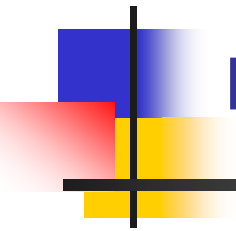


Kapitel 6 (1. Teil MIPS): MIPS Befehle

Technische Grundlagen der Informatik 2
(Rechnertechnologie 2)
SS 2006

Andreas Koch

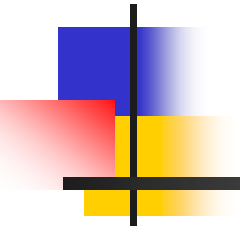
FG Eingebettete Systeme und ihre Anwendungen

Auf Basis von Material von

Rolf Hoffmann

FG Rechnerarchitektur

Technische Universität Darmstadt



In Anlehnung an das Patterson/Hennessy: Computer Organization & Design, 2nd Edition, Chapter 3, 5

Es sind auch die Folien von Dr. M. G. Wahl (Univ. Siegen, Inst. Mikrosystemtechnik) und ähnliche aus den Grundzügen der Informatik II, SS03, von Prof. Dr. Oskar von Stryk verwendet worden.

Inhalt

- Befehlssatz
- RISC und CISC
- MIPS
  - Speicherorganisation
  - Befehle
    - add
    - load, store
    - Unbedingter Sprung jump
    - Bedingter Sprung beq
    - Vergleich slt
    - Register-indirekter Sprung
    - Unterprogramm-Sprung
    - Zeichen und byteweise Zugriffe
  - Konstanten im Befehl
  - Vergleichsbefehl Signed/Unsigned
  - Logische Operationen
  - Konstruktion einer ALU für MIPS

Wiederholung

- Grundprinzipien des Von-Neumann-Rechners:
  - Programme und Daten werden in einem **gemeinsamen Speicher** abgelegt und können dort gelesen und geschrieben werden.
  - Daten und Befehle werden als Binärwörter repräsentiert.
  - Dies ist das Prinzip des speicherprogrammierbaren Rechners (**Stored Program Computer**).
- Stärken dieses Konzepts
  - Daten und Programme können **beliebig im Speicher angeordnet** werden.
  - Programmstart erfolgt einfach durch Angabe einer Speicheradresse
  - Programme können auch als Daten interpretiert und sogar während der Laufzeit modifiziert werden!

Befehlssatz allgemein (1)

- Sichtweise: Formale Sprache
  - Die Maschine versteht (interpretiert) die Maschinenbefehle (in Form des **Binärcodes**)
  - Die Maschinenbefehle definieren eine „**Maschinensprache**“
  - Die **Assemblersprache** ist eine symbolische Form der Maschinensprache, die im wesentlichen 1:1 auf die Maschinensprache durch den Assembler übersetzt wird.

Befehlssatz allgemein (2)

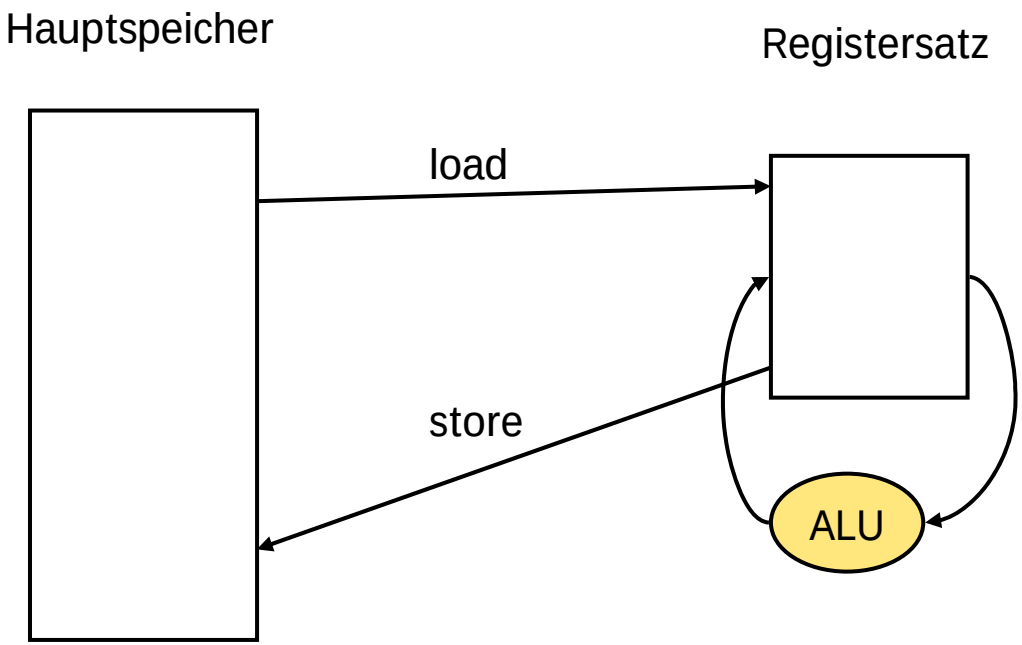
- Entwurf von Befehlssätzen
  - Entwurfsziele
    - Unterstützung
      - bestimmter Anwendungen/Programmiersprachen → **Spezialrechner** oder
      - möglichst vieler Anwendungen/Programmiersprachen → **Universalrechner**
    - maximiere **Leistung** (performance)
    - minimiere **Kosten**
    - reduziere **Entwurfszeit**
  - Nebenbedingungen
    - geeignet für den **Compilerbauer**
    - beschränkte **Hardware-Ressourcen**
    - schnelle und **einfache** Interpretation durch die **Hardware**
    - geringer **Stromverbrauch**

RISC-Rechner

RISC = Reduced Instruction Set Computer

1. (Load-Store-Architektur) Die Rechenoperationen finden nur auf Operanden statt, die sich in den Registern befinden.

2. (Einfache Befehle mit fester Wortlänge): Vereinfacht die Hardware-Implementierung und beschleunigt die Ausführung (Pipelining)



CISC-Rechner

- **Complex Instruction Set Computer**
  - Die Befehlssatz besteht aus einem Mix von einfachen und komplexen Befehlen.
  - Die Wortlänge der Befehle variiert je nach Komplexität und unterzubringender Informationsmenge
  - Operationen können neben Registeroperanden auch auf Speicheroperanden definiert sein.
  - Beispiel für komplexe Befehle
    - **Direkte Operationen auf Speicherzellen**
      m[adr1] := m[adr2] + m[adr3]
    - **Vektorbefehle** können komplette Vektoren auf einmal verarbeiten, implementiert in sogenannten **Vektorrechnern**
    - Beispiel: Pentium **MMX-Befehle** (Multimedia-Erweiterung) zur Unterstützung der Verarbeitung von grafischen Pixel-Vektoren

Vergleich RISC ←→ CISC

	RISC	**CISC**
Anzahl Befehle	klein	groß
Adressierungsarten	wenige	viele
Befehlswortlänge	fest	variabel
Sprachorientierung der Maschinenbefehle	Sprache C	auch andere, z. B. JAVA
komplexe Befehle	nein	ja
Aufwand für Hardware-Optimierung	klein	groß
Hardware-Implementierungsaufwand	klein	groß

Register-Architekturen

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	accumulator	1949
IBM 701	1	accumulator	1953
CDC 6600	8	load-store	1963
IBM 360	16	register-memory	1964
DEC PDP-8	1	accumulator	1965
DEC PDP-11	8	register-memory	1970
Intel 8008	1	accumulator	1972
Motorola 6800	2	accumulator	1974
DEC VAX	16	register-memory, memory-memory	1977
Intel 8086	1	extended accumulator	1978
Motorola 68000	16	register-memory	1980
Intel 80386	8	register-memory	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992
HP/Intel IA-64	128	load-store	2001
AMD64 (EMT64)	16	register-memory	2003

Überleitung zur MIPS-Architektur

- Wir haben den prinzipiellen Aufbau und die Arbeitsweise eines Rechners am Beispiel des DINATOS kennengelernt.
  - Beispiel für eine Akkumulatormaschine und Grundprinzip für CISC-Rechner
  - Implementierung steht als **Softcore** zur Verfügung

- Im folgenden wollen wir die MIPS-Architektur kennenlernen
  - Beispiel für einen RISC-Rechner
  - ISA-Architektur: einfach, überschaubar, mit allen wichtigen Konzepten
  - eingesetzt von Silicon Graphics, NEC, Nintendo, Sony u. a., Embedded Syst.
  - entwickelt seit 1981 J. Hennessy – Stanford Univ.

MIPS = Microprocessor without interlocked pipeline stages

ISA = Instruction Set Architecture

MIPS-Befehle: Erstes Beispiel

„ Addiere die Werte der beiden Variablen ‚b‘ und ‚c‘ und weise das Ergebnis der Variablen ‚a‘ zu “

```
add a, b, c          // a ← b+c
```

- Alle Arithmetikbefehle haben genau 3 Variablen: 2 Argumente und ein Ergebnis
- feste Reihenfolge der Variablen (Ergebnis zuerst)
- jede Zeile ein Befehl

Höhere Programmierspr. ←→ Assembler

Wirkung	MIPS-Assembler
`a ← b + c;`	`add a, b, c`
`d ← a - e;`	`sub d, a, e`

... oder etwas komplizierter:

temporäre Variable

```
f←(g+h)–(i+j)
```

t0 t1

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Register

- allgemein
  - dienen zum Zwischenspeichern der Operanden
  - feste, identische Länge (Wortbreite, word):
    derzeit typischerweise 32 Bit (heute auch schon 64 Bit)
- MIPS:
  - **32 Register mit je 32 Bit**
  - **Register mit spezieller Funktion:**
    Register 0, 31
  - **frei benutzbare Register**
    Register 1, 2, ..., 30

MIPS GNU C Registerbenutzung

Benennung der Register entsprechend der Nutzung im GNU C Compiler
(vgl. Patterson/Hennessy, Seite A-23, Figure A.10)

