MIPS-Programmierung in der WebSPIM-Umgebung (0.3.1)

C. Reichenbach, mailto:reichenbach@cs.uni-frankfurt.de

28. April 2015

1 Einführung

WebSPIM ist ein Web-basierter MIPS32-Simulator, dessen MIPS-Funktionalität auf dem Simulator „SPIM“[1], entwickelt von James Larus (U. Wisconsin) basiert. WebSPIM erweitert SPIM wie folgt:

- Web-basierte Nutzerschnittstelle
- Übungsaufgaben-Management
- Übungsspezifische Speicherfunktion
- Kustomisierungen des simulierten Systems je nach Übungsaufgabe

Dieses Dokument ist eine Kurzbeschreibung zur Verwendung von SPIM. Abschnitt 2 erläutert WebSPIM-Spezifika, während Abschnitt 3 eine Beschreibung der von SPIM verwendeten Assemblersprache ist.

2 WebSPIM

Das WebSPIM-System besteht aus drei wesentlichen Komponenten:

- Der Aufgabenauswahlkomponente
- Der Programmeditor
- Der Simulator

2.1 Aufgabenauswahl

Dies ist die Einstiegsseite des Systems. Die Aufgabe zeigt die möglichen Interaktionen mit dem System an, je nach Konfiguration. Dies kann z.B. die freie Interaktion mit dem Simulator sein, oder auch spezifische Aufgaben mit einem festgesetzten spätesten Abgabetermin.

2.2 Programmeditor

Der Editor (Abbildung 2) besteht aus einem Textfeld mit Assemblercode (linker Hand), einem Block mit der Aufgabenstellung (rechts oben) und (bei Übungen) einen Einsendebereich im rechten unteren Bereich.

Der Editor selbst hat nur grundlegendste Programmmanipulationsfähigkeiten. Es kann (je nach persönlicher Präferenz) einfacher sein, in einem anderen Editor zu arbeiten und den erzeugten Assemblercode dann in den WebSPIM-Editor zu kopieren.

Der Einsendebereich ist nur bei Übungen verfügbar, deren letzter Abgbetermin noch nicht abgelaufen ist. Bei bereits abgelaufenen Übungen wird hier nur die zuletzt abgegebene Übung angezeigt.

Wenn die Übung noch eingereicht werden kann, besteht dieser Eingabebereich aus zwei weiteren Eingabefeldern. Eines dient dazu, eventuelle Namen von mitarbeitenden Studierenden anzugeben. Das zweite, größere Feld kann für verschiedenartige Anmerkungen genutzt werden. Rechts davon ist die aktuellst eingesendete Fassung von Programm, Anmerkungen, und Mitarbeitern. Dieser Eintrag kann durch Drücken des „Einsenden“-Knopfes überschrieben werden, solange der letzte Abgabetermin noch nicht überschritten ist.

Der „Programm Laden“-Knopf links unterhalb dem Programmeditor läd das Programm in den Simulator, ohne es einzusenden.

