Frankfurt University of Applied Sciences

Fachbereich 2

MIPS Cheat Sheet

Dr. Erwin Hoffmann

(hoffmann@fehcom.de)

(Version 0.9.10a)

5. Juli 2023

Zusammenfassung

Dieses MIPS 'Cheat Sheet' enthält eine Zusammenfassung der MIPS Spezifika und soll als Gedankenstütze dienen. Quellen sind das *Spiem Tutorial*, die offiziellen MIPS Architektur Dokumente und Wikipedia.

Die Materialien aus meiner Vorlesung wurden sorgfältig überarbeitet und dort vorhandene Fehler und Unklarheiten beseitigt.

Das Skript ergänzt meine Vorlesung 'Rechnerarchitekturen'; eignet sich aber nicht dazu den MIPS-Assembler zu erlernen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Hardware-Architektur

1.1 Komponenten

Der Standard-MIPS32 Aufbau umfasst die folgenden Komponenten [Abb. 1.1]:

- **ALU** (*Arithmetic Logic Unit*) mit 32 (universellen) 32-Bit Registern (*register file*) sowie einigen Zusatzregistern für Kontrolle und Arithmetik.
- **CP0**: (*Control CPU*) mit 32 (meistens speziellen) 32-Bit Registern verfügbar im Kernel-Mode.
- **CP1**: (*Floating Point Unit*) mit 32 32-Bit Registern im *single precision mode*, die im *dual precision mode* paarweise als insgesamt 16 64-Bit Register genutzt werden können.

Ergänzend beinhaltet die Architektur die

- gemeinsamen Daten- und Instruktionsbusse mit 32 Bit Breite zwischen Registern und den CPU (*von-Neumann Architektur*),
- Busse, die die CPUs untereinander verbinden,
- Busse, die die CPUs mit dem Hauptspeicher koppeln (für *Load & Store* Operationen), sowie die
- Kontroll-Logik.

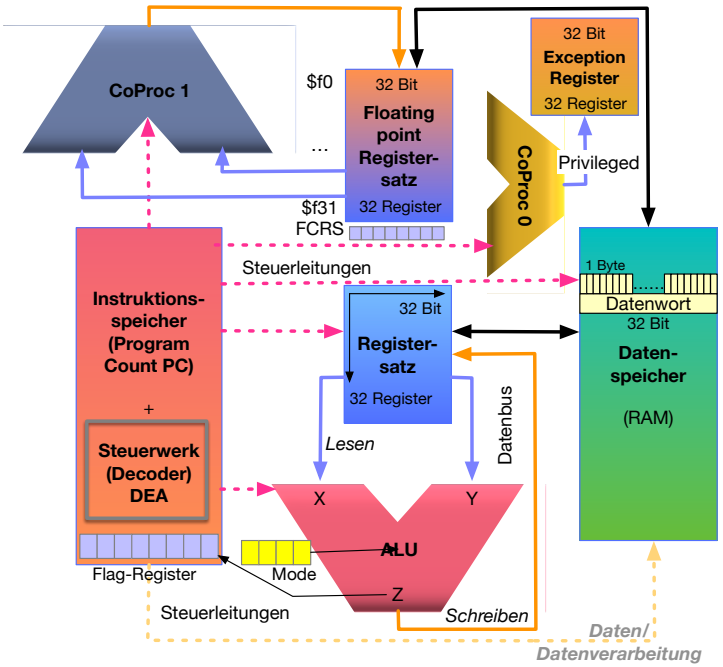

Abbildung 1.1: Hardware-Aufbau eines MIPS-Rechners

Optional kann die Architektur um

- ein *Read-Only Memory* (ROM) mit erweitertem Befehlsumfang sowie
- weitere **FPUs** oder dedizierte Prozessoren

ergänzt werden.

1.2 Load & Store in der 'von-Neumann'-Architektur

Die MIPS-Architektur folgt dem *von-Neumann* Vorbild [Abb. 1.2]:

- Instruktionen und Daten werden gemeinsam im Hauptspeicher gehalten (allerdings an verschiedenen Adressen hier).
- Instruktionen und Daten werden über einen einheitlich 32-Bit breiten Bus zwischen den Registern und dem Hauptspeicher transportiert, wobei die Architektur hier verschiedene 'Ports' zulässt (D-Port und I-Port).
- Die Verarbeitung folgt dem *Load & Store* Paradigma, d.h. Daten und Instruktionen werden sequentiell vom Hauptspeicher zum den CPU-Registern übertragen; Befehle werden 'decodiert' und sequentiell abgearbeitet, sofern der Code (über Verzweigungen) nichts anderes vorschreibt.

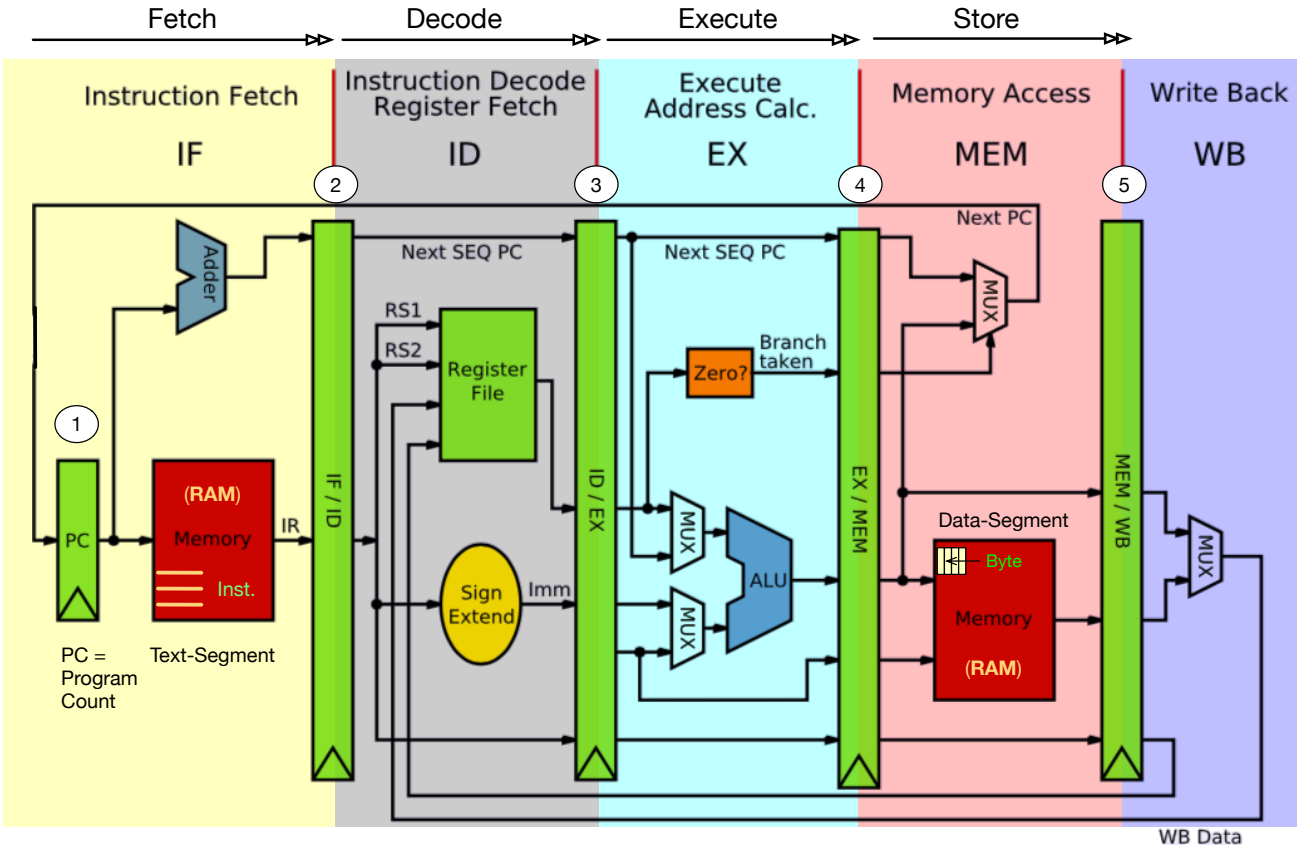

Abbildung 1.2: Load & Store-Architektur eines MIPS-Rechners

Neben dem externen Hauptspeicher (*Random Access Memory* **RAM**), kann die MIPS-Hardware auch über sog. Caches verfügen, als Read-/Write-Memory, dass der CPU zugeordnet ist. Hierdurch können Daten und Instruktionen lokal bevorratet werden und diese in einer 'pipeline' Operation vom Hauptspeicher bezogen werden, um die Ladezeiten zu verringern. Zudem unterstützt die MIPS-Hardware die Abbildung des physischen Hauptspeichers und der virtuellen Speicheradressen durch einen *Memory Controller* bzw. *Memory Management Unit* (**MMU**) und eines *Translation Lookaside Buffers* **TLB**. Die Verwaltung der Hardware, ist Aufgabe der CP0, also des Kontroll-Prozessors.

1.3 Betriebsmodi

Wie die meisten CPU-Architekturen [Abb. 1.3] unterscheidet auch MIPS zwischen

- einem *Supervisor* (bzw. *Kernel*) Mode mit vollem Zugriff auf die Hardware und
- einem *Benutzer* (*User*) Mode, in dem die Anwendungsprogramme laufen.

Die Unterscheidung zwischen einem privilegierten und nicht-privilegierten Modus erfolgt

- unter *Kernel-Kontrolle* über den Benutzer-Kontext des Programms,
- unter *CPU-Kontrolle* durch spezifische Assembler-Befehle, die sich in Hochsprachen nur über das Hinzufügen von bestimmten Start-Up Codes ausführen lassen (als *Object-Files*).

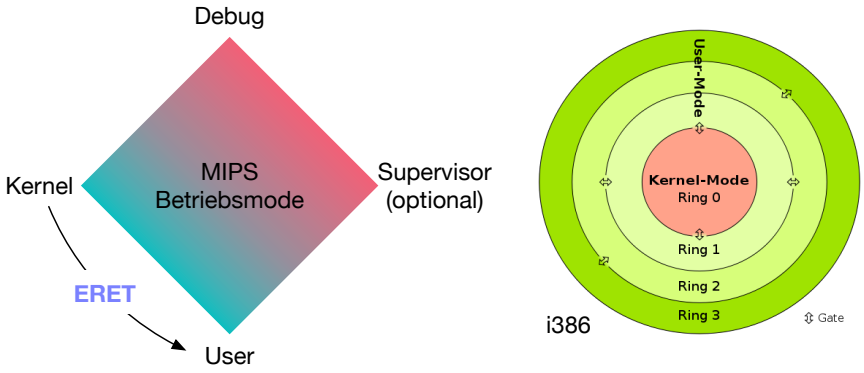

Abbildung 1.3: Betriebsmodi bei der MIPS und i386-Architektur: ERET: Exception Return Anweisung

↪ Bei der Intel-Architektur spricht man von *Ringen*. Relevant sind hier heute nur noch der Kernel- und User-Ring. Beim Betriebssystem *O2/2* wurde Ring 1 zur Graphik-Ausgabe benutzt; während sowohl Linux/Unix als auch Windows (NT) die Graphik-Fähigkeit in den Ring 0 gelegt haben (→ BSoD: **Blue Screen of Death**).

2 CPU Register

Jede CPU der MIPS besitzt mindestens 32 Register mit je 32 Bit Wortlänge. Während die Register der ALU und der FPU in der Regel General Purpose Register sind, die allgemein von einem Assembler-Programm genutzt werden können, sind die Register der Kontroll-CPU CP0 geschützt und nur im privilegierten Mode les- bzw. beschreibbar. Aber es gibt in allen Fällen eine Reihe von Ausnahmen.

2.1 ALU: General Purpose Register (GPR)

Der MIPS verfügt über 32 Register [Tab. 2.1]. 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.

- $zero: Enthält den Wert Null (0). Dieses Register ist auf Schaltkreisebene realisiert und kann nicht beschrieben werden.
- $a0 bis $a3: Prozedurargumente, weitere Argumente müssen über den Stack übergeben werden.
- $v0 und $v1: Funktionsergebnisse, gegebenenfalls können diese Register auch bei der Auswertung von Ausdrücken benutzt werden.
- $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.
- $s0 bis $s7: Die $s-Register dienen der Haltung langlebiger Variablen. Sollen sie in einem Unterprogramm geändert werden, so müssen sie zuvor gesichert werden.
- $gp: Globaler Zeiger auf die Mitte eines 64K großen Speicherblocks im statischen Datensegment, das für alle Dateien sichtbar ist.
- $fp: *Framepointer*.
- $sp: *Stackpointer*, aktuelle Adresse des Stack-Beginns.
- $ra: Returnaddress, Rücksprungadresse nach dem Aufruf von **jal** (**jump**) und anderen Befehlen.

2.1.1 Spezielle ALU Register

General Purpose Register:

- Die $k-(Kernel)-Register werden bei der Unterbrechungsbehandlung verwendet.
- Das $at-Register wird von der ALU für Zwischenbrechnungen benutzt.
- Wie auch das $ra-(Return Adddress)-Register ist es volatil und kann vom laufenden Programm geändert werden.
- Die Register $a0 bis $a3 dienen überlicherweise zur Übergabe von Argumenten an Unterprogramme.
- Demgegenüber beinhalten die Register $v0 und $v1 entsprechende Rückgabewerte.

Register für den Programmablauf:

- pc *Program Counter* speichert die Adresse des gerade ausgeführten Befehls, um bei 'Sprüngen' wieder dorthin zurück zu kommen.
- hi und lo sind keine vom Bennutzer beschreibbaren 'General Purpose Register', sondern werden für arithmetische Ergebnisse (Multiplikation/Division) benutzt, bei der das Ergebnis nicht als 32 Bit Integer darstellbar ist: Bei Multiplikation wird das 64 Bit Ergebnis in einen oberen und unteren 32 Bit Bereich aufgeteilt; bei Division wird der Quotient und der Teilungsrest (remainder) getrennt abgelegt. Diese Register können nur mittels spezieller arithmetischer Instruktionen belegt und und deren Inhalt mit speziellen Instruktionen entnommen werden.

Register Name	Nr.	Nutzung	Bemerkung
$zero	0	enthält Konstante '0'	read-only
$at	1	temporäres Assemblerregister	nur intern vom Assembler genutzt
$v0	2	Funktionsergebnisse	für Zwischenergebnisse
$v1	3	1 und 2	und Ausgabe
$a0	4	Argumente	
$a1	5	1 bis 4	
$a2	6	für den	
$a3	7	Prozeduraufruf	
$t0	8	temporäre Variablen 1 bis 8	können von aufgerufenen
$t1	9		Prozeduren gesetzt werden
$t2	10		
$t3	11		
$t4	12		
$t5	13		
$t6	14		
$t7	15		
$s0	16	langlebige Variablen 1 bis 8	dürfen von aufgerufenen
$s1	17		Prozeduren nicht verändert werden
$s2	18		
$s3	19		
$s4	20		
$s5	21		
$s6	22		
$s7	23		
$t8	24	temporäre Variablen 9 und 10	können von aufgerufenen
$t9	25		Prozeduren gesetzt werden
$k0	26	Kernel-Register	Reserviert für das OS
$k1	27	1 und 2	benötigt beim Task-Wechsel
$gp	28	'Global' Pointer	auf Datensegment
$sp	29	Pointer auf Stack	Zeiger auf erstes freies Element
$fp	30	Pointer auf Frame	Zeiger auf aktiven Prozedur-Frame
$ra	31	Return Adresse	enthält nach Aufruf des Befehls **jal** die Rücksprungadresse

Tabelle 2.1: Liste der Register und ihre Nutzung/Bedeutung

2.2 FPU: Fliesskommma-Register

Die 32 *General Purpose Register* des CoProc1 (auch FPU Register genannt; daher $f*) besitzen ebenfalls eine Breite von 32 Bit.
Zur Hinterlegung von *double precision* Werten, können aber jeweils zwei zusammen gefasst werden (*konkatiniert*): ($f0 || $f1) → $f0/$f1 [Abb. 2.1].
Einige Register besitzen besondere Bedeutung:

- $f0/$f1: Diese werden als Register für die *Benutzereingabe* (→ $a0 bis $a3 bei ALU) genutzt.
- $f12/13: Dienen als *Ausgaberegister* (→ $v0 und $v1 bei ALU) für die Berechnung.