Name	Nr.	Funktion
$zero	0	der konstante Wert 0
$at	1	reserviert für Assembler
$v0, $v1	2, 3	Resultatwerte und Ausdrückeevaluierung (not saved)
$a0 ... $a3	4-7	Argumente
$t0 ... $t7	**8-15**	**Zwischenwerte (temporär, not saved)**
$s0 ... $s7	**16-23**	**gesicherte Werte (saved)**
$t8 ... $t9	**24-25**	**weitere Zwischenwerte (not saved)**
$k0, $k1	26, 27	Betriebssystem (OS kernel)
$gp	28	globaler Zeiger (global pointer)
$sp	29	Zeiger auf Kellerspeicher (stack pointer)
$fp	30	Rahmenzeiger (frame pointer)
$ra	31	Rücksprungadresse (return address)

saved: sollen beim Procedure Call gerettet werden

Das „alte“ Beispiel mit Registern

Programmiersprache C

```
f =(g+h)-(i+j)
```

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

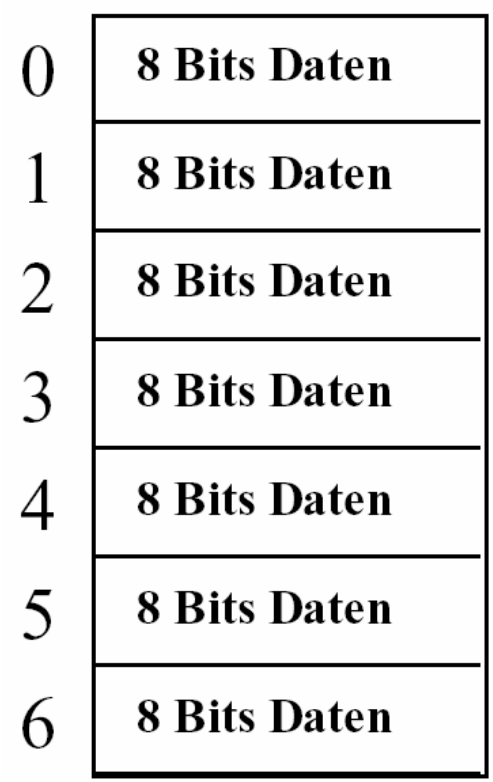
Die Variablen der C-Anweisung wurden der Reihe nach den Registern **`$s0 - $s4`** zugeordnet.

Aber: Wie kommen die Variablenwerte in die Register?

Load und Store

- Variablen, Felder und andere Datenstrukturen werden im Speicher gehalten.
- Zur Verarbeitung werden die Daten aus dem Datenspeicher gelesen und in die Register geladen (load). Die Ergebnisse werden anschließend zurückgeschrieben (store).
- weiteres dazu 6-28

Speicher-Organisation

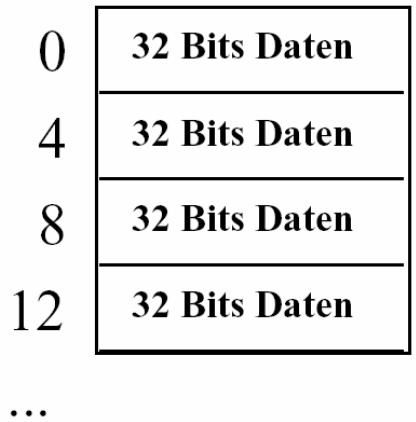


- **Speicher**: vorstellbar als großer, eindimensionaler Vektor von Bytes
- **Speicheradresse**: Der einzelnen Bytes des Speichers werden durchnummeriert.

...

Speicher-Organisation

- Bytes sind brauchbare Einheiten, jedoch werden größere „Worte“ benötigt.
- Für MIPS besteht jedes Wort aus 32 Bits oder 4 Bytes, ein Register kann ein Wort speichern.
- Der Speicher wird entweder als Bytevektor oder als Wortvektor interpretiert.
- Interpretation als **Bytevektor**
  - 2^{32} Bytes mit *Byteadressen* von 0 bis 2^{32} –1
- Interpretation als **Wortvektor**
  - 2^{30} Worten mit *Wortadressen* 0, 4, 8, ... 2^{32} – 4



Eine ***Byteadresse*** dient zum Adressieren eines Bytes.
Eine ***Wortadresse*** dient zum Adressieren eines Wortes.

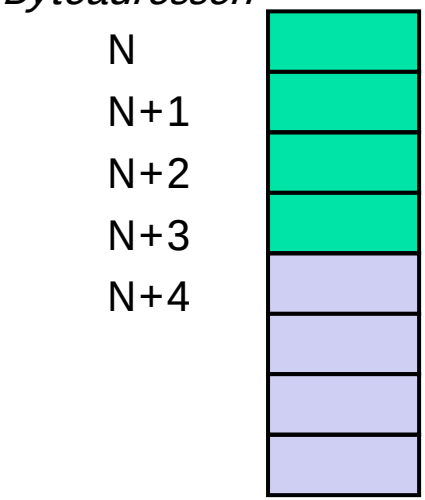
Alignment

- Worte dürfen bei MIPS nur an Adressen beginnen, die durch Wortgröße ohne Rest teilbar sind.
- **Ausgerichtet, Aligned**
  - Die Wortadresse ist durch 4 ohne Rest teilbar. Dadurch effizientere Speicher-(RAM-)Zugriffe, 1 Zugriff/Wort
- Der ***Wortindex*** ist die AlignedWortadresse/4
- Was bedeuten die niedrigsten 2 Bit einer Wortadresse?
  - Definieren ein Byte im Wort, abhängig von der Zählweise
    - von links nach rechts (big endian) oder
    - von rechts nach links (little endian) → weiteres 6-21,22

Byteadressierung und Wortadressierung

Byteadressierung

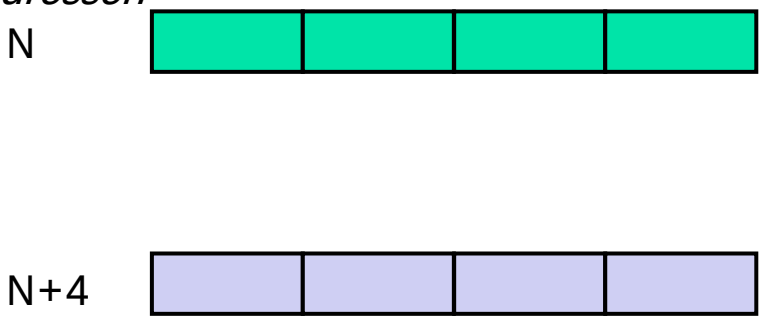
Byteadressen



Der Speicher wird als Bytevektor gesehen

Wortadressierung

Wortadressen



Der Speicher wird als Wortvektor gesehen

Index-Adressierung

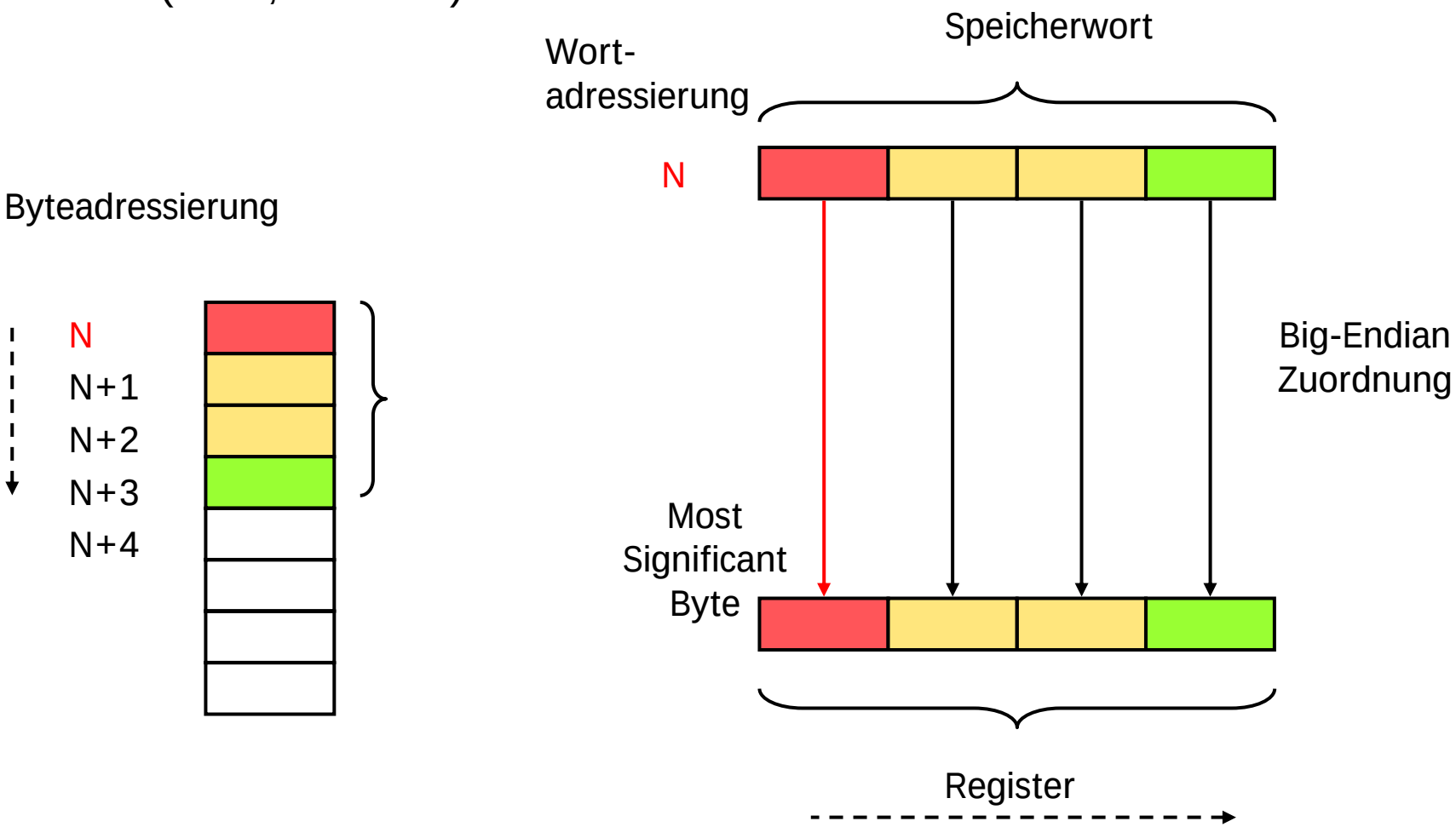
i = N/4

i+1 = (N+4)/4

Wortindex

Big-Endian Byte Ordering

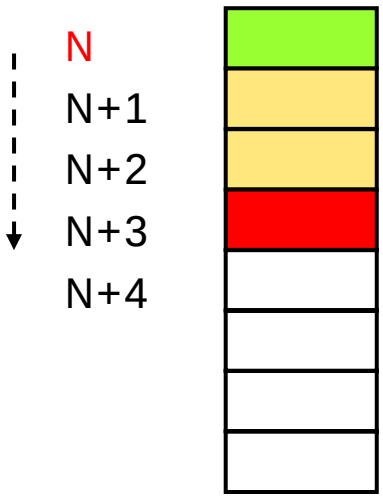
Big-Endian: Der Speicheradresse N ist das **Most Signficant Register Byte** zugeordnet.
(MIPS, Motorola)