[1] www.cs.wisc.edu/~larus/spim.html

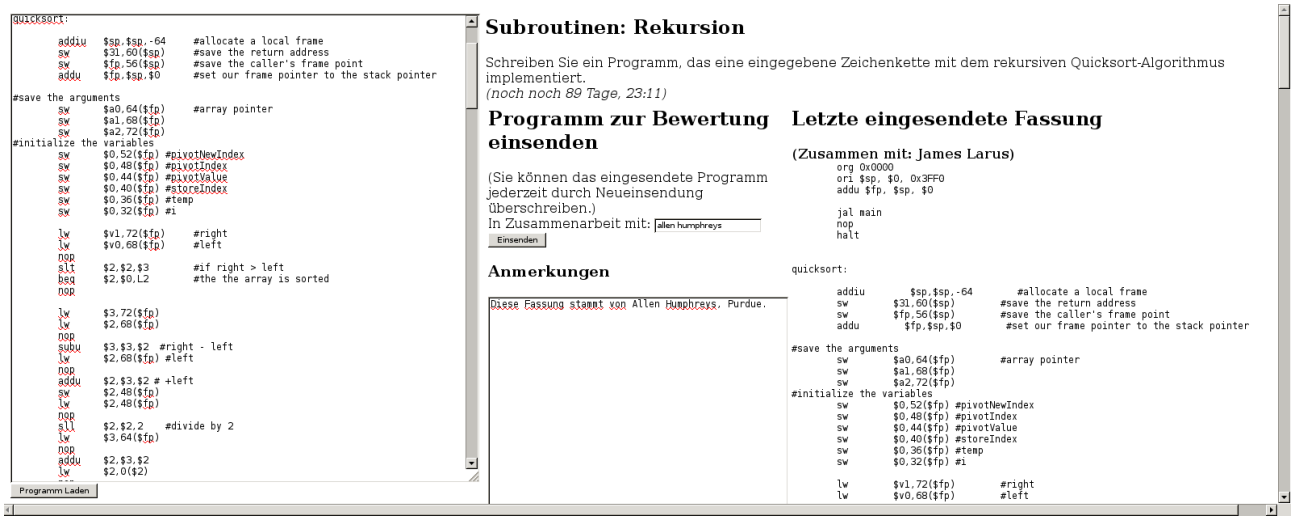

Abbildung 1: Der WebSPIM-Editor.

2.3 Simulator

Der Simulator besteht aus sechs Komponenten:

- Im linken Drittel ist eine Speicheranzeige.
- Im mittleren Drittel ist eine Anzeige der Assemblerbefehle.
- Rechts oben sind Links, die zurück zum Hauptmenü oder zurück zum Programmeditor springen. Wenn das laufende Programm vom Editor aus nicht eingesendet wurde, wird es beim Sprung in das Hauptmenü verworfen, beim Sprung in den Programmeditor aber wieder hergestellt.
- Darunter ist eine Anzeige der aktuellen Registerinhalte.
- Darunter ist eine kleine Leiste mit Ausführknöpfen.
- Ganz rechts unten ist die Konsolenausgabe des Systems. Hier werden Ausgaben des Prozessors gesammelt.

2.3.1 Die Speicheranzeige

Ganz links in eckigen Klammern sind Adressen wie z.B. `[00400000]` angegeben. Diese Adressen geben (hexadezimal notiert) die Speicheradresse des ersten der sechzehn folgenden Bytes (ebenfalls hexadezimal notiert) an. Eine Zeile der Form

```
[00400000] 01 02 03 04 05 06 07 08 00 00 00 00 0A 0B 0C 0D
```

besagt also, daß das Byte an Adresse $\texttt{00400000}_{16}$ im Speicher den Wert $\texttt{01}_{16}$ hat, das Wort an Adresse $\texttt{00400001}_{16}$ den Wert $\texttt{02}_{16}$ usw.

2.3.2 Die Assembleranzeige

Im mittleren Drittel sind alle identifizierten Assemblerbefehle aufgeführt, zusammen mit ihrer Speicheradresse und ihrem Maschinencode. Ein Doppelpfeil (⇒) markiert die Instruktion, auf die der aktuelle Befehlszähler (PC) verweist.

2.3.3 Ausführknöpfe

Die beiden Ausführknöpfe können zur Einzelschrittausführung oder zur Mehrschrittausführung verwendet werden; bei letzterer muß die Anzahl der gewünschten Schritte angegeben werden. Der Einzelausführungsknopf kann durch Drücken der `s`-Taste direkt ausgelöst werden.

Die Simulationsumgebung beobachtet Änderungen von Speicher- und Registerinhalten und markiert diese in der Anzeige durch hellgrüne Unterlegung des geänderten Wertes.