2.2.1 Foating Point Control Register

Pro FPU existieren weitere fünf sog. *Floating Point Control Registers* **FCR**:

- FIR *Floating Point Implementation Register*: Read-Only zur Festlegung der Floating Point Eigenschaften der betreffenden MIPS CPU.

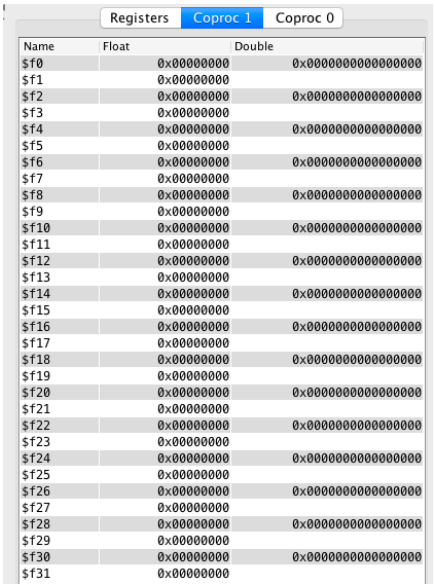

Abbildung 2.1: Anzeige der dual-purpose CP1 Register im MARS Emulator

- FCSR *Floating Point Control and Status Register*: Festlegung der Rundungsverhalten, Traps und IEEE 754 Exception-Verarbeitung.
- FCCR *Floating Point Condition Code Register*: Status der IEEE 754 Bearbeitung, wie *Rundung*, *Denormalisierung* und *Exceptions*.
- FEXR *Floating Point Exceptions Register*: Ergänzende Angaben zum FCSR im Falle von Exceptions mit *Cause* und *Flag* Informationen.
- FENR *Floating Point Enables Register*: Ergänzende Angaben zum FCSR im Falle von Rundungen.

2.3 CP0: Control Register

Die Betriebsmodi der MIPS-CPU sind von der Nutzung einiger CoProc0 (CP0) Register abhängig, von denen der Supervisor-Mode optional implementiert werden kann:

Mode	**Bedingung**	**Auswirkung**
Debug	EJTAG Register; Bit DM = 1 in Debug-Register	Zugang zu allen CPU Ressourcen
Kernel	Bits KSU = 0xb00 in CP0 Status Register; EXL + ERL = 1 im Status Register	Power up Mode, Exception liegt vor
Supervisor	DM = 0 in Debug Register; KSU = 0xb01; ESL + ERL = 0 in Status Register	Implementierung optional
User	Bit DM = 0 in Debug Register; Bits KSU = 0xb10 sowie EXL + ERL = 0 in Status Register	keine Exceptions, Übergang von Kernel durch ERET Aufruf

Tabelle 2.2: Betriebsmodi beim MIPS und die Belegung der CP0-Register

2.3.1 Control Register im Kernel-Mode

Der CoProc0 verfügt ebenfalls über 32 Register [Tab. 2.3], die neben der Behandlung von Exceptions sog. *Application Specific Extension* **ASE** unterstützen:

- *Virtual Process* (VP), bzw. *Virtual Process Entry* (VPE) → Prozess-Management
- *Thread Control* (TC) → Kernel-Thread-Unterstützung
- *Translation Lookaside Buffer* (TLB) → Virtual Memory Management Support

Die CP0-Register [Tab. 2.3] sind – mit Ausnahme der rot hervorgehobenen – nur in den privilegierten Modi les- bzw. beschreibbar.

No.	Name	Bedeutung
$0	VP Control*	TLB und Multithreading Unterstützung
$1	VPE Control*	Virtual Process Unterstützung
$2	Thread Control*	Thread Control
$3	EntryLo1	TLB Unterstützung
$4	Context	Pointer zur Page Table im Memory
$5	Page	Support für variable und kleine Pages
$6	SRS Conf	Shadow Register Configuration
$7	HWREna	Unterstützung Wahl der Hardware Register
$8	BadVAddr	(Virtuelle) Speicheradresse des ungültigen Zugriffs
$9	Count	CPU Zyklus Zähler
$10	EntryHi	Hi-order Anteil des TLB Eintrags
$11	Compare	Timer Interrupt Kontrolle
$12	Status	Prozess Status und Kontrolle
$13	Cause	Art der Unterbrechung und noch zu bearbeitende Unterbrechungen
$14	EPC	Exception PC: Inhalt des *Program Counter*
$15	PRId/EBase	CPU Identifikation. Exception Vektor Base Register
$16	Config	Configurations Register
$17	LLAddr	Load linked address
$18	WatchLo	Watchpoint address
$19	WatchHi	Watchpoint control
$20		XContext in 64-bit implementations
$21		Reserved for future extensions
$22		Available for implementation dependent use
$23	EJTAG	EJTAG Debug Register und Trace control
$24	DEPC	Program counter at last EJTAG debug exception
$25	Perfcnt	Performance counter interface
$26	ErrCtl	Parity/ECC error control and status
$27	CachErr	Cache parity error control and status
$28	Cache	Low-order portion of cache/data tag interface
$29	Tag/Data*	High-order portion of cache/data tag interface
$30	ErrorEPC	Program counter at last error
$31	DESAVE	EJTAG debug exception save register

Tabelle 2.3: Die CopProc0 Register $0 bis $31; die mit einen *) versehen Prozesse besitzen keinen spezifischen Namen

Im Gegensatz zu den 'einfachen' ALU- und CP1-Registern, ist der Aufbau der CP0-Register deutlich inhomogener, wie der Aufbau des `Status`- und `Cause`-Register in Abb. 2.2 verdeutlicht.

2.3.2 Exception Register

Der CoProc0 verfügt über mehrere Register, die die Behandlung von *Exceptions* unterstützen:

No.	Name	Bedeutung
$8	BadVAddr	(Virtuelle) Speicheradresse des ungültigen Zugriffs
$12	Status	Unterbrechungsmaske und Interrupt-Bits
$13	Cause	Art der Unterbrechung und noch zu bearbeitende Unterbrechungen
$14	EPC	Exception PC: Inhalt des *Program Counter*, wie er bei der Unterbrechung auftrat

Tabelle 2.4: Die CopProc0 Exception Register

```
1 # Sichern des PC:
2
3 mtc0 $pc $14  # Atomare Operation
```

Listing 2.1: Kopieren des ALU Program Counters auf das CP0 EPC Register

↪ Die Sicherung der ALU Register im CP0 Register ist immer eine *atomare Operation*; im Gegensatz zur Ablage der Registerinhalte im Hauptspeicher.

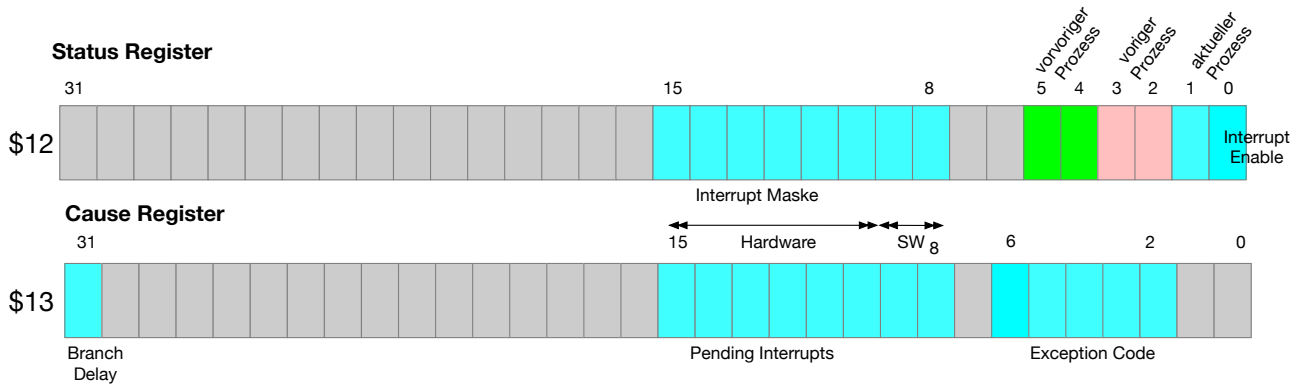

Abbildung 2.2: Aufbau/Nutzung des Status- und Cause-Registers

2.3.3 Cause Register

Im Cause Register $13 [Abb. 2.2] werden nicht nur in den Bits 6 bis 2 die Ursache der Unterbrechung protokolliert [Tab. 2.5], sondern auch noch die sog. *Pending Interrupts*, d.h. Unterbrechungen, die noch abzuarbeiten sind.

Bits 6-2	Name	Exception Ursache
0	INT	(externer) Interrupt
4	ADDRL	Fehler beim Adressenladen (**load**)
5	ADDRS	Fehler beim Speichern (**store**)
6	IBUS	Busfehler beim Laden eines Befehls
7	DBUS	Busfehler beim Laden oder Speichern von Daten
8	SYSCALL	Befehl **syscall**
9	BKPT	Breakpoint
10	RI	Reservered Instruction (priviligierter Mode)
12	OVF	Overflow/math. Überlauf

Tabelle 2.5: Exception Code Bits und ihre Bedeutung

2.3.4 Status Register

Im *Status* Register $12 können bis zu 3 Unterbrechungen protokolliert werden, wobei die letzte Ursache dann im *Cause* Register abgelegt wird [Abb. 2.2].
Die beiden Bits, die den Prozessen zugeordnet sind [Tab. 2.6], haben folgende Bedeutung:

- Bit 1: *Interrupt Enable*: Unterbrechungen sind zugelassen oder nicht (1 / 0).
- Bit 2: *Kernel/User*: Programm läuft im Kernel-/User-Mode (1 / 0).

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

Tabelle 2.6: Belegung des Status Registers

3 Organisation des Virtuellen Speichers

MIPS verfolgt ein lineares Speichermodell, das an Unix angelehnt ist. Zudem wird der virtuelle Speicher – wie in der von-Neumann-Architektur vorgesehen – gemeinsam für die Datenhaltung und die Instruktionen benutzt: *Datensegment*, *Textsegment*.
Der Aufteilung des 4 GByte grossen virtuellen Speichers zeigt Abb. 3.1:

- Die Speicheradressen `0x0040 0000` bis `0x7fff fffc` sind für den *User* vorgesehen.
- Die Speicheradressen `0x8000 0000` bis zum obersten Wert `0xffff ffff` sind für den *Kernel* reserviert.

Somit stehen für den MIPS-Benutzer knapp 2 GByte adressierbare Speicher zur Verfügung [Abb. 3.1]. In allen Fällen wird der virtuelle Speicher

- für die 'kleinen' Adressen als Textsegment für die Instruktionen benutzt und
- bei den 'höheren' Adressen als Datencontainer für Konstanten, Datenbestände variabler Grässe sowie den Stack als 'Daten-Parkplatz' für abzulegende temporäre Daten und auch Return-Adressen.

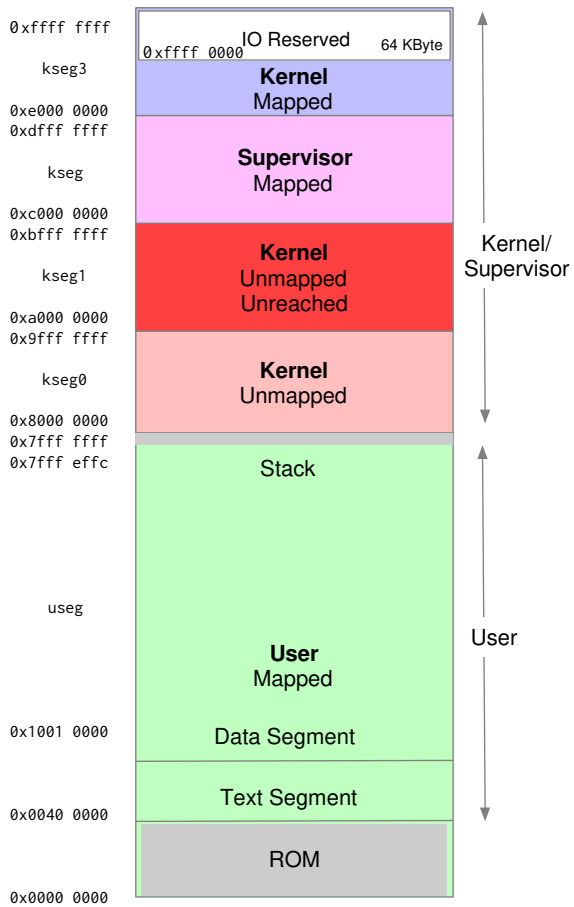

Abbildung 3.1: Organisation den virtuellen Speicher beim MIPS

3.1 Direktiven zur Hauptspeicher-Zuweisung

Die beiden wichtigsten Direktiven lauten:

- .data – Die (nachfolgenden) Daten werden im Data-Segment des Hauptspeichers abgelegt. Werden hier Befehle eingestellt, werden diese ignoriert.
- .text – Die (nachfolgenden) Daten und Instruktionen liegen im Textsegment.

Ergänzend hierzu gibt es für die Ausnahmebehandlung, wo zu einer festen Stelle im Hauptspeicher gesprungen wird, der Adresse 0x8000 0080, dem Beginn des reservierten Kernel-Textsegmentes.

- .ktext – 0x8000 0080 *Funktionen*, die für die Ausnahmebehandlung benötigt werden, müssen im Anschluss an dieses Statement deklariert werden und werden an der angegebenen Speicheradresse eingeblendet.
- .kdata – 0x8000 0080 *Daten*, die bei der Ausnahmebehandlung zur Verfügung stehen sollen, sind anschliessend zu deklarieren.

Zur Unterstützung dieser Aufgaben stehen die Kernel-CPU-Register $k0 und $k1 zur Verfügung und zusätzlich die Register des mathematischen Co-Prozessors 0 ($14 (epc) exception, $13 (cause), $12 (status) $8 (vaddr)).

3.2 Data-Segment: Direktiven zur Speicherplatz-Formatierung -und Bereitstellung

Die wichtigsten Direktiven zur *Typisierung* von Datenstrukturen im Data-Segment lauten:

- .ascii – Start der Adresse einer Zeichenkette (Feld) im Speicher.
- .asciiz – Adresse der Zeichenkette mit abschliessendem 0.
- .byte – Nachfolgende Zahlen besitzen einen 8-Bit Wert.
- .double – Nachfolgende Zahlen besitzen einen ein 64-Bit Wert.
- .float – Nachfolgende Zahlen sind 32-Bit Fliesskomma-Zahlen.
- .half – Nachfolgende Zahlen besitzen einen 16-Bit Wert.
- .word – Die (nachfolgenden) Zahlen besitzen einen 32-Bit Wert.

Bei diesen Direktiven wird der angeforderte Speicherplatz zusätzlich initialisiert. Ergänzend gibt es noch die Direktive

- .space n – Bytes im Speicher ohne besondere Formatierung/Struktur

und auch ohne Initialisierung!