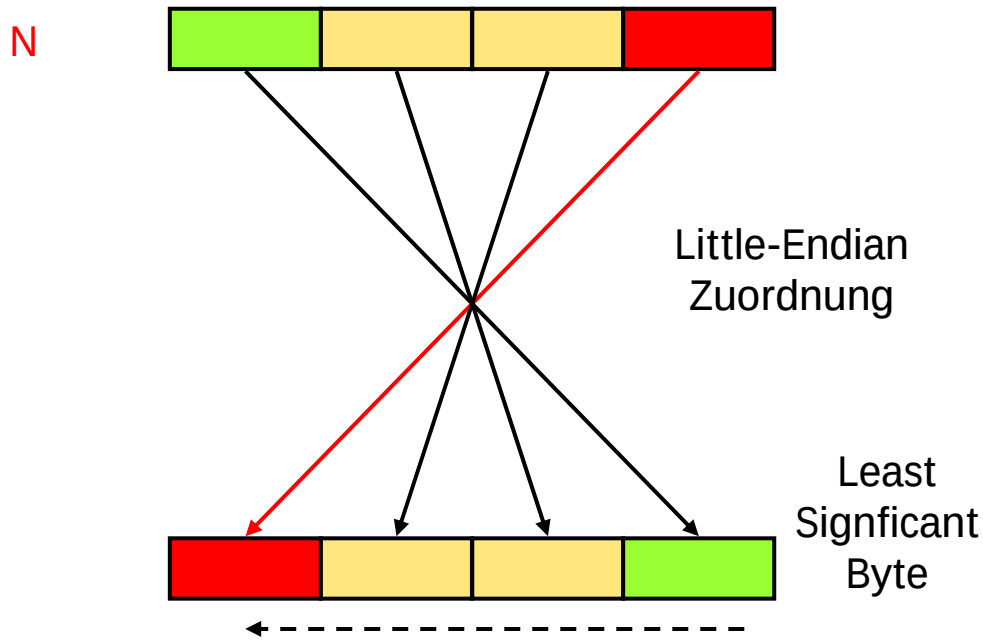
Little-Endian Byte Ordering

Little-Endian: Der Speicheradresse N ist das **Least Significant Register Byte** zugeordnet.
(Intel-Prozessoren)

Speicheradressierung
(Byteadressierung)



Wortadressierung



(Das Most Significant Register Byte
ist der Speicheradresse N+3 zugeordnet.)

Vergleich

- N sei aligned
- LoadWort von der Speicheradresse N in das Register
- Big-Endian und Little-Endian → dasselbe Ergebnis
- aber bei der byteweisen Übertragung
  - bei Big-Endian wird das Register von links aufgefüllt, während es bei Little-Endian von rechts aufgefüllt wird.
  - Wenn schon nach der Übertragung des ersten Bytes mit der Weiterverarbeitung begonnen wird, dann kann z. B. die Addition schneller bei Little-Endian durchgeführt werden.

MIPS Befehle (teilweise)

MIPS assembly language

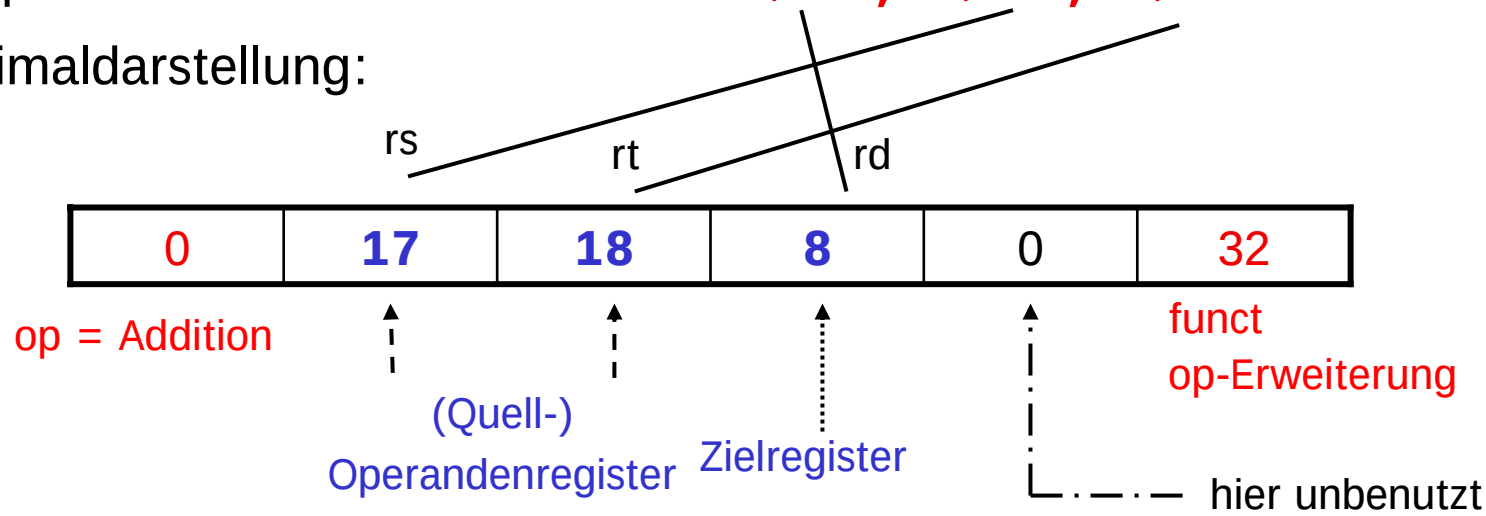
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 * 2^16`	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; for `beq`, `bne`
	set less than immediate	`slti $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Additionsbefehl (add)

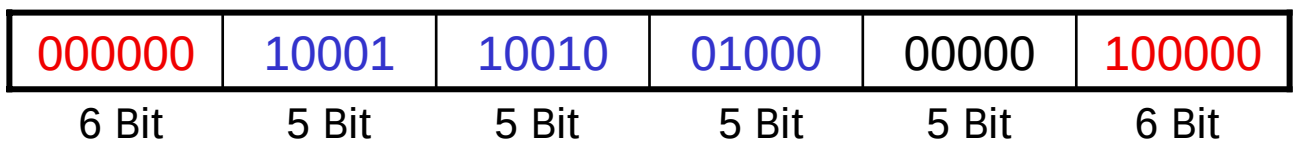
- Nicht nur Register und Speicherworte auch Befehle sind 32 Bits lang
- Jedes Register hat eine eigene Nummer:

  `$t0=8`, ..., **`$t7=15`**, **`$s0=16`**, ..., **`$s7=23`**

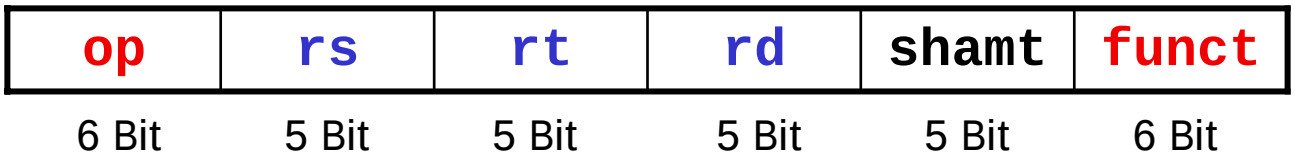
- Beispiel: MIPS-Assembler: **`add $t0, $s1, $s2`**
- Dezimaldarstellung:



- Binärdarstellung:



add: Verwendetes Befehlsformat „R“ (für Register)

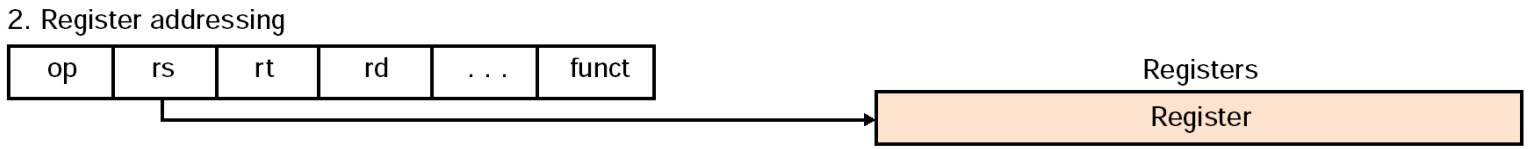


- **Arithmetisches Format, Register-Format**
  - **op**: „**opcode**“ ; Grundtyp des Befehls (instruction)
  - **rs**: erster Registeroperand (Registernummer)
  - **rt**: zweiter Registeroperand (Registernummer)
  - **rd**: Zielregister (Registernummer)
  - **shamt**: „**shift amount**“, für sogenannte Schiebeoperationen wichtig (siehe später)
  - **funct**: „**Funktionscode**“; Ergänzung zum **op**-Feld: genaue Spezifikation der durchzuführenden Operation

  - Zusammenfassung der MIPS-Architektur bis hierher siehe Patterson/Hennessy, Seite 121, Figure 3.6