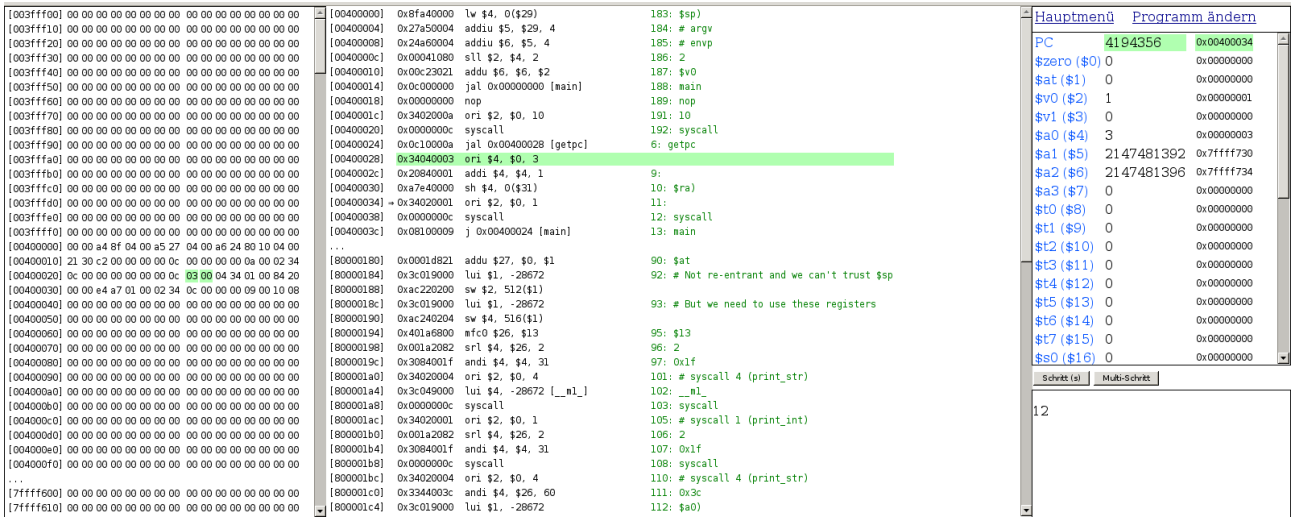

Abbildung 2: Das WebSPIM-Interface zu SPIM.

3 MIPS-Programmierung mit SPIM

Der SPIM-Assembler ist Teil des SPIM-Systems und somit auch Teil von WebSPIM. Seine Aufgabe ist es, Assemblerbefehle in Maschinensprache zu übersetzen, um diese ausführen zu können.

Maschinenspracheprogramme bestehen aus vier Bausteinen:

- Direktiven, die Daten angeben oder Eigenschaften des Programms erläutern
- Marken, als Ziele für Sprung-/Verzweigungsoperationen (Sprungmarken) oder auch für Speicheroperationen (Speichermarken)
- Assemblerbefehle
- Kommentare

Hier ist ein kleines SPIM-Programm, das diese Konzepte illustriert:

```
        .text       # Markiert den folgenden Teil als Assembler
        .globl main # Markiert Sprungmarke "main" als global sichtbar
main:
    li $a0, 2
    li $v0, 1
    syscall
    li $v0, 10
    syscall
```

Die erste Zeile beinhaltet eine Direktive `.text`. Direktive beginnen immer mit einem vorangestellten Punkt. Der Rest der Zeile ist Kommentar, eingeleitet von einem Lattenzaun-Zeichen (`#`): Dieses Zeichen markiert immer das erste Zeichen eines Kommentars, so daß der Rest der Zeile automatisch vom Assembler ignoriert wird.

Zeile Drei besteht aus der Sprungmarke `main`. Sprungmarken werden durch einen Doppelpunkt abgeschlossen. Sie werden vom Assembler verwendet, um die Adresse eines Objektes zu bestimmen. Sie können sowohl relativ (z.B. bei `bgz`, `lw`) als auch absolut (z.B. bei `j`, `jal`) verwendet werden; der Assembler erkennt dies automatisch.

Die letzten fünf Zeilen schließlich sind Assemblerbefehle.

Wir beschränken uns in dieser Beschreibung auf die grundsätzlichsten Fähigkeiten des SPIM-Assemblers, wie sie zum Lösen typischer Programmieraufgaben nötig sind.

3.1 Direktiven

3.1.1 Sektionsmarkierungen

Diese Markierungen geben an, in welchen Teil des Speichers dieser Teil des Assemblerprogrammes geladen werden soll.

.data : Diese Direktive instruiert den Assembler, die folgenden Teile in das Datensegment zu laden. Typischerweise werden hier globale Variablen reserviert, z.B. per `.byte`, `asciiz`, oder `.word`-Direktive.

.text : Diese Direktive instruiert den Assembler, die folgenden Teile in das Textsegment zu laden, also in den zur Programmausführung designierten Speicher.

3.1.2 Sichtbarkeit von Marken

Sprung- und Speichermarken sind normalerweise nur innerhalb des eigenen Moduls sichtbar. In WebSPIM ist das geladene Programm immer ein eigenständiges Modul, somit sind dessen Marken vom Rest des Systems streng getrennt. Dies kann jedoch über einige Deklarationen gesteuert werden.