3.2.1 Daten-Alignment und Datenorganisation

In der Regel ist die Organisation des Datensegmentes *Byte-aligned*; d.h. es wird implizit angenommen, dass in Vielfachen eines Bytes vorgenommen wird:

- .word – Datenworte starten immer an einer Byte-Boundary-Adresse, die durch vier teilbar ist.
- .half – Die Byte-Boundary-Adressen von Halbworten sind durch zwei teilbar.
- .double – Die Byte-Bondary-Adressen von 64-Bit Worten sind durch acht teilbar.

Auf das *Alignment* dieser Datenstrukturen kann mittels des

- align n

Operators Einfluss genommen werden, wobei n eine Potenz von 2 ist. Die Gültigkeit der **align** Anweisung beschränkt sich auf das aktuell deklarierte Datensegment.

3.3 Labels und die Symboltabelle

Labels werden vom Assembler in in einer Symboltabelle geführt und bezeichnen einfach die Adresse des Labels. Äquivalent zu den Labels im Textsegment können Labels auch für die Markierung von Speicherplätzen im Data-Segment genutzt werden.
Somit können Labels für zwei grundlegende unterschiedliche Operationen genutzt werden [Abb. 3.2]:

1. Als *Index* für Datenstrukturen im Data-Segment (→ Pointer).
2. Als (relative) *Sprung-Markierung* im Textsegment.

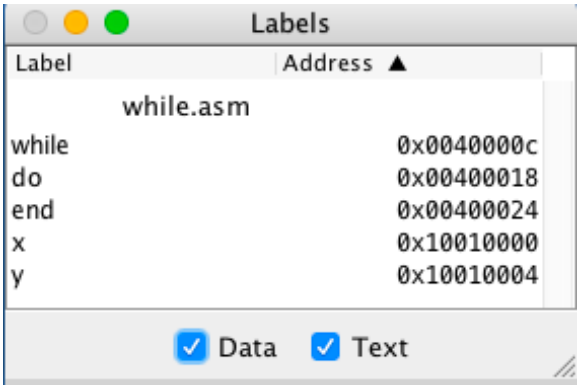

Abbildung 3.2: Symboltabelle für das Text- und Data-Segment

3.3.1 Labels im Data-Segment

Im Data-Segment kann der Speicherplatz von Direktiven mittels eines Labels ausgezeichnet werden:

```
1 .data
2 x: .word
3 y: .word 3, -1, 5, 8 , 32, 21, 79, 14, 99
```

Listing 3.1: Labels zur Auszeichnung von Speicheradressen im Data-Segment

↪ Ein Label stellt keine 'Variable' dar: Es findet weder eine Deklaration, noch Initialisierung des betreffenden Speicherplatzes statt:

1. Erst mit der Direktive, wird der (nachfolgende) Speicherplatz deklariert (Zeile 2 in Lis. 3.1).
2. Anschliessend kann mit der Befüllung der im Data-Segment eine Initialisierung vorgenommen werden (Zeile 3 in Lis. 3.1).

Um dem numerischen Wert eines Labels zu erhalten, kann mittels des **la**-Befehls

```
1 la $a0, label
```

Listing 3.2: Beziehen der Speicheradresse eines Labels und Ablage in $a0

Zugriff auf die Symboltabelle genommen werden, woraus der numerische Wert der Adressen-Markierung entnommen und in einem (beliebigen) Register gespeichert werden kann. Aus Abb. 3.2 geht hervor, dass der Adresswert für '`label`' selbst ein 32 Bit-Wert ist, mit der der gesamte Adressbereich erreicht werden kann. Die numerische Repräsentanz von '`label`' im Kommando **la** – was ja selbst nur eine Grösse von 32 Bit aufweist – muss aber kleiner sein. Daher ist die **la** Anweisung als Pseudoinstruktion hinterlegt und wird in zwei Schritten ausgeführt:

```
1 # Implementierung der Pseudobefehle 'la' und 'li'
2 #la rd, label :=          # label := upper.lower
3 lui rt, upper             # (load upper immediate) upper
4 ori rd, rt, lower         # (or immediate) lower; ori von lower mit rt = rd
5
6 #li rd, lower             # = ori, rd, $0, lower
```

Listing 3.3: Umsetzung des Pseudobefehls 'la' in zwei Schritten

Wird statt **ori** der Pseudobefehl '**li**' (= *load immediate*) genutzt, werden allerdings die obersten 16 Bit geleert, da **li** das Register $zero als `rt` nutzt. Daher seien die wichtigen Befehle mit ihrer Syntax und ihrem Einsatz kurz erläutert:

Befehl	Argumente	Arbeitsweise	Beispiel
la[PB]	`rd, label`	Ermittle Speicheradresse von `label` und speichere in rt	**la** `$a0, info`
lui	`rd, immediate`	Lade 16 Bit `immediate` als obere 16 Bit in Register rd und fülle mit 0 auf	**lui** `$t0, 25`
li[PB]	`rd, immediate`	Lade 16 Bit `immediate` als untere 16 Bit in Register rd; fülle die oberen mt '0'	**li** `$t0, 0x7ffe`

Tabelle 3.1: Load Address, Load Immediate und Load Upper Immediate Instruktionen; mit [PB] ist der sog. *Pseudobefehl* gemeint; d.h. der Assemblerbefehl setzt sich aus elementaren Instruktionen zusammen

3.3.2 Labels im Textsegment

Labels im Textsegment dienen ebenfalls zur Auszeichnung von Instruktionen [Abb. 3.2]. Sie können zur Strukturierung des Codes, aber im besonderen als Sprung-Markieren genutzt werden:

a) Bei bedingten Verzweigungen (*branch*), die einem Test folgen, sowie
b) bei unbedingten Verzweigungen, d.h. einem *jump*: `j end`.

Die Details dieser beiden Instruktionen wird weiter unten erläutert.

3.4 Stack & Heap

Zur Verarbeitung grösserer, temporären Datenmengen benötigen wir zusätzliche Speicher: **Stack & Heap**. Hierzu schauen wir uns das Speicher-Layout bzw. die Speichereinteilung nochmals an.

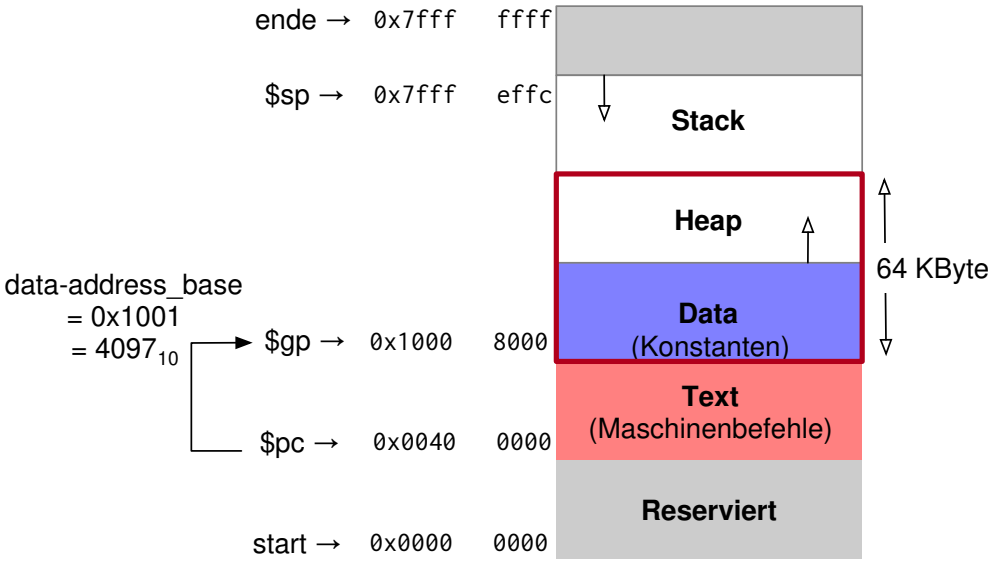

Abbildung 3.3: Layout des Speichers bei der MIPS-Architektur

Hier gibt es zwei 'leere' Bereiche, die genutzt werden können:

- Der *Stack* beginnt ab der Adresse `0xffff fffc` und kann als 'Kellerspeicher' von oben nach unten genutzt werden. Der Beginn des *Stack* wird im Register $sp (*Stack Pointer*) festgehalten.
- Der *Heap* ist ebenfalls ein 'Speicher' für grössere Datenmengen und schliesst sich an das Data-Segment an. Der Beginn des *Data-Segmentes* wird im Register $gp (*Global Pointer*) vermerkt. Nach dem 'Ende' der hier definierten Daten beginnt der *Heap*.

3.4.1 Nutzung des Stacks

Zur Sicherung temporärer Variablen kann der Stack in Richtung niedriger Adressen befüllt werden:

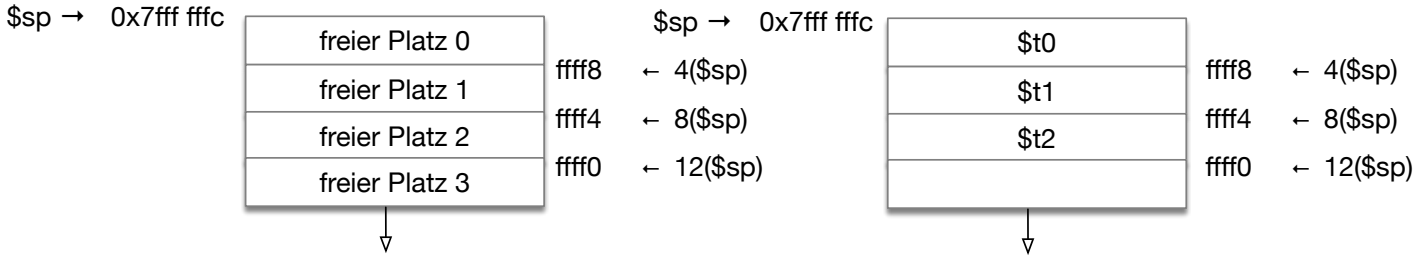

Abbildung 3.4: Aufbau und Nutzung des Stacks

Hierbei ist der *Stack Pointer* $sp ein Wert, der üblicherweise auf den Beginn des Stacks verweist. Operation auf den Stack ergeben sich durch das Inkrementieren der von $sp (um ein Vierfaches):

```
1  lw   $t3, 12($sp)   # $t3 erhält den dritten Wert vom Stack
2  lw   $t2, 8($sp)    # $t2 erhält den zweiten Wert vom Stack
3  lw   $t1, 4($sp)    # $t1 erhält den ersten  Wert vom Stack
4  addi $sp, $sp, 1    # $sp inkrementieren
```

Listing 3.4: Laden von Daten aus dem Stack

↪ Der *Stack Pointer* $sp enthält die Adresse des Stack und ist daher tatsächlich ein 'Pointer'. Der *Stack Pointer* ist aber 'normal' beschreibbar und kann daher sowohl inkrementiert als auch dekrementiert werden.

3.4.2 Der Stack zur Datensicherung von Registerinhalten

Zur Sicherung der Inhalte von Registern (die Daten oder aber auch Adressen beinhalten können) bieten sich an:

- Das *Datensegment*: Dies beginnt üblicherweise ab der Adresse `0x1001 0000` und wird von 'unten nach oben' befüllt. Das Data-Segment dient zur Bevorratung langlebiger, benannter Daten und gestattet, diese zu *allokieren* und und *typisieren*.
- Der *Stack* ist eine Rumpelkammer (verfügbar von Adresse `0x7fff fffc`), in der Daten hinterlegt werden können; aber gelegentlich aufgeräumt werden muss.
- Der *Stack* setzt keine *Typisierung* der Daten voraus; wir müssen uns aber merken, so wir die letzten eingelagerten Daten untergebracht haben.

Zur Verwaltung der Daten im Stack besitzt die MIPS-Architektur folgende Register:

- Der *Stack Pointer* $sp ($31), der auf den nächsten freien Platz verweist.
- Der *Frame Poiner* $fp ($30), der auf das aktuelle Frame weist, wo die genutzten *Register* der Prozedur gesichert sind.

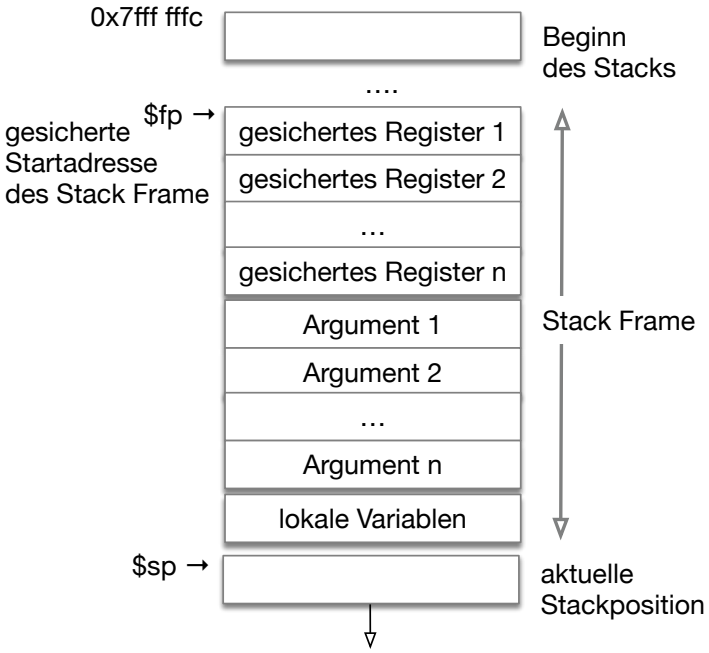

Abbildung 3.5: Aufbau und Nutzung eines Stack Frames

3.4.3 Nutzung des Heap

Der MIPS Assembler kann aber nicht nur dem ihm zugewiesenen Datenplatz im Memory verwalten, sondern auch ergänzenden Speicherplatz vom Betriebssystem anzufordern [Abb. 3.3]:

```
1  .text
2  main:
3
4    addi $a0, $zero, 4 # $a0 fordert 4 Byte Heap an
5    li $v0, 9          # Anforderung Heap 9 -> sbrk
6    syscall            # dynamische Anforderung von Speicher
7    sw $t0, $v0        # $v0 enthaelt Adresse des verfuegbaren Speichers im Heap;
8                       # $t0 wird auf Adresse $v0 abgelegt
9
10   syscall
```

Listing 3.5: Anfordern von Heap-Speicher über den Betriebssystem-Call

- Die Grösse des angeforderten Speicherplatzes wird durch den Inhalt des Registers $a0 mitgeteilt.
- Der Systemaufruf **sbrk** gibt die Adresse im Speicher zurück, die nun genutzt werden kann; z.B an Position `0x1004 0000`,
- An diese Speicherplatzadresse können nun temporäre Daten persistent hinterlegt werden.

↪ Dynamische Speicherplatz-Anforderung; vergleichbar mit **malloc()** in 'C'. Im Gegensatz zu 'C' gibt es aber keine Möglichkeit, den Speicherplatz wieder frei zu geben

3.4.4 Referenzierung von Speicheradressen mittels Labels

Wir berücksichtigen, dass eine MIPS-Instruktion genau eine Wortlänge besitzt (also 32 Bit) und die Register mit jeweils 5 Bit Länge angesprochen werden können (weil nur 32 Registeradressen vorhanden sind und adressiert werden müssen).
Im Rahmen der Load&Store-Architektur muss natürlich auch mit Speicheradressen gearbeitet werden, was voraussetzt, dass die 32 Bit Adresse in der Instruktion übertragen werden muss, was natürlich nur mit 'gekappten' Adressen möglich ist, die zu ihrer vollständigen 32 Bit Grösse einen 'Offset' benötigen. In Lis. 3.5 und Tab. 4.2 wurde gezeigt, welchen 'Trick' man sich hierbei bedient.
Speicheradressen können beim MIPS auf unterschiedliche Arten in Instruktion mitgeben werden:

- Über den Inhalt eines Register, z.B. $t0 → voller 32 Bit Bereich.
- Als i-tes Element (Index: $t0 = i) einer Direktive relativ zu einem Label im Data-Segment: `x($t0)`.
- Als fixer Offset (16) zu einer Direktive gekennzeichnet durch einem Label im Data-Segment: `16 + y`.
- In Kombination eines über ein Label gekennzeichneten Labels mit anschliessendem Offset: `16 + y($t0)`.