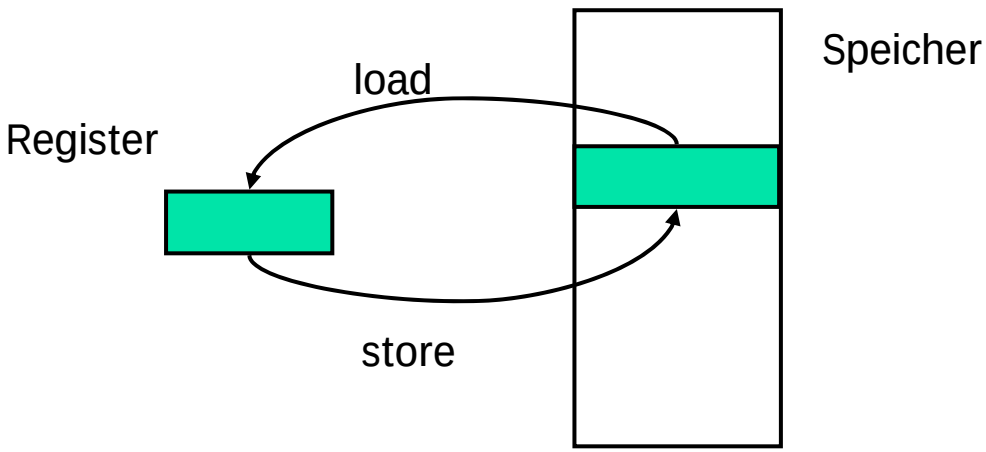
add: Verwendete Adressierungsart

- Registeradressierung: Operand ist Registerinhalt. Im Befehl stehen die Registeradressen.
  - rs : register source1
  - rt : register source2
  - rd: register destination



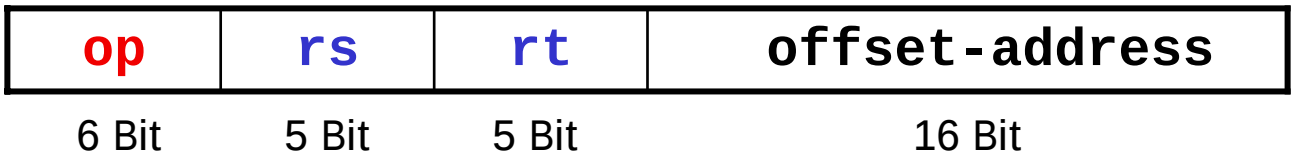
Ladebefehl (lw) und Speicherbefehl (sw)

- Ladebefehl
  - **lw**: **l**oad **w**ord (lade Wort),
  - Datenübertragung Speicher → Register
- Speicherbefehl
  - **sw**: **s**tore **w**ord (speichere Wort)
  - Datenübertragung Register → Speicher



lw und **sw** sind die einzigen Befehlstypen, die auf Speicher zugreifen!
LOAD-STORE-Architektur

load/store: Verwendetes Befehlsformat „I“

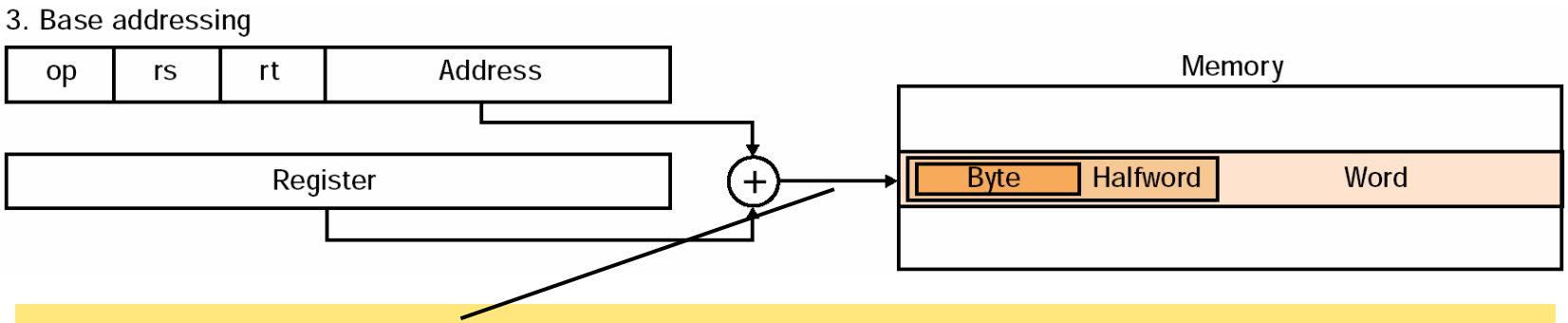


- Datentransfer-Format
  - **op**: opcode
  - **rs**: Nummer des Basisadressregisters
  - **rt**: Zielregister (Registernummer)
  - **offset-address**: Konstante, die Offset bzgl. Basisregister angibt
  - **lw**, **sw** benötigen 2 Register und 1 Konstante als Array-Offset

- **op**-Kodierung von **lw** ist 35 (dezimal) bzw. 100011 (binär)
- **op**-Kodierung von **sw** ist 43 (dezimal) bzw. 101011 (binär)

load/store: Verwendete Adressierungsart

- Basisadressierung
  - Effektive Speicheradresse (des Speicheroperanden) ist die Summe von Registerinhalt und Offset-Address im Befehl.



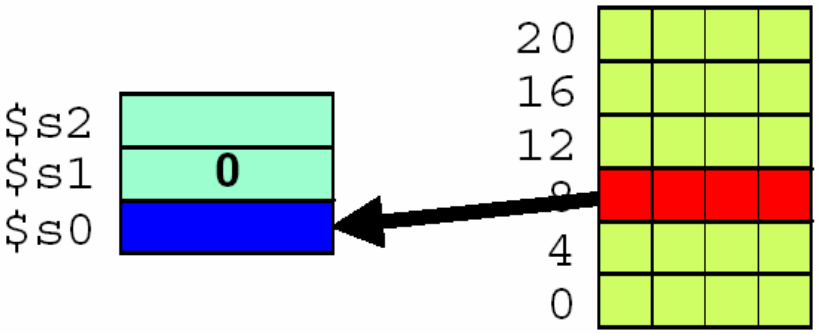
EffektiveAdresse = Konstante + Inhalt des Basisregisters

Konstante (Basisregister)

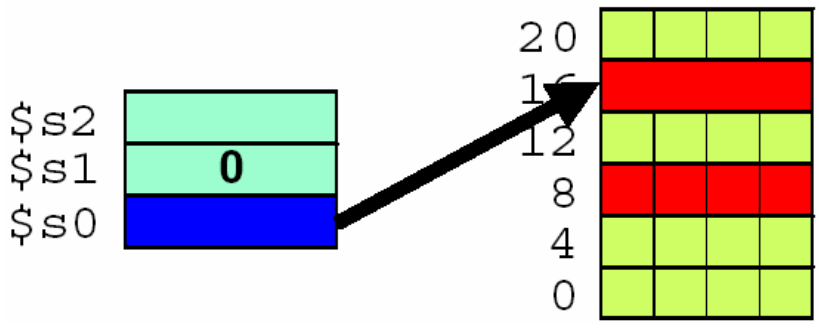
z. B. **32($s3)**

Beispiel load und store

lw $s0, 8($s1) // s0 ← m[8+s1]



sw $s0, 16($s1) // s0 → m[16+s1]



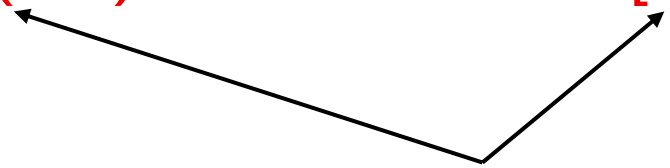
Bemerkung: Die Konstante wird auch als Offset bezeichnet, eine konstante Distanz zu der Basisadresse. Die Basisadresse kann auch als Indexregister fungieren, d. h. durch Inkrementieren der Basisadresse können fortlaufende Elemente eines Vektors adressiert werden.

Beispiel

C: A[8] = h + A[8] // int A[256]; wobei int=1 Wort

MIPS:

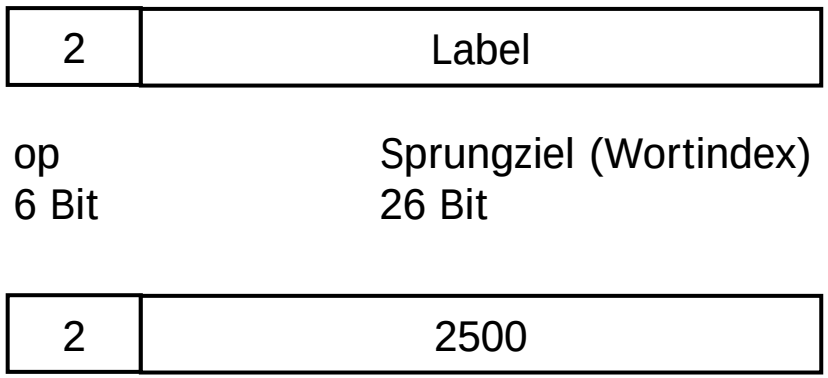
```
lw    $t0, 32($s3)     //t0 ← m[32+s3]
add   $t0, $s2, $t0    //t0 ← s2+t0
sw    $t0, 32($s3)     //t0 → m[32+s3]
```



bei **sw** steht die Zieladresse rechts

Unbedingter Sprungbefehl

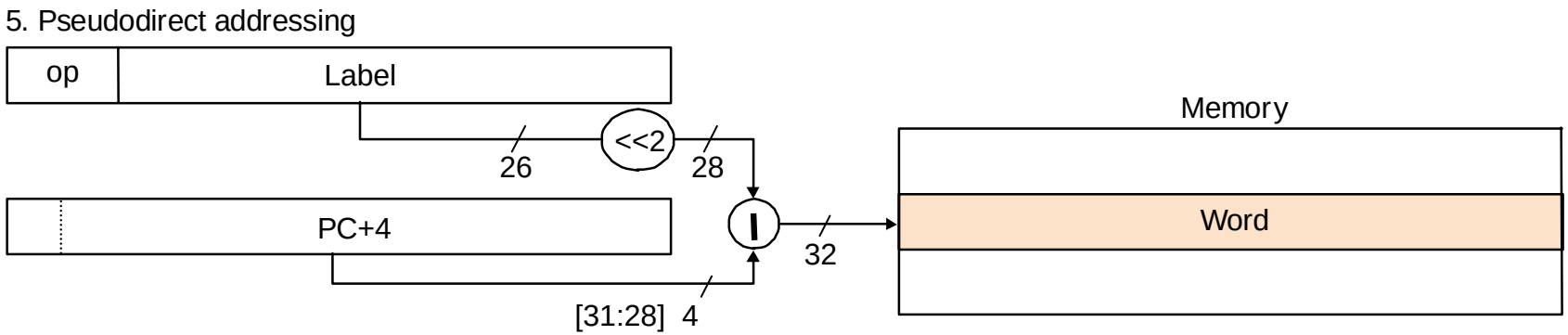
- **j Label** (jump to Label)
- Der Befehlszähler PC wird auf Label gesetzt. Es erfolgt ein unbedingter Sprung an die Stelle Label (Sprungziel, target address).
- Befehlsformat J (für Jump)



Beispiel: j 10000 (Springe an die Programmadresse 10000=4*2500)

jump: Verwendete Adressierungsart

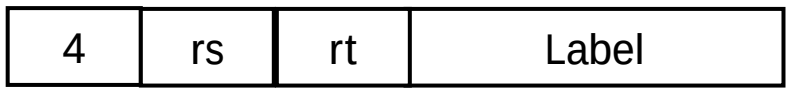
5. Pseudodirekte Adressierung: Sprungzieladresse setzt sich aus 26 Bits im Befehl (Label), verschoben nach links um 2 Bitpositionen und den oberen 4 Bits des PC (inkrementiert um 4) zusammen.