.globl : Diese Direktive wird zusammen mit einer Marke verwendet, z.B.

```
.globl main
```

Sie macht diese Marke global sichtbar, erlaubt also anderen Modulen, sie anzuspringen oder von ihr zu laden. Die Sprungmarke `main` muß im Hauptmodul immer global markiert sein, damit das Betriebssystem weiß, wohin im Programm es springen soll, um das Programm zu starten.

.comm, .lcomm : Diese Direktiven allozieren Speicher (üblicherweise im `.data`-Bereich), geben dem Speicher eine Marke und setzen die Sichtbarkeit. `.comm` setzt globale Sichtbarkeit, während `.lcomm` lokale Sichtbarkeit setzt. Die Direktiven haben zwei Parameter, durch ein Komma getrennt: der Name der Marke, und der reservierte Platz. Zum Beispiel:

```
.lcomm puffer, 40
```

alloziert 40 Bytes an einer Marke `puffer`.

.set : Diese Direktive ändert das Verhalten des Assemblers bezüglich Assemblercode, der auf das `$at`-Register zugreift.

```
.set at
```

verbietet dem Programmierer, dieses Register direkt zu verwenden, aber erlaubt Pseudobefehle, die das Register verwenden.

```
.set noat
```

verbietet dem Programmierer, Pseudobefehle zu verwenden, die `$at` verwenden, aber erlaubt direkten Zugriff auf dieses Register.

3.1.3 Platzallozierungen

Allozierungen finden normalerweise im `.data`-Segment statt. Diese Operationen stellen eine gewisse Menge an Speicher im aktuellen Segment zur Verfügung. Einige von ihnen initialisieren den Speicher auch (d.h., sie setzten den Speicher auf vordefinierte Werte).

.space : alloziert die angegebene Menge an Speicher, initialisiert diesen aber nicht.

```
.space 8
```

alloziert zum Beispiel 8 Bytes.

.byte : alloziert ein Byte für jedes Byte, das dem `.byte` folgt (durch Komma getrennt). Beispiel:

```
.byte 12, 0x0a
```

alloziert zwei Bytes mit den Werten 12 (`0xc`) und 10 (`0xa`).

.half, .word : Diese beiden Direktiven funktionieren wie `.byte`, allerdings für Halbworte und Worte.

.ascii, .asciiz : Diese Direktiven allozieren Zeichenketten. Beide legen ASCII-Zeichen ab; der Unterschied ist, daß `asciiz` hinter diese Zeichen noch ein Null-Byte legt, zum Beispiel:

```
.asciiz "Hallo, Welt!"
```

3.2 Assemblerbefehle

Wir unterscheiden zwischen Hardwarebefehlen und Pseudobefehlen. Pseudobefehle markieren wir mit einem unten angestellten "p", z.B. li$_p$.

3.2.1 Arithmetische und logische Befehle

abs$_p$ $z, $x: $z := |$x| (d.h., es wird der Betrag berechnet.)

add $z, $x, $y: $z := $x + $y. Ausnahmebehandlung bei Überlauf.

addu $z, $x, $y: $z := $x + $y. Überlauf wird abgeschnitten.

addi $z, $x, v: $z := $x + v. Ausnahmebehandlung bei Überlauf. v ist vorzeichenbehaftet.

addiu $z, $x, v: $z := $x + v. Überlauf wird abgeschnitten. v ist nicht vorzeichenbehaftet.

and $z, $x, $y: $z := $x & $y. Bitweise Und-Verknüpfung der Bits.

andi $z, $x, v: $z := $x & v. Bitweise Und-Verknüpfung der Bits.

clo $z, $x: Zählt die führenden Einsen in der Bit-Repräsentierung des Inhaltes von $x.

clz $z, $x: Zählt die führenden Nullen in der Bit-Repräsentierung des Inhaltes von $x.

div $x, $y: LO := $\left\lfloor \frac{\$x}{\$y} \right\rfloor$, HI := $x *mod* $y. Ausnahmebehandlung bei Überlauf. Registerinhalte werden vorzeichenbehaftet interpretiert. Der Divisionsrest in HI ist nur für positive Operanden definiert, bei negativen Operanden gibt die Befehlssatzarchitektur keine Garantien zum Wert von HI.

divu $x, $y: LO := $\left\lfloor \frac{\$x}{\$y} \right\rfloor$, HI := $x *mod* $y. Überlauf wird abgeschnitten. Registerinhalte werden ohne Vorzeichen interpretiert.