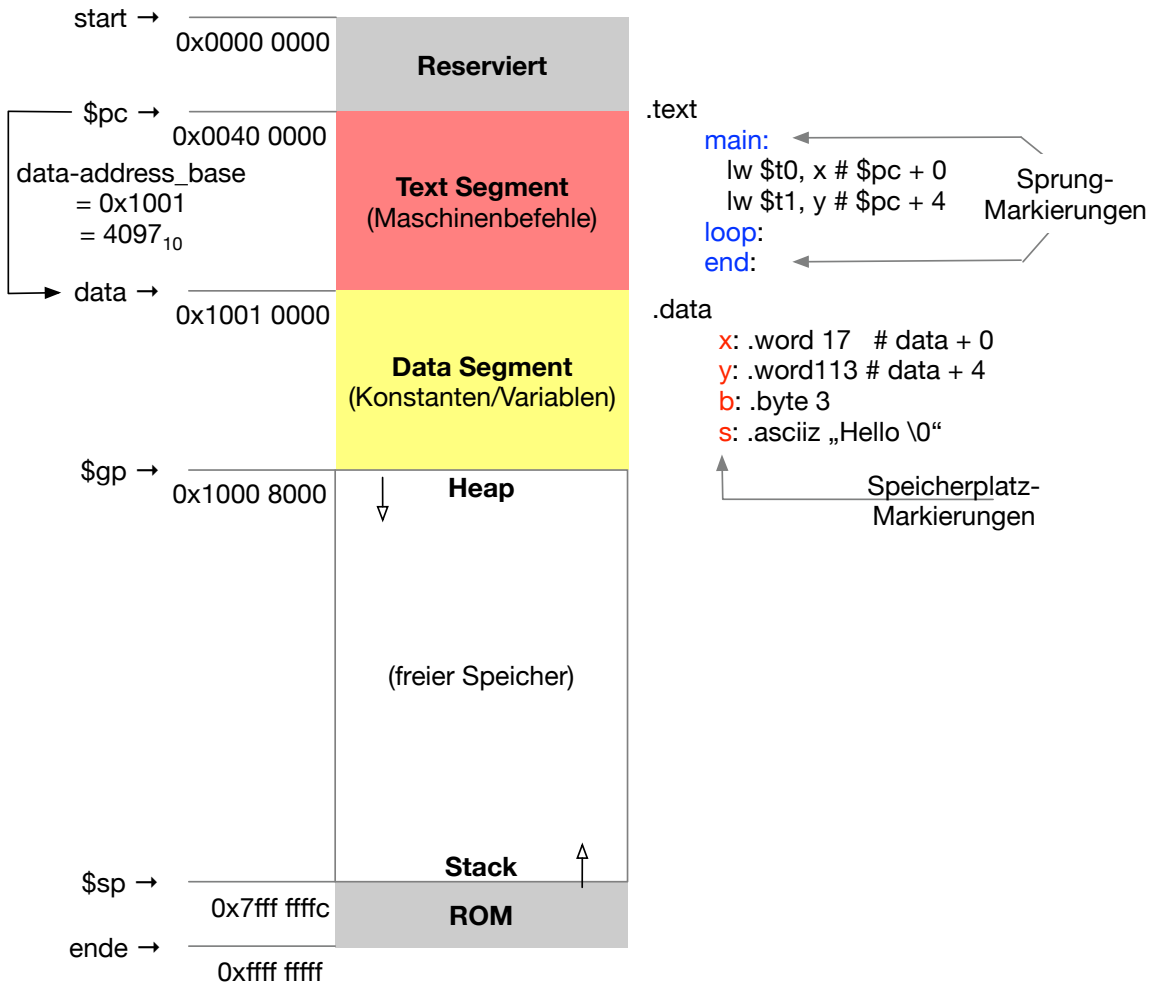

Abbildung 3.6: Aufbau des Speichers beim MIPS und Nutzung von Labels (in umgekehrter Darstellung)

3.4.5 Referenzierung von Speicheradressen im Stack

Der Stack ist so organisiert, dass die Adressen von hohen zu niedrigeren Adressen *abnehmen*. Der Stackpointer $sp bezeichnet den Start des Stacks und kann in Einheiten von einem Byte adressiert werden:

- Ist entsprechend [Abb. 3.3] $sp = `0x7fff fffc`, so
- ist 4($sp) = `0x7fff fff8`, also eine *niedrigere* Adresse.

↪ Wird der Stack von einer Prozedur genutzt, ist der Stackpointer $sp auf den niedrigsten – nun freien - Platz zu setzen, damit die 'höheren' Adressen bzw. die Adress-Inhalte beibehalten werden.

4 MIPS-Instruktionen

Wie wir bereits mit MIPS-Aufbau gesehen haben, werden zumindest drei CPUs eingesetzt, wobei die ALU und die FPU (CoPorc 1, CP1) für den 'normalen' User bereitstehen und der CoProc0 (CP0) für Verwaltungs- und Steuerungsaufgaben vorgesehen ist. Alle diese drei CPUs können per Instruktion angesprochen werden; allerdings nicht in jedem *Betriebsmode*. Der Befehlssatz beim MIPS ist beschränkt; die ALU beherrscht im wesentlichen

- arithmetische Operationen,
- logische Operationen,
- Bitshift-Operationen sowie
- Operationen zum Laden und Speichern von Registerinhalten im Hauptspeicher RAM (und ungekehrt).

Wir sprechen deshalb auch von eine CPU mit beschränktem Befehlssatz, bzw. einem Reduced Instruction Set Computer **RISC**.

4.1 Typen der MIPS-Instruktionen

Zur Gewährleistung der Flexibilität in der Verarbeitung [Abb. 3.5] kennt MIPS drei unterschiedliche *Instruktionsformate*:

- **R**-*Format*: Operationen für CPU/ALU mit Registerangaben.
- **I**-*Format*: Statt Registerangabe kann ein Immediate-Operanden angegeben werden.
- **J**-*Format*: Jump-Befehle zur konditionalen Befehlsverarbeitung der ALU.

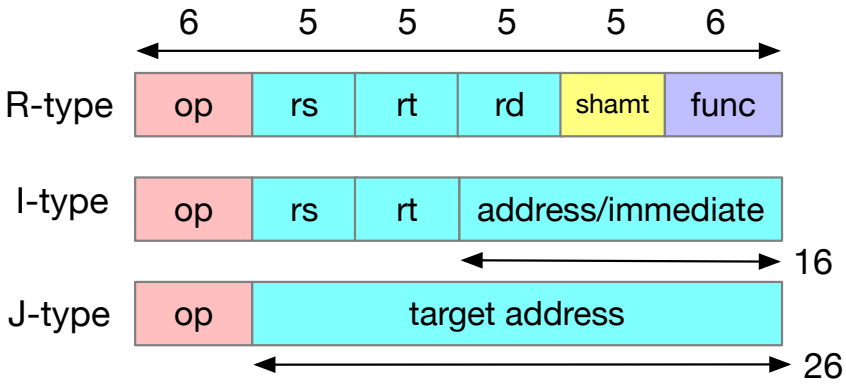

Abbildung 4.1: Die drei Instruktions-Typen der MIPS Architektur; op: `OpCode` [Abb. 4.2], func: `FuncCode` [Abb. 4.3], shamt: `Shift Amount`. Bei der R-Instruktion ist `rd` das Zielregister und `rs` sowie `rt` die Quellregister, bei der I-Instruktion wird `rt` als Ziel- und `rs` als Quellregister genutzt.

Generell gilt:

- Instruktionen sind immer 32 Bit Worte.
- Der Anfang der Instruktion macht immer ein 6 Bit OpCode.
- Es stehen 32 Operandenregister zur Verfügung, die als 5 Bit Wert angesprochen werden.
- Achtung! Beim Erzeugen des binären des Assembler-Code wird zuerst das Zielregister `rd` bzw `rt` und dann die Quellregister `rs` bzw. `rt` angegeben!

Die ersten beiden Bits im **OpCode** werden von der ALU unmittelbar ausgewertet:

00 Addition der gegeben Werte (**addi**, **lw**, **sw** ...).
01 Subtraktion der Werte (**beq**, **bne** ...).
00 Bei R-Typen mit allen OpCode Bits auf '0' wird die Anweisung über den *FuncCode* mitgeteilt.

- Das 32-Bit **Instruktionswort** beinhalten den *OpCode* und als *Argumente* die Adressen der Register, in denen die zu verarbeitenden Daten abgelegt sind. Da nur 32 Register verfügbar sind, reichen 5 Bit aus, was maximal 15 Bit für weitere Parameter bei drei vorhandenen Adressen ergibt.
- Ausnahme sind ***immediate*** *Instruktionen* (wie z.B. **addi**). wo statt Adressen die eigentlichen (max. 16 Bit grossen) Daten (als Parameter) angegeben werden.

4.1.1 Input- und Output-Register in der Instruktion

Bei der Beschreiung des Instruktionswortes werden für die Namen der Register folgende Konventionen verwendet:

- Das Register, das nach dem OpCode folgt wird `rs` genannt und ist immer ein 'Input'-Register.
- Das folgende Register ist als `rt` bezeichnet. Bei I-Instruktionen ist dies das 'Output'- bzw. Ziel-Register.
- Bei R-Instruktionen wird das dritte (nun 'Output'-Register) `rd` benannt.

↪ Bei der Programmierung der Instruktionen ist das erste Argument immer das Ziel-Register!

4.1.2 OpCode und FuncCode

Bei den Immediate- und Jump-Anweisungen kann die ALU die logische, arithemtische oder Sprung-Anweisung direkt den ersten sechs Bits des OpCode-Feldes entnehmen. Dessen Codierung folgt folgendem Schema:

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP1X*[1] δ	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	β	β	β	β	*SPECIAL2* δ	JALX ε	ε	*SPECIAL3*[2] δ⊕
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5	101	SB	SH	SWL	SW	β	β	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	β	LDC1	LDC2 θ	β
7	111	SC	SWC1	SWC2 θ	*	β	SDC1	SDC2 θ	β

Abbildung 4.2: Codierung des OpCodes [Quelle: MIPS32 Architecture For Programmers Volume II, Revision 0.95]

Bei den **R-Typ** Instruktionen liegt als `OpCode 0000 00` vor ('special'). Neben den zu verarbeitenden Registern muss der ALU noch mitgeteilt werden, welchen Befehl sie zu befolgen hat. Dies ist im 6-Bit-Feld `FuncCode` wie folgt untergebracht:

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL[1]	*MOVCI δ*	*SRL δ*	SRA	SLLV	*	*SRLV δ*	SRAV
1	001	JR[2]	JALR[2]	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3	011	MULT	MULTU	DIV	DIVU	β	β	β	β
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	β	β	β	β
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	β	*	β	β	β	*	β	β

1. Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP, EHB and PAUSE functions.
2. Specific encodings of the *hint* field are used to distinguish JR from JR.HB and JALR from JALR.HB

Abbildung 4.3: Der FunctionCode bei den R-Instruktionen [Quelle: MIPS32 Architecture For Programmers Volume II, Revision 0.95]

Die Format sowol des `OpCode` als auch des `FuncCode` ist für eine schnelle bit-weise Verarbeitung optimiert, was aus nachfolgender Tabelle entmommen werden kann:

Befehl	Code		Befehl	Code		Befehl	Code	
ADD	100	000	**MULT**	011	000	**AND**	100	100
ADDU	100	001	**MULTU**	011	001	**OR**	100	101
SUB	100	010	**DIV**	011	010	**XOR**	100	110
SUBU	100	011	**DIVU**	011	011	**NOR**	100	111

Tabelle 4.1: Beispiele für die Codierung des FuncCode für R-Instruktionen mit OpCode '000000'

4.1.3 R-Instruktionen

Der Opcode bei der R-Instruktion ist $00\ \ 0000_2$. Bei R-Instruktionen gilt folgendes:

- Zwei gelesene Register: `rs` und `rt`. Inhalt der Register geht zur ALU (RD1 und RD2).
- Das Resultat der ALU Berechnung wird über die Registeradresse **rd** zur Verfügung gestellt und nach `WD` kopiert.

Weitere Felder:

- `shamt`: '*shift amount*' wird für Verschieboperationen benötigt.
- `func`: *Function Code* zur Erweiterung des OpCode Feldes.

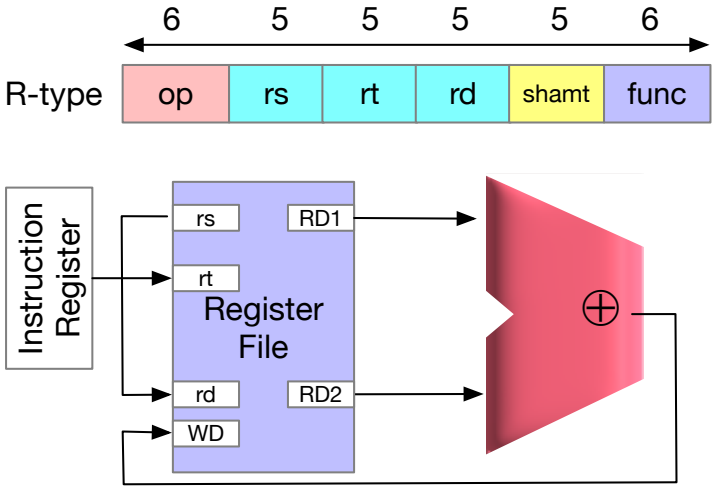

Abbildung 4.4: R-Instruktion und die Nutzung der Register; RD: read data; WD: write data

4.1.4 I-Instruktionen

Die I-(*Immediate*)-Instruktion besitzt zwei Arbeitsmodi:

1. Die direkte Übergabe eines Operanden (Wertes) an die ALU im Feld *immediate*, ohne auf ein Register zugreifen zu müssen.
2. Die Übergabe eines Wertes unter Angabe der Adresse im gemeinsamen Feld in *address/immediate*.

Die I-Instruktion besitzt folgende Merkmale:

- Ein Basis-Register: `rs`.
- Ein Ziel- oder Quell-Register: `rt`.
- Immediate-Operand mit 16 Bit Länge.

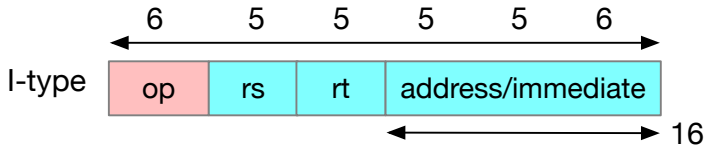

Abbildung 4.5: I-Instruktion, op: OpCode

Die I-Instruktion kann für die Addition (**addi**) aber auch für Load (**lw**) und Store (**sw**) Operationen unter Referenzierung von Adressen genutzt werden. Beispiele:

- `lui $1 4097` → *load unsigned immediate* des Wertes an Adresse `4097 = 0x1001 (0000)` (Start des Data-Segments) in `$1` [Tab. 4.2].
- `addi $s0, $zero, 17` → Schreibe Konstante `17` in Register `$s0`.

4.1.5 J-Instruktionen

Die J- bzw. *Jump-Instruktion* wendet sich ausschliesslich an die ALU und dient zur unbedingten (`goto`) oder bedingten (`if/then/else`) Steuerung des Programmablaufs:

- `OpCode` ist `00 0010`$_2$.
- `Target Address`: Adresse im Textsegment.