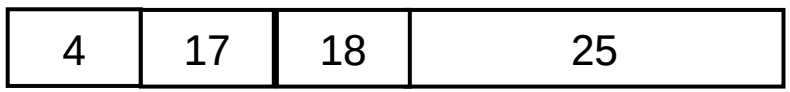
Maximal in 256MB (2^{28} Bytes) Addressbereich springbar.
Weiter nur mit Tricks (siehe später)

Bedingter Sprung

- **beq rs, rt, Label** (branch if equal)
- Wenn die Registerinhalte gleich sind, wird PC auf Label gesetzt.
- Befehlsformat I (für Immediate)

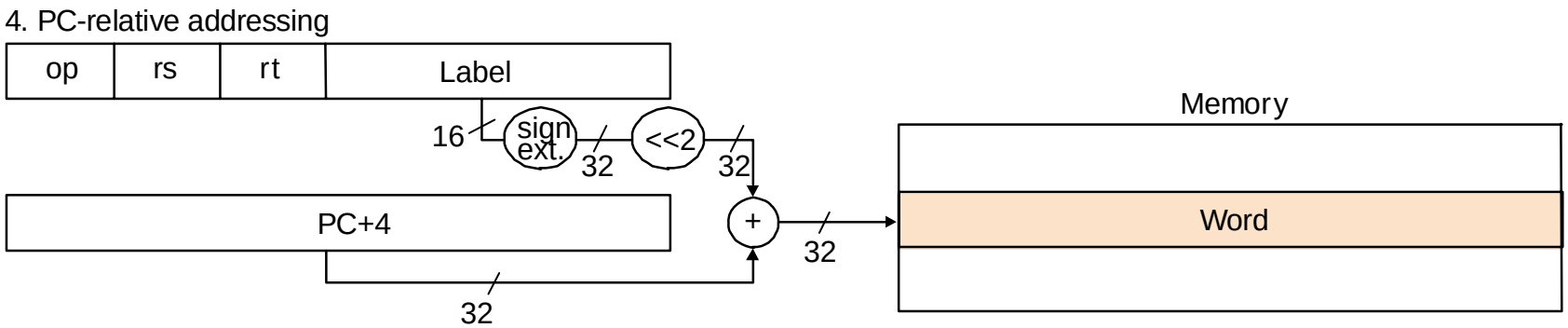


op 6 Bit | reg1 5 Bit | reg2 5 Bit | Sprungziel (Wortindex) 16 Bit | relativ zu PC



beq: Verwendete Adressierungsart

- Wie kann man Sprungziele größer als 2^{16}-1 realisieren?
- Beobachtung: Verzweigungen in Schleifen und Fallunterscheidungen haben Sprungadressen „in der Nähe“ der aktuellen Instruktion
- PC-relative Adressierung: Sprungzieladresse (branch target address) ist Summe von PC und Sprungadresse (Label) im Befehl, verschoben nach links um 2 Bitpositionen plus PC plus 4

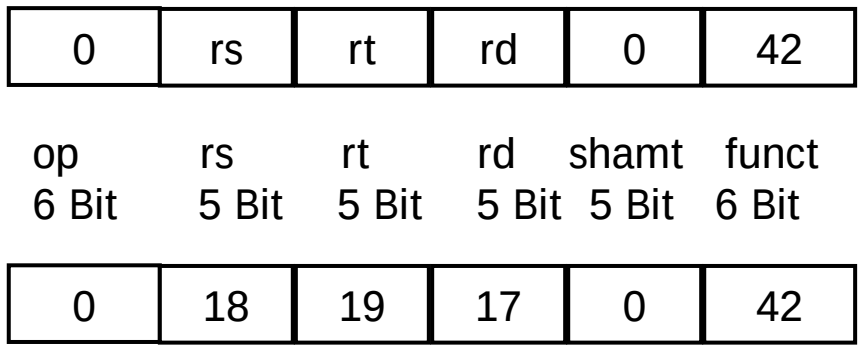


Bedingter Sprung

- **bne rs, rt, Label** (branch if not equal)
- Wenn die Registerinhalte ungleich sind, wird PC auf Label gesetzt.
- op = 5
- ansonsten wie beq

Compare

- **slt rd, rs, rt** (set on less than)
- if rs < rt then rd:=1 else rd:=0
- Die Bedingung < wird berechnet und in rd gespeichert. Sie kann dann anschließend durch beq oder bne zum Verzweigen benutzt werden.
- Format R (für Register)

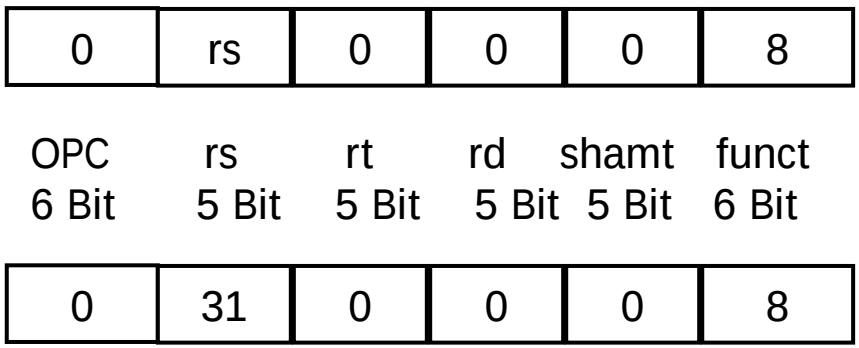


Beispiel: slt $s1,$s2,$s3

vergl. 6-49

Register-indirekter Sprung

- **jr rs** (jump register)
- Springe zu der Programmadresse, die in dem Register rs steht
- Anwendungen: case, Unterprogramm-Rücksprung
- Format R

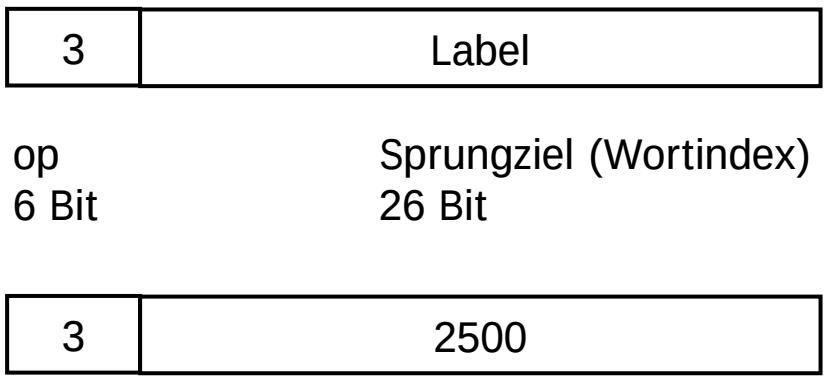


Beispiel: jr $ra

Volle 4GB (2^{32}) anspringbar

Unterprogramm-Sprung

- **jal Label** (jump and link)
- Springe zu der Programmadresse Label (Anfang des Unterprogramms und rette Rücksprungadresse PC+4 in $ra)
- Anwendung: Sprung zu einem Unterprogramm
- Format J



Beispiel: 256: jal 10000 (Springe an die Programmadresse 10000=2500*4)
und merke 256+4 in $ra=31

Schritte bei Ausführung einer Prozedur

(Genaueres über die Programmierung von Prozeduren ist Stoff von GDI 3)

1. Parameter bereit stellen, so daß Prozedur darauf zugreifen kann:
   - Argumentregister **`$a0, $a1, $a2, $a3`**
   - Eingangsparameter bleiben durch Prozedur unverändert
2. Ablaufsteuerung (control) der Prozedur übergeben:
   - **`jal Prozeduradresse`** (jump and link)
   - Sonderbefehl für Prozedurausführung: springt zur angegebenen Adresse und speichert die Adresse der folgenden Instruktion in **`$ra`**
3. Benötigten Speicherbereich für Prozedur bereitstellen
4. Prozedur ausführen
5. Ergebniswert so bereitstellen, daß aufrufendes Programm darauf zugreifen kann:
   - Ergebniswertregister **`$v0,$v1`**
6. Ablaufsteuerung an Aufrufstelle zurückgeben: durch Sprung zurück zum Ausgangspunkt:
   - **`jr $ra`** (jr: jump register, ra: return address)

Zeichen und byteweise Zugriffe

- „Daten“ sind nicht immer nur Zahlen, häufig auch Texte, d. h. Zeichenketten (strings).
- Einzelne Zeichen (character) werden durch einzelne Bytes repräsentiert.
- Instruktionssätze verfügen daher über Byte-Lade- und Byte-Speicher-Operationen zur Zeichenverarbeitung.
- Ladeoperation: **`lb $t0, 0($sp)`** (load byte)
  - Lädt ein Byte aus dem Speicher und platziert es in die rechts gelegenen (niederwertigen) Bits eines Registers.
- Speicheroperation: **`sb $t0, 0($a1)`** (store byte)
  - Speichert die niederwertigen acht Bits eines Registers in die bezeichnete Speicherposition

Bisherige Adressierungsarten

Schwächen der bisherigen Instruktionen:

- Die Verwendung von Konstanten ist unhandlich:
  - Man muß entweder ein spezielles Register bereitstellen ($zero) oder muß die Konstanten im Speicher an bekannten Stellen ablegen, bzw. erst in ein Register laden.
  - Da Konstanten (z. B. für Inkrement/Dekrement-Operationen bei Datenzugriffen in Schleifen) häufig auftreten, ist ein verbesserter Zugriff auf Konstanten wünschenswert.