div$_p$ $z, $x, $y: $z := $\left\lfloor \frac{\$x}{\$y} \right\rfloor$. Ausnahmebehandlung bei Überlauf. Vorzeichenbehaftet.

divu$_p$ $z, $x, $y: $z := $\left\lfloor \frac{\$x}{\$y} \right\rfloor$. Überlauf wird abgeschnitten. Nicht vorzeichenbehaftet.

mult $x, $y: HI:LO := $x × $y. Registerinhalte werden vorzeichenbehaftet interpretiert.

multu $x, $y: HI:LO := $x × $y. Registerinhalte werden ohne Vorzeichen interpretiert.

mul$_p$ $z, $x, $y: $z := $x × $y. Überlauf wird abgeschnitten. Registerinhalte werden vorzeichenbehaftet interpretiert.

mulo$_p$ $z, $x, $y: $z := $x × $y. Ausnahmebehandlung bei Überlauf. Registerinhalte werden vorzeichenbehaftet interpretiert.

mulou$_p$ $z, $x, $y: $z := $x × $y. Ausnahmebehandlung bei Überlauf. Registerinhalte werden ohne Vorzeichen interpretiert.

neg$_p$ **$z, $x:** $z = -$x. Arithmetische Negation. Ausnahmebehandlung bei Überlauf.

negu$_p$ **$z, $x:** $z = -$x. Arithmetische Negation. Überlauf wird ignoriert.

nor $z, $x, $y: $z := ¬($x | $y). Bitweise Nicht-Oder-Verknüpfung der Bits.

or $z, $x, $y: $z := $x | $y. Bitweise Oder-Verknüpfung der Bits.

ori $z, $x, v: $z := $x | v. Bitweise Oder-Verknüpfung der Bits.

rem$_p$ **$z, $x, $y:** $z := $x *mod* $y. Ausnahmebehandlung bei Überlauf. Vorzeichenbehaftet.

remu$_p$ **$z, $x, $y:** $z := $x *mod* $y. Überlauf wird abgeschnitten. Nicht vorzeichenbehaftet.

sll $z, $x, v: $z := $x ≪ v. Verschiebt Bits nach links und füllt mit Nullen auf.

sllv $z, $x, $y: $z := $x ≪ $y. Verschiebt Bits nach links und füllt mit Nullen auf.

sra $z, $x, v: $z := $x $\gg^-$ v. Verschiebt Bits nach rechts und füllt mit Vorzeichenbit auf.

srav $z, $x, $y: $z := $x $\gg^-$ $y. Verschiebt Bits nach rechts und füllt mit Vorzeichenbit auf.

srl $z, $x, v: $z := $x ≫ v. Verschiebt Bits nach rechts und füllt mit Nullen auf.

srlv $z, $x, $y: $z := $x ≫ $y. Verschiebt Bits nach rechts und füllt mit Nullen auf.

rol$_p$ **$z, $x, v:** Verschiebt Bits nach links und füllt rechts mit den hinausgeschobenen Bits auf.

ror$_p$ **$z, $x, $y:** Verschiebt Bits nach rechts und füllt links mit den hinausgeschobenen Bits auf.

sub $z, $x, $y: $z := $x − $y. Ausnahmebehandlung bei Überlauf.

subu $z, $x, $y: $z := $x − $y. Überlauf wird abgeschnitten.

xor $z, $x, $y: $z := $x ^ $y. Bitweise Exklusiv-Oder-Verknüpfung der Bits.

xori $z, $x, v: $z := $x ^ v. Bitweise Exklusiv-Oder-Verknüpfung der Bits.

3.2.2 Ladeoperationen

move$_p$ **$z, $x:** $z := $x

li$_p$ **$z, v:** Lade 32 Bit-Wert v in Register $z

la$_p$ **$z, ziel:** Lade Adresse der Sprungmarke ziel in Register $z

lui $z, v: Lade obere 16 Bit von $z mit v, setze untere 16 Bit auf 0.

mfhi $z: $z := HI

mflo $z: $z := LO

mthi $z: HI := $z

mtlo $z: LO := $z

3.2.3 Vergleichsbefehle

slt $z, $x, $y: Setzt $z auf 1 gdw $x < $y (mit Vorzeichen), sonst 0.

sltu $z, $x, $y: Setzt $z auf 1 gdw $x < $y (ohne Vorzeichen), sonst 0.