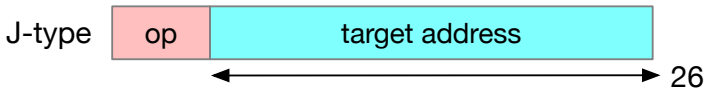

Abbildung 4.6: J-Instruktion; op: OpCode

Eigenschaften:

- Die hier angegebene Adresse besitzt nur eine Länge von 26 Bit, da 6 Bit für den OpCode benötigt werden.
- Die Zieladresse muss immer ein Vielfaches von 4 betragen, was als *Alignment* bezeichnet wird.
- Hierdurch liegt die oberste Sprungadresse bei `0x1000 0000` = 2^{28}.

4.2 Instruktionsformate im Maschinencode

Instruktionen und Programmausführung beginnen ab Adresse `0x0040 0000`:

- Beim *I-Type* ist die letzte Angabe die signed 16 Bit Zahl, die *immediate* zu verarbeiten ist.
- Beim *J-Type* folgt im Anschluss an den OpCode die 26 Bit *Jump Adresse*.

Format	Adresse	OpCode	rs-Register	rt-Register	rd-Register	shamt	FuncCode
	32 Bit	6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit
R-Type	0x0040 0000	000000	00000	11111	00001	00000	10 0000
R-Type	0x0040 0004	000000	11111	00000	00001	00000	10 0010
R-Type	0x0040 0008	000000	00000	00001	11111	00000	10 0010
I-Type	0x0040 000c	001000	00000	00111	1010 1111 1111 1110		
I-Type	0x0040 0010	101011	00001	01000	1010 1111 1111 0000		
J-Type	0x00400014	000001	00 0001 0000 0000 0000 0000 0000				

Tabelle 4.2: Aufbau von R-, I- und J-Instruktionen mit konsekutiver Folge von Befehlen im Textsegment

- Beim *R-Type* können zwei Input-Register (rs, rt) und das Output-Register (rd) zusammen mit einem ergänzenden **FuncCode** spezifiziert werden.

In einer geeignet aufbereiteten Binärdarstellung können wird Maschinencode auch 'lesen' und quasi dis-assemblieren. Tab. 4.2 liest sich dann zeilenweise wie folgt:

```
1  add  $at, $zero, $at # Opcode: 000000, FuncCode: 10 0000
2  sub  $at, $ra, $zero # Opcode: 000000, FuncCode: 10 0010 sub
3  sub  $ra, $zero, $at # Opcode: 000000, FuncCode: 10 0010
4
5  addi $a3, $0 45054   # Opcode: 001000; immediate = 0xaffe
6  sw $t0, Label        # Opcode: 101011; rs = $at; 16-Bit Label: 0xaff0
7
8  j LABEL              # Opcode: 000001; 28-Bit LABEL: 0x10 00000
```

Listing 4.1: Interpretation von Maschinecode

Bemerkung: Die **sw**-Anweisung wird zwei-stufig aufgelöst: zunächst **lui** zur Bestimmung des Offset (in `$at` abgelegt) und dann Abspeichern mittels **sw** des Register-Inhalts an der relativen Adresse ergänzt um den Offset.

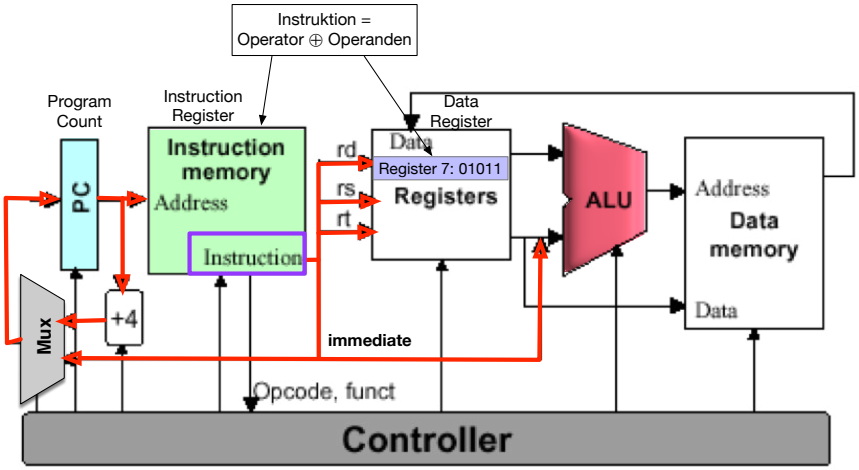

Abbildung 4.7: Die Instruktionen enthalten den OpCode und stehen im Instruction Register zur Verfügung

4.3 Pseudoinstruktionen

Als *Pseudoinstruktion* bezeichnen wir Instruktionen, die der Assembler versteht, aber mehrstufig in elementare MIPS-Operationen umsetzt. Der Pseudoinstruktionen machen den Code lesbarer und überlässt es dem Assembler, die fehlerfrei Umsetzung vorzunehmen.

Pseudoinstruktions Anweisungen kommen bei der Adress-Arithmetik aber aber bei elementaren Operationen, wie z.B. der Addition, Multiplikation und Division vor.

```
1  mulo $s0, $t0, $t1       # Pseudoinstruktion
2
3  mult $t0, $t1
4  mfhi $at
5  beq $at, $zero, 4
6  break $zero
7  mflo $s0
8
9  mulou $s0, $t0, $t1      # Pseudoinstruktion
10
11 multu  $t0, $t1
12 mfhi, $at, $0
13 beq $at, $zero, 4
14 mflo $s0
```

Listing 4.2: Auflösung von Pseudoinstruktionen

5 MIPS-Pipelineverarbeitung

MIPS steht für das Akronym '*Micropocessor without Interlocked Pipeline States*' was darauf hindeutet, dass sehr wohl die Varbeitung der Instruktionen in einer Pipeline über mehrere 'Stages', sprich Zwischenstufen, stattfindet, die allerdings unabhängig voneinander arbeiten, also nicht 'gelocked' sind.
Dieses Konzept wollen wir uns nun anschauen; und aus diesem Konzept heraus weist die MIPS-Architektur auch eine gewisse Eleganz und Einfachheit auf.

5.1 Fetch/Decode/Execute/Store-Zyklus

Der Verarbeitungsablauf eines Programms in der CPU/ALU folgt dem Schema [Abb. 1.2]:

IF	**Fetch**: Hole Instruktionen und Daten aus dem Speicher (RAM).
ID	**Decode**: Stelle fest (dekodiere), welche Instruktion auszuführen ist und bereite die Register vor.
EX	**Execute**: Hole Daten aus den Registern und führe die Instruktion/Berechnung durch.
Mem	**Mem**: Übertrage das Resultat in das RAM (optional).
WB	**Write Back**: Schreibe das Ergebnis in das Ausgaberegister (falls notwendig).

Die MIPS Architektur bildet diese Aufgaben in Hardware ab, was als **Stages** bezeichnet wird. Die Verarbeitungsdauer in einer *Stage* ist identisch, was durch Puffern der Daten in Flip Flops realisiert wird. Das Gesamtsystem wird durch eine gemeinsame Clock-Rate synchronisiert.
Die ersten drei Schritte werden auch als **FDX** (***F**etch/**D**ecode/e**X**ecute*) bezeichnet.

Der **Instruction Fetch**-Schritt umfasst:

- Holen der Instruktionen aus dem RAM (dem Textsegment),
- Überführung in das Instruction Register,
- Erhöhung des ALU *Program Count* Registers um den Wert $+4$ ($pc+4).

Der **Decode**-Schritt umfasst:

- Die Dekodierung der durchzuführenden *Instruktion*,
- Überprüfung, welche nächste Instruktion (*Program Count*) durchgeführt werden soll, falls dies von der gegebenen Reihenfolge abweicht (*Branch Prediction*),
- welche Operanden (also *Register*) genutzt werden sollen und
- Bereitstellung dieser.

Der **Execute**-Teil beinhaltet:

- Überführung der Registerinhalte an die ALU (oder Copro),
- Durchführung die *Instruktion* bzw. Adress-Berechnung,
- befüllt das *Flag-Register* mit evtl. Ergebnissen der Berechnung.

Der **Memory**-Teil beinhaltet:

- Lesen vom RAM (Data-Segment bzw. Stack)
- Schreiben in das RAM.

Der **Write Back**-Teil macht schliesslich:

- Überführen des Ergebnisses in das Zielregister.
- Bereitstellung der Daten aus dem RAM für die nachfolgende Operation.

5.2 MIPS Pipeline Architektur

Die MIPS Hardware ist so aufgebaut, dass die Verarbeitung in diesen Phasen immer gleich lang dauert, da wir es hier mit einem synchron getakteten Rechenwerk zu tun haben. Die Synchronisation wird per Flip-Flops (Latches) zwischen den Baugruppen erzielt, die die Daten puffern Abb. 1.1. Somit sind fünf Stages und vier Puffer von Bedeutung, sieht man einmal vom Instruktionsregister ab, dass über den *Program Count* (PC) Kenntnis über die nächste zu verarbeitende Instruktion besitzt.
Bedingt durch die fünf Stages, können die Verarbeitungsschritte hierin unabhängig wirken. Betrachten wir die Verarbeitungskette:

$$IF \rightarrow ID \rightarrow EX \rightarrow Mem \rightarrow WB$$

so heisst dass, dass in jeder *Stage* unterschiedliche Instruktionen vorliegen können:

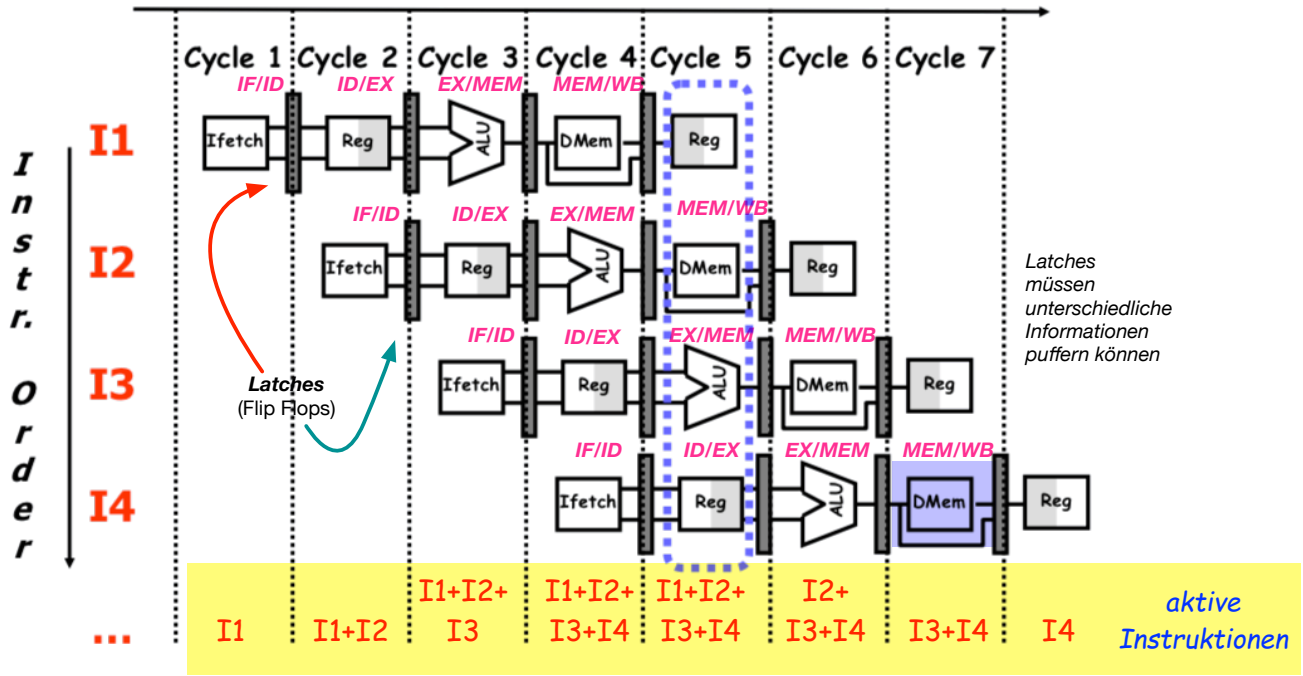

Abbildung 5.1: Instruktionsverarbeitung in der Pipeline: Mehrere Instruktionen sind simultan aktiv

6 Assembler: MIPS-Befehlsreferenz

In diesem Kapitel sollen die wichtigsten MIPS-Instruktionen – die die ALU betreffen – vorgestellt und erläutert werden. Hierbei ist folgendes zu beachten:

6.1 Syntax und Konventionen der Assembler-Befehle

In diesem Kapitel werden die MIPS-Instruktionen mit ihren Argumenten vorgestellt. Hierbei werden von einigen Konventionen Gebrauch gemacht, die im folgenden erläutert werden.

6.1.1 Assembler Syntax-Regeln

Beim Einsatz des Assemblers in der MARS-Umgebung gelten folgende allgemeinen Ausnahmen:

1. Die Benennung der Instruktionen ist *case-insensitive*; d.h. Gross- und Kleindarstellung der Instruktionen (nicht der Labels!) ist irrelevant für das Ergebnis.
2. Die Argumente der Instruktionen können – müssen aber nicht – mittels eines *Kommas* getrennt werden. Ein 'white space' Zeichen (Leerzeichen, Tabulator) ist ausreichend.

Es ist empfehlenswert, die Instruktionen in Kleinbuchstaben darzustellen und bei den Argumenten – zwecks Kompatibilität mit anderen MIPS-Assemblern – die Kommas zu setzen.

6.1.2 Benennung der MIPS-Register in Operationen

MIPS-Operationen beziehen sich in der Regel auf die Verknüpfung zweier Input-Register und mit dem Ergebnis im Output-Register. Wir verwenden hier entsprechend den MIPS-Originaldokumenten:

- (Logische) Input-Register: rs und ggf. rt
- (Logisches) Output-Register: rd oder auch rt

Wir werde dieses allgemeine Bennungsschema bei den anschliessend diskutierten MIPS-Operationen auf der ALU nutzen. Bei der FPU gilt folgendes:

- (Logische) Input-Register: fs, ft
- (Logisches) Output-Register: fd

Für alle anderen Koprozessoren nutzen wir:

- (Logische) Input-Register: cs, ct
- (Logisches) Output-Register: cd

6.2 ALU Load & Store

Daten zwischen dem Hauptspeicher und den Registern der ALU zu transferieren kann auf Wort- und Byte-Baseis vorgenommen werden:
Ein *un-aligned* Speicherzugriff auf Daten im Hauptspeicher ist mit folgenden Befehlen möglich:

Befehl	Argumente	Beispiel	Speicher-Lokation	Erläuterung
lw	`rd, Addr`	**lw** `$s1,100($s2)`	Memory-Adresse($s2+100) = $s1	Lade Wort aus einer Position im RAM in ein Register
sw	`rs, Addr`	**sw** `$s1,100($s2)`	Memory-Adresse($s2+100) = $s1	Speichere Wort aus Register ins RAM
lb	`rd, Addr`	**lb** `$s1,100($s2)`	$s1 = Memory-Adresse($s2+100)	Kopiere Byte vom RAM ins Register
sb	`rs, Addr`	**sb** `$s1,100($s2)`	Memory-Adresse($s2+100) = $s1	Speichere Byte aus Register ins RAM
lui	`rd, Addr`	**lui** `$s1,100`	$s1 = 100 * 2^{16}	Kopiere Konstante in die oberen 16 Bits

Tabelle 6.1: Load und Store Anweisungen für Datenwörter und Bytes für die ALU

Befehl	Argumente	Erläuterung	Erläuterung
lwl	`rd, offset(base)`	Load Word Left	lade den Inhalt der Speicheradresse `base+offset` in Register von `rd` von links nach recht
lwr	`rd, offset(base)`	Load Word Right	lade den Inhalt der Speicheradresse `base+offset` in Register von `rd` von rechts nach links
swl	`rs, offset(base)`	Store Word Left	speichere den Registerinhalt von `rt` an die Speicheradresse `base+offset` von links nach rechts
swr	`rs, offset(base)`	Store Word Right	speichere den Registerinhalt von `rt` an die Speicheradresse `base+offset`

Tabelle 6.2: Load Word und Store Word Instruktionen

Hierbei ist bei `lwl/lwr rt` das zu schreibende Register und `base` das zu lesende und `offset` ein vorzeichenbehafteter 16 Bit Wert.

