der *ASCII-Code*[4], der in seiner Erweiterung 256 Zeichen (1 Byte) umfaßt.

`.asciiz`

Texte werden mit der `.asciiz`-Direktive in den Speicher geschrieben:

```
marke: .asciiz "Hallo Welt"
```

Nullterminierung

Der Text wird in doppelte Anführungszeichen eingeschlossen. Während des Assemblierens werden die einzelnen Zeichen des Textes kodiert und je vier von ihnen in ein Doppelwort geschrieben. An das Ende des Textes wird das Zeichen mit dem Code 0 (Chr 0) gesetzt, damit bei der Textausgabe das Ende des Textes erkannt werden kann. Dieses Verfahren nennt man *Nullterminierung*. Ein anderes Verfahren, das von vielen Modula-2-Compilern verwendet wird, ist im ersten Byte (Zeichen) der Zeichenkette deren Länge abzulegen. Zeichenketten können bei diesem Verfahren maximal 255 Zeichen enthalten.

`.ascii`

Es gibt auch eine `.ascii`-Direktive, die das Chr 0 nicht anhängt. Sie ist beispielsweise hilfreich, wenn längere Texte im Assemblerprogramm auf mehrere Zeilen verteilt werden müssen. In diesem Fall teilen wir die Zeichenkette in mehrere Assemblerzeilen auf, wobei nur die letzte Zeile die Direktive `.asciiz` enthält und alle anderen Textteile mit `.ascii` abgelegt werden:

```
marke: .ascii  "Der Mond ist aufgegangen, die goldnen Sternl"
       .asciiz "ein prangen, am Himmel hell und klar."
```

Um besondere Zeichen im Speicher abzulegen, müssen die in Abbildung 5 auf dieser Seite aufgeführten Kombinationen verwendet werden.

Abbildung 5: Zeichenkombinationen

`\n`:	Neue Zeile
`\t`:	Sprung zum nächsten Tabulator
`\"`:	Das doppelte Anführungszeichen

Ebenso wie die `.word`-Direktive sollten die Direktiven `.asciiz` und `.ascii` nur im Datensegment verwendet werden.

Übungen

Übung 1
Setze das folgende Modula-2-Codefragment in MIPS-Code um:

```
VAR
i,j:   CARDINAL;
Summe: LONGCARD;
s1:    ARRAY[0..20] OF CHAR;
```

[4] ASCII: American Standard Code for Information Interchange

Verwende die gleichen Bezeichnernamen und Datentypen (CARDINAL: 16 Bit, LONGCARD: 32 Bit), initialisiere `i` *und* `j` *mit 0,* `Summe` *mit 200 und* `s1` *mit dem Text „Alle meine Entchen". Die Obergrenze der Modula-Zeichenkette ist für die Umsetzung hier irrelevant.*

Übung 2
Durch eine Eingabemaske sollen Vor- und Nachname, Straße, Postleitzahl und Ort einer Person nacheinander erfaßt werden. Lege die dafür nötigen Texte jeweils einzeln im Datensegment ab.

4.3 Die Datenausrichtung im Datensegment

Die meisten Transferbefehle, die wir im folgenden Abschnitt kennen lernen werden, arbeiten nur auf *ausgerichteten Daten (aligned Data)*. Ein Datum ist ausgerichtet, wenn seine Hauptspeicheradresse ein ganzzahliges Vielfaches seiner Größe ist. aligned Data

Zum Beispiel ist ein Halbwort (2 Byte Größe) ausgerichtet, wenn es an einer geraden Hauptspeicheradresse liegt, ein Vollwort (4 Byte Größe), wenn es an einer durch vier teilbaren Hauptspeicheradresse liegt.

Man kann diese Datenausrichtung durch die Direktive `.align` beeinflussen. Für unsere Zwecke genügt es zu wissen, dass die Direktive `.word`, `.half` etc. die Daten stets richtig ablegen und wir uns daher nicht um deren Ausrichtung zu kümmern brauchen. `.align`

5 Transferbefehle

Wie im vorangegangenen Abschnitt erwähnt hat der SPIM eine Load-Store-Architektur. Wollen wir also mit denen im Datensegment abgelegten Daten arbeiten, so müssen wir sie zunächst in ein Register im Prozessor laden. Hierfür gibt es eine Reihe von Befehlen, die wir jetzt kennen lernen werden.

5.1 Ladebefehle

Es gibt eine recht große Zahl von Ladebefehlen für verschiedene Datentypen. Der am häufigsten verwendete Befehl ist sicherlich der Befehl `lw`:

Befehl	Argumente	Wirkung	Erläuterung
`lw`	`Rd, Adr`	Rd:=MEM[Adr]	Load word

`lw` lädt den Wert, der an der durch `Adr` angegebenen Adresse im Hauptspeicher steht, in das Register `Rd`. Die Möglichkeiten, mit denen man diese Adresse angeben kann, lernen wir sofort kennen:

5.1.1 Adressierungsmodi I: Register Indirekte, direkte und indexierte Adressierung

Der **MIPS** kennt mehrere Adressierungsmodi, von denen uns im Zusammenhang mit Ladebefehlen nur drei interessieren:

Register-indirekt

- `(Rs)`: Der gesuchte Wert steht im Hauptspeicher an der Adresse, die im Register `Rs` angegeben ist (*Register-indirekt*).

direkt

- `label` oder `label+Konstante`: Der gesuchte Wert steht im Hauptspeicher an der Adresse der angegebenen Marke, die ggf. zuvor um die Konstante erhöht oder verringert wird (*direkt*).

indexiert

- `label(Rs)` oder `label+Konstante(Rs)`: Der gesuchte Wert steht im Hauptspeicher an der Adresse der angegebenen Marke plus dem Inhalt des Registers `Rs`, die Konstante wird ggf. zuvor addiert oder subtrahiert (*indexiert*).

Dies sind die drei Möglichkeiten, durch `Adr` angegebene Parameter zu notieren.

Die vorangegangenen drei Adressierungsarten werden vom Assembler alle auf die indexierte Adressierung zurückgeführt. So wird bei direkter Adressierung das Register `$zero` mit dem Wert 0, bei der Register indirekten Adressierung für den Wert der Marke der Wert 0 verwendet. Die Konstanten werden zur Assemblierzeit bzw. beim Binden des Programmes, also vor der Programmausführung, ausgewertet.

Beispiel 2: Beispiele zur direkten Adressierung

```
# ADRMODE1.S
.data
var:    .word   20, 4, 22, 25, 7
#
.text
main:   lw  $t1, var         # $t1 enthaelt "20" (direkte Adr.)
        lw  $t1, 4+var       # $t1 enthaelt "4"  (direkte Adr.)
        lw  $t2, var($t1)    # $t2 enthaelt "4"  (indexierte Adr.)
        lw  $t2, 8+var($t1)  # $t2 enthaelt "25" (indexierte Adr.)
```

Übungen

Übung 3
Lade den Wert 7 aus Beispiel 2 auf der vorangegangenen Seite in das Register `$t4`

Übung 4
Register-indirekte und direkte Operanden werden vom Assembler auf die indexierte Adressierung umgesetzt. Gebe einen Algorithmus hierfür an und überprüfe ihn anhand eines Beispielprogramms.

5.1.2 Weitere Ladebefehle

Die folgenden Ladebefehle arbeiten analog zu `lw`:

Befehl	Argumente		Wirkung	Erläuterung
`lb`	`Rd, Adr`		Rd:=MEM[Adr]	Load byte
`lbu`	`Rd, Adr`		Rd:=MEM[Adr]	Load unsigned byte
`lh`	`Rd, Adr`		Rd:=MEM[Adr]	Load halfword
`lhu`	`Rd, Adr`		Rd:=MEM[Adr]	Load unsigned halfword
`ld`Ⓟ	`Rd, Adr`		Lädt das Doppelwort an der Stelle `Adr` in die Register `Rd` und `Rd`+1	Load double-word
		`:=`	`lw Rd, Adr` `lw Rd+1, Adr+4`	

Die Ladebefehle für Halbwörter (16 Bit) und Bytes (8 Bit) gibt es auch in einer *unsigned*-Version. Die normalen Befehle (`lb`, `lh`) wandeln automatisch negative 8- bzw. 16-Bit-Zahlen in Doppelwörter um: aus dem Byte `0xF0` (= −16) wird dann `0xFFFF FFF0`. Diese Umwandlung ist erwünscht, wenn es sich tatsächlich um vorzeichenbehaftete Zahlen handelt, aber unerwünscht, wenn die gespeicherten Zahlen vorzeichenlos sind, wenn also `0xF0` für die Zahl 240 steht. In diesem Fall soll im Register nach dem Laden eben `0x0000 00F0` stehen. Hierfür stehen die `unsigned`-*Befehle* zur Verfügung, die Bytes und Halbwörter unverändert laden. unsigned unsigned-Befehle

Die oben genannten Ladebefehle gehen davon aus, dass die Daten ausgerichtet (`align`*ed*) sind. Die Adressen von Wörtern müssen also immer durch vier teilbar sein, andernfalls wird das Programm zur Laufzeit abgebrochen. Die Adressen von Halbwörtern müssen dementsprechend gerade sein. Gelegentlich sollen aber auch Wörter oder Halbwörter über ihre Grenzen hinweg geladen werden. Hierfür stehen dem fortgeschrittenen Assemblerprogrammierer die folgenden Befehle zur Verfügung: aligned

Befehl	Argumente	Wirkung	Erläuterung
`ulw`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load word
`ulh`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load halfword
`ulhu`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load unsigned halfword
`lwr`	`Rd, Adr`	Rd:=MEM[Adr] DIV 2^{16}	Load word right
`lwl`	`Rd, Adr`	Rd:=MEM[Adr] MOD 2^{16}	Load word left

Übungen

Übung 5
Lege im Datensegment die Werte `0xFFFF 0000` *und* `0x0000 FFFF` *ab und lade sie mit den Befehlen* `lb`, `lbu`, `lh` *und* `lhu`. *Erkläre die Ergebnisse!*

5.2 Speicherbefehle

Nachdem wir Werte in den Hauptspeicher geladen haben und mit ihnen Berechnungen angestellt haben, wollen wir sie häufig wieder in den Hauptspeicher zurückschreiben um später nochmals auf sie zugreifen zu können.

Analog zu den Ladebefehlen gibt es für jeden Datentyp ein oder mehrere Speicherbefehle. Der wichtigste ist der Befehl `sw`:

Befehl	Argumente	Wirkung	Erläuterung
`sw`	`Rs, Adr`	MEM[Adr]:=Rs	store word

Er speichert den Wert des Registers `Rs` an der angegebenen Stelle im Hauptspeicher. Beachte, dass der Zieloperand diesmal an *zweiter* Stelle steht und nicht wie bisher an *erster*.

Adressierungsmodi

Als *Adressierungsmodi* stehen wieder die unter Abschnitt 5.1.1 auf Seite 18 genannten Modi zur Verfügung.

Weitere Speicherbefehle sind:

Befehl	Argumente	Wirkung	Erläuterung
`sb`	`Rs, Adr`	MEM[Adr]:=Rs MOD 256	store byte
`sh`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store halfword
`sd`Ⓟ	`Rs, Adr`	MEM[Adr]:=Rs + 2^{16} Rs+1 := `sw Rd, Adr` `sw Rd+1, Adr+4`	store double-word

Wie bei der ersten Gruppe der Ladebefehle müssen auch bei diesen Speicherbefehlen die Adressen ausgerichtet sein. Aber auch unter den Speicherbefehlen gibt es Befehle, die nicht ausgerichtete Adressen verarbeiten können:

Befehl	Argumente	Wirkung	Erläuterung
`swl`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store word left
`swr`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store word right
`ush`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	unaligned store halfword
`usw`	`Rs, Adr`	MEM[Adr]:=Rs	unaligned store word

5.3 Register-Transfer-Befehle

Außer dem Laden von Daten aus dem Hauptspeicher in Register (Abschnitt 5.1) und dem Speichern von Registerinhalten in den Hauptspeicher (Abschnitt 5.2) wollen wir häufig Daten in einem Register manipulieren, ohne auf den Hauptspeicher zuzugreifen. Dies ermöglichen uns die folgenden Befehle:

Befehl	Argumente	Wirkung	Erläuterung
`move`Ⓟ	`Rd, Rs`	Rd:=Rs	move
		:= `addu Rd $zero Rs`	
`li`Ⓟ	`Rd, Imm`	Rd:=Imm	load immediate
		:= `ori Rd, $zero, Imm` oder: `lui $at, Imm DIV` 2^{16} `ori Rd, $at, Imm MOD` 2^{16}	
`lui`	`Rs, Imm`	Rd:=Imm 2^{16}	load upper immediate

Die in den Befehlen `li` und `lui` bereits auftretende *unmittelbare* (*immediate*) Adressierung lernen wir in Abschnitt 6.1 auf der folgenden Seite kennen.

Der Vollständigkeit halber seien an dieser Stelle noch vier Befehle für die beiden bisher noch nicht erwähnten Register `hi` und `lo` genannt. Die genannten Register werden für Multiplikation und Division benutzt.

`hi` und `lo`

Befehl	Argumente	Wirkung	Erläuterung
`mfhi`	`Rd`	Rd:=hi	move from `hi`
`mflo`[5]	`Rd`	Rd:=lo	move from `lo`
`mthi`	`Rs`	hi:=Rs	move to `hi`
`mtlo`	`Rs`	lo:=Rs	move to `lo`

Übungen

Übung 6
Lege im Speicher je ein Wort, ein Halbwort und ein Byte an. Speichere mit Hilfe des Befehls `li` *und den Speicherbefehlen den Wert 20 nacheinander mit den passenden Speicherbefehlen.*

Übung 7
Was passiert wenn du einen zu großen Wert, z.B. 256 mit `sb` *speicherst? Und was passiert, wenn du 256 mit* `sw` *auf ein mit* `.byte` *angelegte Speicherstelle schreibst? Was ändert sich daran, wenn du* `.byte` *als erste Direktive ins Datensegment schreibst?*

[5] [Pat, A-65] nennt den falschen Befehl `mfloi` und eine falsche Erläuterung

6 Arithmetische Befehle (ganze Zahlen)

In den beiden vorangegangenen Abschnitten haben wir gelernt, wie man Daten im Hauptspeicher anlegt und sie zwischen dort und den Registern der CPU hin und her bewegt. Wir wenden uns nun der arithmetischen Verarbeitung der Daten zu.

6.1 Adressierungsmodi II: Register direkte und unmittelbare Adressierung

Wie bereits verschiedentlich erwähnt, kann der **MIPS** Operationen ausschließlich auf Registern durchführen. Kein arithmetischer Befehl hat als Parameter Hauptspeicheradressen! Die aus Abschnitt 5.1.1 auf Seite 18 bekannten Adressierungsmodi reichen uns nun also nicht mehr aus.

Register-direkte Adressierung

Zunächst benötigen wir einen Modus, der es uns erlaubt auf den Inhalt eines Registers direkt zuzugreifen. Wir nennen ihn *Register-direkte Adressierung* (*register direct*). Register-direkte Operanden werden durch Angabe des Registernamens oder der Registernummer notiert. Derartige Operanden sind in den Befehlstabellen mit `Rd`, `Rs`, `Rs1` und `Rs2` bezeichnet.

unmittelbare Adressierung

Außerdem erscheint es zweckmäßig Konstanten direkt verwenden zu können ohne sie erst aufwendig in ein Register schreiben zu müssen. Diese Adressierungsart nennen wir *unmittelbare Adressierung* oder *immediate*. Diese Operanden sind in den Befehlstabellen mit `Imm` bezeichnet. Unmittelbare Adressierung ist nur im letzten Operanden möglich.

6.2 Addition und Subtraktion

Die folgenden drei Befehle bedürfen keiner weiteren Erklärung:

Befehl	Argumente	Wirkung	Erläuterung
`add`	`Rd, Rs1, Rs2`	Rd := Rs1 + Rs2	addition (with overflow)
`addi`	`Rd, Rs, Imm`	Rd := Rs + Imm	addition immediate (with overflow)
`sub`	`Rd, Rs1, Rs2`	Rd := Rs1 - Rs2	subtract (with overflow)

Obwohl es einen eigenen Befehl `addi` gibt, können wir auch beim Befehl `add` unmittelbare Operanden verwenden. Der Assembler verwendet dann automatisch den korrekten Befehl `addi`. Auch den Befehl `sub` können wir mit unmittelbaren Operanden verwenden. Der Assembler lädt in diesem Fall den Wert zunächst in sein `$at`-Register und führt dann die Subtraktion aus. Diese praktische Umwandlung geschieht auch bei den anderen arithmetischen Befehlen automatisch.

Ausnahmebehandlung

Wenn bei den zuvor genannten Befehlen ein Überlauf auftritt (Summe zweier positiver Zahlen wird negativ o.ä.), wird eine *Ausnahmebehandlung* (exception handler) aufgerufen, die das Programm abbricht. Unter Ausnahmen versteht man Fehler, die zur Laufzeit des Programmes auftreten und die vom Programm selbst verursacht werden. Das klassische Beispiel für eine Ausnahme ist die Division durch Null, eine Aufgabe, die der Prozessor nicht lösen kann. Von der Ausnahme abzugrenzen ist die *Unterbrechung*, die extern erfolgt, beispielsweise der Warmstart des Rechners durch eine bestimmte Tastenkombination. Mehr über Unterbrechungen und Ausnahmen und deren Behandlung lernen wir in Abschnitt 15 auf Seite 66.

Unterbrechung

Zurück zu unseren Additionsbefehlen und den dabei eventuell auftretenden Überläufen. Handelt es sich bei den Zahlen um vorzeichenlose Zahlen oder soll der Überlauf anders bearbeitet werden, so verwenden wir einen der folgenden Befehle:

Befehl	Argumente	Wirkung	Erläuterung
`addu`	`Rd, Rs1, Rs2`	Rd := Rs1 + Rs2	addition (without overflow)
`addiu`	`Rd, Rs, Imm`	Rd := Rs + Imm	addition immediate (without overflow)
`subu`	`Rd, Rs1, Rs2`	Rd := Rs1 - Rs2	subtract (without overflow)

Der Buchstabe `u` scheint zwar auf 'unsigned' hinzuweisen, aber die Beschreibung 'overflow' ist treffender, da auch bei vorzeichenlosen Zahlen Überläufe auftreten können. Durch die Vernachlässigung der Überläufe werden diese Befehle aber eben auch für vorzeichenlose Arithmetik brauchbar. Deutlich wird dies, wenn man versucht, zur größten vorzeichenbehafteten Zahl 1 hinzuzuaddieren. Als Beispiel betrachten wir den übersichtlicheren Fall von Halbbytes:

$$\begin{array}{rll} a = & 0111 & \\ b = & 0001 & + \\ \hline carry = & 0\,1110 & \\ s = & 1000 & \\ \hline\hline \end{array}$$

Das Ergebnis +8 kann als vorzeichenbehaftete Zahl nicht dargestellt werden, es wird als -8 interpretiert und ist somit falsch. Haben wir jedoch mit vorzeichenlosen Zahlen gerechnet, so ist das Ergebnis richtig.

Übungen

Übung 8
Als erstes komplettes Programm sollst du ein Programm schreiben, das den Term $(w + x + 3) - (y + z - 5)$ ohne diesen zuvor zu optimieren berechnet. Halte die Operanden w, x, y, z als vorzeichenbehaftete von dir zur Assemblierzeit initialisierte ganze Zahlen im Hauptspeicher und lege dein Ergebnis in einem fünften Speicherplatz ab.

Übung 9
Ändere das Programm aus Übung 8 so, dass es mit 8-Bit-Zahlen arbeitet

Übung 10
Initialisiere dein Programm aus Übung 9 mit $w = y = 128$ und $x = z = 127$. Wie lautet das korrekte Ergebnis und was passiert bei der Programmausführung?

Übung 11
Schreibe dein Programm aus Übung 9 so um, dass es für den Fall $w \approx y, x \approx z, w > 100, z > 100$ etwas sicherer läuft.

6.3 Multiplikation und Division

Der **MIPS** verfügt lediglich über die folgenden vier Maschinenbefehle, mit denen alle anderen Divisions- und Multiplikationsbefehle realisiert werden:

Befehl	Argumente	Wirkung	Erläuterung
`div`	`Rd, Rs`	hi:=Rd MOD Rs; lo:=Rd DIV Rs	Divide (with overflow)
`divu`	`Rd, Rs`	hi:=Rd MOD Rs; lo:=Rd DIV Rs	Divide (without overflow)

Befehl	Argumente	Wirkung	Erläuterung
`mult`	`Rd, Rs`	hi:=Rd $\times$ Rs DIV 2^{16}; lo:=Rd $\times$ Rs MOD 2^{16}	multiply
`multu`	`Rd, Rs`	hi:=Rd $\times$ Rs DIV 2^{16}; lo:=Rd $\times$ Rs MOD 2^{16}	Unsigned multiply

Alle Befehle verwenden zwei Register, die wir bisher noch nicht benutzt haben: `hi`(gh) und `lo`(w). Sie gehören nicht zu den 32 Registern, die wir mit dem `$`-Zeichen adressieren können. Es gibt aber zwei Befehle (`mfhi` und `mflo`), mit denen wir ihren Inhalt in ein normales Register kopieren können. Wir haben sie bereits in Abschnitt 5.3 auf Seite 21 kennen gelernt.