  → *Effizienter Zugriff speziell auf kleine Konstanten wünschenswert !*

Konstanten als Direkt-Operanden im Befehl

- R-Befehlsformat (Wdh. Wert im Register):

op	rs	rt	rd	shamt	funct
6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit

Beispiel: MIPS: `add $t0, $s1, $s2` # $t0 = $s1 + $s2

Dezimal-darstellung						
	0	17	18	8	0	32

- **I-Befehlsformat** (I: immediate, unmittelbar, direkt):

op	rs	rt	immediate
6 Bit	5 Bit	5 Bit	16 Bit

Beispiel: MIPS: `addi $sp, $sp, 4` # $sp = $sp + 4

Dezimal-darstellung				
	8	29	29	4

Konstanten als Direkt-Operanden

Weiteres Beispiel:

- bisher **`slt $t0, $s1, $s2`** # falls $s1 < $s2: $t0=1, sonst $t0=0
- Immediate Adressierung

`slti $t0, $s1, 10` # falls $s1 < 10: $t0=1, sonst $t0=0

Dezimal-darstellung	10	8	17	10
Binär-darstellung	001010	01000	10001	0000 0000 0000 1010

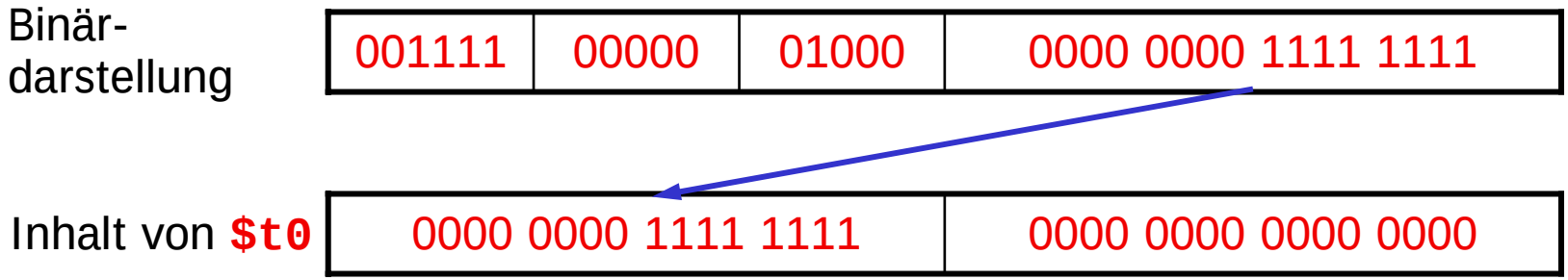
- Vorteil der direkten Adressierung: Die Verwendung häufig vorkommender, kleiner Konstanten wird effizient (schnell) ausgeführt !

Frage: Was tun, wenn Konstante größer als $2^{16}-1 = 65535$ ist, und mehr als 16 Bits zur Darstellung benötigt werden?

Nicht ganz richtig, Verfeinerung auf 6-50!

Behandlung großer Konstanten

- maximal 32 Bits große Konstanten (d. h. $2^{32}-1 = 4{,}29... \times 10^9$)
- Der Befehl **`lui`** (load upper half word immediate) lädt angegebene Konstante in die höherwertigen 16 Bits des Zielregisters.
- Beispiel: **`lui $t0, 255`**

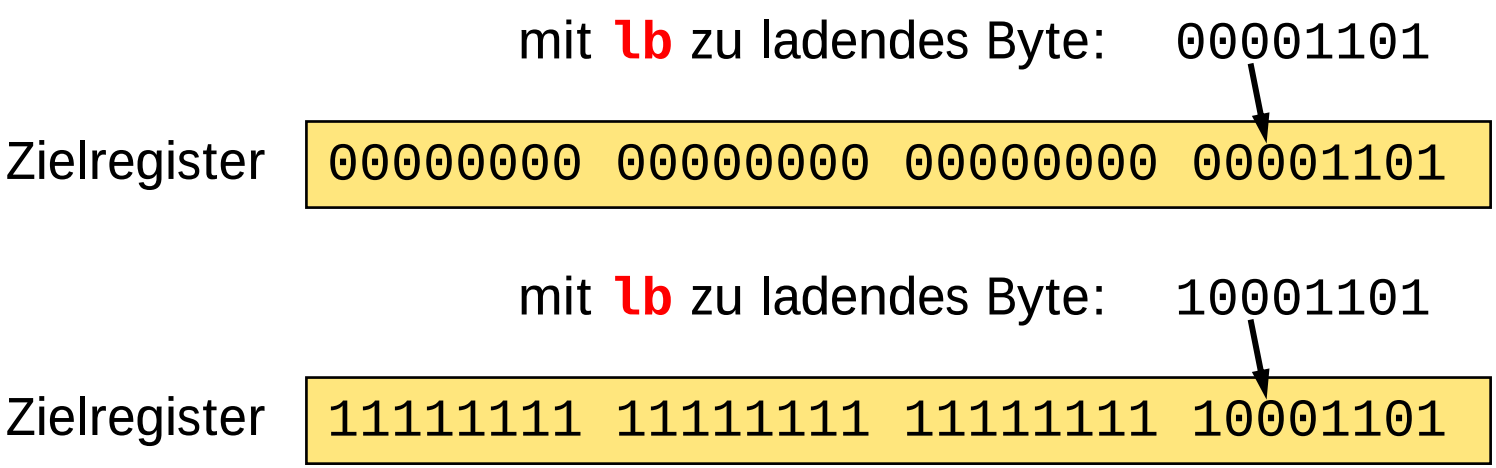


- Setzen der niederwertigen 16 Bits von **`$t0`** z. B. mit

`addi $t0, $t0, 2304`

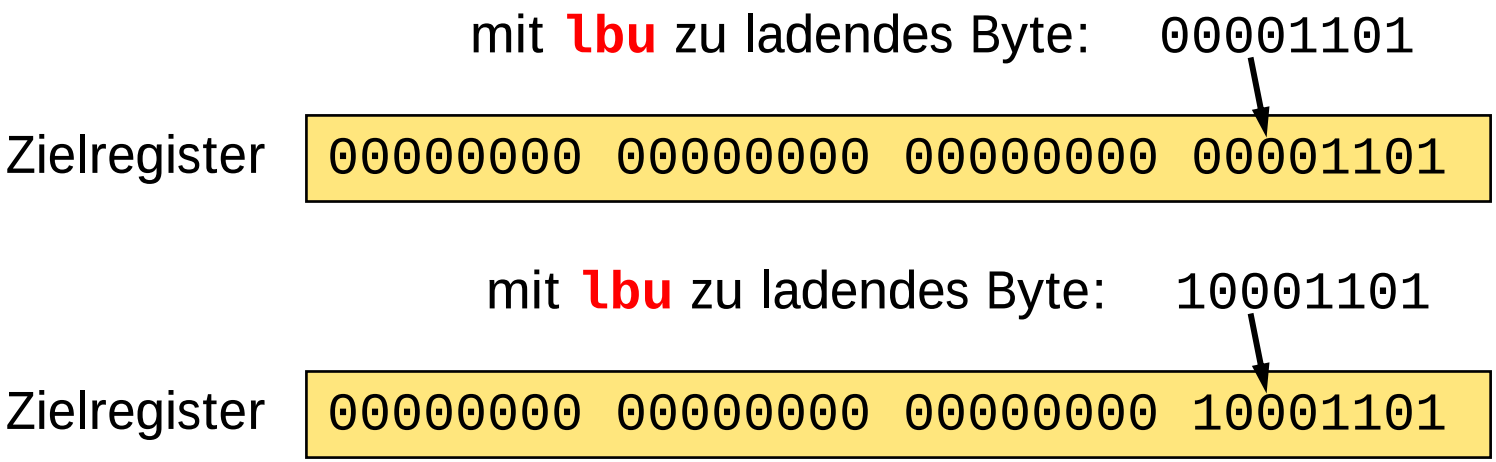
Byte-Ladebefehl Signed

- `lb` (load byte) interpretiert das zu ladende Byte als vorzeichenbehaftete Zahl, d. h.
- wenn das vorderste Bit „0“ ist, müssen die vorderen 24 Bit des Zielregisters ebenfalls „0“ sein,
- wenn das vorderste Bit „1“ ist, müssen die vorderen 24 Bit des Zielregisters ebenfalls „1“ sein.
- Die Operation, die das durchführt, heißt „sign extension“.



Byte-Ladebefehl Unsigned

- **lbu** (load byte unsigned) interpretiert das zu ladende Byte als vorzeichenlose Zahl, d. h.
- unabhängig vom Wert des zu ladenden Bytes werden die oberen 24 Bits des Zielregisters auf „0“ gesetzt.



- Da ein Zeichen (character) in der Regel durch ein Byte dargestellt wird (und Zahlen durch mehr als ein Byte), wird **lbu** fast ausschließlich verwendet.