slti $z, $x, v: Setzt $z auf 1 gdw $x < v (mit Vorzeichen), sonst 0.

sltiu $z, $x, v: Setzt $z auf 1 gdw $x < v (ohne Vorzeichen), sonst 0.

seq$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x = $y (mit Vorzeichen), sonst 0.

sge$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x ≥ $y (mit Vorzeichen), sonst 0.

sgeu$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x ≥ $y (ohne Vorzeichen), sonst 0.

sgt$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x > $y (mit Vorzeichen), sonst 0.

sgtu$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x > $y (ohne Vorzeichen), sonst 0.

sle$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x ≤ $y (mit Vorzeichen), sonst 0.

sleu$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x ≤ $y (ohne Vorzeichen), sonst 0.

sne$_p$ $z, $x, $y: Setzt $z auf 1 gdw $x ≠ $y (mit Vorzeichen), sonst 0.

3.2.4 Sprung- und Verzweigungsbefehle

b ziel: PC := ziel

bnez $x, ziel: PC := ziel falls $x ≠ 0

beqz $x, ziel: PC := ziel falls $x = 0

beq $x, $y, ziel: PC := ziel falls $x = $y

bne $x, $y, ziel: PC := ziel falls $x ≠ $y

bgez $x, ziel: PC := ziel falls $x ≥ 0 (mit Vorzeichen)

bgezal $x, ziel: $ra := PC + 4; PC := ziel falls $x ≥ 0 (mit Vorzeichen)

bgtz $x, ziel: PC := ziel falls $x > 0 (mit Vorzeichen)

blez $x, ziel: PC := ziel falls $x ≤ 0 (mit Vorzeichen)

bltz $x, ziel: PC := ziel falls $x < 0 (mit Vorzeichen)

bltzal $x, ziel: $ra := PC + 4; PC := ziel falls $x < 0 (mit Vorzeichen)

bge$_p$ $x, $y, ziel: PC := ziel falls $x ≥ $y (mit Vorzeichen)

bgeu$_p$ **$x, $y, ziel:** PC := ziel falls $x ≥ $y (ohne Vorzeichen)

bgt$_p$ **$x, $y, ziel:** PC := ziel falls $x > $y (mit Vorzeichen)

bgtu$_p$ **$x, $y, ziel:** PC := ziel falls $x > $y (ohne Vorzeichen)

ble$_p$ **$x, $y, ziel:** PC := ziel falls $x ≤ $y (mit Vorzeichen)

bleu$_p$ **$x, $y, ziel:** PC := ziel falls $x ≤ $y (ohne Vorzeichen)

blt$_p$ **$x, $y, ziel:** PC := ziel falls $x < $y (mit Vorzeichen)

bltu$_p$ **$x, $y, ziel:** PC := ziel falls $x < $y (ohne Vorzeichen)

j ziel: PC := ziel

jal ziel: $ra := PC + 4; PC := ziel

jr $x: PC := $x

jalr $x: $ra := PC + 4; PC := $x

3.2.5 Speicheroperationen (Laden und Schreiben)

Lade- und Schreibbefehle für den Speicher greifen jeweils auf Adresse v+$x zu. Sie addieren also den Inhalt von Register $x mit dem Direktoperanden v und legen dadurch die Speicheradresse fest, auf die sie schreiben bzw. von der sie lesen. Solche Zugriffe müssen entsprechend den Bedürfnissen des Befehls im Speicher ausgerichtet sein, sonst wird eine Ausnahmebehandlung im Betriebssystemkern angestoßen.

Ladebefehle:

Befehl	Name	Ausrichtung	B	Vorzeichenerweiterung
lb $z, v($x)	load **b**yte	8	1	ja
lbu $z, v($x)	load **b**yte **u**nsigned	8	1	nein
lh $z, v($x)	load **h**alfword	16	2	ja
lhu $z, v($x)	load **h**alfword **u**nsigned	16	2	nein
lw $z, v($x)	load **w**ord	32	4	—

Schreibbefehle:

Befehl	Name	Ausrichtung	Bytes
sb $z, v($x)	**s**tore **b**yte	8	1
sh $z, v($x)	**s**tore **h**alfword	16	2
sw $z, v($x)	**s**tore **w**ord	32	4