Der **MIPS**-Assembler stellt einige Pseudoinstruktionen zur Verfügung, die das Kopieren der Ergebnisse aus den Registern `lo` und `hi` übernehmen:

Befehl	Argumente		Wirkung	Erläuterung
`mul`Ⓟ	`Rd, Rs1, Rs2`		Rd:=Rs1 $\times$ Rs2	Multiply (without overflow)
		:=	`mult Rs1, Rs2` `mflo Rd`	
`mulo`Ⓟ	`Rd, Rs1, Rs2`		Rd:=Rs1 $\times$ Rs2	Multiply (with overflow)
		:=	`mult Rs1, Rs2` `mfhi $at` `beq $at, $zero 4` `break $zero` `mflo Rd`	
`mulou`Ⓟ	`Rd, Rs1, Rs2`		Rd:=Rs1 $\times$ Rs2	Unsigned multiply (with overflow)
		:=	`multu Rs1, Rs2` `mfhi $at` `beq $at, $zero 4` `break $zero` `mflo Rd`	

Und für die Division:

Befehl	Argumente		Wirkung	Erläuterung
`div`Ⓟ	`Rd, Rs1, Rs2`		Rd:=Rs1 DIV Rs2	Divide (with overflow)
		:=	`bne Rs2, $zero, 4` `break $zero` `div Rs1, Rs2` `mfl Rd`	
`divu`Ⓟ	`Rd, Rs1, Rs2`		Rd:=Rs1 DIV Rs2	Divide (without overflow)
		:=	`bne Rs2, $zero, 4` `break $zero` `divu Rs1, Rs2` `mfl Rd`	

Der Befehl `div` kann also zwei oder drei Parameter haben, je nachdem ob der Pseudobefehl oder der Maschinenbefehl vom Anfang dieses Abschnitts gemeint ist.

Werfen wir einen Blick auf die Übersetzung dieser Befehle. Das Grundprinzip ist bei allen diesen Pseudobefehlen dasselbe: Das Ergebnis wird zunächst mit einem der Maschinenbefehle berechnet. Danach befinden sich in den Registern `lo` und `hi` die Teilergebnisse. In Abhängigkeit von diesen Werten wird dann eine Ausnahmebehandlung ausgelöst oder nicht.

Hierfür werden die Befehle `bne` (branch if not equal) und `beq` (branch if equal) verwendet, die wir in Abschnitt Abschnitt 9.1 auf Seite 32 kennen lernen werden. Hier genügt erst einmal deren Effekt: Wenn der Inhalt der beiden Register gleich bzw. ungleich ist, werden die nächsten n Befehle des Programms übersprungen, wobei n der dritte Parameter der Befehle ist und durch vier teilbar sein muss.[6] Die erste Zeile der Übersetzung des Befehls `div` überspringt also den folgenden Befehl, falls der Teiler Null ist. Womit wir einen weiteren neuen Befehl kennen lernen: `break`, er wird in Abschnitt Abschnitt 15 auf Seite 66 näher erläutert. Er verursacht den Aufruf einer Ausnahmebehandlungsroutine, die die Programmausführung nach der Ausgabe einer entsprechenden Meldung stoppt.

Übungen

Übung 12
Schreibe ein Programm, das den Umfang und den Flächeninhalt eines Rechteckes berechnet. Die Seiten des Rechteckes sollen im Hauptspeicher gehalten werden, die Ergebnisse dort abgelegt werden.

6.4 Sonstige arithmetische Befehle

Befehl	Argumente		Wirkung	Erläuterung
`abs`[P]	`Rd, Rs`		Rd:=ABS (Rs)	Absolute value
		:=	`add Rd, $zero, Rs` `bgez Rd 4` `sub Rd, $zero, Rs`	
`neg`[P]	`Rd, Rs`		Rd:=–Rs	Negate value (with overflow)
		:=	`sub Rd, $zero, Rs`	
`negu`[P]	`Rd, Rs`		Rd:=–Rs	Negate value (without overflow)
		:=	`subu Rd, $zero, Rs`	
`rem`[P]	`Rd, Rs1, Rs2`		Rd:=Rs1 MOD Rs2	Remainder
		:=	`bne Rs2, $zero, 4` `break $zero` `div Rs1, Rs2` `mfhi Rd`	
`remu`[P]	`Rd, Rs1, Rs2`		Rd:=Rs1 MOD Rs2	Unsigned remainder
		:=	`bne Rs2, $zero, 4` `break $zero` `divu Rs1, Rs2` `mfhi Rd`	

Warum gibt es zwei Negationsbefehle? In der Tat gibt es eine Zahl, die man nicht negieren kann: die kleinste darstellbare Zahl, beim SPIM also `0x8000 0000`. Dies liegt daran, dass jede Kombination von Bits eine gerade Anzahl von Kombinationsmöglichkeiten hat. Eine dieser Möglichkeiten wird aber für den Wert 0 benötigt, so dass für den negativen und den positiven Zahlraum unterschiedlich viele Zahlen übrig bleiben. Im 2er-Komplement führt dies dazu, dass der Betrag der kleinsten darstellbaren Zahl um eins größer als die größte darstellbare Zahl ist. Aus diesem Grunde kann auch bei der Verwendung des Befehls `abs` ein Überlauf auftreten.

Die Befehle `rem` und `remu` berechnen den Rest bei der ganzzahligen Division.

[6] Diese Erklärung ist nicht ganz korrekt, auf Seite 34 werden wir mehr darüber erfahren!

7 Das SPIM-Betriebssystem (Ein- und Ausgabe)

SPIM stellt ein Betriebssystemfragment zur Verfügung, mit dem u.a. einfachste Ein- und Ausgaben realisiert werden können. Diese Funktionen stehen uns aber nur dann zur Verfügung, wenn wir auf die Möglichkeit der Simulation von memory-mapped-io (siehe Abschnitt 15 auf Seite 66, Option `-mapped-io`) verzichten.

`syscall`

Die Betriebssystemfunktionen (system calls) werden mit dem Befehl syscall aufgerufen. In das Register `$v0` müssen wir zuvor einen Code für die gewünschte Funktion schreiben. Einen Überblick über die möglichen Funktionen gibt Abbildung 6 auf der gegenüberliegenden Seite.

Befehl	Argumente	Wirkung	Erläuterung
`syscall`		führt die Systemfunktion in `$v0` aus	vgl. Abbildung 6 auf der folgenden Seite

Parameter von Betriebssystemfunktionen werden in den Registern `$a0`, `$a1` und `$f12`, einem Register eines der Koprozessoren, die wir in 14 näher kennen lernen werden, übergeben.

Die Ergebnisse von Betriebssystemfunktionen werden im Register `$v0` bzw. dem Koprozessorregister `$f0` zurückgegeben.

Für das erste reichen uns aber die Funktionen `print_int` (1), `print_string` (4), `read_int` (5) und `read_string` (6), die im nächsten Beispiel verwendet werden:

Beispiel 3: Ein- und Ausgabe mit Betriebssystemfunktionen

```
# IO.S
# Beispielprogramm zur Ein- und Ausgabe
.data
        txt1:   .asciiz "Zahl= "
        txt2:   .asciiz "Text= "
        input:  .ascii  "Dieser Text wird nachher ueber"
                .asciiz "schrieben!                    "
                                #
.text
                                # Eingabe...
main:   li      $v0, 4          # 4: print_str
        la      $a0, txt1       # Adresse des ersten Textes in $a0
        syscall
        li      $v0, 5          # 5: read_int
        syscall
        move    $s0,$v0
                                #
        li      $v0, 4          # 4: print_str
        la      $a0, txt2       # Adresse des zweiten Textes in $a0
        syscall
        li      $v0, 8          # 8: read_str
        la      $a0, input      # Adresse des einzugebenden Textes
        li      $a1, 256        # maximale Laenge
        syscall
                                # Ausgabe...
        li      $v0, 1          # 1: print_int
        move    $a0, $s0
        syscall
        li      $v0, 4          # 4: print_str
        la      $a0, input
        syscall
        li      $v0, 10         # Exit
        syscall
```

Abbildung 6: Betriebssystemfunktionen des SPIM (system calls)

Funktion	Code in `$v0`	Argument(e)	Ergebnis
`print_int`	1	Wert in `$a0` wird dezimal ausgegeben	
`print_float`	2	Wert in `$f12/13`[1] wird als 32-Bit-Gleitkommazahl ausgegeben	
`print_double`	3	Wert in `$f12`[1] wird als 64-Bit-Gleitkommazahl ausgegeben	
`print_string`	4	Die mit Chr 0 terminierte Zeichenkette, die an der Stelle (`$a0`) beginnt, wird ausgegeben	
`read_int`	5		Die auf der Konsole dezimal eingegebene ganze Zahl in `$v0`
`read_float`	6		Die auf der Konsole dezimal eingegebene 32-Bit-Gleitkommazahl in `$f0`[2]
`read_double`	7		Die auf der Konsole dezimal eingegebene 64-Bit-Gleitkommazahl in `$f0/1`[2]
`read_string`	8	Adresse, ab der die Zeichenkette abgelegt werden soll in `$a0`, maximale Länge der Zeichenkette in `$a1`	Speicher von (`$a0`) bis (`$a0`)+`$a1` wird mit der eingelesenen Zeichenkette belegt.
`sbrk`	9	Größe des Speicherbereichs in Bytes in `$a0`	Anfangsadresse eines freien Blocks der geforderten Größe in `$v0`
`exit`	10		

[1]: `$f12` und `$f13` sind Register des Koprozessors 1, siehe 14)

[2]: `$f0` und `$f1` sind Register des Koprozessors 1, siehe 14)

Übungen

Übung 13

Ergänze deine Lösung aus Übung 11 so, dass die Werte vom Benutzer eingegeben werden können. Die Haltung der Werte w bis z im Hauptspeicher entfällt also. Gebe das Ergebnis mit einem Hinweistext aus.

Übung 14

Ergänze deine Lösung aus Übung 12 so, dass die Werte vom Benutzer eingegeben werden können. Gebe die Ergebnisse mit einem Hinweistext aus.

8 Logische Befehle

In den vorangegangenen Abschnitten haben wir gelernt, Daten zwischen Hauptspeicher und Prozessor hin und her zu bewegen und mit ihnen Arithmetik zu betreiben. Wir werden uns nun den logischen Operationen zuwenden.

bitweise Bearbeitung

Der aus Hochsprachen bekannte Datentyp `boolean` ist auf Assembler-Ebene nicht vorhanden. Die kleinste Zuordnungseinheit im Hauptspeicher ist 1 Byte, auf Prozessorebene gar ein ganzes Wort mit 4 Bytes. Die logischen Befehle arbeiten wie die arithmetischen alle auf Registern. Die Register werden *bitweise* bearbeitet:

$$\texttt{OP R1, R2} := (\texttt{R1}_0\,\texttt{OP}\,\texttt{R2}_0)(\texttt{R1}_1\,\texttt{OP}\,\texttt{R2}_1)\ldots(\texttt{R1}_{31}\,\texttt{OP}\,\texttt{R2}_{31})$$

z.B.:

$$0101_2 \wedge 1100_2 = 0100_2$$

8.1 Elementare logische Befehle

Die folgenden Operationen implementieren die Basisoperationen, deren Funktionalität bekannt sein sollte:

Befehl	Argumente	Wirkung	Erläuterung
`and`	`Rd, Rs1, Rs2`	Rd := Rs1 ∧ Rs2	and
`andi`	`Rd, Rs, Imm`	Rd := Rs ∧ Imm	and immediate
`nor`	`Rd, Rs1, Rs2`	Rd := $\overline{\text{Rs1} \vee \text{Rs2}}$	nor
`or`	`Rd, Rs1, Rs2`	Rd := Rs1 ∨ Rs2	or
`ori`	`Rd, Rs, Imm`	Rd := Rs ∨ Imm	or immediate
`xor`	`Rd, Rs1, Rs2`	Rd := Rs1 ⊕ Rs2	exclusive or
`xori`	`Rd, Rs, Imm`	Rd := Rs ⊕ Imm	exclusive or immediate
`not`Ⓟ	`Rd, Rs`	Rd := $\overline{\text{Rs}}$	not
	:=	`xori Rd, Rs, -1`	

Übungen

Übung 15
Der Pseudobefehl `move` *wird mit Hilfe des Befehls* `addu` *übersetzt. Welche Übersetzungen wären noch denkbar?*

Übung 16
Wie könnte man einen Pseudobefehl `swap` *definieren, der den Inhalt zweier Register austauscht? Verzichte auf das* `$at`*-Register und beweise die Korrektheit deines Befehls! Tip: Verwende nur den Befehl* `xor`.

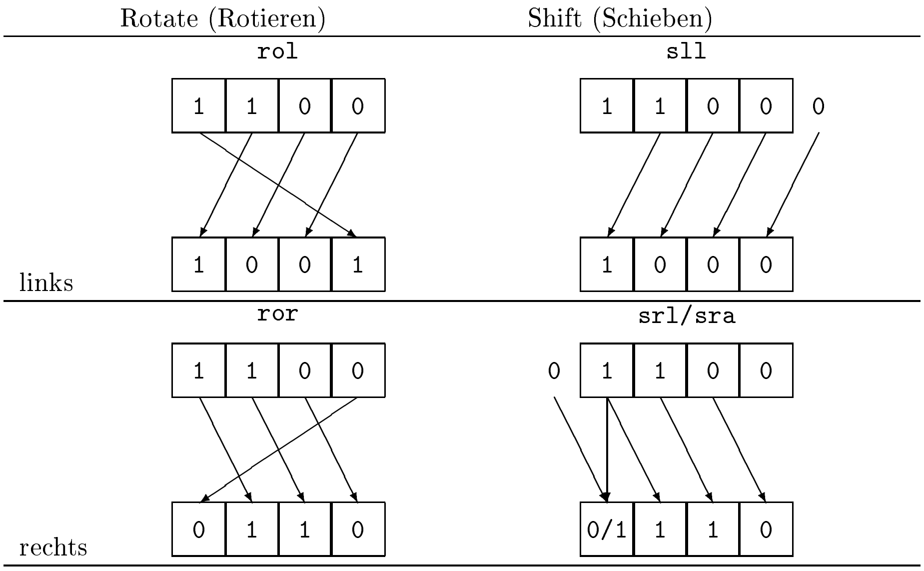
8.2 Rotations- und Schiebebefehle

Rotations- und Schiebebefehle

Vielfach wird bei der Assemblerprogrammierung auf sogenannte *Rotations-* und *Schiebebefehle* zurückgegriffen. Sie manipulieren die Reihenfolge der Bits in einem Register. Beim Rotieren werden die Bits um einen gewählten Betrag nach links oder rechts verschoben, die überzähligen Bits am anderen Ende jedoch wieder angehängt. Bei Schiebebefehlen werden die überzähligen Bits ignoriert und die freien Bits mit Nullen oder Einsen aufgefüllt. Das Prinzip wird in der ersten Spalte der Abbildung 7 auf der gegenüberliegenden Seite dargestellt.

Genutzt werden können die Rotations- und Schiebebefehle beispielsweise bei der Auswertung von Schnittstellen.

Abbildung 7: Rotations- und Schiebebefehle



Werfen wir zunächst einen Blick auf die uns zur Verfügung stehenden Rotationsbfehle:

Befehl	Argumente	Wirkung	Erläuterung
`rol`Ⓟ	`Rd, Rs1, Rs2`	Rd := Rs1 um Rs2 Stellen nach links rotiert	rotate left
	:=	`subu $at, $zero, Rs2` `srlv $at, Rs1, $at` `sllv Rd, Rs2, Rs1` `or Rd, Rd, $at`	
`ror`Ⓟ	`Rd, Rs1, Rs2`	Rd := Rs1 um Rs2 Stellen nach rechts rotiert	rotate right
	:=	`subu $at, $zero, Rs2` `srlv $at, $at, Rs1` `srlv Rd, Rs1, Rs2` `or Rd, Rd, $at`	

logische und arithmetische Schiebebefehle

Bei den Schiebebefehlen müssen wir zwischen *logischen* und *arithmetischen* Befehlen differenzieren. Der Unterschied liegt in der Behandlung der hochwertigen Bits. Bei vorzeichenbehafteten Zahlen repräsentiert das höchstwertige Bit das Vorzeichen. Die logischen Schiebebefehle füllen die freigewordenen Bits mit Nullen auf, während der arithmetische Schiebebefehl das höchstwertige Bit der Quelle übernimmt. In vielen Fällen soll das Vorzeichen jedoch erhalten bleiben, wir müssen dann einen der arithmetischen Schiebebefehle wählen. Es gibt nur einen arithmetischen Schiebebefehl (`sra`), da beim Schieben nach links das Vorzeichen nicht beachtet werden muss. (Selber ausprobieren mit 0100, 1000 und 1101!)

Alle Schiebebefehle liegen zudem in einer Form für fixe Verschiebungen, und in einer für variable durch ein Register angegebene Verschiebungen vor. Die variablen Befehle sollten nur verwendet werden, wenn der Betrag der Verschiebung tatsächlich erst zur Laufzeit bestimmt wird. Bei den fixen Befehlen sind für den Verschiebungsbetrag 5 Bit, also die Werte 0 bis 32, vorgesehen. Andere Werte werden beim Assemblieren umgerechnet.

Befehl	Argumente	Wirkung	Erläuterung
`sll`	`Rd, Rs, Imm`	Rd:=Rs $\times$ 2^{Imm}	Shift left logical
`sllv`	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ 2^{Rs2}	Shift left logical variable
`srl`	`Rd, Rs, Imm`	Rd:=Rs DIV 2^{Imm}	Shift right logical
`srlv`	`Rd, Rs1, Rs2`	Rd:=Rs1 DIV 2^{Rs2}	Shift right logical variable
`sra`	`Rd, Rs, Imm`	Rd:=Rs $\times$ 2^{Imm}	Shift right arithmetic
`srav`	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ 2^{Rs2}	Shift right arithmetic variable

8.3 Vergleichsbefehle

Dieser Abschnitt soll kein Vorgriff auf Abschnitt 9 auf Seite 32 sein, obwohl die Vergleichsbefehle auch in Kontrollstrukturen Anwendung finden. Für kleinste Entscheidungen sind die folgenden Befehle jedoch oft auch ohne größere Kontrollstrukturen nützlich.

Der **MIPS** verfügt nur über Vergleichsbefehle, die die Relation „<" realisieren. Im nächsten Abschnitt werden wir noch Maschinenbefehle kennen lernen, die auch die Relation „=" sowie Vergleiche mit 0 ermöglichen. Diese spartanische Ausstattung wird durch einige unterschiedlich aufwendige Pseudobefehle so erweitert, dass uns alle Relationen zur Verfügung stehen.

Die Vergleichsbefefehle setzen alle das Zielregister auf 1, falls ihre Bedingung erfüllt ist, sonst auf 0. Sie sind zwar alle nur mit register-direkter Adressierung definiert, aber der SPIM-Assembler übersetzt unmittelbare Operanden, indem er sie zuvor in das `$at`-Register lädt. Die in der folgenden Tabelle genannten Übersetzungen der Pseudobefehle beziehen sich jedoch nur auf die rein register-direkten Adressierungen.

Tests auf Gleichheit:

Befehl	Argumente	Wirkung	Erläuterung
`seq`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1=Rs2, 0 sonst	(=) set equal
	:=	`beq Rs2, Rs1, 8` `ori Rd, $zero, 0` `beq $zero, $zero, 4` `ori Rd, $zero, 1`	
`sne`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≠Rs2, 0 sonst	(≠) set not equal
	:=	`beq Rs2, Rs1, 8` `ori Rd, $zero, 1` `beq $zero, $zero, 4` `ori Rd, $zero, 0`	

Tests auf größer:

Befehl	Argumente	Wirkung	Erläuterung
`sge`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≥Rs2, 0 sonst	(≥) set greater than equal
	:=	`bne Rs2, Rs1, 8` `ori Rd, $zero, 1` `beq $zero, $zero, 4` `slt Rd, Rs2, Rs1`	

Befehl	Argumente	Wirkung	Erläuterung
`sgeu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≥Rs2, 0 sonst	(≥) set greater than equal unsigned
	:=	`bne Rs2, Rs1, 8` `ori Rd, $zero, 1` `beq $zero, $zero, 4` `sltu Rd, Rs2, Rs1`	
`sgt`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1>Rs2, 0 sonst	(>) set greater than
	:=	`slt Rd, Rs2, Rs1`	
`sgtu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≥Rs2, 0 sonst	(>) set greater than unsigned
	:=	`sltu Rd, Rs2, Rs1`	

Tests auf kleiner:

Befehl	Argumente	Wirkung	Erläuterung
`sle`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≤Rs2, 0 sonst	(≤) set less than equal
	:=	`bne Rs2, Rs1, 8` `ori Rd, $zero, 1` `beq $zero, $zero, 4` `slt Rd, Rs1, Rs2`	
`sleu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1≤Rs2, 0 sonst	(≤) set less than equal unsigned
	:=	`bne Rs2, Rs1, 8` `ori Rd, $zero, 1` `beq $zero, $zero, 4` `sltu Rd, Rs1, Rs2`	
`slt`	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1 < Rs2, 0 sonst	(<) set less than
`sltu`	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1 < Rs2, 0 sonst	(<) set less than unsigned
`slti`	`Rd, Rs, Imm`	Rd := 1, falls Rs < Imm, 0 sonst	(<) set less than immediate
`sltui`	`Rd, Rs, Imm`	Rd := 1, falls Rs < Imm, 0 sonst	(<) set less than unsigned immediate

Bei der Umsetzung der Pseudo-Befehle in Maschinenbefehle finden wieder die Befehle `beq` und `bne` Anwendung. Wir lernen sie erst im nächsten Abschnitt kennen, haben aber schon im Abschnitt 6.3 auf Seite 24 von ihnen gehört.

Zur Erinnerung: der Maschinenbefehl `ori Rd, $zero, 0` ist die Übersetzung des Befehls `li` (load immediate), vgl. Abschnitt 5.3 auf Seite 21.