Vergleichsbefehl Signed/Unsigned

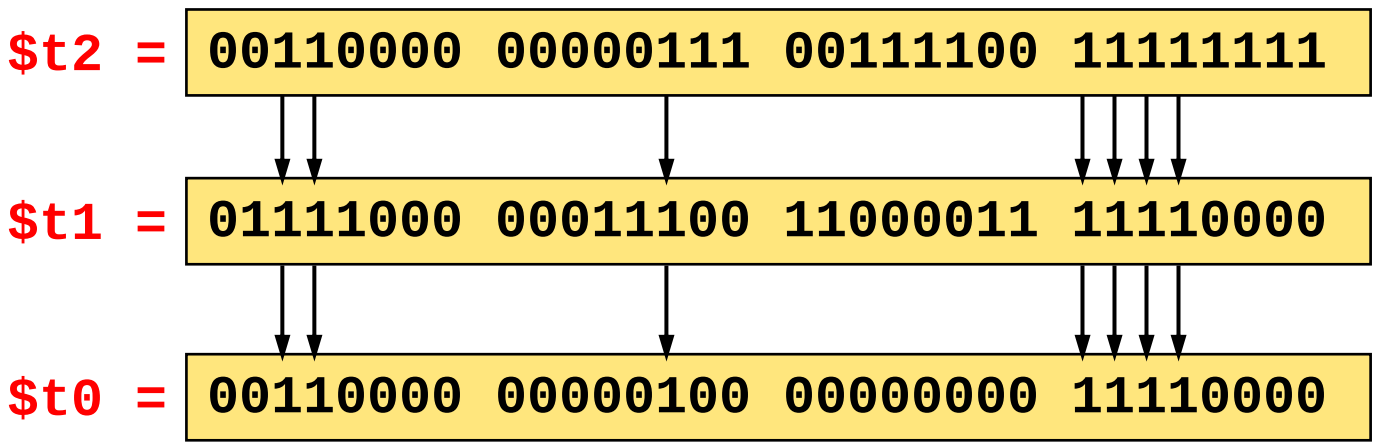
- **slt** und **slti** (set on less than [immediate]) interpretieren die zu vergleichenden Bitfolgen als vorzeichenbehaftet,
  - d. h. Zahlen mit einer „1“ im vordersten Bit sind kleiner als Zahlen mit einer „0“ ganz vorne. -> Zweierkomplement (2K)
- **sltu** und **sltiu** (set on less than [immediate] unsigned) interpretieren die zu vergleichenden Bitfolgen als vorzeichenlos,
  - d. h. Zahlen mit einer „1“ im vordersten Bit sind größer als Zahlen mit einer „0“ ganz vorne.
- Beispiel:

```
$s0 = 11111111 11111111 11111111 11111111
$s1 = 00000000 00000000 00000000 00000001
```

```
slt  $t0, $s0, $s1  →  $t0 = 1
sltu $t0, $s0, $s1  →  $t0 = 0
```

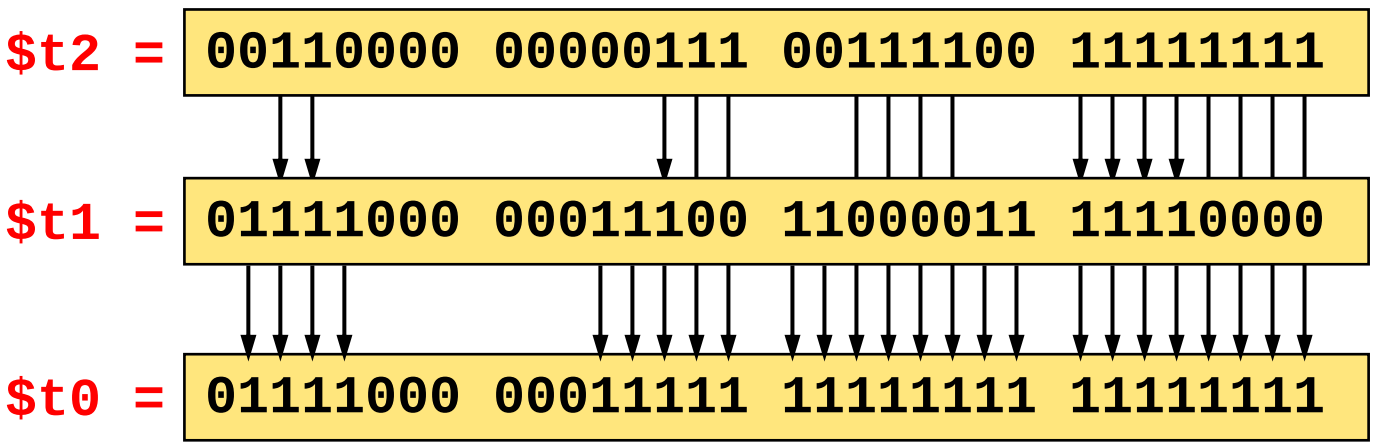
Logische Operationen

- Bitweises logisches UND (**and**):
  - Setzt alle Bit-Positionen des Zielregisters auf „1“, die sowohl im ersten als auch im zweiten Quellregister auf „1“ stehen.
  - Beispiel: `and $t0, $t1, $t2`



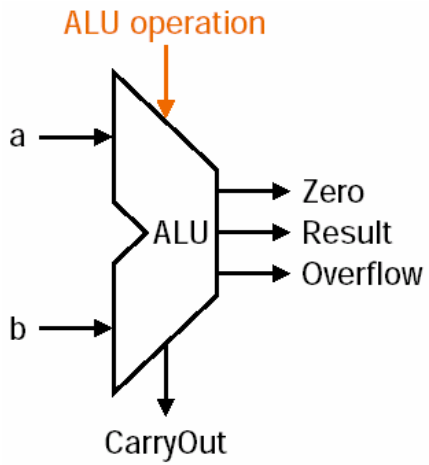
Logische Operationen

- Bitweises logisches ODER (**`or`**):
  - Setzt alle Bit-Positionen des Zielregisters auf „1“, die entweder im ersten oder im zweiten Quellregister auf „1“ stehen.
  - Beispiel: **`or $t0, $t1, $t2`**

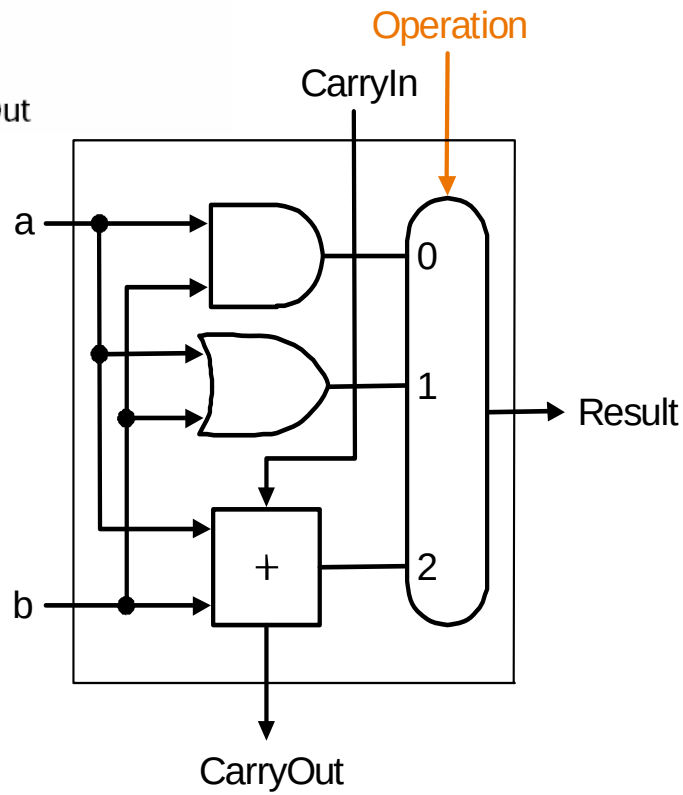


Konstruktion der ALU für MIPS

Ziel



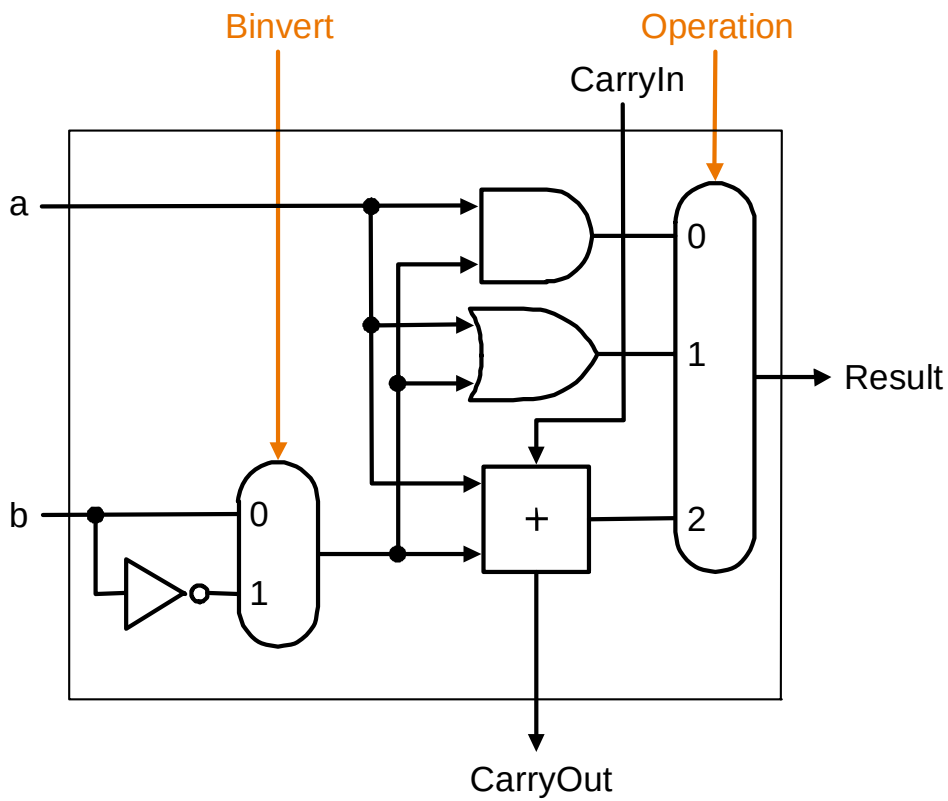
1-Bit ALU



- Im folgenden wird der Entwurf einer ALU für MIPS beschrieben. Dabei wird die ALU schrittweise vervollständigt.
- (1. Schritt) : 1-Bit-ALU
- Diese 1-Bit-ALU realisiert nur die Operationen AND, OR und Addition.
- Auswahl der Operation über einen Multiplexer

Erweiterung 1-Bit ALU für Subtraktion (a-b)

(2. Schritt)



- a - b = a + (-b)
- Subtraktion wird auf Addition und Komplementbildung zurück geführt.
- (-b): Zweier-Komplement bedeutet: bitweise Invertierung von b und Addition von 1