3.2.6 Systemaufrufe

syscall : Springe in die Unterbrechungsbehandlungsroutine des Betriebssystems und bitte um einen Systemdienst. Der genaue Systemdienst wird durch eine Kennzahl in Register $v0 ausgewählt. Die verfügbaren Systemdienste variieren je nach Konfiguration von WebSPIM. Hier sind die häufigst unterstützten Dienste:

$v0	Parameter	Beschreibung	Wirkung
1	$a0	Zahl ausgeben	Druckt die Zahl in $a0 auf der Ausgabekonsole aus.
4	$a0	Zeichenkette ausgeben	Druckt die null-terminierte ASCII-Zeichenkette, die im Speicher ab Adresse $a0 abgelegt ist, auf der Ausgabekonsole aus.
5		Zahl einlesen	Liest eine Zahl ein und schreibt diese in Register $v0.
8	$a0, $a1	Zeichenkette einlesen	Liest eine null-terminierte ASCII-Zeichenkette ein. Die Zeichenkette wird ab Adresse $a0 in den Speicher geschrieben. Wenn mehr als ($a1-1) Zeichen eingegeben wurden, wird die Zeichenkette gekürzt: maximal $a1 Zeichen werden geschrieben.
10		System anhalten	Beauftragt das Betriebssystem, den Prozessor anzuhalten.

3.3 Register

Register	Name	Bedeutung	Erhalten?
	PC	Programmzähler	—
$0	$zero	Konstante 0	—
$1	$at	Assembler-Hilfsregister	*nein*
$2–$3	$v0–$v1	Rückgabewerte	*nein*
$4–$7	$a0–$a3	Argumente (Parameter)	*nein*
$8–$15	$t0–$t7	Temporäre Register	*nein*
$16–$23	$s0–$s7	Gesicherte Register	*ja*
$24–$25	$t8–$t9	Temporäre Register	*nein*
$26–$27	$k0–$k1	Kernelregister	*nein*
$28	$gp	Globaler Zeiger	*ja*
$29	$sp	Stapelzeiger	*ja*
$30	$fp	Rahmenzeiger	*ja*
$31	$ra	Rücksprungadresse	*nein*
	HI	Mult.-Einheit, obere 32 Bit	*nein*
	LO	Mult.-Einheit, untere 32 Bit	*nein*

3.4 Aufrufkonventionen

Die Aufrufkonventionen stellen eine reibungslose Koexistenz verschiedener Programmteile sicher. Sie beschreiben, wie der *Aufrufer* vor einer jal oder jalr-Instruktion das System vorbereitet, und welche Verantwortung der *Aufgerufene*, also die per jal oder jalr-Instruktion aufgerufene Subroutine hat.

Die hier wiedergegebenen Aufrufkonventionen sind vereinfachte MIPS64-Aufrufkonventionen (so haben wir z.B. die Übergabe von Fließkommazahlen ausgelassen).

iese Konventionen werden vom Prozessor nicht erzwungen; sie sind eine standardisierte und formalisierte Vereinbarung zwischen MIPS-Programmierern.

3.4.1 Aufrufer

Der *Aufrufer* stellt beim Aufruf einer Subroutine mit n Parametern folgendes sicher:

- Der Stapelzeiger $sp ist 64-Bit-ausgerichtet
- Auf dem Stapel ist ab $sp Platz für mindestens n Worte
- Parameter 0–3, soweit vorhanden, liegen in Registern $a0–$a3
- Jeder Parameter i ($i > 3$), soweit vorhanden, liegt im Speicher an der Adresse $sp + 4 \times i$

Da der Aufgerufene alle in Abschnitt 3.3 als ***Erhalten***=*nein* markierten Register ändern kann, muß der Aufrufer zudem sicherstellen, daß die Inhalte aller solcher Register, die nach dem Subroutinenaufruf noch relevant sind, anderweitig gesichert werden.

3.4.2 Aufgerufener

Der *Aufgerufene* stellt Eigenschaften sicher, die sich auf den *Subroutinenbeginn* beziehen. Mit Subroutinenbeginn ist dabei der Zustand des Systems zum Zeitpunkt unmittelbar vor der Ausführung der ersten Instruktion der Subroutine gemeint.

- Am Ende der Subroutine sind die Inhalte aller Register, die in Abschnitt 3.3 als ***Erhalten***=*ja* markiert sind, identisch mit den Inhalten, die diese Register zum Subroutinenbeginn hatten.
- Am Ende der Subroutine springt diese an die Adresse, die das Register `$ra` zum Subroutinenbeginn hatte.