Übungen

Übung 17
Berechne mit Hilfe der dir jetzt bekannten Entscheidungsbefehle das Entgelt für einen Brief. Gehe vereinfachend davon aus, dass das Porto wie folgt berechnet wird:

Gewicht bis	*20g*	*50g*	*250g*	*500g*
Entgelt	*110*	*220*	*300*	*440*

9 Kontrollstrukturen

Entscheidungen

Grundlage vieler Algorithmen sind *Entscheidungen*:

> Wenn der Betrag größer als 1.000,– DM ist, dann gewähre 3% Rabatt. Sonst beträgt der Rabatt 2%.

In MODULA würden wir dafür schreiben:

```
IF Betrag > 1000
   THEN Rabatt := 3
   ELSE Rabatt := 2
END;(*IF*)
```

Schleifen

Manche Anweisungen sollen mehrmals hintereinander ausgeführt werden. Wir sprechen dann von *Schleifen*. Z.B.:

> Addiere solange die Folge der natürlichen Zahlen bis die Summe 100 überschreitet.

In MODULA:

```
summe := 0;
i := 0;
WHILE summe <= 100
    i := i + 1;
    summe := summe + i
END;(*WHILE*)
```

Kontrollstrukturen

Hochsprachen stellen für solche Entscheidungen und Schleifen eine Fülle verschiedener ausgefeilter *Kontrollstrukturen* zur Verfügung. Der Assemblerprogrammierer muss sich diese Strukturen jedoch aus einer ganzen Reihe von Befehlen zusammensetzen. Die häufigsten Kontrollstrukturen lernen wir in diesem Abschnitt in die Befehle des **MIPS** zu übersetzen.

9.1 Programmverzweigungen (einfache Fallunterscheidungen) oder: Sprungbefehle

In dem Rabatt-Beispiel zu Beginn dieses Abschnitts wurde ein Teil des Programms nur bedingt ausgeführt, nämlich dann, wenn die Variable `Betrag` größer als 1 000 ist. Diese Bedingung ist eine sehr einfache. Hochsprachen erlauben die Verkettung mehrerer Bedingungen, sogar komplexe Funktionsaufrufe sind erlaubt. Der **MIPS** erlaubt nur eine ganz geringe Zahl von Bedingungen: Lediglich Größenvergleiche zwischen den Werten zweier Register oder einem Register und einem unmittelbaren Wert sind erlaubt.

Es gibt zwei unterschiedliche Arten in einem Programm Sprünge auszuführen:

branching

jumping

- Springe um n Befehle nach vorne oder nach hinten (*branching*)
- Springe an die Stelle x (*jumping*).

Der SPIM verfügt über vier unbedingte Jump-Befehle, 21 bedingte und einen unbedingten Branch-Befehl, von denen wir die meisten jetzt kennen lernen werden. Beim Programmieren muss man sich um den Unterschied der beiden Sprungarten nicht kümmern, da es der Assembler bei allen Sprungbefehlen erlaubt, das Sprungziel durch eine Marke (label) anzugeben.

Die folgenden Tabellen liefern eine Übersicht über die möglichen Sprunganweisungen.

Unbedingte Sprunganweisungen:

Befehl	Argumente		Wirkung	Erläuterung
bⓅ	`label`		unbedingter Sprung nach `label`	branch
		:=	`bgez $zero label-PC`	
j	`label`		unbedingter Sprung nach `label`	jump

Sprung bei Gleichheit (oder Ungleichheit):

Befehl	Argumente		Wirkung	Erläuterung
`beq`	`Rs1, Rs2, label`		Sprung nach `label`, falls `Rs1`=`Rs2`	$(=)$ branch on equal
`beqz`Ⓟ	`Rs, label`		Sprung nach `label`, falls `Rs`=0	$(=0)$ branch on equal zero
		:=	`beq $zero, Rs, label-PC`	
`bne`	`Rs1, Rs2, label`		Sprung nach `label`, falls `Rs`$\neq$0	$(\neq)$ branch on not equal zero
`bnez`Ⓟ	`Rs, label`		Sprung nach `label`, falls `Rs`$\neq$0	$(\neq 0)$ branch on not equal zero
		:=	`bne $zero, Rs, label-PC`	

Sprung bei größer oder gleich:

Befehl	Argumente		Wirkung	Erläuterung
`bge`Ⓟ	`Rs1, Rs2, label`		Sprung nach `label`, falls `Rs1`$\geq$`Rs2`	$(\geq)$ branch on greater than equal
		:=	`slt $at, Rs1, Rs2` `beq $at, $zero, label-PC`	
`bgeu`Ⓟ	`Rs1, Rs2, label`		Sprung nach `label`, falls `Rs1`$\geq$`Rs2`	$(\geq)$ branch on greater than equal unsigned
		:=	`sltu $at, Rs1, Rs2` `beq $at, $zero, label-PC`	
`bgez`	`Rs, label`		Sprung nach `label`, falls `Rs`$\geq$0	(≥ 0) branch on greater than equal zero

Sprung bei größer:

Befehl	Argumente		Wirkung	Erläuterung
`bgt`Ⓟ	`Rs1 Rs2 label`		Sprung nach `label`, falls `Rs1`$>$`Rs2`	$(>)$ branch on greater than
		:=	`slt $at, Rs2, Rs1` `bne $at, $zero, label-PC`	
`bgtu`Ⓟ	`Rs1 Rs2 label`		Sprung nach `label`, falls `Rs1`$>$`Rs2`	$(>)$ branch on greater than unsigned
		:=	`sltu $at, Rs2, Rs1` `bne $at, $zero, label-PC`	
`bgtz`	`Rs, label`		Sprung nach `label`, falls `Rs`$>$0	(>0) branch on greater than zero

Spung bei kleiner oder gleich:

Befehl	Argumente		Wirkung	Erläuterung
`ble`Ⓟ	`Rs1 Rs2 label`		Sprung nach `label`, falls `Rs1`$\leq$`Rs2`	$(\leq)$ branch on less than equal
		:=	`slt $at, Rs2, Rs1` `beq $at, $zero, label-PC`	

Befehl	Argumente	Wirkung	Erläuterung
`bleu`Ⓟ	`Rs1 Rs2 label`	Sprung nach `label`, falls `Rs1`≤`Rs2`	(≤) branch on less than equal unsigned
	:=	`sltu $at, Rs2, Rs1` `beq $at, $zero, label-PC`	
`blez`	`Rs, label`	Sprung nach `label`, falls `Rs`≤0	(≤ 0) branch on less than equal zero

Spung bei kleiner:

Befehl	Argumente	Wirkung	Erläuterung
`blt`Ⓟ	`Rs1, Rs2, label`	Sprung nach `label`, falls `Rs1`<`Rs2`	(<) branch on less than
	:=	`slt $at, Rs1, Rs2` `bne $at, $zero, label-PC`	
`bltu`Ⓟ	`Rs1 Rs2 label`	Sprung nach `label`, falls `Rs1`<`Rs2`	(<) branch on less than unsigned
	:=	`sltu $at, Rs1, Rs2` `bne $at, $zero, label-PC`	
`bltz`	`Rs, label`	Sprung nach `label`, falls `Rs`<0	(< 0) branch on less than zero

PC-relative Adressierung

Die Branchbefehle erwarten eigentlich eine Distanz (Offset) zum aktuellen Programmzähler (PC). Man spricht auch von *PC-relativer Adressierung*. Da diese Adressierung für den Assemblerprogrammierer sehr aufwendig und extrem fehleranfällig ist, erlaubt der SPIM-Assembler Marken als Parameter. Leider entfällt damit die Möglichkeit zur PC-relativen Adressierung völlig, da der Assembler nicht mehr zwischen der Marke „4“ und dem Wert 4 unterscheiden kann.

Nicht ganz konsistent ist die Angabe der Distanz:

- In der *Maschinensprache*, die für den Assemblerprogrammierer lediglich beim manuellen disasemblieren oder assemblieren interessant ist, wird in den letzten 16 Bit des Befehls (also in der niedrigwertigen Hälfte) die Zahl der zu überspringenden *Befehle* angegeben.
- Der SPIM-Disassemblierer gibt jedoch im Text-Fenster die Zahl der zu überspringenden *Bytes* an, also jeweils das vierfache der rechten Befehlshälfte. Aus Konsistenzgründen habe ich bei der Angabe der Übersetzung der Pseudobefehle in diesem Tutorial ebenfalls diese Notation gewählt.
- Auf der Assembler-Ebene ist lediglich die Verwendung von Marken zugelassen, die Berechnung der Distanz übernimmt der Assembler.

einfache Verzweigungen

Bei der Implementation *einfacher Verzweigungen* sollte man sich zunächst die Struktur der Fallunterscheidung deutlich machen. Eine gute Möglichkeit, die auch für die Dokumentierung dieser Assembler-Passage gut geeignet ist, ist eine Formulierung in MODULA oder einer anderen Hochsprache.

Nehmen wir beispielsweise das Problem vom Anfang dieses Kapitels:

```
IF Betrag > 1000
   THEN Rabatt := 3
   ELSE Rabatt := 2
END;(*IF*)
```

Gehen wir einmal davon aus, dass die Variable `Betrag` im Register `$t0` und die Variable `Rabatt` im Register `$t1` abgelegt sind. Es bietet sich an, zunächst einmal die Bedingung für den 3%-Rabatt in Assembler umzusetzen:

```
        bgt     $t0, 1000, then           # IF Betrag > 1000
```

Die Marke `then` haben wir noch nicht definiert, sie soll am Anfang des „Dann"-Zweiges stehen, den wir uns nun vornehmen:

```
then:   li      $t1, 3                    #   THEN Rabatt := 3
```

Als nächstes übersetzen wir den „Sonst"-Zweig:

```
        li      $t1, 2                    #   ELSE Rabatt := 2
```

Würden wir diese drei Zeilen einfach hintereinander schreiben, so würden sowohl der „Dann"- als auch der „Sonst"-Zweig *immer* ausgeführt werden, was nicht im Sinne unserer Aufgabe ist. Eine korrekte, aber ineffiziente Lösung wäre:

Beispiel 4: Einfache Verzweigung (Erster Versuch)

```
· # BSP001.S
· main:   bgtu    $t0, 1000, then # IF Betrag > 1000
·         b       else
· then:   li      $t1, 3          #    THEN Rabatt := 3
5         b       endif
· else:   li      $t1, 2          #    ELSE Rabatt := 2
· endif:                          # END;(*IF*)
```

Betrachten wir das Programm genauer, so fällt uns auf, dass wir auf einen der unbedingten Sprünge verzichten können, wenn wir statt der Bedingung für den „Dann"-Zweig die Bedingung für den „Sonst"-Zweig verwenden. Unser Programm sieht nun so aus:

Beispiel 5: Einfache Verzweigung (Zweiter Versuch)

```
· # BSP002.S
· main:   ble     $t0, 1000, else # IF Betrag > 1000
·         li      $t1, 3          #    THEN Rabatt := 3
·         b       endif
5 else:   li      $t1, 2          #    ELSE Rabatt := 2
· endif:                          # END;(*IF*)
```

Wir sparen eine Zeile im „Sonst"-Zweig. Eine andere Vereinfachung wäre:

Beispiel 6: Einfache Verzweigung (Zweiter Versuch, Variante)

```
· # BSP003.S
· main:   bgt     $t0, 1000, then # IF Betrag > 1000
· else:   li      $t1, 2          #    ELSE Rabatt := 2
·         b       endif
5 then:   li      $t1, 3          #    THEN Rabatt := 3
· endif:                          # END;(*IF*)
```

Welche der Varianten man in der Regel wählt, ist Geschmackssache. Sollte einmal einer der Zweige leer sein (in der Regel wohl der „Sonst"-Zweig), so wäre allerdings die erste Variante zu bevorzugen, da hier die Zeilen 3 und 4 völlig weggelassen werden könnten.

Häufig wird die Sprungbedingung nicht in einem einzigen Vergleich bestehen. In diesem Fall muss sie zunächst seperat ausgewertet werden. Am Ende des Auswertungsabschnittes steht dann einer der zur Verfügung stehenden einfachen Verzweigungsbefehle.

Übungen

Übung 18
Schreibe ein Programm, das die Länge der Seiten eines Dreieckes in beliebiger Reihenfolge einliest und dann ausgibt ob es sich um ein allgemeines (alle Seiten verschieden), ein gleichschenkliges (zwei gleich lange Seiten) oder ein gleichseitiges Dreieck (drei gleiche Seiten) handelt.

Übung 19
Ein Betrieb verkauft sein Produkt zu einem Preis von 1 000 DM pro Stück. Bei der Bestellung von 100 Stück gewährt er einen Rabatt von 5%, bei der Bestellung von 500 Stück einen Rabatt von 10%. Es handelt sich um einen Gesamtrabatt, d.h. 99 Stück kosten 99 000 DM, 100 Stück 95 000 DM, und 500 Stück 450 000 DM.

Schreibe ein Programm, das nach Eingabe der Bestellmenge den Gesamtpreis ausgibt.

Übung 20
Der Betrieb aus 19 führt alternativ zu dem bestehenden Rabattsystem ein weiters ein. Dabei handelt es sich um einen Stufenrabatt, d.h. der Rabatt bezieht sich jeweils nur auf die Menge, die über die Rabattgrenze hinaus gekauft wurde: Der Rabatt für das 20. bis 49. Stück beträgt 2%, der Rabatt vom 50. bis 100. Stück 5% und ab dem 101. Stück 10%.

19 Stück kosten also 19 000 DM, 20 Stück 19 980 DM.

Schreibe ein Programm, das die Preise nach beiden Rabattsystemen berechnet und ausgibt.

9.2 Schleifen

Neben Fallunterscheidungen stellen Schleifen eine wesentliche Grundlage von Algorithmen dar. Wir kennen drei Grundtypen von Schleifen:

Abweisende Schleife

- Die *Abweisende Schleife* (While-Schleife), die *vor* jedem Ausführen der Schleife prüft, ob die Schleifenbedingung noch wahr ist. Abweisende Schleifen werden gar nicht ausgeführt, wenn die Schleifenbedingung zu Beginn unwahr ist.

Nachprüfende Schleife

- Bei der *Nachprüfenden Schleife* (Repeat-Until-Schleife) wird erst nach dem ersten und jedem weiteren Ausführen der Schleife geprüft, ob die Schleifen(abbruch)bedingung noch wahr ist. Sie wird also mindestens einmal ausgeführt.

Zählschleife

- Bei der *Zählschleife* (For-Schleife) wird vor Eintritt in die Zählschleife festgelegt, wie oft sie ausgeführt werden soll. Die Zahl der absolvierten Schleifendurchläufe wird in einer Zählvariable gezählt. Viele Hochsprachen erlauben auch das Zählen mit einer bestimmten, ggf. auch negativen Schrittweite.

9.2.1 Abweisende Schleifen

Betrachten wir das zweite Beispiel vom Anfang dieses Kapitels:

```
summe := 0;
i := 0;
WHILE summe <= 100
```

```
    i := i + 1;
    summe := summe + i
END;
```

Die ersten beiden Zeilen dieses (ineffizienten) Programmfragments sind schnell übersetzt:

```
li    $t0, 0        # summe := 0;
li    $t1, 0        # i := 0;
```

Für die Schleife benötigen wir sicherlich eine Marke, schließlich wollen wir am Ende der Schleife stets wieder an ihren Anfang springen. Falls die Eintrittsbedingung nicht mehr erfüllt ist, müssen wir dann von dort wieder an das Ende der Schleife springen. Wir benötigen also auch dort eine Marke. Unser Assemblerprogramm könnte also so aussehen:

Beispiel 7: Eine Abweisende Schleife

```
# BSP004.S
        li      $t0, 0          # summe := 0;
        li      $t1, 0          # i := 0;
while:  bgt     $t0, 100, elihw # While summe <= 100
        addi    $t1, $t1, 1     #       i := i + 1;
        add     $t0, $t1, $t0   #       summe := summe + i
        b       while           # END;
elihw:
```

Zu beachten ist die umgekehrte Sprungbedingung wie wir sie auch schon bei den Fallunterscheidungen kennen gelernt haben. Ungünstig wirkt sich der unbedingte Rücksprung am Schleifenende aus. Dieser Befehl kann eingespart werden, wenn man statt ihm einen bedingten Sprungbefehl mit der negierten Abweisungsbedingung einsetzt, der an den ersten Befehl in der Schleife springt (`addi`). Diese effizientere Version ist jedoch etwas unübersichtlicher und damit fehleranfällig.

9.2.2 Nachprüfende Schleifen

Die nachprüfenden Schleifen sind etwas einfacher zu programmieren als die abweisenden Schleifen, weil hier der unbedingte Rücksprung zur Prüfung entfällt. Da dieser Rücksprung bei jedem Schleifendruchlauf ausgeführt wird, empfiehlt es sich, wenn möglich, die nachprüfenden Schleifen den abweisenden vorzuziehen.

Beispiel 8: Eine nachprüfende Schleife

```
# BSP005.S
        li      $t0, 0           # summe := 0;
        li      $t1, 0           # i := 0;
                                 # REPEAT
repeat: addi    $t1, $t1, 1      #       i := i + 1;
        add     $t0, $t1, $t0    #       summe := summe + i
        ble     $t0, 100, repeat # UNTIL summe > 100;
```

9.2.3 Zählschleifen

Es gibt Zählschleifen, bei denen bereits zum Entwicklungszeitpunkt bekannt ist, wie oft sie ausgeführt werden. Ist die Zahl der Ausführungen sehr gering und die Schleife sehr kurz, so kann es unter Umständen günstiger sein, den zu wiederholenden Code mehrfach hintereinander zu schreiben. Im Regelfall sollte aber auch bei bekannter Durchlaufanzahl auf eine Zählschleife zurückgegriffen werden.

Die Zählschleifen lassen sich auch als abweisende Schleifen formulieren, nur liegt dann die Bearbeitung der Zählvariable in der Hand des Programmierers (in Hochsprachen). Im Assemblerprogramm unterscheiden sich die beiden Schleifen kaum. Zu Dokumentierungszwecken sollten die beiden Schleifenarten jedoch auseinander gehalten werden.

Wollen wir beispielsweise die geraden Zahlen von 0 bis 100 aufsummieren, so würden wir in MODULA schreiben:

```
summe := 0;
FOR i := 0 TO 100 STEP 2 DO
   summe := summe + i
END;
```

Die Übersetzung der ersten Zeile bereitet uns keine Schwierigkeiten. Wir entscheiden uns, die Laufvariable `i` im Register `$t1` zu halten. Den Rest der Schleife können wir übersetzen wie eine abweisende Schleife. Natürlich müssen wir auch die Verwaltung der Laufvariablen bedenken, die uns in MODULA der MODULA-Compiler abgenommen hätte. Die Laufvariable wird grundsätzlich als letztes in der Schleife um die Schrittweite erhöht. In Abschnitt 7 auf der vorangegangenen Seite hatten wir erfahren, dass die abweisenden Schleifen effizienter sind, wenn sie keine unbedingte Sprunganweisung enthalten. Im folgenden Beispiel sehen wir jetzt eine solche effiziente Version.

Beispiel 9: Eine effiziente Zählschleife

```
# BSP006.S
        li      $t0, 0            # summe := 0;
        li      $t1, 0            # i := 0;
        bgt     $t1, 100, rof     # FOR i := 0 TO 100 STEP 2 DO
for:    add     $t0, $t0, $t1     #    summe := summe + i
        addi    $t1, $t1, 2
        blt     $t1, 100, for     # END;
rof:
```

9.2.4 Schleifen verlassen

Wenn wir in Assembler programmieren, haben wir die Möglichkeit, an jeder Stelle einer Schleife die Schleifenbearbeitung abzubrechen und an das Ende der Schleife zu springen. Dieses Ende ist in der Regel ja durch eine Marke gekennzeichnet. Diese Möglichkeit besteht in vielen Hochsprachen nicht (in MODULA gibt es eine wenig bekannte `LOOP`-Schleife, die mit dem Befehl `EXIT` an jeder beliebigen Stelle verlassen werden kann).

Sicherlich tragen solche Schleifenabbrüche zu einer günstigeren Laufzeit bei, aber sie machen das Programm unübersichtlich. Daher sollte darauf wenn irgendmöglich verzichtet werden. Wenn Schleifen außerhalb der Abbruchbedingungen verlassen werden müssen, dann nur

- durch Sprung an den auf die Schleife unmittelbar folgenden Befehl und
- unter Dokumentierung, am besten mit Hilfe eines Befehls ähnlich dem `EXIT`-Befehl von MODULA.

Sind mehrere Schleifen verschachtelt, so kann es auch reizvoll sein, nicht an das Ende der gerade aktiven Schleife zu springen, sondern darüber hinaus an das Ende einer die aktive Schleife einschachtelnden Schleife zu springen. Es sollte jedoch genau abgewogen werden, ob sich dies lohnt, denn die Wartung und Fehlersuche werden dadurch sehr erschwert.

Übungen

Übung 21
Gebe ein aus Sternchen bestehendes Dreieck der Höhe n aus. n soll vom Benutzer eingegeben werden und zwischen 1 und 20 liegen. Für $n = 3$ soll folgendes Dreieck ausgegeben werden:

```
  *
 ***
*****
```

Übung 22
Implementiere die Fibonacci-Zahlen iterativ. Zur Erinnerung:

$$fib(n) := fib(n-1) + fib(n-2), fib(2) = fib(1) = 1$$

10 Datenhaltung II: Komplexe Datentypen

Wir haben nun gelernt, mit dem **MIPS** ganze Zahlen verschiedener Länge zu bearbeiten. Höhere Programmiersprachen bieten aber auch die Möglichkeit einfache Datentypen zusammenzufassen. Datentypen verschiedenen oder gleichen Typs können miteinander verbunden werden und mit einer einzigen (Hochsprachen-) Operation mit Werten belegt werden. Diese *Verbunde (Record, structure)* bieten die Möglichkeit, alle Daten, die zu einem Objekt gehören, in einer einzigen Variable vorliegen zu haben.