6.3 ALU Register Transfer

Es besteht nicht nur die Möglichkeit, Instruktionen und Daten zwischen Hauptspeicher und Registern in beiden Richtungen zu transferieren, sondern auch die Dateninhalte zwischen den (Eingangs-)Registern zu kopieren, was teilweise mit Hilfe von Pseudobefehlen erfolgt. Die ALU verfügt zudem über die Spezialregister hi und lo, die für Multiplikation und Division genutzt werden.

Befehl	Argumente	Erläuterung
move[PB]	`rd, rs`	Kopiere Inhalt von Register `rs` ins Register `rd`
mfhi	**rd**	Kopiere Inhalt von Register **hi** nach **rd**
mflo	`rd`	Kopiere Inhalt von Register **lo** nach **rd**
mthi	`rs`	Kopiere Inhalt von Register `rs` nach **hi**
mtlo	`rs`	Kopiere Inhalt von Register `rs` nach **lo**

Tabelle 6.3: ALU Register Transfer-Befehle; *PB*: Pseudeobefehl

Der Pseudobefehl **move** ist denkbar einfach implementiert:

```
1 add $s0, $zero, $t0 # move $s0, $t0
```

Listing 6.1: Implementierung der Pseudoinstruktion 'move'

Achtung! **move** macht eine Kopie der Registerinhalte; das ursprüngliche Register bleibt unangetastet.

6.4 Load Address

Viele arithmetische Assembler-Operationen beziehen sich nicht auf die Werte in den Register, sondern auf Speicheradressen von

- *Feldern* im Data-Segment oder dem Stack, sowie
- *Markierungen* von Positionen im Textsegment; z.B. durch eine Label gekennzeichnet.

Zur Unterstützung letzter Operation kann auf zwei interne Informationen zurück gegriffen werden:

1. *Program Counter* (**PC**): Das PC-Register ($pc) enthält die Adresse der aktuellen Instruktion. Der Inhalt des PC-Registers spiegelt daher den dynamischen Ist-Zustand des Programmablaufs dar.
2. *Symboltabelle*: Diese wird vom Assembler angelegt, falls entsprechende Labels im Code vorgesehen sind. Diese Einträge dienen als Referenzen und sind statisch.

Zur Ablaufsteuerung eines Assembler-Programms werden beide Informationen eingesetzt:

- Labels dienen als Markierung in der Programm-Landschaft,
- Der PC beschreibt in Register `$pc` die aktuelle Position, also unseren Standort darin.

In der praktischen Umsetzungen reicht es, relative Adressen einzusetzen: *Wie weit bin ich von X entfernt? Wie viele Schritte muss ich bis nach Y gehen?*

↪ Wir bezeichnen dies als **Offset**.

Um die Speicheradresse von einem Label zu beziehen, kann der Pseudobefehl *Load Address* (**la**) genutzt werden:

Befehl	Argumente	Arbeitsweise	Beispiel
la	`label`	Lade Adresse des Labels; nicht dessen Wert	`la proc`

Tabelle 6.4: Load Address Pseudobefehl

Der **la**-Befehl »`la label`« ist folgendermassen umgesetzt:

```
1 lui $at, upper      # upper halfword of label
2 ori rd, $at, lower  # lower halfword of label
```

Listing 6.2: Implementierung der Pseudoinstruktion 'la'

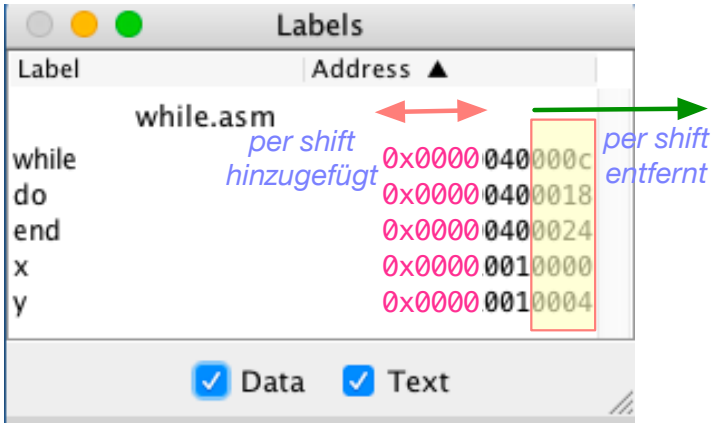

Abbildung 6.1: Symboltabelle → Wert für Label für **la**-Operation

Zugrunde liegen die I-Instruktion *Load Upper Immediate* (**lui**) verknüpft mit der Bit-Operation *Or Immediate* (**ori**):

Befehl	Argumente	Arbeitsweise	Erläuterung
lui	`rs`, Label	`rs` = Label shifted um 16 Bits nach Rechts aufgefüllt mit '0'	Load Upper Immediate
ori	`rd`, `rs`, Label	`rd` = `rs` ˇ Label mit ersten 16 Bit auf '0' maskiert	logisches OR mit Wert

Tabelle 6.5: Syntax und Arbeitsweise der **lui** und **ori** Instruktionen für die Pseudoinstruktion **la**

6.4.1 Verkürzte Adressen und Offsets

Bei der Nutzung von Speicheradressen bei Instruktionen treten folgende Situationen auf:

- Die Speicheradresse wird in einem Register übermittelt: → 32 Bit-Adresse mit Zugriff auf den gesamten Speicherbereich.
- Die Speicheradresse bei *J-Typ Instruktionen*: → Die Zieladresse kann maximal mit 26 Bit dargestellt werden (da 6 Bit für den *OpCode*).
- Die Speicheradresse bei *I-Typ Befehlen*: → Die Adresse kann hier nur als Halbwort mit 16 Bit übergeben werden (zusätzlich zu den 6 Bit für den *OpCode* werden jeweils 2 * 5 Bit für Register).

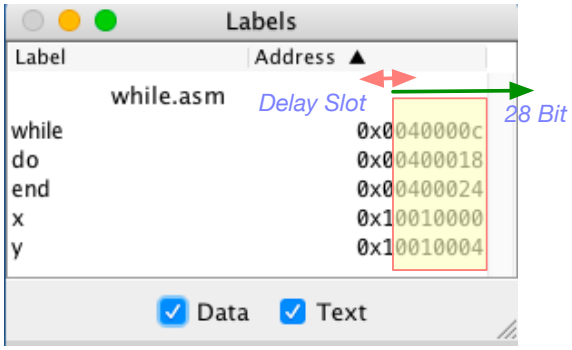

Abbildung 6.2: Nutzung der Symboltabelle für 26 Bit-Adressen

Um bei den beiden letzten Fällen zu einer gültigen Speicheradresse zu kommen, wird folgendes vorgenommen:

1. *26 Bit Adressen*: Es wird von der Adresse des **PC** ausgegangen und die obersten vier Bit, als '*Delay Slot*' verstanden. Die verbleibenden 28 Bit werden um 2 Bits nach links verschoben und der so erzielte 26 Bit-Wert als '*Instruction Index*' verstanden. Hierdurch können $\pm$ 256 MByte in Memory adressiert werden.
2. *16 Bit Adressen*: Der 16 Bit-Wert wird as Differenz bzw. *Offset* zum aktuellen **PC**+1 verstanden. Da der Offset auch negativ (zu kleineren Adressen) sein kann, wird er als Vorzeichenbehaftet betrachtet, um zwei Bit nach links verschoben und zum Wert von **PC**+1 addiert. Dies ergibt einen maximalen Adressenbereich von $\pm$ 128 KByte.

6.4.2 16 Bit Offset bei 32 Bit Adressen

Um trotzdem den gesamten 32-Bit Bereich zu adressieren, wird zu einem Trick gegriffen und das Vorzeichenbit 'missbraucht':

- Bei einer vorzeichenbehafteten 16 Bit Zahl beträgt der grösste Wert:
  $\mathbf{0}111\ 1111\ 111\ 1111_2 = (2^{15}\text{-}1)_{10}$.
- Diese Adresse wird *sign extended*; also:
  $0000\ 0000\ 0000\ 0000\ 0111\ 1111\ 111\ 1111_2 = (2^{15}\text{-}1)_{10}$.
  Hiermit lassen sich die 'unteren' Speicheradressen abdecken:
  $0000\ 0000\ 0000\ 0000\ 0xxx\ xxxx\ xxxx\ xxxx_2$
- Wird hingegen das erste Bit von den 16 Bits (dem *Bitvektor*) auf Eins gesetzt:
  $\mathbf{1}111\ 1111\ 111\ 1111_2 = \text{-}1_{10}$,
  würde das eigentlich im *Zweierkomplement* der '-1' entsprechen.
- In diesem Fall wird aber angenommen, dass alle *Most-Significant Bits* auf Eins gesetzt sind:
  $\mathbf{1111\ 11111\ 1111\ 1111}\ \mathbf{1}xxx\ xxxx\ xxx\ xxxx_2$
  und damit auch der 'obere' Speicherbereich adressierbar gemacht.

↪ Mit diesem 'Trick' kann aus einer 16 Bit Adresse eine qualifizierte 32 Bit Adresse abgeleitet werden.

6.5 ALU Arithmetik

Bei der Integer Arithmetik stehen die Befehle, die keine Vorzeichen beachten mit dem ergänzenden Buchstaben 'u' zur Verfügung. Hierbei ist aber Vorsicht geboten: *Immediate*-Instruktionen besitzen immer eine Konstante, die als Zweier-Komplement aufgefasst wird (mit der Ausnahme, falls es sich im eine Adresse handelt). Das 'u' bezieht sich somit auf den Fall, dass kein arithmetischer 'Überlauf' vorgenommen wird.

- Generell wird der Inhalt zweier Input-Register (`rs`, `rt`) über die Operation verknüpft und das Ergebnis auf dem Register `rd` ausgegeben.
- Bei der Multiplikation und Division werden per Default die Register `hi` und `lo` genutzt, die anschliessend entsprechend weiter zu verarbeiten sind.

6.5.1 ALU Addition und Subtraktion

Befehl	Argumente	Arbeitsweise	Erläuterung
add	`rd`, `rs`, `rt`	`rd` = `rs` + `rt`	Addition (mit Überlauf)
addi	`rd`, `rs`, Immediate	`rd` = `rs` + Immediate	Addition mit Immediate (mit Überlauf)
addu	`rd`, `rs`, `rt`	`rd` = `rs` + `rt`	Addition (mit Überlauf)
addiu	`rd`, `rs`, Immediate	`rd` = `rs` + Immediate	Addition mit Immediate (ohne Überlauf)
sub	`rd`, `rs`, `rt`	`rd` = `rs` − `rt`	Subtraktion (mit Überlauf)
subu	`rd`, `rs`, `rt`	`rd` = `rs` − `rt`	Subtraktion (ohne Überlauf)

Tabelle 6.6: Elementare Maschinen-Instruktionen für arithmetische Operationen

↪ Abweichend von diesem Schema, kann bei der Addition '**addiu**' genutzt werden, indem der in der Instruktion übergebene Wert zur (*unsigned*) Addition genutzt werden kann. Dieser Befehl bietet sich für Adressen-Operation an. Allerdings ist der Werte-Bereich von '`Wert`' auf 16 Bit beschränkt.

Dies markiert den Start des ausführbaren Programms und somit identische mit dem Inhalt der Registers $pc vor Beginn der Ausführung.

6.5.2 Arithmetische Pseudoinstruktion: Multiplikation

Bei der Multiplikation kann zudem auf folgende Pseudoinstruktionen gesetzt werden:

Befehl	Argumente	Arbeitsweise	Erläuterung
mult	`rs`, `rt`	`rd` = `rs` ∗ `rt`	Multiplikation; Ergebnis: `lo`, `hi`
multu	`rs`, `rt`	`rd` = `rs` ∗ `rt`	Vorzeichenlose Multiplikation; Ergebnis: `lo`, `hi`
mul^{PB}	`rd`, `rs`, `rt`	`rd` = `rs` ∗ `rt`	mathematische Multiplikation (ohne Überlauf)
mulo^{PB}	`rd`, `rs`, `rt`	`rd` = `rs` ∗ `rt`	mathematische Multiplikation (mit Überlauf)
mulou^{PB}	`rd`, `rs`, `rt`	`rd` = `rs` ∗ `rt`	Vorzeichenlose arith. Multiplikation (mit Überlauf)

Tabelle 6.7: Pseudoinstruktionen für Multiplikation; *PB* = Pseudobefehl

6.5.3 Arithmetische Pseudoinstruktion: Division

In allen Fällen wird das Ergebnis in zwei Schritten berechnet, indem zunächst die Register lo und hi benutzt werden, den Ergebniswert zu bevorraten und anschliessend das Ergebnis ins angegebene Zielregister kopiert wird.

Die Division kann ebenfalls einige Pseudoinstruktionen nutzen:

Befehl	Argumente	Arbeitsweise	Erläuterung
div^{PB}	`rd`, `rs`, `rt`	`rd` = `rs` : `rt`	mathematische Division (mit Überlauf)
divu^{PB}	`rd`, `rs`, `rt`	`rd` = `rs` : `rt`	mathematische Division (ohne Überlauf)

Tabelle 6.8: Pseudoinstruktionen für Multiplikation; *PB*: Pseudobefehl

Hier schauen wir uns den Aufbau der Pseudoinstruktion genauer an:

```
1  div $s0, $t0, $t1       # Pseudoinstruktion
2  bne $t0, $zero, 4
3  break $zero
4  div $t0, $t1
5  mflo $s0
6
7  divu $s0, $t0, $t1      # Pseudoinstruktion
8  bne $t0, $zero, 4
9  break $zero
10 divu $t0, $t1
11 mflo $s0
```

Listing 6.3: Implementierung der Pseudoinstruktion 'div' und 'divu'

Die Auflösung der Pseudoinstruktionen ist Implementierungsabhängig kann auf andere Weise erfolgen; die gezeigten Bespiele entsprechen nicht unbedingt dem MARS-Emulator!