Verbunde

Häufig brauchen wir aber auch eine ganze Reihe Variabler gleichen Typs, etwa eine Liste mit Personen. Hierfür stellen fast alle Hochsprachen *Felder (Array, List)* zur Verfügung. Innerhalb des Feldes haben die Feldelemente eine feste Reihenfolge und sind über ihren Index zugreifbar.

Felder

Beide komplexen Datentypen können in Hochsprachen mehr oder weniger beliebig miteinander kombiniert werden. Änderungen an ihrer Struktur sind oft mit wenigen Handgriffen möglich. Auf Assemblerebene ist dies ganz anders, wie wir jetzt sehen werden.

10.1 Felder

10.1.1 Erstellung und Initialisierung von Feldern

Abgesehen von sehr kleinen Feldern müssen Felder grundsätzlich im Hauptspeicher abgelegt sein. Nur zum Lesen werden die Feldelemente in die Prozessorregister geladen. Aus Abschnitt 4.1 auf Seite 15 kenner wir bereits einige Direktiven, die im Datensegment Platz für Zahlen belegen. Zur Erinnerung:

`.byte` legt die nachfolgenden Zahlen als 8-Bit-Zahlen ab,

`.half` legt die nachfolgenden Zahlen als 16-Bit-Zahlen ab,

`.word` legt die nachfolgenden Zahlen als 32-Bit-Zahlen ab und

`.double` legt die nachfolgenden Zahlen als 64-Bit-Zahlen.

Ein Feld von 13 Wörtern können wir also wie folgt erstellen:

```
        .data
feld:   .word   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Damit sind alle Feldelemente zugleich mit dem Wert 0 initialisiert. Leider gibt es keine Direktive, die dieses wiederholte Eintragen der Werte erleichtert.

.space

Es gibt aber auch die Möglichkeit mit dem Befehl `.space` ganze Bereiche zu reservieren, ohne sie gleichzeitig zu initialisieren:

```
         .data
feld:    .space   52
```

Der Parameter gibt die Zahl von Bytes an, die reserviert werden. Beide Direktiven legen also ein Feld mit 13 Wörtern an. Wesentlich ist die Feststellung, dass `.space` die Feldelemente *nicht* initialisiert.

Die Initialisierung durch `.word` u.ä. Direktiven kostet zwar keine Laufzeit, da sie beim Assemblieren durchgeführt wird, ist aber auch nicht zwingend notwendig, da der erste Zugriff auf eine Variable sowieso nicht lesend, sondern besser schreibend erfolgen sollte. Schließlich kann man sonst oft nicht mit letzter Sicherheit bestimmen, mit welchem Wert die Variable belegt ist. Es gibt sicher gute Gründe von diesem Prinzip abzuweichen. Aber bei Feldern sind solche Gründe sehr selten.

Wenn man wie wir den Simulator SPIM benutzt, gibt es ein weiteres Argument gegen die einmalige Initialisierung: Vor dem nächsten Programmdurchlauf werden die Daten im Datensegment nicht gelöscht, wenn man nicht das gesamte Programm neu lädt.

10.1.2 Arbeiten mit Feldern

Von dem Feld ist uns nur die Anfangsadresse bekannt. In der Regel wird erst zur Laufzeit entschieden, auf welches Feldelement wir zugreifen wollen. Die Hauptspeicheradresse, auf die wir zugreifen wollen, ist also variabel.

Die Lade- und Speicherbefehle erlauben alle die indexierte Adressierung (Vgl. Abschnitt 5.1.1 auf Seite 18). Wollen wir beispielsweise alle Feldelemente unseres 13-elementigen Feldes mit ihrem Index belegen, so könnten wir dies so tun:

Beispiel 10: Indexieren eines Feldes

```
# BSP007.S
        .data
feld:   .space  52              # VAR:
                                #   feld: ARRAY [0..12] OF INTEGER;
        .text
main:   li      $t0, 0
for:    bgt     $t0, 48, rof    # FOR i := 0 TO 12 DO
        sw      $t0, feld($t0)  #    feld[i] := i;
        addi    $t0, $t0, 4
        b       for             # END;(*FOR*)
rof:
```

Das Beispielprogramm zeigt das Prinzip und lässt sich leicht effizienter gestalten.

Feldzuweisung

Soll ein Feld einem anderen *zugewiesen* werden, so müssen mühselig die Daten des einen Feldes in den Speicher des anderen kopiert werden.

Bereichsgrenzen

Großes Augenmerk müssen wir den *Bereichsgrenzen* widmen. Keinesfalls darf außerhalb des Feldes gelesen oder geschrieben werden. Derartige Fehler sind sehr schwierig zu finden und können unberechenbare Folgen haben. Daher ist hier Selbstdisziplin unabdingbar. Viele Compiler fügen vor jedem Zugriff auf ein Feldelement, egal ob lesend oder schreibend, Code ein, der überprüft, ob der Index im zulässigen Bereich liegt.

Übungen

Übung 23
Lese eine Folge von Zahlen über die Tastatur ein. Die Eingabe wird beendet, wenn die 0 eingegeben wird oder wenn die 20. Zahl gelesen wurde. Sortiere diese Folge aufsteigend in einem beliebigen Verfahren und gebe sie aus.

Übung 24
Lege im Datensegment zwei null-terminierte Zeichenketten ab. Kopiere den Inhalt der zweiten Zeichenkette in die erste Zeichenkette. Was für ein Problem kann sich dabei ergeben?

10.2 Verbunde

Verbunde werden wir nur kurz behandeln, da sie im Gegensatz zu den Feldern zwar umständlich aber unproblematisch zu behandeln sind.

Es soll hier ausreichen sich folgende Gedanken zu machen:

- Bei der Haltung im Datensegment reicht in der Regel die Belegung mit der `.space`-Direktive aus.
- Die Zuordnung der einzelnen Komponenten eines Verbundes zu den Bytes muss dokumentiert werden.
- Ein Record sollte ausgerichtet, d.h. Wörter sollten keinesfalls über die Wortgrenzen hinaus abgelegt werden. Komponenten von nicht ganzzahliger Wortanzahl sollten am Ende des Verbundes abgelegt werden.
- Änderungen an der Verbundstruktur sind in Assemblersprache wesentlich aufwendiger als in Hochsprachen.
- Das Gleichsetzen von zwei Verbunden kann wortweise, also nicht Komponente für Komponente erfolgen.

11 Mehrfache Fallunterscheidung (CASE-Anweisung)

In Abschnitt 9.1 auf Seite 32 haben wir die einfache Fallunterscheidung kennen gelernt. Häufig benötigen wir jedoch eine *mehrfache Fallunterscheidung*, in der in Abhängigkeit vom Wert eines Ausdrucks an mehr als zwei verschiedene Stellen im Programm gesprungen wird, etwa nach einer Menüauswahl.

mehrfache Fallunterscheidung

Eine Lösung dieses Problems wäre ein Folge einfacher Fallunterscheidungen:

Beispiel 11: Mehrfache Fallunterscheidung mit einfachen Fallunterscheidungen

```
# CASE1.S - Programmfragment, nicht lauffaehig!
# Menueauswahl in $v0, 0 <= $v0 <= 4
        beq     $v0, 0, case0
        beq     $v0, 1, case1
        beq     $v0, 2, case2
        beq     $v0, 3, case3
        b       case4
```

Der Nachteil dieses Verfahrens ist offensichtlich: Bei vielen Alternativen müssen alle vor der gewünschten Alternative auftretenden Verzweigungsbefehle abgearbeitet werden. Wir können zwar etwas nachhelfen und die Alternativen in der Reihenfolge ihrer voraussichtlichen Häufigkeit sortieren oder wir können uns durch Intervallschachtelung sehr viel schneller der gewünschten Alternative nähern.

Wir sollten aber auch einen Blick auf die *Sprungtabelle (jump address table)* werfen, einem Verfahren, das eine sehr effiziente Sprungverzweiung mit nur einer einzigen Sprunganweisung ermöglicht.

Sprungtabelle (jump address table)

Dafür benötigen wir einen weiteren Befehl:

Befehl	Argumente	Wirkung	Erläuterung
`jr`	`Rs`	unbedingter Sprung an die Adresse in `Rs`	jump register

`jr` ermöglicht uns den Sprung an eine erst zur Laufzeit ermittelte Stelle im Programm. Die Idee der Sprungtabelle ist nun, im Datensegment ein Feld anzulegen, das die Adressen der Sprungziele enthält. Da diese Adressen bereits zur Assemblierzeit feststehen, muss zur Laufzeit lediglich noch die richtige Adresse geladen werden.

Beispiel 12: Mehrfache Fallunterscheidung mit Sprungtabelle

```
# CASE2.S - Mehrfache Fallunterscheidung mit Sprungtabelle
.data
jat:    .word case0, case1, case2, case3, case4
        # Sprungtabelle wird zur Assemblierzeit belegt.

.text
main:   li      $v0, 5          # RdInt
        syscall
        blt     $v0, 0, error   # Eingabefehler abfangen
        bgt     $v0, 4, error
        mul     $v0, $v0, 4     # 4-Byte-Adressen
        lw      $t0, jat($v0)   # $t0 enthält Sprungziel
        jr      $t0
case0:  li      $a0, 0          # tu dies und das
        j       exit
case1:  li      $a0, 1          # tu dies und das
        j       exit
```

```
 · case2:  li      $a0, 2          # tu dies und das
 ·         j       exit
 · case3:  li      $a0, 3          # tu dies und das
20         j       exit
 · case4:  li      $a0, 4          # tu dies und das
 ·         j       exit
 · error:  li      $a0, 999        # tu dies und das
 · exit:   li      $v0, 1          # WrInt
25         syscall
 ·         li      $v0, 10         # Exit
 ·         syscall
```

Zu beachten ist, dass die Auswahl in die korrekte Speicheradresse umgewandelt wird. Da die Adressen jeweils ein Wort breit sind, muss die Auswahl mit vier multipliziert werden. Größte Sogfalt müssen wir darauf verwenden, das die berechnete Adresse tatsächlich innerhalb der Sprungtabelle liegt. Ein Laden eines benachbarten Wertes führt zu einer Fortsetzung des Programmes an einer falschen Stelle. Fehler dieser Art können extrem schwierig zu lokalisieren sein.

Die Sprungtabelle ist auch dann verwendbar, wenn für eine ganze Wertegruppe ein Sprung an dieselbe Stelle ausgeführt werden soll. In diesem Fall braucht lediglich die Zeile 3 in dem vorangegangenen Beispiel entsprechend verändert zu werden.

Übungen

Übung 25

Der folgenden Tabelle können die (vereinfachten) Entgelte für weltweite Luftpostbriefe entnommen werden:

Gewichtsstufe:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Entgelt in DM:	*3*	*6*	*10*	*16*	*24*	*32*	*40*	*56*	*72*

Nach Eingabe des Gewichts soll der Benutzer das Entgelt ausgegeben bekommen. Verwende bei der Lösung eine Sprungtabelle, obwohl sie hier nicht unbedingt notwendig wäre!

12 Kellerspeicher (Stack)

12.1 Das Grundprinzip

Häufig benötigen wir *temporäre Variablen* für Zwischenergebnisse bei der Auswertung von Ausdrücken o.ä. Meist stehen hierfür die Register `$t0` bis `$t9` zur Verfügung. Diese reichen aber gelegentlich nicht aus und für die Auswertung von Ausdrücken variabler Länge sind sie ungeeignet. temporäre Variablen

Abhilfe schafft der *Stack (Kellerspeicher)*, ein Bereich im Hauptspeicher, der je nach Bedarf wächst oder schrumpft. Bei dem Stack handelt es sich um einen sogenannten *LIFO (Last-in-First-out)-Speicher*: Bevor auf das vorletzte Element zugegriffen werden kann, muss zunächst das letzte Element vom Stack entfernt werden. Stack (Kellerspeicher) LIFO-Speicher

Bei dem **MIPS** wie auch bei den meisten anderen Rechnersystemen wird der Stack am Ende des adressierbaren Speichers angeordnet und wächst von dort mit geringer werdenden Hauptspeicheradressen in Richtung auf die Adresse 0. Eine andere Möglichkeit wäre, den Stack an das Ende des Programms zu setzen und ihn von dort mit gößer werdenden Adressen auf das Ende des Hauptspeichers hinwachsen zu lassen.

Die gewählte Alternative hat den Vorteil, dass die Startadresse des Stacks nicht abhängig von der Länge des geladenen Programms ist, was das dynamische Linken fast unmöglich machen würde. Aufgabe des Betriebssystems ist es zu überprüfen, ob der Stack in das Programm hineinwächst. Tritt dieser Fall ein, kann das Programm nur noch abgebrochen werden.

Noch komplizierter wird das Ganze, wenn die Programme auch noch zur Laufzeit weiteren (Daten-)Speicher anfordern dürfen. In der Regel wird diese Anforderung erfüllt, indem ein „umgedrehter“ Stack, ein *Heap* (Haufen) dem Stack entgegen wächst. In diesem Fall kann es jedoch passieren, dass dieser dynamische Datenbereich in den Stack hineinwächst. Dies ist leicht festzustellen, da beim Einkellern und bei der Speicheranforderung lediglich der Stackpointer, ein Register, das auf den Stack zeigt, und der Heappointer auf Gleichheit verglichen werden müssen. Dieses Kriterium ist dann vom Betriebssystem zu überprüfen. Heap

Das Speicherlayout des **MIPS** ist Abbildung 8 auf der folgenden Seite zu entnehmen. In dem Bereich oberhalb von `0x7FFF FFFF` befinden sich Daten und Programme des Betriebssystems (Kernel).

12.2 Die Stackprogrammierung: Einkellern

Der *Stackpointer* (`$sp`) zeigt stets auf das *erste freie* Wort auf dem Stack. Dies ist eine Konvention, die *unbedingt* eingehalten werden muss. Nach dem Laden eines Programmes zeigt der Stackpointer also auf `0x7FFF FFFC`, da dies das oberste Wort des Stacksegmentes ist.[7] Die Programmierung mit dem Stack ist fehleranfällig und sollte besonders sorgfältig vorgenommen werden, da Fehler sehr schwer zu lokalisieren sind. Stackpointer

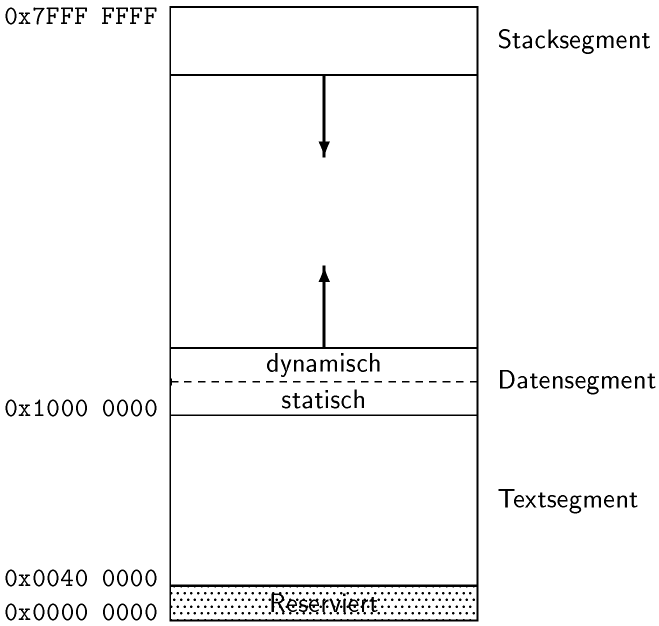
Einige CISC-Prozessoren, wie z.B. die Motorola 68 000er-Serie, stellen für Stackoperationen eigene Befehle wie z.B. `PUSH` für das „Einkellern“ oder `POP` für das Entfernen von Stackelementen zur Verfügung. Der **MIPS** kennt derartige Befehle nicht. Wir müssen also mit den uns bereits zur Verfügung stehenden Befehlen auskommen.

Wollen wir ein Element auf den Stack legen, so könnten wir wie folgt vorgehen:

```
        sw      $t0, ($sp)      # $t0 liegt auf dem erste freien Platz
        addi    $sp, -4         # Stackpointer dekrementieren
```

[7] Tatsächlich wird der Stackpointer mit `0x7FFF FEFC` initialisiert. Der Grund hierfür ist mir nicht bekannt.

Abbildung 8: Das Speicherlayout des **MIPS**



In der ersten Zeile wird der Inhalt des Registers `$t0` an den Speicherplatz geschrieben, der im Stackpointer angegeben ist. In der zweiten Zeile wird der Stackpointer verringert, da der Stack in Richtung auf die Speicherstelle `0` wächst. Das Dekrementieren des Stackpointers ist sehr wichtig, da andernfalls das oberste Element überschrieben werden kann. Da der Stackpointer auf Speicherplätze zeigt, muss er jeweils um ganze Worte erhöht oder verringert werden, weshalb der Stackpointer in der zweiten Zeile um `4` verringert wird.

Die hier vorgestellte Version ist jedoch gefährlich, da theoretisch nach *jedem* Befehl eine Unterbrechung (z.B. Tastatur gedrückt o.ä.) auftreten kann. Dann wird automatisch ein Betriebssystemteil ausgeführt, das eventuell den Stack manipuliert. Tritt eine solche Unterbrechung nun unglücklicherweise nach dem `sw`-Befehl auf, so kann unser `$t0` durch das Betriebssystem überschrieben werden. Ein solcher Fehler ist praktisch nicht reproduzierbar und muss daher unbedingt verhindert werden.

Durch zwei geringfügige Änderungen läßt sich aus dem vorangegangen unsicheren Code ein sicherer Code machen:

```
addi    $sp, -4         # Stackpointer dekrementieren
sw      $t0, 4($sp)     # $t0 liegt auf dem erstem freien Platz
```

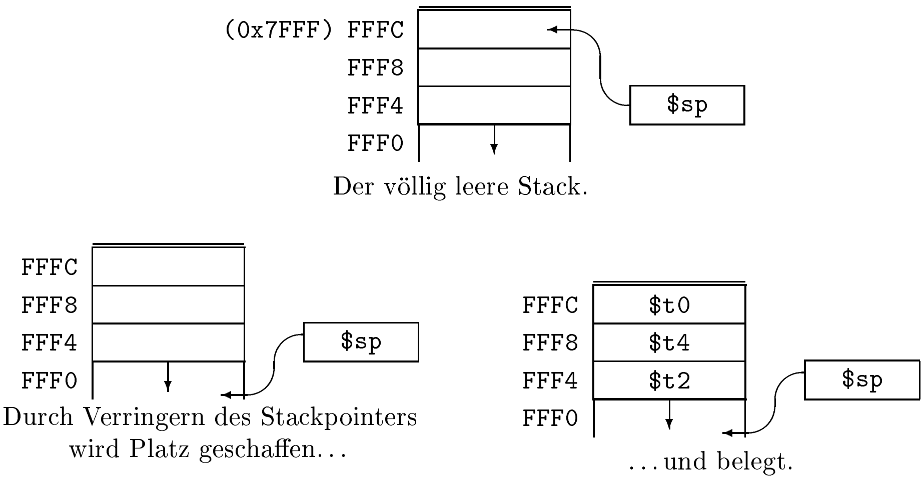
Nun darf unser Programm auch zwischen den beiden betrachteten Zeilen unterbrochen werden, ohne dass dies zu Problemen führen kann. Ein Pseudobefehl `push` könnte auf diese Weise definiert werden.

Häufig müssen wir gleich mehrere Werte im Stack ablegen. Man könnte dann die obige Sequenz entsprechend häufig hintereinander schreiben. Es bietet sich aber an, statt dessen wie folgt zu programmieren:

```
addi    $sp, -12        # $sp dekrementieren
sw      $t0, 12($sp)    # $t0 liegt auf erstem freien Platz
sw      $t4, 8($sp)     # $t4 liegt auf zweitem freien Platz
sw      $t2, 4($sp)     # $t2 liegt auf drittem freien Platz
```

Man beachte, dass der erste freie Platz nun durch den größten Index (12) adressiert wird. Zur Verdeutlichung ist die Entwicklung des Stacks in der Abbildung 9 auf dieser Seite dargestellt.

Abbildung 9: Die Entwicklung des Stacks beim „Einkellern“



Gegenüber dem naiven Ansatz, jedes Register durch die beiden Zeilen unseres nicht vorhandenen `push`-Pseudobefehls zu ersetzen, sparen wir zwei `addi`-Befehle. Aber sparen wir auch Rechenzeit? Schließlich muss der Stackpointer in den Zeilen zwei bis vier ebenfalls inkrementiert werden, um die richtige Hauptspeicheradresse zu erhalten. Diese Inkrementierung wird aber auf Schaltkreisebene realisiert und benötigt somit keine zusätzliche Rechenzeit.

12.3 Die Stackprogrammierung: Lesen vom Stack

Analog zum Schreiben auf den Stack funktioniert das Abräumen des Stacks:

```
lw      $t2, 12($sp)     # $t2 erhält den dritten Wert vom Stack
lw      $t4, 8($sp)      # $t4 erhält den zweiten Wert vom Stack
lw      $t0, 4($sp)      # $t0 erhält den ersten Wert vom Stack
addi    $sp, 12          # $sp inkrementieren
```

Die Entwicklung des Stacks ist der Abbildung 10 auf der folgenden Seite zu entnehmen.

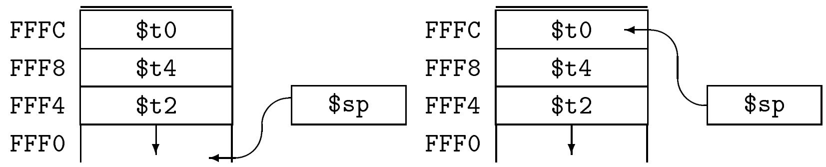
Zu beachten ist, dass die Werte bis zum nächsten Schreibvorgang erhalten bleiben. Sie sind jedoch nicht mehr sicher, da nach dem Inkrementieren des Stackpointers auftretende Unterbrechungen zu ihrer Veränderung führen können!

Natürlich kann auf die Stackelemente auch ohne sofortiges Inkrementieren des Stackpointers zugegriffen werden, etwa für die weitere Speicherung der temporären Variablen. Diese Zugriffe können sowohl lesend als auch schreibend vorgenommen werden, sie sollten aber gut dokumentiert werden.

12.4 Anwendungen und Beispiele

Eine der wesentlichen Anwendungen des Stacks ist die Realisierung beliebig häufiger Unterprogramm- oder Prozedur-Aufrufe. Diesem Thema werden wir

Abbildung 10: Die Entwicklung des Stacks beim „Abräumen“



Der belegte Stack… …wird geleert.

im nächsten Abschnitt einige Aufmerksamkeit schenken.

Doch kann der Stack auch sonst hilfreich sein. Werden zum Beispiel in einem Programmabschnitt große Datenmengen benötigt, die nicht in Registern gehalten werden können und nach Verlassen dieses Programmabschnittes nicht mehr benötigt werden, so könnte man für diesen Zweck Platz im Datensegment reservieren. Dieser Speicherplatz geht aber für andere Daten verloren. Es ist zwar denkbar diesen Speicherplatz auch von anderen Stellen im Programm aus zu benutzen, aber dies stellt eine gefährliche Fehlerquelle dar. Der Stack schafft Abhilfe und stellt ausreichend Speicherplatz zur Verfügung.

Eine weitere Anwendung ist die Auswertung von langen Ausdrücken. Statt jedes Zwischenergebnis in einem Register zu halten, können diese Zwischenergebnisse auch auf dem Stack abgelegt werden. Dies gilt besonders, wenn der Ausdruck erst zur Laufzeit festgelegt wird, etwa durch Eingabe des Benutzers.

Übungen

Übung 26
Bei der Postfix-Schreibweise werden Operatoren nach *den Operanden geschrieben, z.B.*

$$234+-\mathrel{\hat{=}}(3+4)-2$$

Schreibe ein Programm, das einen Postfix-Ausdruck als Zeichenkette übergeben bekommt und ihn auswertet. Die Zeichenkette enthält nur die Ziffern `0` *bis* `9` *und die Zeichen für die vier Grundrechenarten* `+`, `-`, `*` *und* `/`. *Als Eingabewerte sind nur Ziffern zugelassen, also die Zahlen von* 0 *bis* 9.

12.5 Vor- und Nachteile der CISC-Befehle PUSH und POP

In Abschnitt 12.2 auf Seite 45 haben wir die CISC-Befehle PUSH und POP angesprochen. Ihr Vorteil ist, dass der Programmierer den Stackpointer nicht mehr direkt manipulieren muss und somit die Fehlergefahr abnimmt. Die Reihenfolge des Einkellerns und Abräumens des Stacks ist deutlicher und kann vom Leser schneller erfasst werden.

Nachteilig an diesen Befehlen ist der (für den Fall $n \to \infty$) doppelte Rechenaufwand durch das n-malige Ausführen der Additionsbefehle. Soll auf eine anderes als das letzte Element des Stacks zugegriffen werden, so muss auch bei der Verwendung der Befehle PUSH und POP die Adressierung über den Abstand zum Stackpointer vorgenommen werden.

13 Prozeduren

Einer der bereits frühzeitig formulierten Grundsätze der Programmierung lautet '*Share identical code!*', frei übersetzt: „*Schreibe nichts doppelt!*“. Code, der an mehreren Stellen benötigt wird, soll nur an einer Stelle im Programm stehen. Daraus ergibt sich eine Verringerung

1. des Wartungsaufwands,
2. des Programmier- und Prüfungsaufwands und
3. des Platzbedarfs.

Bereits frühzeitig wurde daher das Konzept des *Unterprogrammes* entwickelt. Der Name Unterprogramm rührt von dem Umstand her, dass diese Unterprogramme in vielen Programmiersprachen (z.B. in der Sprache C) hinter, also *unter* dem Hauptprogramm standen. Bei MODULA-2 ist es bekanntlich umgekehrt. Im einfachsten Fall erlaubt ein Rechner bzw. eine Programmiersprache nur den Sprung an eine bestimmte Stelle im Programm und ermöglicht den *Rücksprung* an die auf den aufrufenden Befehl *folgende* Anweisung. Dies entspricht den BASIC-Befehlen `GOSUB` und `RETURN`.

Unterprogramm

Rücksprung

Moderne Hochsprachen erlauben die Übergabe von Daten an Unterprogramme. Solche Unterprogramme nennt man im Allgemeinen *Prozeduren* oder –wenn sie einen Wert zurückliefern– *Funktionen*. Auf Assemblerebene sind beide Verfahren gleich kryptisch und fehleranfällig.

Prozeduren/Funktionen

Bei Unterprogrammen unterscheidet man zwischen dem *caller*, dem Aufrufer und dem *callee*, dem Aufgerufenen. Ein callee kann selbst zum caller werden, indem er ein weiteres Unterprogramm aufruft.

caller/callee

13.1 Einfache Unterprogrammaufrufe

Häufig lohnt sich die Programmierung nach den strengen Formalien, die wir im nächsten Abschnitt kennen lernen werden, nicht. Dies ist z.B. der Fall bei sehr kleinen Assemblerprogrammen und nur wenigen Unterprogrammen, die mit sehr wenigen Parametern auskommen. Wir behandeln diese Technik an dieser Stelle nur, um die Möglichkeit aufzuzeigen und um einige Befehle einzuführen. Von der Benutzung ist eher abzuraten.

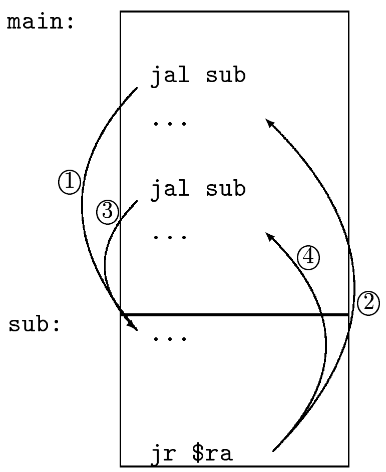
Folgende Befehle werden uns im nützlich sein:

Befehl	Argumente	Wirkung	Erläuterung
`jal`	`label`	unbedingter Sprung nach `label`, Adresse des nächsten Befehls in `$ra`	jump and link
`jalr`	`Rs, Rd`	unbedingter Sprung an die Stelle `Rs`, Adresse des nächsten Befehls in `Rd`, Default: `$ra`	jump and link register
`bgezal`	`Rs, label`	Sprung nach `label` wenn `Rs` ≥ 0, Adresse des nächsten Befehls in `$ra`	Branch on greater than equal zero and link
`bltzal`	`Rs, label`	Sprung nach `label` wenn `Rs` < 0, Adresse des nächsten Befehls in `$ra`	Branch on less than and link

Diese Befehle erlauben uns ein Unterprogramm aufzurufen. Die Rücksprungadresse (Adresse des ersten nach Abarbeitung des Unterprogramms auszuführenden Befehls) wird bei Benutzung dieser Befehle im Register `$ra` abgelegt. Eine eventuell dort bereits befindliche Adresse wird logischerweise dabei überschrieben. Dies muss natürlich bei verschachtelten Unterprogrammaufrufen verhindert werden.

Mit dem uns bereits bekannten Befehl `jr` kann nach Abarbeitung des Unterprogramms die Programmausführung fortgesetzt werden. Das Grundprinzip ist in Abbildung 11 auf dieser Seite dargestellt: das aufrufende Hauptprogramm (caller) ruft mittels `jal` das Unterprogramm (callee) auf (1). Nach Abarbeitung des Unterprogramms gibt dieses die Kontrolle wieder an das Hauptprogramm ab (2). Später wiederholt sich dieser Vorgang (3, 4).

Abbildung 11: Das Grundprinzip des Unterprogrammaufrufs



Werfen wir einen Blick auf den zughörigen SPIM-Code:

Beispiel 13: Ein einfacher Unterprogrammaufruf

```
# UNTER1.S - Einfacher Unterprogrammaufruf - nicht lauffaehig!
#
.text
main:   # ... tu dies ...
        jal sub              # Aufruf des Unterprogramms
        # ... tu das ...
        jal sub              # Aufruf des Unterprogramms
                             #
        li $v0, 10           # Exit
        syscall
#
sub:    # ... tu dies und das ...
        jr $ra               # Zurück zum Aufrufer
```

Dieses Beispiel zeigt jedoch nur das Prinzip. Das Unterprogramm soll eine Befehlsfolge ausführen. Da der MIPS eine Load-and-Store-Architektur hat, verändern fast alle Befehlsfolgen mindestens ein Register, zumindest greifen sie lesend auf den Inhalt eines Registers zu. Arbeiten wir mit solchen einfachen Unterprogrammaufrufen, so müssen alle gelesenen und veränderten Register im Assemblertext dokumentiert werden! Das vorangegangene Beispiel könnte also wie folgt aussehen:

Beispiel 14: Ein einfacher kommentierter Unterprogrammaufruf

```
# UNTER2.S - Einfacher Unterprogrammaufruf, kommentiert - nicht lauffaehig!
#
.text
main:   # ... tu dies ...
        jal sub                 # Aufruf des Unterprogramms
        # ... tu das ...
        jal sub                 # Aufruf des Unterprogramms
                                #
        li $v0, 10              # Exit
        syscall
                                #
                                #
                                #
# Unterogramm sub
# Eingabe:      $a0: a > 0
#               $a1: b > 0
# Ausgabe:      $v0: 0, wenn a und b teilerfremd, 1 sonst
#               $a0, $a1: unverändert
#               $t0: verändert, Datenschrott
#               $t1: verändert, Datenschrott
#               $t2: verändert, Datenschrott
#               alle anderen: unverändert
sub:    # ... tu dies und das ...
        jr $ra                  # Zurück zum Aufrufer
```

In diesem Fall müsste also der Aufrufer dafür Sorge tragen, dass er relevante Daten in den Registern `$t0` bis `$t2` zuvor sichert (*caller saved*). Hierfür bietet sich der Stack an.

caller saved

Es wäre natürlich auch denkbar, dass das Unterprogramm alle Register wieder so hinterlässt wie es sie vorgefunden hat (*callee saved*). Beide Verfahren haben Vor- und Nachteile. Muss der callee alle von ihm benutzten Register sichern, so sichert er vielleicht auch Register, die vom caller nie benutzt wurden oder deren Inhalt jetzt nicht mehr benötigt wird. Lässt man alle Register vom caller sichern, so sichert dieser vielleicht Register, die der callee gar nicht verändert. Ein Mittelweg, auf dem Register teilweise vom caller und teilweise vom callee gesichert werden, scheint viel versprechend zu sein.

callee saved

Sicherlich könnten die Autoren der Unterprogramme entsprechend genaue Angaben über veränderte Register machen. Die Einhaltung dieser Spezifikationen durch die Programmierer der caller ist jedoch fehlerträchtig. Zudem wäre es wünschenswert, dass einige Register vom caller nie gesichert werden müssten, während andere stets gesichert werden müssten. Erstere Register würden der Haltung langlebiger Variabler dienen, letztgenannte der Haltung von Zwischenergebnissen.

Eine entsprechende Konvention wurde für den **MIPS** vereinbart. Wir werden sie in dem folgenden Abschnitt kennen lernen.

13.2 Verschachtelte Unterprogrammaufrufe (Prozedur-Konvention)

Die **MIPS**-Prozedur-Konvention (*Procedure Call Convention*) läßt sich in vier Teile gliedern:

Procedure Call Convention

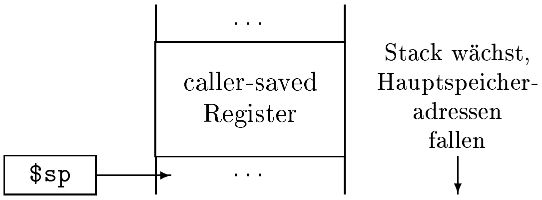
1. den Prolog des Callers,
2. den Prolog des Callees,
3. den Epilog des Callees und
4. den Epilog des Callers.

13.2.1 Prolog des Callers

(a) Sichere alle caller-saved Register:

Wir müssen alle Register, die vom callee verändert werden dürfen, die wir aber nach Beendigung des callees noch benötigen, sichern. Dies sind die Register `$a0`-`$a3`, `$t0`-`$t9` sowie `$v0` und `$v1`. Alle weiteren Register muss der callee im gleichen Zustand hinterlassen wie er sie vorgefunden hat.[8]

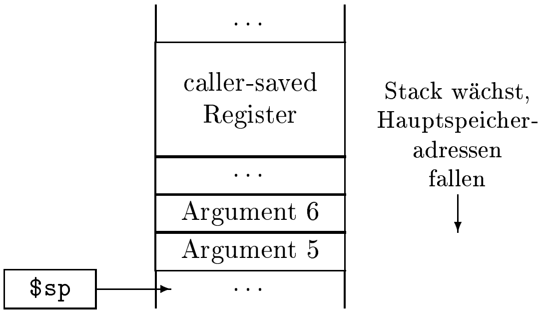
Unser Stack sieht nun so aus:



(b) Übergebe die Argumente: Die ersten vier Argumente übergeben wir in den Registern `$a0` bis `$a3`. Sollen weitere Argumente übergeben werden, so werden diese in *umgekehrter* Reihenfoge (das fünfte als letztes) auf dem Stack übergeben.

Hierbei ist die Art der Parameterübergabe (call-by-value bzw. call-by-reference) zu beachten, siehe Ende dieses Abschnittes!

Der Stack sieht jetzt so aus:



(c) Starte die Prozedur (`jal`)

Wert- und Variablenparameter

Unter Nr. 2 haben wir erfahren, wo und in welcher Reihenfolge Argumente an die Prozedur übergeben werden. Bei einfachen Datentypen ist dieses Verfahren auch zufrieden stellend. Was aber soll mit größeren Argumenten, etwa Feldern, geschehen? Aus Modula-2 kennen wir *Wert- und Variablenparameter*. Wie können diese auf Assemblerebene realisiert werden?

call-by-value

Zunächst zum Fall des *call-by-value* oder Wertparameters: Hier wird einfach der *Wert* des Parameters übergeben.

call-by-reference

Der zweite Fall ist fast noch simpler, denn er erlaubt die Übergabe komplexer Strukturen durch die schlichte Übergabe der *Adresse*, an der sich das Objekt befindet. Dieses Verfahren nennt man *call-by-reference* oder Variablenparameter. Wir benötigen dafür noch einmal einen neuen Befehl:

Befehl	Argumente		Wirkung	Erläuterung
`la`Ⓟ	`label`		Lade die Adresse des Labels, nicht seinen Wert	load address
		:=	`lui Rd, upper` `ori Rd, $at, lower` `# upper = label DIV` s^{32} `# lower = label MOD` 2^{32}	

[8] [Pat, S. A-25] gibt eine ungünstige Reihenfolge an. Richtig ist die Reihenfolge 2-1-3. `$v0` und `$v1` werden dort nicht gesichert, weil es unüblich ist, dort Daten zu halten. Tatsächlich dürfen diese aber verändert werden.

Wesentlicher Unterschied zwischen beiden Verfahren ist, dass beim Variablenparameter der Parameter selbst vom callee verändert werden kann, was beim Wertparameter nicht möglich ist.

13.2.2 Prolog des Callees

(a) Schaffe Platz für den Stackframe:

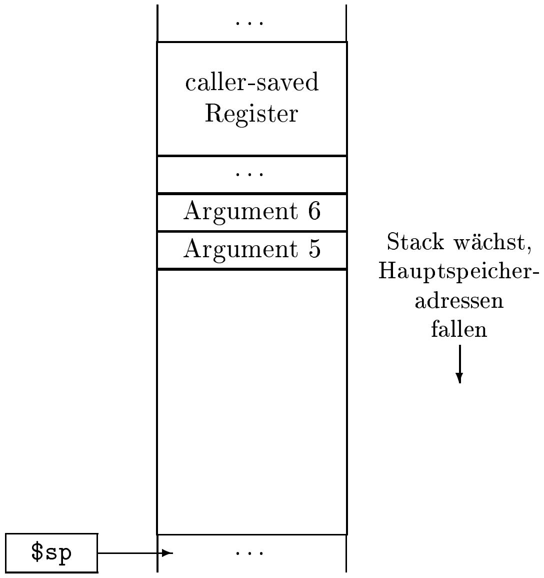
Unsere Prozedur-Konvention sieht einen sogenannten *Stackframe* vor, einen Bereich auf dem Stack, der in (fast) jeder Prozedur aufgebaut wird und der ein „Fenster" auf dem Stack bezeichnet, in dem Argumente und lokale Variablen des callees zu sehen sind. Für diesen Stackframe müssen wir zunächst Platz schaffen, indem wir den Stackpointer um die Größe des Stackframes verringern. Der Stackframe hat die Größe

Stackframe

$$(\text{Zahl der zu sichernden Register} + \text{Zahl der lokalen Variablen}) \times 4$$

Der Stackpointer ist übrigens das einzige Register, das der Callee nicht sichern muss, obwohl es in der Regel verändert wird. Dies liegt daran, dass der Stackpointer nach Abarbeitung des Callees wieder im Ursprungszustand sein sollte.

Der Stack sieht jetzt so aus:



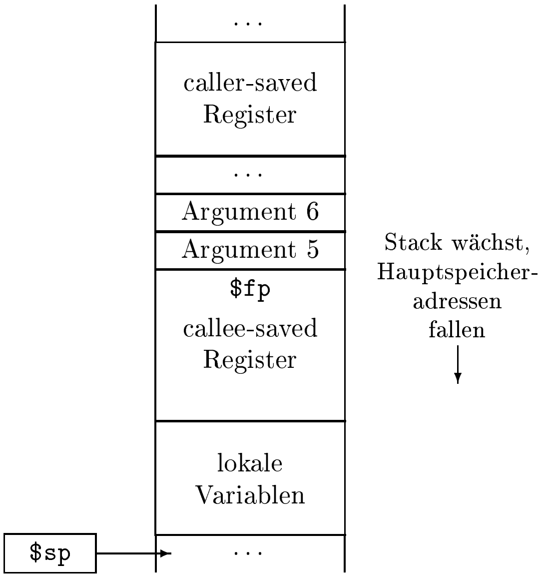
(b) Sichere alle callee-saved Register, die in der Prozedur verändert werden:

Verändert der Callee eines der Register `$fp`, `$ra` und `$s0`-`$s7`, so müssen diese zuvor gesichert werden. Dabei darf das Register `$ra` nicht vergessen werden, obwohl es nur indirekt durch den Befehl `jal` geändert wird!

Das Register `$fp` sollte als erstes gesichert werden, und zwar auf den Speicherplatz, auf den zu Prozedurbeginn der Stackpointer zeigte. Daran schließen sich die oben genannten Register in beliebiger Reihenfolge an.

Alle caller-saved Register, insbesondere `$a0`-`$a3`, dürfen verändert werden!

Ein Blick auf die Lage auf dem Stack:

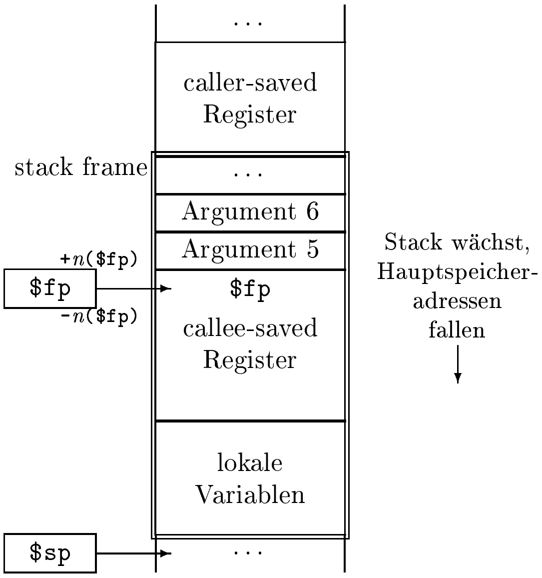


(c) Erstelle den Framepointer:

Framepointer

Die letzte Maßnahme innerhalb des callee-Prologs ist das Erstellen des *Framepointers*, einem besonderen Register (`$fp`), das den Wert des Stackpointers zu Beginn der Prozedur erhält. Da dieser Wert bereits in Schritt Nr. 1 verändert wurde, müssen wir ihn erst wieder durch Addition der Größe des Stackframe zum Stackpointer ermitteln. Das Ergebnis legen wir in `$fp` ab.

Der Stack sieht jetzt so aus:



Besondere Beachtung verdient der Umstand, dass der aktuelle Framepointer auf genau die Stelle auf dem Stack zeigt, an der der *vorherige* Framepointer liegt. So können wir auf die Argumente und lokalen Variablen der vorherigen Prozedur zugreifen. Ein effizientes, aber etwas fehleranfälliges, Verfahren!

13.2.3 Der Callee

Der Callee hat nun durch positive Indizes (z.B. `8($fp)`) die Möglichkeit auf den Wert der Argumente 5 ff. zuzugreifen und durch negative Indizes (z.B. `-12($fp)`) auf seine lokalen Variablen. Wobei bei letzteren natürlich die Werte der gesicherten Register keinesfalls überschrieben werden dürfen.

13.2.4 Epilog des Callees

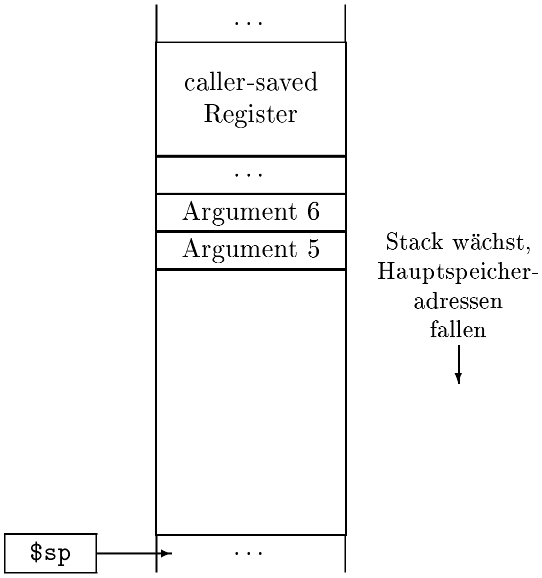
(a) Rückgabe des Funktionswertes:

Handelt es sich bei dem Callee um eine Funktion, so müssen wir das Funktionsergebnis im Register `$v0` (bzw. in `$v0` und `$v1`) abglegen. Natürlich bietet es sich an diese Register dann bereits bei der Berechnung des Funktionsergebnisses zu verwenden.

(b) Stelle die gesicherten Register wieder her:

Alle vom Callee gesicherten Register werden wieder hergestellt. Zweckmäßigerweise stellen wir den Framepointer als letztes Register wieder her.

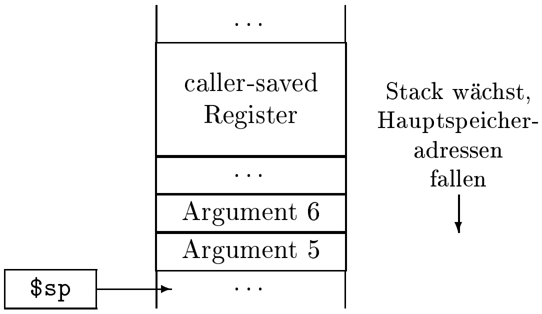
Der Stack sieht nun wieder so aus:



(c) Entferne den Stackframe:

Wir stellen den Stackpointer wieder her, indem wir die Größe des Stackframes zum Stackpointer addieren[9].

Der Stack sieht nun so aus wie zu Beginn der Prozedur:



(d) Springe zum Caller zurück:

```
jr $ra
```

13.2.5 Epilog des Callers

(a) Stelle die gesicherten Register wieder her:

Alle vom Caller gesicherten Register werden wieder hergestellt. Achtung: Eventuell über den Stack übergebene Argumente bei der Berechnung des Abstandes zum Stackpointer beachten!

[9][Pat, S. A-25]: '3. Pop the stack frame by subtracting[sic!] the frame size from `$sp`' ist falsch.

(b) Stelle den ursprünglichen Stackpointer wieder her:

Multipliziere die Zahl der Argumente und gesicherten Register mit vier und addiere sie zum Stackpointer.

Der Prozeduraufruf ist abgeschlossen.

13.2.6 Ein Beispiel zum Prozeduraufruf

Wir wollen uns nun einmal die Prozedur-Konvention an einem Beispiel verdeutlichen. Um uns auch eines kleineren Beispiels bedienen zu können, werden wir davon ausgehen, dass die Register `$a1` bis `$a3` nicht existent sind, da wir sonst die Parameterübergabe auf dem Stack nicht mit einem handlichen Beispiel demonstrieren könnten.

Ballot-Zahlen

Eines der Standard-Beispiele für Prozeduren –und ganz besonders für rekursive– sind die Fibonacci-Zahlen. Nur wenig komplizierter sind die *Ballot-Zahlen*, die gleich zwei Parameter haben und deshalb an dieser Stelle als Beispiel dienen sollen.

Die Ballot-Zahlen (engl. 'ballot', frz. 'ballotage': Stichwahl) geben die Zahl der Möglichkeiten an, nach einer schriftlichen Stichwahl zwischen zwei Kandidaten die Stimmen so auszuzählen, dass bei der Auszählung stets derselbe Kandidat vorne liegt. Als Parameter dienen die Zahl der Stimmen für jeden der Kandidaten.

Das Bildungsgesetz lautet $a_{n,m} = a_{n,m-1} + a_{n-1,m}$ wobei $a_{n,1} = -1, n > 1$, $a_{1,m} = 1, m > 1$ und $a_{1,1} = 0$.

Es ergibt sich dann folgende Wertetabelle:

		m					
		1	*2*	*3*	*4*	*4*	*6*
	1	0	−1	−1	−1	−1	−1
	2	1	0	−1	−2	−3	−4
	3	1	1	0	−2	−5	−9
n	*4*	1	2	2	0	−5	−14
	5	1	3	5	5	0	−14
	6	1	4	9	14	14	0

Beispiel 15: Beispiel zur Prozedur-Konvention: Ballot-Zahlen

```
# BALLOT.S                                          (c) 04.97
# Beispiele fuer die SPIM-Prozedurkonvention
# Ballot-Zahlen
# Zur Verdeutlichung der Konvention seien die Register $a1 bis $a3
# nicht existent.
.data
input1: .asciiz "\nBallot (n,m)\nn=0: Ende \n\tn= "
input2: .asciiz "\tm= "
output: .asciiz "\nBallot (n,m) = "
.text
main:
# Testumgebung fuer die Prozedur ballot
# Ruft ballot mit den vom Benutzer eingegebenen Werten
# auf bis dieser mit n oder m = 0 beendet.
        li      $v0, 4          # Eingaben anfordern
        la      $a0, input1
        syscall
        li      $v0, 5
        syscall
        beqz    $v0, exit
        move    $t0, $v0        # $t0 := n
        li      $v0, 4
```

```
        la      $a0, input2
        syscall
        li      $v0, 5
        syscall
        beqz    $v0, exit
        sub     $sp, $sp, 4     # Prolog (1)
                                # 1(a) Register sichern (hier: keine)
        sw      $v0, 4($sp)     # 1(b) Argumente uebergeben
        move    $a0, $t0        #
        jal     ballot          # 1(c) Prozedurstart
                                # Epilog (4)
                                # 4(a) Register wiederherstellen (keins)
        add     $sp, $sp, 4     # 4(b) Stackpointer wiederherstellen
        move    $t0, $v0        # $t0:=ballot(n,m)
        li      $v0, 4          # Ergebnis ausgeben
        la      $a0, output
        syscall
        li      $v0, 1
        move    $a0, $t0
        syscall
        j       main
exit:   li      $v0, 10
        syscall
                                #
ballot:
# Berechnet die Ballot-Zahl ballot (n,m)
# Argument 1 ($a0)  : n
# Argument 2 (Stack): m => 4($fp)
# Ergebnis in $v0
# Registerbelegung innerhalb der Prozedur:
# $a0: n
# $s0: m (besser waere $t0, hier nur zur Demonstration!)
# $t0: Zwischenergebnis
                                # Prolog (2)
        sub     $sp, $sp, 12    # 2(a) Stackframe erstellen
        sw      $fp, 12($sp)    # 2(b) Register sichern
        sw      $ra, 8($sp)
        sw      $s0, 4($sp)
        addi    $fp, $sp, 12    # 2(c) Framepointer erstellen
        lw      $s0, 4($fp)     # m laden
                                # Ballot-Zahlen berechnen...
        beq     $a0, 1, min1    # IF n = 1 THEN ballot = -1
        beq     $s0, 1, plus1   # ELSIF m = 1 THEN ballot = 1
                                # ELSE ballot (n,m) =
                                #   ballot (n, m-1) +
                                #   ballot (n-1, m) END;
                                # Aufruf ballot (n, m-1)
                                # Prolog (1)
        sub     $sp, $sp, 12    # 1(a) Register sichern
        sw      $a0, 12($sp)    # $a0 sichern, $t0 noch nicht benutzt
        sw      $s0, 8($sp)     # Rueckrechnung waere einfacher!
        sub     $s0, $s0, 1     # 1(b) Arg. ueberg., $a0 ist bereits ok,
        sw      $s0, 4($sp)     # (m-1) muss berechnet werden
        jal     ballot          # 1(c) Prozedurstart
                                # Epilog (4)
        lw      $a0, 12($sp)    # 4(a) Register wiederherstellen
        lw      $s0, 8($sp)
        addi    $sp, $sp, 12    # 4(b) Stackpointer wiederherstellen
        move    $t0, $v0        # Ergebnis sichern
        sub     $a0, $a0, 1     # Aufruf ballot (n-1, m)
                                # Prolog (1)
        sub     $sp, $sp, 8     # 1(a) Register sichern
        sw      $t0, 8($sp)     # $t0 sichern, $a0 nicht mehr benutzt
        sw      $s0, 4($sp)     # 1(b) Arg. ueberg., $a0 ist bereits ok
        jal     ballot          # 1(c) Prozedurstart
                                # Epilog (4)
        lw      $t0, 8($sp)     # 4(a) Register wiederherstellen
        addi    $sp, $sp, 8     # 4(b) Stackpointer wiederherstellen
        add     $v0, $v0, $t0   # Funktionsergebnis berechnen
tollab:                         # Epilog (3)
                                # 3(a) Funktionswert (liegt in $v0)
        lw      $fp, 12($sp)    # 3(b) Register wiederherstellen
        lw      $ra,  8($sp)
```

```
          lw      $s0,  4($sp)
          addi    $sp, $sp, 12    # 3(c) Stackframe entfernen
          jr      $ra             # 3(d) Rücksprung
  min1:   li      $v0, -1         # ballot = -1
          j       tollab
  plus1:  li      $v0, 1          # ballot = 1
          j       tollab
```

13.2.7 Zusammenfassung: Prozedur-Konvention beim SPIM

Am Ende dieses Abschnittes hier noch eine Zusammenfassung der Konvention:

1. Prolog des Callers:
   (a) Sichere alle caller-saved Register:
   Sichere den Inhalt der Register `$a0`-`$a3`, `$t0`-`$t9` sowie `$v0` und `$v1`. Der Callee darf ausschließlich diese Register verändern ohne ihren Inhalt wieder herzustellen.[10]
   (b) Übergebe die Argumente:
   Die ersten vier Argumente werden in den Register `$a0` bis `$a3` übergeben, alle weiteren werden auf dem Stack. Das fünfte Argument kommt *zuletzt* auf den Stack. Achtung: Parameterübergabe (Call-by-Value bzw. Call-by-Reference) beachten!
   (c) Starte die Prozedur (`jal`)
2. Prolog des Callees:
   (a) Schaffe Platz für den Stackframe:
   Subtrahiere die Größe des Stackframes vom Stackpointer.
   (b) Sichere alle callee-saved Register, die in der Prozedur verändert werden:
   Der Callee muss die Register `$fp`, `$ra` und `$s0`-`$s7` sichern, wenn sie innerhalb der Prozedur verändert werden. Achtung: das Register `$ra` wird durch den Befehl `jal` geändert!
   (c) Erstelle den Framepointer:
   Addiere die Größe des Stackframe zum Stackpointer und lege das Ergebnis in `$fp` ab.
3. Epilog des Callees:
   (a) Rückgabe des Funktionswertes:
   Handelt es sich bei dem Callee um eine Funktion, so muss das Funktionsergebnis in den Registern `$v0` und `$v1` abgelegt werden.
   (b) Stelle die gesicherten Register wieder her:
   Alle vom Callee gesicherten Register werden wieder hergestellt. Achtung: den Framepointer als letztes Register wieder herstellen!
   (c) Entferne den Stackframe:
   Addiere[11] die Größe des stack frames vom Stackpointer.
   (d) Springe zum Caller zurück:
   `jr $ra`

[10] [Pat, A-25] gibt eine ungünstige Reihenfolge an. Richtig: 2-1-3. `$v0` und `$v1` werden dort nicht gesichert, weil es unüblich ist, dort Daten zu halten. Tatsächlich dürfen diese aber verändert werden.

[11] [Pat, S. A-25]: '3. Pop the stack frame by subtracting[sic!] the frame size from `$sp`' ist falsch.

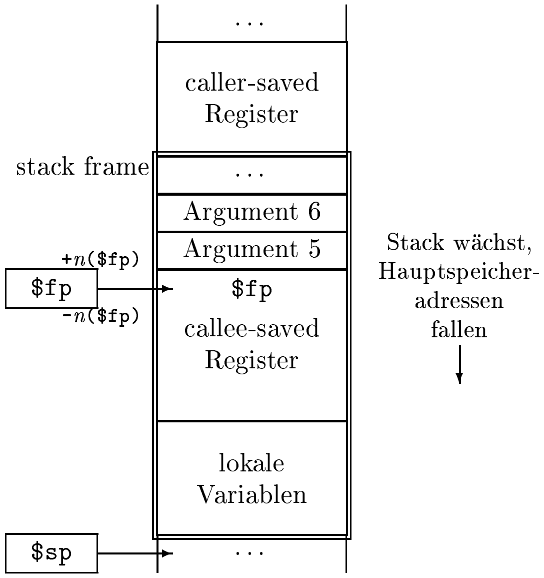
4. Epilog des Callers:

   (a) Stelle die gesicherten Register wieder her:

   Alle vom Caller gesicherten Register werden wieder hergestellt. Achtung: Evtl. über den Stack übergebene Argumente bei der Berechnung des Abstandes zum Stackpointer beachten!

   (b) Stelle den ursprünglichen Stackpointer wieder her:

   Multipliziere die Zahl der Argumente und gesicherten Register mit vier und addiere sie zum Stackpointer.



14 Gleitkommazahlen

Wenn wir uns bisher mit Zahlen beschäftigt haben, dann handelte es sich stets um ganze Zahlen. In der realen Welt nehmen diese jedoch nur einen kleinen Bereich ein, so dass in vielen Anwendungen die Arbeit mit rationalen Zahlen notwendig ist. Computer arbeiten nur mit einer kleinen Teilmenge der rationalen Zahlen, ebenso wie sie nur mit einer winzigen Teilmenge der ganzen Zahlen arbeiten (beim **MIPS** mit denjenigen ganzen Zahlen zwischen -2^{31} und $+2^{31}-1$). Das Pendant der rationalen Zahlen im Computerbereich heißt *Gleitkommazahlen oder Floating-Point-Zahlen*. Die Repräsentation von Gleitkommazahlen ist [Pat, S. 226f] zu entnehmen, der gleiche Abschnitt 4.8 beschäftigt sich auch mit Rechenverfahren für Gleitkommazahlen.

Gleitkommazahlen

Der **MIPS**-Prozessor kann selber keine Gleitkommazahlen verarbeiten. Er bedient sich dazu eines *(mathematischen) Koprozessors* für Gleitkommazahlen, er wird auch *FPU (floating point unit)* genannt. Der **MIPS**-Prozessor kann über mehrere Koprozessoren für verschiede Aufgaben verfügen. Im SPIM-Simulator werden zwei von ihnen simuliert:

Koprozessor
floating point unit

1. Koprozessor 0:

   Er wird bei der Behandlung von Unterbrechungen und Ausnahmen verwendet. Im Original-Prozessor wird dieser Prozessor auch bei der Hauptspeicherverwaltung verwendet. Diese wird jedoch vom SPIM nicht simuliert. Er wird in Abschnitt 15 auf Seite 66 weiter behandelt.

2. Koprozessor 1:

   Er ist der mathematische Koprozessor und zuständig für Berechnungen mit Gleitkommazahlen.

Der mathematische Koprozessor verfügt wie der Hauptprozessor über 32 Register von je 32 Bit Breite. Die Register sind von `$0` bis `$31` durchnummeriert und haben die Namen `$f0` bis `$f31`. Es handelt sich um echte general purpose register, alle Register sind also beschreibbar und für jeden beliebigen Zweck einsetzbar. Mit zwei Einschränkungen: Rechnet man mit doppelt genauen Gleitkommazahlen, so werden hierfür 64 Bit benötigt, die wir durch Zusammenfassen je zweier Register erhalten. Zusammengefasst werden ein Register mit einer geraden Registernummer und das folgende Register, also `$f0` und `$f1`, `$f2` und `$f3` usw.

Die zweite Einschränkung betrifft die Register `$f0` und `$f1` (im Folgenden werden wir für dies „Doppelregister" `$f0/1` schreiben), in die das Betriebssystem die Benutzereingaben schreibt (Betriebssystemfunktionen 6 und 7) und die Register `$f12/13`, aus denen die Ausgaben gelesen werden (Betriebssystemfunktionen 2 und 3). Vergleiche hierzu die Abbildung 6 auf Seite 27 bzw. Abschnitt A.5 auf Seite 78!

> Leider funktioniert das Registerfenster nicht unter allen Windows-Versionen, es werden lediglich die Werte 0.000... bzw. -0.000... angezeigt. Dieses Problem betrifft jedoch nur die *Anzeige* der Werte!

14.1 Datenhaltung III: Gleitkommazahlen

Gleitkommazahlen werden wie alle anderen Daten auch im Datensegment gehalten. Für das Ablegen von Gleitkommazahlen stehen zwei Direktiven zur Verfügung:

- `.float`

  `.float`

  Die Direktive `.float` legt die folgenden Werte als 32-Bit-Gleitkommazahlen ab. Man spricht auch von Gleitkommazahlen *einfacher Genauigkeit* oder *single precision*.

  einfache Genauigkeit

  Die Werte können in nicht normalisierter Form in Exponentialschreibweise eingetragen werden: `123.45e67`, `0.0003e-9`. Das `e` muss klein geschrieben sein. Auf die Exponentialschreibweise kann auch verzichtet werden, dann muss allerdings ein Dezimalpunkt vorhanden sein: `1234567.` und `0.09999` sind erlaubt, `1234567` jedoch nicht.

- `.double`

  `.double`

  Die Direktive `.double` legt die folgenden Werte als 64-Bit-Gleitkommazahlen ab. Man spricht auch von Gleitkommazahlen *doppelter Genauigkeit* oder *double precision*.

  doppelte Genauigkeit

14.2 Transferbefehle

Daten können zwischen einem der Koprozessoren und dem Hauptprozessor oder dem Speicher hin- und herbewegt werden. Hierfür dienen die folgenden Befehle.

Für die Notation der Koprozessorregister verwenden wir die Notation aus Abbildung 12 auf dieser Seite.

Abbildung 12: Neue Argumentarten

Notation	Erläuterung	Beispiel
`Cd, Cs`	Koprozessor-Registeroperanden, d: destination (Ziel), s: source (Quelle) Achtung: Register*namen* wie z.B. `$f10` sind hier nicht erlaubt!	`$1`, `$3`
`Fd, Fs,` `Fs1, Fs2`	Registeroperanden des *mathematischen* Koprozessors	`$f0`, `$f13`

14.2.1 Ladebefehle

Für jeden Koprozessor gibt es einen eigenen Ladebefehl `lwc`$\langle z \rangle$, wobei für $\langle z \rangle$ eine Zahl zwischen 0 und 3 zu wählen ist. Der mathematische Koprozessor hat die Nummer 1.

Befehl	Argumente		Wirkung	Erläuterung
`lwc`$\langle z \rangle$	`Cd, Adr`		Cd := Mem[Adr]; $\langle z \rangle$: [0..3]	load word coprocessor
`l.d`$^{(P)}$	`Fd, Adr`		Fd := Mem[Adr]; Fd+1 := Mem[Adr+4]	Load floating-point double
		:=	`lui $at, Adr` `lwc1 Fd, 0($at)` `lui $at, Adr` `lwc1 Fd+1, 4($at)`	
`l.s`$^{(P)}$	`Fd, Adr`		Fd := Mem[Adr]	Load floating-point single
		:=	`lui $at, Adr` `lwc1 Fd, 0($at)`	

Befehle, die ausschließlich der Verarbeitung von Gleitkommazahlen dienen, sind an den Punkten im Befehlsnamen zu erkennen. Die meisten Befehle gibt es in einer Version für einfach genaue Zahlen und in einer für doppelt genaue Zahlen. Die beiden Versionen werden in der Regel durch `.s` (single) und `.d` (double) auseinander gehalten.

Wie eingangs erwähnt, werden doppelt genaue Zahlen in zwei aufeinanderfolgenden Registern abgelegt, beginnend mit einer geraden Registernummer. Die Befehle für doppelt genaue Zahlen arbeiten zwar auch mit ungeraden Registernummern, jedoch basieren die Anzeigen des SPIM auf der Annahme, dass die doppelt genauen Zahlen in geraden Registern beginnen.

14.2.2 Speicherbefehle

Die Speicherbefehle entsprechen weitgehend den bereits bekannten Ladebefehlen:

Befehl	Argumente	Wirkung	Erläuterung
`swc`$\langle z\rangle$	`Cs, Adr`	Mem[Adr] := Cs; $\langle z\rangle$: [0..3]	store word coprocessor
`s.d`[P]	`Fs, Adr`	Mem[Adr] := Fs; Mem[Adr+4] := Fs+1	Store floating-point double
	:=	`lui $at, Adr` `swc1 Fd, 0($at)` `lui $at, Adr` `swc1 Fd+1, 4($at)`	
`s.s`[P]	`Fs, Adr`	Mem[Adr] := Fs	Store floating-point single
	:=	`lui $at, Adr` `swc1 Fd, 0($at)`	

14.2.3 Register-Transfer-Befehle

Mit den folgenden Befehlen können Daten zwischen dem Haupt- und dem Koprozessor hin- und herbewegt werden:

Befehl	Argumente	Wirkung	Erläuterung
`mfc`$\langle z\rangle$	`Rd, Cs`	Rd := Cs; $\langle z\rangle$: [0..3]	Move from coprocessor
`mfc1.d`[P]	`Rd, Cs`	Rd := Cs; Rd+1 := Cs+1	Move double from coprocessor 1
	:=	`mfc1 Rd, Cs` `mfc1 Rd+1, Cs+1`	
`mtc`$\langle z\rangle$	`Rs, Cd`	Cd := Rs; $\langle z\rangle$: [0..3]	Move to coprocessor

> *Achtung:* Der Befehl `mtc`$\langle z\rangle$funktioniert in der Windows-Version nicht! Zum Laden eines unmittelbaren Wertes müssen wir diesen also zuvor im Speicher ablegen und dann von dort laden.

Innerhalb des mathematischen Koprozessors dienen die folgenden beiden Befehle zum Verschieben des Inhalts von Registern:

Befehl	Argumente	Wirkung	Erläuterung
`mov.d`	`Fd, Fs`	Fd/Fd+1 := Fs/Fs+1	Move floating-point double
`mov.s`	`Fd, Fs`	Fd := Fs	Move floating-point single

Häufig ist es notwendig ganze Zahlen in Gleitkommazahlen, doppelt genaue Zahlen in einfach genaue Zahlen o.ä., umzuwandeln. Hierfür stehen die folgenden Konvertierungsbefehle zur Verfügung:

Befehl	Argumente	Wirkung	Erläuterung
`cvt.d.s`	`Fd, Fs`	Fd/Fd+1 := Fs	Convert single to double
`cvt.d.w`	`Fd, Fs`	Fd/Fd+1 := Fs	Convert integer to double
`cvt.s.d`	`Fd, Fs`	Fd := Fs/Fs+1	Convert double to single
`cvt.s.w`	`Fd, Fs`	Fd := Fs	Convert integer to single
`cvt.w.d`	`Fd, Fs`	Fd := Fs/Fs+1	Convert double to integer
`cvt.w.s`	`Fd, Fs`	Fd := Fs/Fs+1	Convert single to integer

14.3 Arithmetische Befehle für Gleitkommazahlen

Die Additionsbefehle für Gleitkommazahlen entsprechen im Wesentlichen denen für Ganzzahlen, nur dass es keine Versionen ohne Überlaufbehandlung gibt. Dafür existieren sie in jeweils einer Variante für doppelt und einfach genaue Gleitkommazahlen.

Addition und Subtraktion:

Befehl	Argumente	Wirkung	Erläuterung
`add.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 + Fs2/Fs2+1	Floating-point addition double
`add.s`	`Fd, Fs1, Fs2`	Fd := Fs1 + Fs2	Floating-point addition single
`sub.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 - Fs2/Fs2+1	Floating-point subtract double
`sub.s`	`Fd, Fs1, Fs2`	Fd := Fs1 - Fs2	Floating-point subtract single

Multiplikation und Division:

Befehl	Argumente	Wirkung	Erläuterung
`mul.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 * Fs2/Fs2+1	Floating-point multiply double
`mul.s`	`Fd, Fs1, Fs2`	Fd := Fs1 * Fs2	Floating-point multiply single
`div.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 / Fs2/Fs2+1	Floating-point divide double
`div.s`	`Fd, Fs1, Fs2`	Fd := Fs1 / Fs2	Floating-point divide single

sonstige arithmetische Befehle:

Befehl	Argumente	Wirkung	Erläuterung
`abs.d`	`Fd, Fs`	Fd/Fd+1 := ABS (Fs/Fs+1)	Floating-point absolute value double
`abs.s`	`Fd, Fs`	Fd := ABS (Fs)	Floating-point absolute value single

Befehl	Argumente	Wirkung	Erläuterung
`neg.d`	`Fd, Fs`	Fd/Fd+1 := - Fs/Fs+1	Floating-point negate double
`neg.s`	`Fd, Fs`	Fd := - Fs	Floating-point negate single

14.4 Vergleichsbefehle für Gleitkommazahlen

condition-flag

Wir haben sechs verschiedene Vergleichsbefehle zur Verfügung, die jeweils zwei Zahlen vergleichen und als Ergebnis das *Bedingungs-Bit* (*condition-flag*) des mathematischen Koprozessors auf 1 (=wahr) oder auf 0 (=falsch) setzen. Dieses condition-flag kann von der CPU als Sprungkriterium verwendet werden. Die beiden folgenden Befehle verzweigen die Programmausführung in Abhängigkeit von dem condition-flag des Koprozessors:

Befehl	Argumente	Wirkung	Erläuterung
bc$\langle z \rangle$t		Sprung, wenn condition-flag des Koprozessors $\langle z \rangle$ wahr (1) ist	Branch coprocessor $\langle z \rangle$ true
bc$\langle z \rangle$f		Sprung, wenn condition-flag des Koprozessors $\langle z \rangle$ falsch (0) ist	Branch coprocessor $\langle z \rangle$ false

Die Vergleichsbefehle für Gleitkommazahlen sind:

Befehl	Argumente	Wirkung	Erläuterung
`c.eq.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 = Fs2/Fs2+1	Compare equal double
`c.eq.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 = Fs2	Compare equal single
`c.le.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 $\leq$ Fs2/Fs2+1	Compare less than equal double
`c.le.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 $\leq$ Fs2	Compare less than equal single
`c.lt.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 $<$ Fs2/Fs2+1	Compare less than double
`c.lt.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 $<$ Fs2	Compare less than single

14.5 Ein Beispiel zu Gleitkommazahlen

Das Hornerschema dient u.a. zur Berechnung von Polynomen n-ter Ordnung. Durch fortgesetztes Ausklammern des x ergibt sich:

$$\begin{aligned} & a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 \\ = \; & ((((((a_n)\, x) + a_{n-1})\, x + a_{n-2})\, x + \ldots)\, x + a_1)\, x + a_0 \end{aligned}$$

Die rechte Seite des Terms lässt sich einfach iterativ berechnen.

Beispiel 16: Das Hornerschema mit doppelt genauen Gleitkommazahlen

```
# HORNER.S                                                  (c) 06/97
# Implementation des Hornerschemas fuer Gleitkommazahlen doppelter
# Genauigkeit.
#
.data
Koeffs: .double 1.,     5.,     -2.     # ein Testfeld
last:   .double -24.
.text
#
# Hauptprogramm main
# Nach Eingabe des x-Wertes wird der Wert des Polynoms zwischen
# Koeffs und last an der Stelle x berechnet.
#
__start:
main:  la      $a0, Koeffs      # Initialisierung
       la      $a1, last
       li      $v0, 7           # Read_double
       syscall
       jal     poly
       li      $v0, 3           # Print_double
       syscall
       li      $v0, 10          # Exit
       syscall
# Funktion poly
# Berechnet den Wert eines Polynoms f an einer gegeben Stelle x
# Eingabe:
# $a0:    Adresse des hoechstrangigen Koeffizienten a_n des Polynoms.
#         Die Koeffizienten sind hintereinander mit sinkender Wertigkeit
#         als doppelt genaue Gleitkommazahlen abgelegt.
# $a1:    Adresse des niedrigstranigen Koeffizienten a_0 (Konstante)
# $f0/1:  Stelle x, an der das Polynom berechnet werden soll.
# $f12/13:Ergebnisakkumulation (akk)
# zusaetzlich verwendetes Register:
# $f10/11:temporaer a_n (tmp); wird gesichert!
.data
dbl0:   .double 0.
.text
poly:   addiu   $sp, $sp, -8    # Platz fuer $f10/11
        s.d     $f10, 8($sp)    # $f10/11 sichern
        l.d     $f12, dbl0      # Initialisierung, Windows-Version kann
                                # kein mtc1!
                                # besser waere: Zeile 35/36 loeschen,
                                # Zeile 40 ff:
                                # li      $t0, 0
                                # mtc1    $t0, $12
                                # cvt.d.w $f12, $f12
                                # oder einfach durch
                                # sub.d $f12, $f12, $f12
        bgt     $a0, $a1, exit  # Fehler in der Adressierung
repeat: l.d     $f10, ($a0)     # tmp := a_n
        mul.d   $f12, $f12, $f0 # akk := akk * x
        add.d   $f12, $f12, $f10# akk := akk + a_n
        addiu   $a0, $a0, 8     # n := n + 1; 64-Bit-Zahlen!
        ble     $a0, $a1, repeat# solange nicht am Ende der Koeffizienten
exit:   l.d     $f10, 8($sp)    # $f10/11 wiederherstellen
        addiu   $sp, $sp, 8     # $sp wiederherstellen
        jr      $ra             # Bye-bye
```

15 Unterbrechnungen und Ausnahmen

In Abschnitt 6.2 auf Seite 22 haben wir bereits von Unterbrechungen und Ausnahmen gehört. Sie sollen nun noch Gegenstand eines ganzen Abschnittes in diesem Tutorial sein.

Es ist jedoch anzumerken, dass SPIM hier deutliche Schwächen aufweist. So werden etliche Befehle, die wir benötigen, nicht ganz korrekt simuliert. Um unnötigen Frust zu vermeiden, sollten wir uns daher auf eine theoretische Behandlung dieses interessanten Teils der Assemblerprogrammierung beschränken. Mir bekannte Fehler des Simulators sind in Fußnoten erläutert.

Dieser Abschnitt soll lediglich ein Leitfaden sein, keinesfalls ist das Thema Uneterbrechnungen und Ausnahmen hier erschöpfend behandelt.

15.1 Zu den Begriffen „Unterbrechung" und „Ausnahme"

Interrupt, Trap, Ausnahme, exception usw., all dies sind Begriffe, die fast jede(r) Informatikstudent(in) schon mal gehört und zumindest eine ungefähre Vorstellung von ihrer Bedeutung hat. Bevor wir uns mit ihnen beschäftigen, wollen wir versuchen das Begriffswirrwarr etwas zu lichten und diese Begriffe im Sinne des **MIPS** zu definieren, denn nicht überall sind die Begriffe gleich definiert.

Interrupt, Unterbrechung

Ein *Interrupt* oder eine *Unterbrechung* ist ein Ereignis, das *asynchron* zum Programmablauf eintritt, also nicht in direkter Abhängigkeit zu bestimmten Befehlen steht, und eine (sofortige) Behandlung durch das Betriebssystem notwendig macht. Typische Beispiele sind Ein- und Ausgabegeräte, z.B. die Tastatur. Unterbrechungen sind nicht reproduzierbar, sie lauern immer und überall während der gesamten Programmausführung.

exception, Ausnahme, Trap

Eine *exception* oder eine *(Ausnahme)*, gelegentlich auch *Trap*, ist ein Ereignis, das *synchron* zum Programmablauf eintritt, also in Zusammenhang mit bestimmten Befehlen, und eine Behandlung durch das Betriebssystem notwendig macht. Typisches Beispiel sind arithmetische Fehler, wie z.B. die Division durch Null, Überläufe etc. Ebenfalls zu den Ausnahmen zählen ungültige Instruktionen, wie sie beim Sprung ins Datensegment auftreten können, oder der Aufruf des Betriebssystems durch das Benutzerprogramm, beim **MIPS** mit dem bereits bekannten Befehl `syscall`.[12] Bei gleichem Ausgangszustand des Rechnersystems sind Ausnahmen im Gegensatz zu Unterbrechungen reproduzierbar, d.h. sie treten immer wieder an derselben Stelle auf.

Hardwarefehler können sowohl asynchrone Unterbrechungen verursachen als auch synchrone Ausnahmen, je nachdem wo und wie sie auftreten.

15.2 Die Behandlung von Unterbrechungen und Ausnahmen

Da die Stellen, an denen Ausnahmen auftreten können, bekannt sind, könnte man Ausnahmen vollständig durch das Benutzerprogramm abarbeiten lassen, etwa durch vorherige Tests der Operanden oder durch Überprüfung des Ergebnisses. Da der Benutzer aber fast immer korrekte Ergebnisse haben will, würde bei fast jedem arithmetischen Befehl Code zur Überprüfung des Ergebnisses stehen, eine ineffziente Lösung, zumal sich gerade die arithmetischen Ausnahmen sehr gut auf Schaltkreisebene erkennen lassen.

[12] Im SPIM werden die Betriebssystemaufrufe allerdings nicht simuliert, sondern vom Simulator direkt verarbeitet.

Folglich behandelt man Unterbrechungen und Ausnahmen gemeinsam durch das Betriebssystem. Beim Auftreten eines solchen Ereignisses wird nach Abarbeitung des aktuellen Befehls ein Sprung an eine von der CPU-Hardware festgelegte Stelle erfolgen, beim SPIM nach 0x8000 0080. Dort liegt eine sogenannte *Unterbrechungsbehandlungsroutine* oder *Interrupthandler*, die die Unterbrechung bzw. die Ausnahme behandelt. Hierfür stehen im Koprozessor 0 einige Register zur Verfügung, in die die CPU Informationen über den Grund der Unterbrechung oder der Ausnahme schreibt. Von den zahlreichen Registern des Koprozessors 0 werden die vier in Abbildung 13 auf dieser Seite vom SPIM simuliert.

Unterbrechungs-behandlungsroutine, Interrupthandler

Abbildung 13: Register des Koprozessors 0

Register-name	Register-nummer	Inhalt
BadVAddr	8	Hauptspeicheradresse, auf welche zugegriffen werden sollte. Dieses Register ist zur Verwaltung des virtuellen Speichersystems notwendig.
Status	12	Unterbrechungsmaske und interrupt bits
Cause	13	Ausnahmetyp und wartende (noch nicht bearbeitete) Unterbrechungen
EPC	14	Adresse des Befehls, der die Ausnahme auslöste.

Auf den Inhalt dieser Register kann mit denen aus dem Abschnitt 14 bekannten Register-Transfer- und Ladebefehlen, diesmal mit der Koprozessornummer 0, zugegriffen werden: `lwc0`, `mfc0`, `mtc0` und `swc0`.

15.2.1 Das Statusregister

Im Statusregister ist in den Bits 15 bis 8 eine Interruptmaske abgelegt. Ist ein Bit 1, so erlaubt es auf dieser Ebene Unterbrechungen. Fünf Hardware- und drei Softwareunterbrechungen sind möglich.

Die Bits 5 bis 0 (siehe Abbildung 14 auf dieser Seite) enthalten Informationen über den Status des Systems. Die sechs Bits sind in drei Zweiergruppen eingeteilt, die für das aktuelle Programm, das vorangegangene und vorvorangegangene Programm enthalten. Das Interrupt-enable-Bit ist 1, wenn Unterbrechungen zugelassen sind, 0 sonst. Das Kernel/User-Bit ist 1, wenn das Programm unter Kernel-Berechtigung lief.

Abbildung 14: Das Statusregister

Bit	Bedeutung	
0	Interrupt enable	aktueller
1	Kernel/User	Prozess
2	Interrupt enable	vorheriger
3	Kernel/User	Prozess
4	Interrupt enable	vorvorheriger
5	Kernel/User	Prozess
6-7	—	
8-15	Interruptmaske	
16-31	—	

15.2.2 Das Cause-Register

Pending Interrupts
exception Code

Vom Cause-Register werden vom SPIM die Bits 15 bis 8[13], die die noch nicht bearbeiteten Unterbrechungen (*Pending Interrupts*), und die Bits 5 bis 2, die den Grund der Unterbrechung (*exception Code*) enthalten, simuliert. Die Ausnahmecodes sind der Abbildung 15 auf dieser Seite zu entnehmen.

Abbildung 15: Ausnahmecodes (exception codes) (Bit 5 bis 2 des Cause-Registers)

Code	Name	Erläuterung
0	INT	Externe Unterbrechung
4	ADDRL	Adressfehler (Ladebefehl, auch beim Laden des Befehls!)
5	ADDRS	Adressfehler (Speicherbefehl)
6	IBUS	Busfehler beim Laden des Befehls
7	DBUS	Busfehler beim Laden oder Speichern von Daten
8	SYSCALL	Betriebssystemaufruf durch den Befehl `syscall`
9	BKPT	Breakpoint
10	RI	Verwendung einer priviligierten Instruktion (reserved instruction)
12	OVF	arithmetischer Überlauf (Overflow)

Für die nicht bearbeiteten Unterbrechungen stehen fünf Hardware-Level und drei Software-Level zur Verfügung. Die Bits 15 bis 8 entsprechen den Bits 15 bis 8 des Status-Registers.

15.2.3 Die Unterbrechungsbehandlungsroutine

Die Unterbrechungsbehandlungsroutine soll hier nicht im Detail beschrieben, sondern nur in ihren Grundfunktionen erläutert werden.

Kernelsegment

`.ktext`-Direktive

Die Ausführung der Unterbrechungsbehandlungsroutine muss unter besondereren Betriebssystemrechten geschehen, deshalb wird sie im *Kernelsegment* abgelegt, einem Segment, in dem der Betriebssystemkern untergebracht wird. Um Befehle ins Kernelsegment zu schreiben steht uns die `.ktext`-*Direktive* zur Verfügung. Ihr Syntax entspricht dem der bereits bekannten `.text`-Direktive.

Da bei Unterbrechungen und Ausnahmen zur Adresse 0x8000 0080 gesprungen wird, muss dafür Sorge getragen werden, dass die Unterbrechungsbehandlungsroutine an dieser Stelle liegt. Wir müssen die `.ktext`-Direktive also mit Hilfe des optionalen Paramaters die Stelle angeben, an der unsere Routine liegen soll: `.ktext 0x8000 0080`.

`.kdata`-Direktive

Analog zur `.data`-Direktive gibt es auch eine `.kdata`-*Direktive*, die Daten im Kerneldatensegment ablegt.

Im Kernelsegment stehen die für uns bisher verbotenen CPU-Register `$k0` und `$k1` zur Verfügung. Andere Register dürfen nur dann verwendet werden, wenn sie zuvor gesichert werden!

Aus dem Statusregister kann nun ermittelt werden, ob es sich um eine Unterbrechung handelt, dann ist es $> 0x44$ oder eine Ausnahme ($< 0x44$). Entsprechend kann nun durch Fallunterscheidungen ermittelt werden, welche Unterbrechung oder welche Ausnahme aufgetreten ist und diese dann bearbeitet werden. Schlimmstenfalls muss das System auch angehalten werden, bzw. der Prozess gelöscht werden.

[13] [Pat, S. A-32, Fig. A.13] gibt falsch Bits 15 bis *10* an!

Soll die Programmausführung nach der Unterbrechungsbehandlung fortgesetzt werden, so muss zunächst die korrekte Rücksprungadresse ermittelt werden. Da die Adresse des Befehls, an der die Ausnahme auftrat, im Register `EPC` steht, kann diese einfach um vier erhöht und dann mittels des Befehls `jr` angesprungen werden.

Zuvor muss jedoch noch das Statusregister wieder hergestellt werden, wofür es einen eigenen Befehl gibt:

Befehl	Argumente	Wirkung	Erläuterung
`rfe`		Stellt das Statusregister wieder her.	return from exception

Zu Testzwecken kann der folgende Befehl `break`[14] verwendet werden, den wir bereits in einigen Pseudobefehlen kennen gelernt haben:

Befehl	Argumente	Wirkung	Erläuterung
`break`	`Imm`	Verursacht die Ausnahme `Imm`	break

[14] Der Befehl arbeitet leider fehlerhaft.

A Zusammenfassende Tabellen

A.1 Befehlsübersicht

Ladebefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`lw`	`Rd, Adr`	Rd:=MEM[Adr]	Load word	18
`lb`	`Rd, Adr`	Rd:=MEM[Adr]	Load byte	19
`lbu`	`Rd, Adr`	Rd:=MEM[Adr]	Load unsigned byte	19
`lh`	`Rd, Adr`	Rd:=MEM[Adr]	Load halfword	19
`lhu`	`Rd, Adr`	Rd:=MEM[Adr]	Load unsigned halfword	19
`ld`^{P}	`Rd, Adr`	Lädt das Doppelwort an der Stelle `Adr` in die Register `Rd` und `Rd`+1	Load double-word	19
`ulw`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load word	19
`ulh`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load halfword	19
`ulhu`	`Rd, Adr`	Rd:=MEM[Adr]	unaligned Load unsigned halfword	19
`lwr`	`Rd, Adr`	Rd:=MEM[Adr] DIV 2^{16}	Load word right	19
`lwl`	`Rd, Adr`	Rd:=MEM[Adr] MOD 2^{16}	Load word left	19

Speicherbefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`sw`	`Rs, Adr`	MEM[Adr]:=Rs	store word	20
`sb`	`Rs, Adr`	MEM[Adr]:=Rs MOD 256	store byte	20
`sh`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store halfword	20
`sd`^{P}	`Rs, Adr`	MEM[Adr]:=Rs + 2^{16} Rs+1	store double-word	20
`swl`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store word left	20
`swr`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	store word right	20
`ush`	`Rs, Adr`	MEM[Adr]:=Rs MOD 2^{16}	unaligned store halfword	20
`usw`	`Rs, Adr`	MEM[Adr]:=Rs	unaligned store word	20

Register-Transfer-Befehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`move`^{P}	`Rd, Rs`	Rd:=Rs	move	20
`li`^{P}	`Rd, Imm`	Rd:=Imm	load immediate	20
`lui`	`Rs, Imm`	Rd:=Imm 2^{16}	load upper immediate	20
`la`	`label`	Lade die Adresse des Labels, nicht seinen Wert	load address	52
`mfhi`	`Rd`	Rd:=hi	move from `hi`	21
`mflo`[15]	`Rd`	Rd:=lo	move from `lo`	21
`mthi`	`Rs`	hi:=Rs	move to `hi`	21
`mtlo`	`Rs`	lo:=Rs	move to `lo`	21

Addition und Subtraktion

Befehl	Argumente	Wirkung	Erläuterung	S.
`add`	`Rd, Rs1, Rs2`	Rd := Rs1 + Rs2	addition (with overflow)	22
`addi`	`Rd, Rs, Imm`	Rd := Rs + Imm	addition immediate (with overflow)	22
`sub`	`Rd, Rs1, Rs2`	Rd := Rs1 - Rs2	subtract (with overflow)	22
`addu`	`Rd, Rs1, Rs2`	Rd := Rs1 + Rs2	addition (without overflow)	22
`addiu`	`Rd, Rs, Imm`	Rd := Rs + Imm	addition immediate (without overflow)	22
`subu`	`Rd, Rs1, Rs2`	Rd := Rs1 - Rs2	subtract (without overflow)	22

Multiplikation und Division

Befehl	Argumente	Wirkung	Erläuterung	S.
`div`	`Rd, Rs`	hi:=Rd MOD Rs; lo:=Rd DIV Rs	Divide (with overflow)	23
`divu`	`Rd, Rs`	hi:=Rd MOD Rs; lo:=Rd DIV Rs	Divide (without overflow)	23
`mult`	`Rd, Rs`	hi:=Rd $\times$ Rs DIV 2^{16}; lo:=Rd $\times$ Rs MOD 2^{16}	multiply	23
`multu`	`Rd, Rs`	hi:=Rd $\times$ Rs DIV 2^{16}; lo:=Rd $\times$ Rs MOD 2^{16}	Unsigned multiply	23
`mul`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ Rs2	Multiply (without overflow)	24
`mulo`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ Rs2	Multiply (with overflow)	24
`mulou`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ Rs2	Unsigned multiply (with overflow)	24