6.5.4 Ergänzende arithmetische Pseudoinstruktionen

Die folgenden logischen (Pseudo-)Instruktionen können bei Bedarf eingesetzt werden:

Befehl	Argumente	Arbeitsweise	Erläuterung
abs	rd, rs	rd = |rs|	Betrag
neg	rd, rs	rd = −rs	Negativer Wert (mit Überlauf)
negu	rd, rs	rd = −rs	Negativer Wert (ohne Überlauf)
rem	rd, rs, rt	rd = mod rs ≡ rt	Modulo (Restwert)
remu	rd, rs, rt	rd = mod rs ≡ rt	Vorzeichenloses Modulo

Tabelle 6.9: Syntax und Arbeitsweise logischer Verknüpfungen

Diese Pseudobefehle lösen sich wie folgt auf:

```
1  abs $s0, $t0            # Pseudoinstruktion
2  add $s0, $zero, $t0
3  begz $s0 4
4  sub $s0, $zero, $to
5
6  negu $s0, $t0           # Pseudoinstruktion
7  subu $s0, $zero, $t0
8
9  rem  $s0, $t0, $t1      # Pseudoinstruktion
10 bne $t2, $zero, 4
11 break $zero
12 div $t0, $t0
13 mfhi $s0
14
15 remu  $s0, $t0, $t1     # Pseudoinstruktion
16 bne $t2, $zero, 4
17 break $zero
18 divu $t0, $t0
19 mfhi $s0
```

Listing 6.4: Implementierung der Pseudoinstruktion 'abs', 'negu', 'rem' sowie 'remu'

6.5.5 Bit-Operationen

Die folgenden logischen Bitfeld-Operationen können genutzt werden:

Befehl	Argumente	Arbeitsweise	Erläuterung
and	rd, rs, rt	rd = rs $\wedge$ rt	logisches AND
andi	rd, rs, Wert	rd = rs $\wedge$ Wert	logisches AND mit Wert
or	rd, rs, rt	rd = rs $\vee$ rt	logisches OR
ori	rd, rs, Wert	rd = rs $\vee$ Wert	logisches OR mit Wert
xor	rd, rs, rt	rd = rs $\oplus$ rt	logisches XOR
xori	rd, rs, Wert	rd = rs $\oplus$ Wert	logisches XOR mit Wert
nor	rd, rs, rt	rd = $\neg$(rs $\vee$ rt)	logisches NOR
not	rd, rs	rd = $\neg$ rs	logisches NOT

Tabelle 6.10: Syntax und Arbeitsweise logischer Verknüpfungen

↪ *Bit-Shift-Operationen* können nach 'links' oder 'rechts' durchgeführt werden:

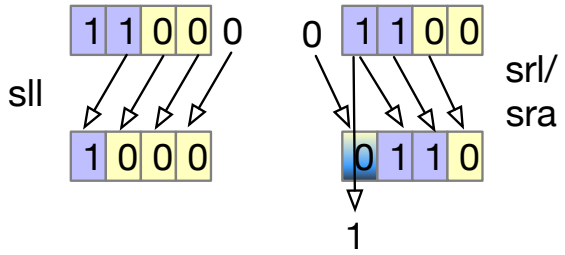

Abbildung 6.3: Bit-Shift-Operationen am Beispiel eines Halb-Byte

Bei einem 'arithmetischen' Rechts-Shift bleibt der Wert des höchsten Bits erhalten.

Befehl	Argumente	Arbeitsweise	Erläuterung	OpCode
sll	rd, rs, Wert	rd = rs $\times$ 2^{Wert}	Shift Word Left Logical (max. 2^5 Bits)	000 000
sllv	rd, rs, rt	rd = rs $\times$ 2^{rt}	Shift Word Left Logical Value	000 100
srl	rd, rs, Wert	rd = rs / 2^{Wert}	Shift Word Right Logical (max. 2^5 Bits)	000 010
srlv	rd, rs, rt	rd = rs / 2^{rt}	Shift Word Right Logical Value	000 110
sra	rd, rs, Wert	rd = rs / 2^{Wert}	Shift Word Right Arithmetic (max. 2^5 Bits)	000 011
srav	rd, rs, rt	rd = rs / 2^{rt}	Shift Word Right Arithmetic Variable	000 111

Tabelle 6.11: Syntax und Arbeitsweise logischer Verknüpfungen

↪ *Rotationen* in einem Bitfeld sind als Pseudoinstruktionen von Bit-Shift implementiert:

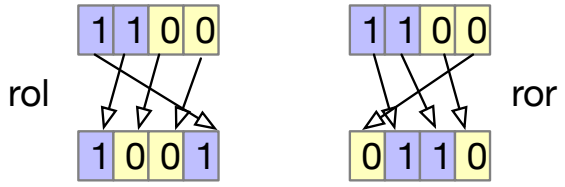

Abbildung 6.4: Bit-Rotations-Operationen am Beispiel eines Halb-Byte

Befehl	Argumente	Arbeitsweise	Erläuterung
rol	rd, rs, rt	rd = $\overset{rt}{\circlearrowleft}$ rs	rotate left (by rt)
ror	rd, rs, rt	rd = $\overset{rt}{\circlearrowright}$ rs	rotate right (by rt)

Tabelle 6.12: Syntax und Arbeitsweise logischer Verknüpfungen

Die Bit-Rotations-Operationen sind Pseudoinstruktionen, die wie folgt umgesetzt sind.

```
  rol $s0, $t0, $t1       # Pseudoinstruktion
  subu $at, $zero, $t1
  srlv $at, $t1, $at
  sllv $s2, $t1, $t0
  or $s0, $s0, $at

# und auch fuer ror

  ror $s0, $t0, $t1       # Pseudoinstruktion
  subu $at, $zero, $t1
  srlv $at, $at, $0
  sllv $s2, $t0, $t1
  or $s0, $s0, $at
```

Listing 6.5: Implementierung der Pseudoinstruktion 'rol' und 'ror'

7 Labels, Verzweigungen und Kontrollstrukturen

Mit *Labels* oder im Deutschen *Marken* können Sprungziele im Assembler-Programm deklariert werden. Hierdurch lassen sich eigene Prozeduren oder Makros realisieren bzw. 'anspringen'.
Labels sind also Marken, die vom Programmierer zu deklarieren sind. Hierzu werden sie an den Anfang einer Zeile gesetzt und mit dem Doppelpunkt abgeschlossen:

```
1 .text
2 label: li $t0, x
```

Listing 7.1: Setzen eines Labels im Textsegment

7.1 Sprung zu Labels im Textsegment

Wollen wir zu diesem Label springen, reicht folgendes Statement:

```
1 j label
```

Listing 7.2: Sprung an ein Anweisungs-Label im Textsegment

Ein wichtiges, oft genutzte generisches Label ist

```
1 .text
2 main:
3
4   jal main     # jump and link
```

Listing 7.3: Markieren des Einsprungs in das Textsegment mittels 'main:'

Bei den einfachen Verschachtelungen gelten die folgenden Paradigmen:

- Die Rücksprungadresse für den *Caller* wird im Register `$ra` automatisch abgelegt.
- Es bieten sich die Argumenten-Register `$a0` bis `$a3` zur Übergabe von Werten an.
- Das Ergebnis kann über die Register `$v0` und ggf. `$v1` mitgeteilt werden.

↪ Diese Arbeitsweise bezeichnen wir als 'Call-by-Value'.

Wir können dies mit folgenden einfachen Sprungbefehlen realisieren:

Befehl	Argumente	Arbeitsweise	Beispiel
jr	`rs`	Springe zur Adresse in Register `rs`	`ja $ra`
jal	`label`	Springe zu `label`; Rücksprungadresse in `$ra`	`jal proc`
jalr	`rs, rd`	Springe zur Adresse im Register `rs`; Rücksprungadresse in `rd` (Default: `$ra`)	`jalr $s0, $s7`

Tabelle 7.1: Syntax und Arbeitsweise der unbedingten Sprunganweisungen

Befehl	Argumente	Arbeitsweise	Beispiel
beq	`rs, rt, label`	Springe zu `label`, falls `rs` $=$ `rt`	`beq $t0, $t1, proc`
bne	`rs, rt, label`	Springe zu `label`, falls `rs` $\neq$ `rt`	`bne $t0, $t1, proc`
beqz	`rs, label`	Springe zu `label`, falls `rs` $= 0$	`beqz $t0, proc`
bnez	`rs, label`	Springe zu `label`, falls `rs` $\neq 0$	`bnez $t0, proc`
bge	`rs, rt, label`	Springe zu `label`, falls `rs` $\geq$	`bge $t0, $t1 proc`
bgeu	`rs rt, label`	Springe zu `label`, falls `rs` $\geq$ rt (unsigned)	`bgeu $t0, $t1, proc`
bgez	`rs, label`	Springe zu `label`, falls `rs` ≥ 0	`bgez $t1, proc`
bgt	`rs, rt, label`	Springe zu `label`, falls `rs` $>$ rt	`bgt $t0, $t1 proc`
bgtu	`rs rt, label`	Springe zu `label`, falls `rs` $>$ rt (unsigned)	`bgtu $t0, $t1, proc`
bgtz	`rs, label`	Springe zu `label`, falls `rs` > 0	`bgtz $t1, proc`
blt	`rs, rt, label`	Springe zu `label`, falls `rs` $<$ rt	`blt $t0, $t1 proc`
bltu	`rs rt, label`	Springe zu `label`, falls `rs` $<$ rt (unsigned)	`bltu $t0, $t1, proc`
bltz	`rs, label`	Springe zu `label`, falls `rs` < 0	`bltz $t1, proc`
ble	`rs, rt, label`	Springe zu `label`, falls `rs` $\leq$ rt	`ble $t0, $t1 proc`
bleu	`rs rt, label`	Springe zu `label`, falls `rs` $\leq$ rt (unsigned)	`bleu $t0, $t1, proc`
blez	`rs, label`	Springe zu `label`, falls `rs` ≤ 0;	`blez $t1, proc`
b	`label`	Springe zu `label` (unbedingt)	`b Done`

Tabelle 7.2: Syntax und Arbeitsweise einiger bedingter (und unbedingter) Sprunganweisungen

Hinweis: Beim Sprung auf ein *Label* im Textsegment ist der Bereich auf $\pm$ 256 MByte beschränkt.

7.1.1 Verzweigungen

Es gibt zwei Arten von Verzweigungen:

- *Unbedingte Verzweigungen* **b** (*branch*) und **j** (*jump*), mit denen immer zu `label` gesprungen wird, wobei aber **b** eine Pseudoinstruktion darstellt.
- *Bedingte Verzweigungen*, die von der Auswertung eines arithmetischen Ausdrucks, wie z.B. $a \geq b$ abhängen.

Die Implementierung von **b** ist eigentlich:

```
1 b := bgez $zero label
```

Listing 7.4: Implementierung der Pseudoinstruktion 'b'

und nutzt somit den arithmetischen Vergleich '`branch greater equal zero`' im Vergleich zu `$zero`, der immer 'wahr' ist.

7.2 Vergleiche und Kontrollstrukturen

Bei Vergleichen werden die Inhalte der Input-Register verglichen und das Ergebnis des Vergleichs als 'logischer' Wert im Ausgaberegister abgelegt:

Befehl	Argumente	Arbeitsweise	Erläuterung
seq	rd, rs, rt	rs = rt ⇒ rd = 1 (0)	"=": set equal
sne	rd, rs, rt	rs ≠ rt ⇒ rd = 1 (0)	"≠": set not equal
sge	rd, rs, rt	rs ≥ rt ⇒ rd = 1 (0)	"≥": set greater equal
sgeu	rd, rs, rt	rs ≥ rt ⇒ rd = 1 (0)	"≥": set greater equal unsigned
sgt	rd, rs, rt	rs > rt ⇒ rd = 1 (0)	">": set greater than
sgtu	rd, rs, rt	rs > rt ⇒ rd = 1 (0)	">": set greater than unsigned
sle	rd, rs, rt	rs ≤ rt ⇒ rd = 1 (0)	"≤": set less equal
sleu	rd, rs, rt	rs ≤ rt ⇒ rd = 1 (0)	"≤": set less equal unsigned
slt	rd, rs, rt	rs < rt ⇒ rd = 1 (0)	">": set less than
sltu	rd, rs, rt	rs < rt ⇒ rd = 1 (0)	">": set less than unsigned
slti	rd, rs, Wert	rs < Wert ⇒ rd = 1 (0)	"<": set less than
sltui	rd, rs, Wert	rs < Wert ⇒ rd = 1 (0)	"<": set less than unsigned

Tabelle 7.3: Syntax und Arbeitsweise logischer Verknüpfungen

Auch diese Befehle sind als Pseudoinstruktionen implementiert:

```
1   seq $s0, $t0, $t1       # Pseudoinstruktion
2   ori $s0, $zero, 0
3   beq $zero, $zero, 4
4   ori $s0, $zero, 1
5
6  # und auch fuer set equal less
7
8   sleu $s0, $t0, $t1      # Pseudoinstruktion
9   bne $t1, $t0, 8
10  ori $s0, $zero, 1
11  beq $zero, $zero, 4
12  slt $s0, $t0, $t1
```

Listing 7.5: Pseudoinstruktion für 'seq' und 'sleu'

7.3 Traps

Gelegentlich ist es notwendig den Programmfluss zu unterbrechen und eine sog. *Ausnahmebehandlung* durchzuführen: **Trap**. Traps werden in der Regel dann gesetzt, wennn eine Operation zu einem unerwünschten Ergebnis führt, z.B. die Division durch Null.

Eine wichtige Trap-Instruktion ist der **SYSCALL**, der das laufende Programm unterbricht und den Supervisor aufruft. Nebem dem **SYSCALL** und der **BREAK**-Anweisung MIPS besitzt hierfür eine umfangreiche Trap-Instruktionen, die sich auf den Inhalt von Registern oder das Ergebniss von Berechnungen beziehen.

Befehl	**Argumente**	**Erläuterung**
teq	`rs`, `rt`	`rs` $=$ `rt` $\Rightarrow$ *trap*
teqi	`rs`, Wert	`rs` $=$ `Wert` $\Rightarrow$ *trap*
tge	`rs`, `rt`	`rs` $\geq$ `rt` $\Rightarrow$ *trap*
tgei	`rs`, Wert	`rs` $\geq$ *Wert* $\Rightarrow$ *trap*
tgeiu	`rs`, Wert	`rs` $>$ (*unsigned*) `Wert` $\Rightarrow$ *trap*
tgeu	`rs`, `rt`	`rs` $>$ (*unsigned*) `rt` $\Rightarrow$ *trap*
tlt	`rs`, `rt`	`rs` $<$ $\Rightarrow$ *trap*
tlti	`rs`, Wert	`rs` $<$ `Wert` $\Rightarrow$ *trap*
tltiu	`rs`, Wert	`rs` $<$ (*unsigned*) `Wert` $\Rightarrow$ *trap*
tltu	`rs`, `rt`	`rs` $<$ (*unsigned*) `rt` $\Rightarrow$ *trap*
tne	`rs`, `rt`	`rs` $\neq$ `rt` $\Rightarrow$ *trap*
tnei	`rs`, Wert	`rs` $\neq$ Wert $\Rightarrow$ *trap*

Tabelle 7.4: Trap-Instruktionen für Register

8 Nutzung der FPU (CoProc1)

Der MIPS-Aufbau beinhaltet neben der ALU noch (mandatorisch) vier weitere Koprozessoren:

- *CoProcessor 0*: Dieser ist zuständig für die Behandlung von *Exceptions* (nächste Vorlesung) und dient als *System Control Coprocessor*.
- *CoProcessor 1*: Mathematischer Koprozessor (*Floating Point Unit* **FPU**) mit der Möglichkeit, *IEEE 754 Fliesskommazahlen* nativ zu verarbeiten. Der CoProc1 verfügt über 32 Register, die als 32 Bit Register ($f*) oder in Kombination als nun 16 64 Bit Register für die Verarbeitung von *double*-Worten genutzt werden können.
- *CoProcessor 2*: Implementierung spezifischer Instruktionen.
- *CoProcessor 3*: Ergänzende *Floating Point Operationen* speziell in der MIPS64 Architektur.

↪ Jeder Koprozessor kann separat angesprochen werden, wobei manche Operationen auf unterschiedlichen Prozessoren abgewickelt werden können. Hierbei wird durch die unterschiedliche Benennung der Register mitgeteilt, welche FPU involviert ist. Bei einigen Befehlen muss aber explizit mitgeteilt werden, für welchen CoProc sie gedacht ist:

- `sw[x] $1 Adresse`: **store word** von Koprozessor `x` (0 bis 3) an *Adresse*.
- `lw[x] $1 Adresse`: **load word** für Koprozessor `x` (0 bis 3) von *Adresse*.

8.1 FPU Fliesskomma-Register

Die 32 G*eneral Purpose Register* des CoProc1 (auch FPU Register genannt; daher $f*) besitzen ebenfalls eine Breite von 32 Bit.

Zur Hinterlegung von *double precision* Werten, können aber jeweils zwei zusammen gefasst werden (*konkatiniert*): ($f0 || $f1) → $f0/1.

Einige Register besitzen besondere Bedeutung:

- $f0/$f1: Diese werden als Register für die Benutzereingabe genutzt.
- $f12/$f13: Dienen als Ausgaberegister für die Berechnung.

Pro FPU existieren weitere fünf sog. *Floating Point Control Registers* **FCR**:

- FIR *Floating Point Implementation Register*: Read-Only zur Festlegung der Floating Point Eigenschaften der betreffenden MIPS CPU.
- FCSR *Floating Point Control and Status Register*: Festlegung der Rundungsverhalten, Traps und IEEE 754 Exception-Verarbeitung.
- FCCR *Floating Point Condition Code Register*: Status der IEEE 754 Bearbeitung, wie *Rundung*, *Denormalisierung* und *Exceptions*.
- FEXR *Floating Point Exceptions Register*: Ergänzende Angaben zum FCSR im Falle von Exceptions mit *Cause* und *Flag* Informationen.
- FENR *Floating Point Enabled Register*: Ergänzende Angaben zum FCSR im Falle von Rundungen.

8.2 Fliesskomma-Operationen

Der CoPro1 des MIPS kann folgende Datenformate (*Direktiven*) verarbeiten:

- .float: 32 Bit Fliesskommazahl (*single precision*). Eingabeformate benötigen einen Punkt als Dezimal-Kennzeichen: `1234.56`, `0.09876`, `-123.45e56`, `0.02e-10`
- .double: 64 Bit Fliesskommazahl, also *double precision*.

Das Register des CoPro1 kann sowohl *single*, als auch *double precision* Werte im IEEE 754 Format verarbeiten.

- Es stehen spezielle *Load&Store*-Befehle zur Verfügung diese Register aus dem Hauptspeicher zu befüllen, bzw. Werte dort abzulegen.
- Operationen für den CoPro1 unterscheiden sich im OpCode von denen für die ALU; wobei die Syntax vergleichbar ist.
- Der CoProc1 nutzt naturgemäss unterschiedliche Operationen für *single* und *double precision* Werte.
- Zwischen diesen Formaten kann umgewandelt werden und ebenso ist ein Transfer der Daten von bzw. zu den Registern der ALU mit entsprechender Transformation möglich

8.2.1 Load & Store in FPU

Das Beziehen der Daten aus dem Hauptspeicher ist unter Einbeziehung des CoProc1 aufgrund des zusätzlichen Double-Datenformates etwas komplexer.

Befehl	Argumente	Arbeitsweise	Erläuterung
lwc[x]	`cd`, Adresse	4-Byte Wort[Adresse] → `cd`	Lade Wort an [Adr] in CPx
l.d	`fd`, Adresse	8-Byte Wort[Adresse] → (`fd` || `fd+1`)	Lade FP Double an [Adr]
l.s	`fd`, Adresse	4-Byte Wort[Adresse] → `fd`	Lade FP Single an [Adr]
swc[x]	`cd`, Adresse	`cd` → 4-Byte Wort[Adresse]	Speichere CPx Register an [Adr]
s.d	`fd`, Adresse	(`fd` || `fd+1`) → 8-Byte Wort[Adresse]	Speichere FP Double an [Adr]
s.s	`fd`, Adresse	`fd` → 4-Byte Wort[Adresse]	Speichere FP Single an [Adr]
mfc[x]	`rd`, `cs`	`cs` → `rd`	Kopiere CPx zu ALU Register
mfc.d	`rd`, `cs`	(`cs` || `cs+1`) → (`rd` || `rd+1`)	Kopiere ALU Double Register zu CP1
mtc[x]	`rs`, `cd`	`rs` → `cd`	Kopiere ALU auf CPx Register
mov.d	`fd`, `fs`	(`fs` || `fs+1`) → (`fd` || `fd+1`)	Kopiere CP1 Double Register
mov.s	`fd`, `fs`	`fs`→ `fd`	Kopiere CP1 Single Register

Tabelle 8.1: Syntax und Arbeitsweise der CoProc1 Load und Store Befehle; FP: Floating Point, Adr: Adresse, CP1: CoProc1

Folgende Sachverhalte sind zu beachten:

- Werden *double precision* Register angesprochen, werden immer zwei normale Register zu Ablage der Daten benötigt.
- Das 'unterste' Register muss immer geradzahlig sein!

8.2.2 Daten-Konvertierung

Zur Umwandlung von Floating Point und Integer Zahlen mit einfacher Genauigkeit und solchen mit doppelter Genauigkeit (also 64 Bit) – Word Fixed Point (WFP) – bietet MIPS folgende Befehle an:

Befehl	Argumente	Arbeitsweise	Erläuterung
cvt.d.s	`fd`, `fs`	`fs` → `fd`	Konvertiert Single nach Double
cvt.d.w	`fd`, `fs`	`fs` → (`fd` || `fd+1`)	Konvertiert WFP nach Double
cvt.s.d	`fd`, `fs`	(`fs` || `fs+1`) → `fd`	Konvertiert Double nach Single
cvt.s.w	`fd`, `fs`	`fs` → `fd`	Konvertiert WFP nach Single
cvt.w.d	`fd`, `fs`	(`fs` || `fs+1`) → `fd`	Konvertiert Double nach WFP
cvt.w.s	`fd`, `fs`	`fs` → `fd`	Konvertiert Single nach WFP

Tabelle 8.2: Konvertierungs-Routinen für Word Fixed Point ↔ Float, Word Fixed Point ↔ Double und Float ↔ Double Registerinhalte

Konvertierung von *Float* nach *Integer* (Word):

1. Umwandlung des Float/Double in einen 'Word Fixed Point' Wert und speichern in einem `$f*` Register.
2. 'Moven' des 'Word Fixed Point' `$f*` Registers in ein ALU GPR (z.B. `$t0`).

Konvertierung von *Integer* (Word) nach *Float*:

1. Kopieren des Integers (aus einem ALU Register) mittels `mtc1` Tab. 8.1 im 'Word Fixed Point' Format in ein `$f*` Register.
2. Umwandlung 'Word Fixed Point' Wert mittels des **cvt.w.s** Befehls in das IEEE 784 Floating Point Format und Abspeichern in einem weiteren `$f*` Register (dies kann das gleiche sein).

Für die 'Doubles' gilt entsprechendes; allerdings natürlich nun mit zwei notwendigen $f* Registern als Ziel.

Rundungsverhalten:
Für die Art und Weise der Rundung (Float/Double → Integer) ist die zwei RM (Rounding Mode) Bits im CoProc1 *Floating Point Control and Status Register* FCSR massgeblich:

Wert	Mnemonik	Bits	Rundungsverhalten
0	RN	00	Round To Nearest (Round to Even) – Default
1	TZ	01	Round To Zero
2	RP	10	Round Towards Plus Inifinity
3	RM	11	Round Towards Minus Inifinity

Tabelle 8.3: Bedeutung der 'Rounding Modes' (RM) im FCSR

Der MARS kennt nur 'Round to Nearest'; andere Rundungsarten werden nicht unterstützt.

8.2.3 FPU-Berechnungen

Die Operationen für die arithmetische Berechnung erlauben die Angabe, ob sie für *single* oder *double precision* Werte (mit den dazu gehörigen zusammengeführten Registern) durchzuführen sind.

Befehl	Argumente	Arbeitsweise	Erläuterung
add.d	`fd, fs, ft`	`(fd || fd+1) = (fs || fs+1) + (ft || ft+1)`	FP Double Addition
add.s	`fd, fs, ft`	`fd = fs + ft`	FP Single Addition
sub.d	`fd, fs, ft`	`(fd || fd+1) = (fs || fs+1) - (ft || ft+1)`	FP Double Subtraktion
sub.s	`fd, fs, ft`	`fd = fs - ft`	FP Single Subtraktion
mul.d	`fd, fs, ft`	`(fd || fd+1) = (fs || fs+1) * (ft || ft+1)`	FP Double Multiplikation
mul.s	`fd, fs, ft`	`fd = fs * ft`	FP Single Multiplikation
div.d	`fd, fs, ft`	`(fd || fd+1) = (fs || fs+1) : (ft || ft+1)`	FP Double Division
div.s	`fd, fs, ft`	`fd = fs : ft`	FP Single Division
abs.d	`fd, fs`	`(fd || fd+1) = |(fs || fs+1)|`	FP Double Absolut Betrag
abs.s	`fd, fs`	`fd := |fs|`	FP Single Absolut Betrag
neg.d	`fd, fs`	`(fd || fd+1) = −(fs || fs+1)`	FP Double negativer Wert
neg.s	`fd, fs`	`fd := −fs`	FP Single negativer Wert

Tabelle 8.4: Arithmetischen Operationen des CoProc1 mit einfacher und doppelter Genauigkeit

8.2.4 FPU-Vergleiche

Der CoProc1 besitzt auch Vergleichs-Operationen mit bedingten Verzweigungen, ähnlich wie die ALU.

Befehl	Argumente	Arbeitsweise	Erläuterung
bc[x]**t**	Wert	Falls Condition-Flag [x] → Jump	Branch CPx, falls [CF] wahr
bc[x]**f**	Wert	Falls Condition-Flag [!x] → Jump	Branch CPx, falls [CF] falsch
c.eq.d	`fs`, `ft`	(`fs` || `fs+1`) = −(`ft` || `ft+1`) → [CF]	Vergleiche Double
c.eq.s	`fs`, `ft`	`fs` = −`ft` → [CF]	Vergleiche Single
c.le.d	`fs`, `ft`	(`fs` || `fs+1`) ≤ (`ft` || `ft+1`) → [CF]	Kleiner/gleich Double
c.le.s	`fs`, `ft`	`fs` ≤ `fst` → [CF]	Kleiner/gleich Single
c.lt.d	`fs`, `ft`	(`fs` || `fs+11`) < (`ft` || `ft+1`) → [CF]	Kleiner Double
c.lt.s	`fs`, `ft`	`fs` < `ft` → [CF]	Kleiner Single

Tabelle 8.5: CoProc1 Vergleichsoperationen mit Auswertung bzw. Setzen des Condition Flags **CF**

↪ Sowohl die ALU als auch jeder Koprozessor verfügt über ein Flag-Register. Bei den FPUs liegen fünf *Floating Point Control Register* vor, bei denen das *Floating Point Control and Status Register* (FCSR) und speziell das Feld FCC (*Floating Point Control Codes*) hier ausgewertet wird.

9 System Calls, Interrupts und Exceptions

Die Betriebssystemfunktionen (*system calls*) werden mit dem Befehl **syscall** aufgerufen. In das Register `$v0` ist zuvor der Code für die gewünschte Funktion zu schreiben:

syscall führt die in `$v0` per Code angegebene Systemfunktion aus.

Code von $v0	**System-funktion**	**zu übergebende Argumente**	**Ergebnis**
1	print_int	`$a0`	wird dezimal ausgegeben
2	print_float	`$f12`	wird als 32 bit Float ausgegeben
3	print_double	`$f12/$f13`	wird als 64 bit Float ausgegeben
4	print_string	`$a0`	wird mit 0 terminiert ausgegeben
5	read_int		Einlesen ganzer 32 bit Zahl von Konsole auf `$v0`
6	read_float		Einlesen 32 bit Float von Konsole auf `$f0`
7	read_double		Einlesen 64 bit Float von Konsole auf `$f0/$f1`
8	read_string		Einlesen Zeichenkette auf `$a0` + Länge in `$a1`
9	sbrk	n [byte]	Anfangsadresse freier Block mit n [byte] in `$v0`
10	exit		beende Programm
11	print_char	`$a0`	gibt einzelnen Charakter aus
12	read_char	`$v0`	Lese einzelnen Charakter ein
13	open (file)	`$a0` = file name, `$a1` = flags, $a2 = mode	file descriptor in $a0
14	read (file)	`$a0` = filedescriptor, `$a1` = buffer, `$a2` = length	Anzahl gelesener Zeichen in `$a0`
15	write (file)	`$a0` = filedescriptor, `$a1` = buffer, `$a2` = length	Anzahl geschriebener Zeichen in `$a0`
16	close (file)	`$a0` = filedescriptor, `$a1` = buffer, `$a2` = length	Datei geschlossen
17	exit2	`$a0` = return code	beende Programm
34	print_int hex	`$a0`	wird hexadezimal ausgegeben; float Register sind auf `$a0` zu kopieren
35	print_int binary	`$a0`	wird binär ausgegeben; sollen alle 32 Bit angezeigt werden, sind die obersten mit '0' zu füllen

Tabelle 9.1: Liste einiger **syscalls** und ihre Nutzung/Bedeutung

10 Pseudocode Darstellung

Bei vielen Programmierproblemen wird die Logik der Fragestellung zunächst 'abstrakt' formuliert, ohne dass man sich auf eine Programmiersprache bezieht. Bekannt sind zwei Ansätze:

1. Nassi-Shneiderman-Diagramme[1].
2. Pseudocode-Implementierung[2].

Für ein Problem, das ich neulich zu bearbeiten hatte, gab es in einer früheren Veröffentlichung[3] folgenden Pseudocode:

```
for all $DOMAIN in DS records of COM and NET zones do
  check _443._tcp.$DOMAIN
  check _443._tcp.www.$DOMAIN

  for SMTP $PORT 25, 465, 587 do
    if MX record points to $MX then
      check _$PORT._tcp.$MX
    else
      check _$PORT._tcp.$DOMAIN
```

Wir sehen, dass hier keine konkrete Syntax beschrieben ist: *Pseudocode*. Das Problem wird aber korrekt und vollständig beschrieben, sodass es nachvollziehbar ist.

10.1 Pseudocode für MIPS Assembler

Wie betrachten folgenden Pseudocode aus einem Aufgabenblatt:

```
(1) r4 <-- 30
(2) r5 <-- 25
(3) r2 <-- r5
(4) if (r4 == 0) goto (10)
(5) if (r5 == 0) goto (9)
(6) if (r4 > r5) r4 <-- r4 - r5
(7) else         r5 <-- r4 - r5
(8) goto (5)
(9) r2 <-- r4
(10) goto (10)
```

Was bedeutet das?

- Es ist klar das 'r' für 'Register' steht: `r4` ist also ALU-Register `$4` bzw. `$a0`.
  - Evtl. können Floating-Point-Register mit 'f' und
  - Floating-Point-Register mit doppelter Genauigkeit mit 'd' bezeichnet werden.
- Das Zeichen `<--` ist ein Zuweisungsoperator.
  - Typischerweise wird dieser per **add** umgesetzt, wobei links das Zielregister steht.
  - In Beispiel auf Zeile (1) folgt eine Zahl, also ist hier **addi** anzuwenden, wobei als zweites Input-Register `$0` zu wählen ist.
- Der arithmetische Vergleich `if (r* == N)` muss mit der Adresse einer Zielinstruktion verknüpft sein.

[1] vgl. https://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm
[2] vgl. https://de.wikipedia.org/wiki/Pseudocode
[3] Quelle: Zhu2015_Chapter_MeasuringDANETLSADeplyoment.pdf

  - Für den Vergleich steht die bedingte Verzweigungen *branch* in den verschiedenen Variationen zur Verfügung.
  - Im Assemblercode erreicht man die Adresse einer Instruktion über ein zugehöriges *Label* im Quellcode.
- Die `else` Anweisung braucht nicht extra codiert zu werden: MIPS folgt dem Load&Store-Paradigma. Wird nicht verzweigt, folgt automatisch die nächste Anweiseung.
- `goto` ist eine unbedingte Verzweigung; also ein 'Jump' mit einer Zieladresse (einem Label).