`div`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 DIV Rs2	Divide (with overflow)	24
`divu`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 DIV Rs2	Divide (without overflow)	24

sonstige arithmetische Befehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`abs`^{P}	`Rd, Rs`	Rd:=ABS (Rs)	Absolute value	25
`neg`^{P}	`Rd, Rs`	Rd:=–Rs	Negate value (with overflow)	25
`negu`^{P}	`Rd, Rs`	Rd:=–Rs	Negate value (without overflow)	25
`rem`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 MOD Rs2	Remainder	25
`remu`^{P}	`Rd, Rs1, Rs2`	Rd:=Rs1 MOD Rs2	Unsigned remainder	25

Betriebssystemfunktion

Befehl	Argumente	Wirkung	Erläuterung	S.
`syscall`		führt die Systemfunktion in `$v0` aus	vgl. Abbildung 6 auf Seite 27	26

elementare logische Befehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`and`	`Rd, Rs1, Rs2`	Rd := Rs1 $\wedge$ Rs2	and	28
`andi`	`Rd, Rs, Imm`	Rd := Rs $\wedge$ Imm	and immediate	28
`nor`	`Rd, Rs1, Rs2`	Rd := $\overline{\text{Rs1} \vee \text{Rs2}}$	nor	28
`or`	`Rd, Rs1, Rs2`	Rd := Rs1 $\vee$ Rs2	or	28
`ori`	`Rd, Rs, Imm`	Rd := Rs $\vee$ Imm	or immediate	28
`xor`	`Rd, Rs1, Rs2`	Rd := Rs1 $\oplus$ Rs2	exclusive or	28
`xori`	`Rd, Rs, Imm`	Rd := Rs $\oplus$ Imm	exclusive or immediate	28
`not`Ⓟ	`Rd, Rs`	Rd := $\overline{\text{Rs}}$	not	28

Rotations- und Schiebebefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`rol`Ⓟ	`Rd, Rs1, Rs2`	Rd := Rs1 um Rs2 Stellen nach links rotiert	rotate left	29
`ror`Ⓟ	`Rd, Rs1, Rs2`	Rd := Rs1 um Rs2 Stellen nach rechts rotiert	rotate right	29
`sll`	`Rd, Rs, Imm`	Rd:=Rs $\times$ 2^{Imm}	Shift left logical	29
`sllv`	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ 2^{Rs2}	Shift left logical variable	29
`srl`	`Rd, Rs, Imm`	Rd:=Rs DIV 2^{Imm}	Shift right logical	29
`srlv`	`Rd, Rs1, Rs2`	Rd:=Rs1 DIV 2^{Rs2}	Shift right logical variable	29
`sra`	`Rd, Rs, Imm`	Rd:=Rs $\times$ 2^{Imm}	Shift right arithmetic	29
`srav`	`Rd, Rs1, Rs2`	Rd:=Rs1 $\times$ 2^{Rs2}	Shift right arithmetic variable	29

Vergleichsbefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`seq`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1=Rs2, 0 sonst	(=) set equal	30
`sne`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\neq$Rs2, 0 sonst	($\neq$) set not equal	30
`sge`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\geq$Rs2, 0 sonst	($\geq$) set greater than equal	30
`sgeu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\geq$Rs2, 0 sonst	($\geq$) set greater than equal unsigned	30
`sgt`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1>Rs2, 0 sonst	(>) set greater than	30
`sgtu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\geq$Rs2, 0 sonst	(>) set greater than unsigned	30
`sle`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\leq$Rs2, 0 sonst	($\leq$) set less than equal	31
`sleu`Ⓟ	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1$\leq$Rs2, 0 sonst	($\leq$) set less than equal unsigned	31
`slt`	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1 < Rs2, 0 sonst	(<) set less than	31
`sltu`	`Rd, Rs1, Rs2`	Rd := 1, falls Rs1 < Rs2, 0 sonst	(<) set less than unsigned	31
`slti`	`Rd, Rs, Imm`	Rd := 1, falls Rs < Imm, 0 sonst	(<) set less than immediate	31

Befehl	Argumente	Wirkung	Erläuterung	S.
sltui	Rd, Rs, Imm	Rd := 1, falls Rs < Imm, 0 sonst	(<) set less than unsigned immediate	31

unbedingte Sprünge

Befehl	Argumente	Wirkung	Erläuterung	S.
bⓅ	label	unbedingter Sprung nach label	branch	32
j	label	unbedingter Sprung nach label	jump	32

bedingte Sprünge

Befehl	Argumente	Wirkung	Erläuterung	S.
beq	Rs1, Rs2, label	Sprung nach label, falls Rs1=Rs2	(=) branch on equal	33
beqzⓅ	Rs, label	Sprung nach label, falls Rs=0	$(=0)$ branch on equal zero	33
bne	Rs1, Rs2, label	Sprung nach label, falls Rs≠0	(≠) branch on not equal zero	33
bnezⓅ	Rs, label	Sprung nach label, falls Rs≠0	$(\neq 0)$ branch on not equal zero	33
bgeⓅ	Rs1, Rs2, label	Sprung nach label, falls Rs1≥Rs2	(≥) branch on greater than equal	33
bgeuⓅ	Rs1, Rs2, label	Sprung nach label, falls Rs1≥Rs2	(≥) branch on greater than equal unsigned	33
bgez	Rs, label	Sprung nach label, falls Rs≥0	(≥ 0) branch on greater than equal zero	33
bgtⓅ	Rs1 Rs2 label	Sprung nach label, falls Rs1>Rs2	(>) branch on greater than	33
bgtuⓅ	Rs1 Rs2 label	Sprung nach label, falls Rs1>Rs2	(>) branch on greater than unsigned	33
bgtz	Rs, label	Sprung nach label, falls Rs>0	(> 0) branch on greater than zero	33
bleⓅ	Rs1 Rs2 label	Sprung nach label, falls Rs1≤Rs2	(≤) branch on less than equal	33
bleuⓅ	Rs1 Rs2 label	Sprung nach label, falls Rs1≤Rs2	(≤) branch on less than equal unsigned	33
blez	Rs, label	Sprung nach label, falls Rs≤0	(≤ 0) branch on less than equal zero	33
bltⓅ	Rs1, Rs2, label	Sprung nach label, falls Rs1<Rs2	(<) branch on less than	34
bltuⓅ	Rs1 Rs2 label	Sprung nach label, falls Rs1<Rs2	(<) branch on less than unsigned	34
bltz	Rs, label	Sprung nach label, falls Rs<0	(< 0) branch on less than zero	34

Prozedursprunganweisungen

Befehl	Argumente	Wirkung	Erläuterung	S.
jr	Rs	unbedingter Sprung an die Adresse in Rs	jump register	43

Befehl	Argumente	Wirkung	Erläuterung	S.
jal	label	unbedingter Sprung nach label, Adresse des nächsten Befehls in $ra	jump and link	49
jalr	Rs, Rd	unbedingter Sprung an die Stelle Rs, Adresse des nächsten Befehls in Rd, Default: $ra	jump and link register	49
bgezal	Rs, label	Sprung nach label wenn Rs ≥ 0, Adresse des nächsten Befehls in $ra	Branch on greater than equal zero and link	49
bltzal	Rs, label	Sprung nach label wenn Rs < 0, Adresse des nächsten Befehls in $ra	Branch on less than and link	49

Gleitkommazahlen: Lade- und Speicherbefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
lwc$\langle z\rangle$	Cd, Adr	Cd := Mem[Adr]; $\langle z\rangle$: [0..3]	load word coprocessor	61
l.d Ⓟ	Fd, Adr	Fd := Mem[Adr]; Fd+1 := Mem[Adr+4]	Load floating-point double	61
l.s Ⓟ	Fd, Adr	Fd := Mem[Adr]	Load floating-point single	61
swc$\langle z\rangle$	Cs, Adr	Mem[Adr] := Cs; $\langle z\rangle$: [0..3]	store word coprocessor	62
s.d Ⓟ	Fs, Adr	Mem[Adr] := Fs; Mem[Adr+4] := Fs+1	Store floating-point double	62
s.s Ⓟ	Fs, Adr	Mem[Adr] := Fs	Store floating-point single	62

Gleitkommazahlen: Register-Transfer- und Konvertierungs-Befehle

Befehl	Argumente	Wirkung	Erläuterung	S.
mfc$\langle z\rangle$	Rd, Cs	Rd := Cs; $\langle z\rangle$: [0..3]	Move from coprocessor	62
mfc1.d Ⓟ	Rd, Cs	Rd := Cs; Rd+1 := Cs+1	Move double from coprocessor 1	62
mtc$\langle z\rangle$	Rs, Cd	Cd := Rs; $\langle z\rangle$: [0..3]	Move to coprocessor	62
mov.d	Fd, Fs	Fd/Fd+1 := Fs/Fs+1	Move floating-point double	62
mov.s	Fd, Fs	Fd := Fs	Move floating-point single	62
cvt.d.s	Fd, Fs	Fd/Fd+1 := Fs	Convert single to double	63
cvt.d.w	Fd, Fs	Fd/Fd+1 := Fs	Convert integer to double	63
cvt.s.d	Fd, Fs	Fd := Fs/Fs+1	Convert double to single	63
cvt.s.w	Fd, Fs	Fd := Fs	Convert integer to single	63
cvt.w.d	Fd, Fs	Fd := Fs/Fs+1	Convert double to integer	63
cvt.w.s	Fd, Fs	Fd := Fs/Fs+1	Convert single to integer	63

Gleitkommazahlen: Addition und Subtraktion

Befehl	Argumente	Wirkung	Erläuterung	S.
`add.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 + Fs2/Fs2+1	Floating-point addition double	63
`add.s`	`Fd, Fs1, Fs2`	Fd := Fs1 + Fs2	Floating-point addition single	63
`sub.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 - Fs2/Fs2+1	Floating-point subtract double	63
`sub.s`	`Fd, Fs1, Fs2`	Fd := Fs1 - Fs2	Floating-point subtract single	63

Gleitkommazahlen: Multiplikation und Division

Befehl	Argumente	Wirkung	Erläuterung	S.
`mul.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 * Fs2/Fs2+1	Floating-point multiply double	63
`mul.s`	`Fd, Fs1, Fs2`	Fd := Fs1 * Fs2	Floating-point multiply single	63
`div.d`	`Fd, Fs1, Fs2`	Fd/Fd+1 := Fs1/Fs1+1 / Fs2/Fs2+1	Floating-point divide double	63
`div.s`	`Fd, Fs1, Fs2`	Fd := Fs1 / Fs2	Floating-point divide single	63

Gleitkommazahlen: sonstige arithmetische Befehle

Befehl	Argumente	Wirkung	Erläuterung	S.
`abs.d`	`Fd, Fs`	Fd/Fd+1 := ABS (Fs/Fs+1)	Floating-point absolute value double	63
`abs.s`	`Fd, Fs`	Fd := ABS (Fs)	Floating-point absolute value single	63
`neg.d`	`Fd, Fs`	Fd/Fd+1 := - Fs/Fs+1	Floating-point negate double	63
`neg.s`	`Fd, Fs`	Fd := - Fs	Floating-point negate single	63

Gleitkommazahlen: Vergleichsbefehle

Befehl	Argumente	Wirkung	Erläuterung	S.
bc$\langle z \rangle$t		Sprung, wenn condition-flag des Koprozessors $\langle z \rangle$ wahr (1) ist	Branch coprocessor $\langle z \rangle$ true	64
bc$\langle z \rangle$f		Sprung, wenn condition-flag des Koprozessors $\langle z \rangle$ falsch (0) ist	Branch coprocessor $\langle z \rangle$ false	64
`c.eq.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 = Fs2/Fs2+1	Compare equal double	64
`c.eq.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 = Fs2	Compare equal single	64
`c.le.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 $\leq$ Fs2/Fs2+1	Compare less than equal double	64
`c.le.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 $\leq$ Fs2	Compare less than equal single	64

Befehl	Argumente	Wirkung	Erläuterung	S.
`c.lt.d`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1/Fs1+1 < Fs2/Fs2+1	Compare less than double	64
`c.lt.s`	`Fs1, Fs2`	Setze condition-flag auf wahr, wenn Fs1 < Fs2	Compare less than single	64

Unterbrechungen und Ausnahmen

Befehl	Argumente	Wirkung	Erläuterung	S.
`rfe`		Stellt das Statusregister wieder her.	return from exception	69
`break`	`Imm`	Verursacht die Ausnahme `Imm`	break	69

A.2 Alphabetische Befehlsübersicht

abs.s	blez	cvt.w.d	li	mulou	sge	sub.d
add.s	blt	cvt.w.s	lui	mult	sgeu	sub.s
addi	bltu	div	lw	multu	sgt	subu
addiu	bltz	div.d	lwc⟨z⟩	neg	sgtu	sw
addu	bltzal	div.s	lwl	neg.s	sh	swc⟨z⟩
andi	bne	divu	lwr	negu	sle	swl
bc⟨z⟩f	bnez	divu	mfc⟨z⟩	nop	sleu	swr
bc⟨z⟩t	break	j	mfc1.d	nor	sll	ulh
beqz	c.eq.d	jal	mfhi	not	sllv	ulhu
bge	c.eq.s	jalr	mflo	or	slt	ulw
bgeu	c.le.d	l.d	mov.d	ori	slti	ush
bgez	c.le.s	l.s	mov.s	rem	sltui	usw
bgezal	c.lt.d	la	mtc⟨z⟩	remu	sne	xor
bgt	c.lt.s	lb	mthi	ror	sra	xori
bgtu	cvt.d.s	lbu	mtlo	s.d	srav	
bgtz	cvt.d.w	ld	mul	s.s	srl	
ble	cvt.s.d	lh	mul.s	sb	srlv	
bleu	cvt.s.w	lhu	mulo	sd	sub	

A.3 Direktiven-Übersicht

Direktive	Argumente	Erläuterung
`.text`	[⟨Adr⟩]	Legt die folgenden Befehle bzw. Direktivenresultate im Textsegment ab. Mit dem optionalen Argument ⟨Adr⟩ kann eine Startadresse angeben werden. Defaultwert ist die nächste freie Adresse.
`.ktext`	[⟨Adr⟩]	Wie `.text`, jedoch erfolgen die Eintragungen im Textsegment des Kernels.
`.data`	[⟨Adr⟩]	Legt die folgenden Befehle bzw. Direktivenresultate im Datensegment ab. Mit dem optionalen Argument ⟨Adr⟩ kann eine Startadresse angeben werden. Defaultwert ist die nächste freie Adresse.
`.kdata`	[⟨Adr⟩]	Wie `.data`, jedoch erfolgen die Eintragungen im Datensegment des Kernels, welches aber vom SPIM nicht vom Datensegment unterschieden wird.

Direktive	Argumente	Erläuterung
`.sdata`	[⟨*Adr*⟩]	Wie `.data`, jedoch erfolgen die Eintragungen im Datensegment des Stacks, welches aber vom SPIM nicht vom Datensegment unterschieden wird.
`.align`	⟨*n*⟩	Richtet die folgenden Eintragungen an 2^n-Byte-Grenzen aus. ⟨*n*⟩=0 schaltet die automatische Ausrichtung bis zur nächsten `.(k)data`-Direktive aus.
`.asciiz`	"⟨*Str*⟩"	Legt die Zeichenkette ⟨*Str*⟩ ab und hängt Chr(0) an.
`.ascii`	"⟨*Str*⟩"	Legt die Zeichenkette ⟨*Str*⟩ ab, hängt Chr(0) aber *nicht* an.
`.word`	$\langle w_1 \rangle$ [, $\langle w_2 \rangle$ ...]	Legt die folgenden Werte als 32-Bit-Worte ab.
`.half`	$\langle h_1 \rangle$ [, $\langle h_2 \rangle$ ...]	Legt die folgenden Werte als 16-Bit-Worte ab.
`.byte`	$\langle b_1 \rangle$ [, $\langle b_2 \rangle$ ...]	Legt die folgenden Werte als 8-Bit-Worte ab.
`.float`	$\langle f_1 \rangle$ [, $\langle f_2 \rangle$ ...]	Legt die folgenden Werte als 32-Bit-Gleitkommazahlen ab.
`.double`	$\langle d_1 \rangle$ [, $\langle d_2 \rangle$ ...]	Legt die folgenden Werte als 64-Bit-Gleitkommazahlen ab.
`.space`	⟨*n*⟩	Läßt die nächsten ⟨*n*⟩ Byte frei.
`.set`	`noat`	Schaltet die SPIM-Warnung vor dem Benutzen von `$at` *aus*.
`.set`	`at`	Schaltet die SPIM-Warnung vor dem Benutzen von `$at` *ein*.
`.globl`	⟨*label*⟩	Exportiert die Marke ⟨*label*⟩, so dass sie von anderen Dateien aus benutzt werden kann.
`.extern`	⟨*label*⟩ ⟨*Größe*⟩	Legt das mit ⟨*label*⟩ bezeichnete Objekt mit der Größe ⟨*Größe*⟩ Bytes in den durch `$gp` bezeichneten Bereich.

A.4 Registerübersicht

Register- -name	-nr.	vereinbarte Nutzung	beschreibbar	caller-saved	callee-saved
`$zero`	0	Enthält den Wert 0	ja	-	-
`$at`	1	temporäres Assemblerregister	nein	-	-
`$v0`–`$v1`	1-2	Funktionsergebnisse 1 und 2	ja	×	-
`$a0`–`$a3`	4-7	Argumente 1 bis 4 für den Prozeduraufruf	ja	×	-
`$t0`–`$t9`	8-15, 24-25	temporäre Variablen 1-9	ja	×	-
`$s0`–`$s7`	16-23	langlebige Variablen 1-8	ja	-	×
`$k0`–`$k1`	26-27	Kernel-Register 1 und 2	nein	-	-
`$gp`	28	Zeiger auf Datensegment	nein	-	-
`$sp`	29	Stackpointer	ja	-	-
`$fp`	30	Framepointer	ja	-	×
`$ra`	31	Return address	ja	-	×

Register der Koprozessoren: Siehe Seite 60 (mathematischer Koprozessor) und Abbildung 13 auf Seite 67 (Koprozessor 0).

A.5 Betriebssystemfunktionen

Funktion	Code in $v0	Argument(e)	Ergebnis
print_int	1	$a0	
print_float	2	$f12	
print_double	3	$f12/13	
print_string	4	$a0 enthält die Anfangsadresse der mit Chr(0) terminierten Zeichenkette	
read_int	5		$v0
read_float	6		$f0
read_double	7		$f0/1
read_string	8	$a0 enthält die Adresse ab der die Zeichenkette abgelegt wird, $a1 ihre maximale Länge	Zeichenkette ab ($a0)
sbrk	9	$a0 benötigte Größe	$v0 Anfangsadresse des Speicherbereichs
exit	10		

B Literaturverzeichnis

Literatur

[Pat] Patterson, David A. und John L. Hennessy: Computer Organization and Design. The Hardware/Software Interface. San Francisco 1994.

Das Buch enthält [Lar] als Anhang A.

[Lar] Larus, James R.: SPIM S20: A MIPS R2000 Simulator. Madison 1993. `http://www.cs.wisc.edu/~larus/SPIM_manual/spim-manual.html`

Das Dokument ist als Anhang A in [Pat] enthalten.

[Tan] Tanenbaum, Andrew S.: Structured Computer Organisation. Third Edition (1990). Amsterdam 1976.

[Coy] Coy, Wolfgang: Aufbau und Wirkungsweise von Rechenanlagen. 2. Auflage. Braunschweig/Wiesbaden 1992.

C Verzeichnisse

Abbildungsverzeichnis

Beispielverzeichnis

Stichwortverzeichnis

Die unterstrichenen Seitenzahlen verweisen auf eine Definition oder eine Begriffserklärung, die **fett** gesetzten Seitenzahlen auf Übersichten oder Abbildungen.