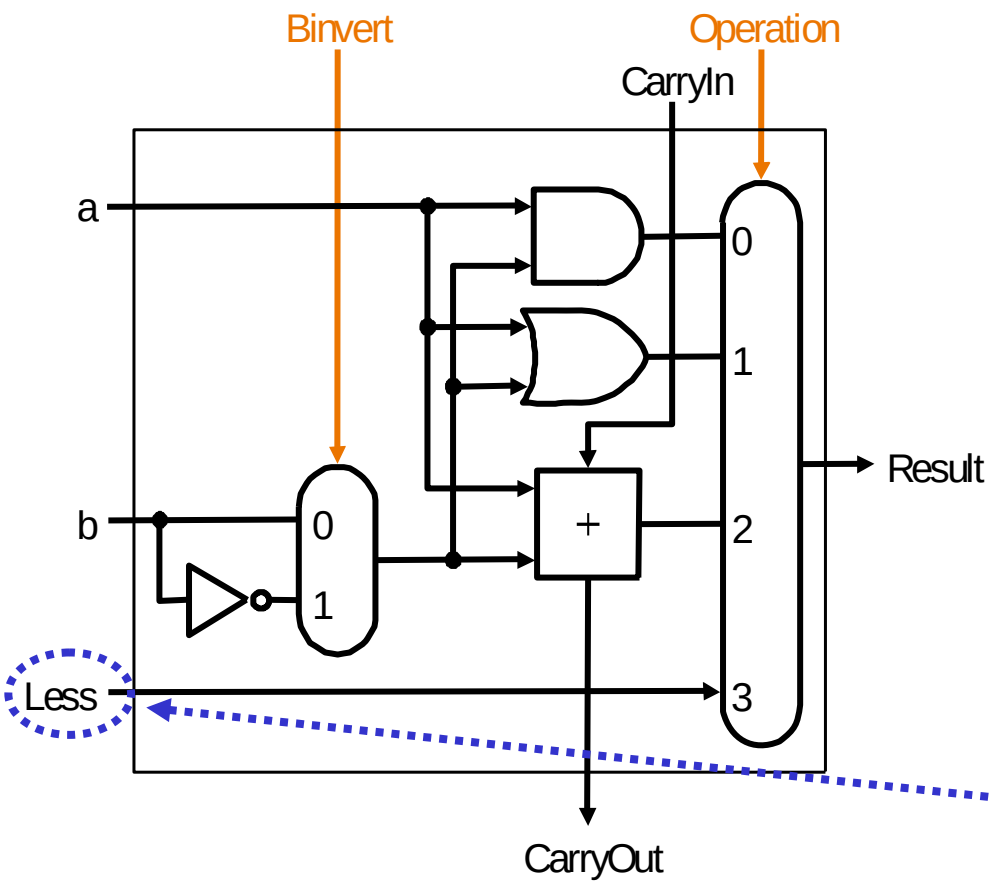
Hardware-Erweiterungen:

- Jeder 1-Bit-Block wird an einem Eingang um eine Inverterschaltung und einen Multiplexer erweitert.
- Bei der Gesamtschaltung wird der CarryIn-Eingang der niederwertigsten Bit-Stufe bei einer Subtraktion auf „1“ gesetzt.

Vergleich durch Subtraktion

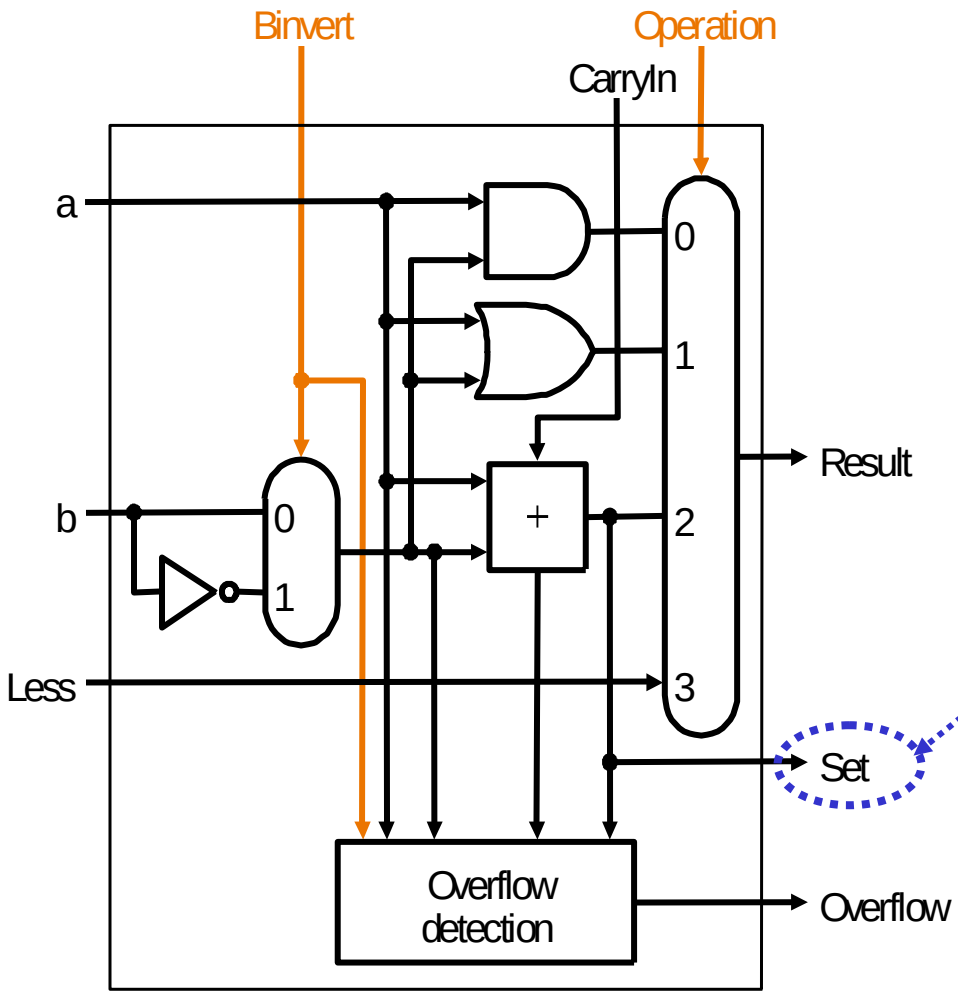
- Zusätzlich wird die Subtraktion benötigt, die auch zum Vergleichen verwendet wird.
- Realisierung der set-on-less-than Instruktion (**`slt`**)
  - Zur Erinnerung: **`slt`** ist eine arithmetische Instruktion
  - produziert 1, falls rs < rt, und 0 sonst
  - verwende Subtraktion: (a – b) < 0 impliziert a < b
- Realisierung des Gleichheitstests:
  - **`beq $t5, $t6, $t7`**
  - verwende Subtraktion: (a – b) = 0 impliziert a=b

(3. Schritt) Erweiterungen für `slt` (1)



- Die Operation
  `slt $t0, $s0, $s1`
  ist definiert als:
- ```
  if ($s0 < $s1)
      $t0 = 1;
  else
      $t0 = 0;
  ```
- Um dies zu realisieren, muß die ALU die 32-Bit Konstanten „0“ und „1“ erzeugen können.
- Dazu wird jede Bit-Stelle zunächst um einen „Less“-Eingang erweitert, der für alle Bits (außer dem niederwertigsten) den Wert „0“ trägt.
  ```

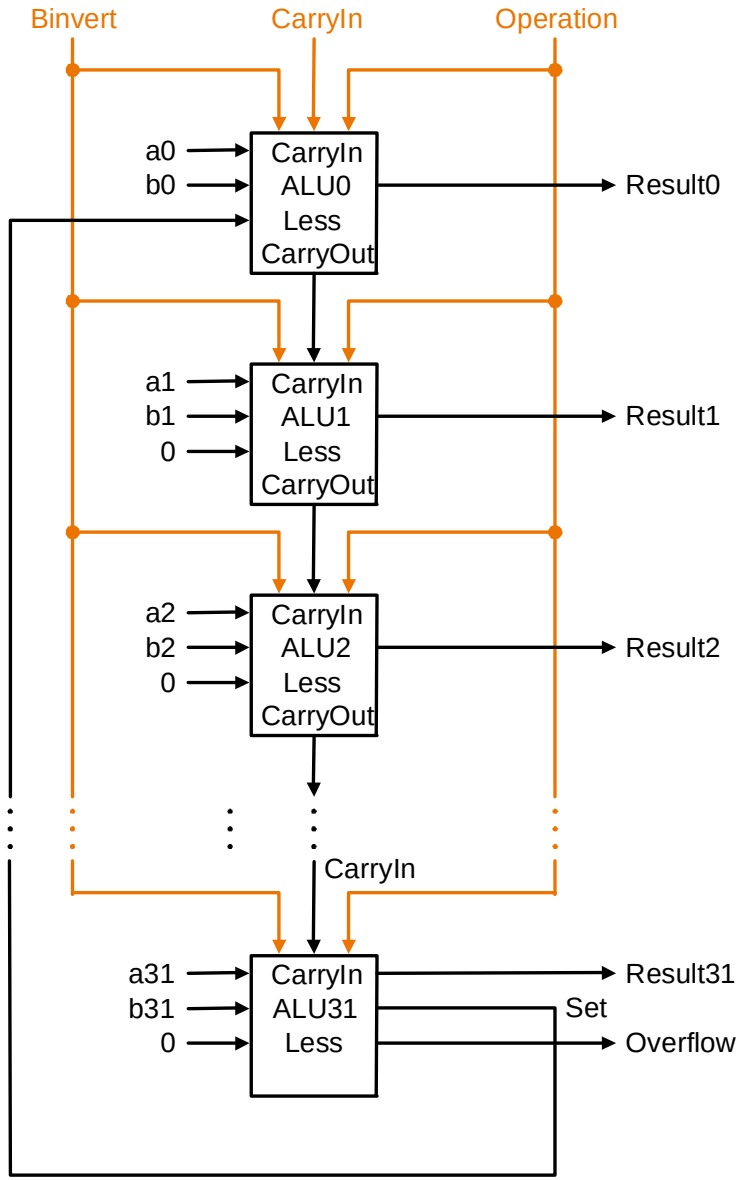
Erweiterungen für `slt` (2)



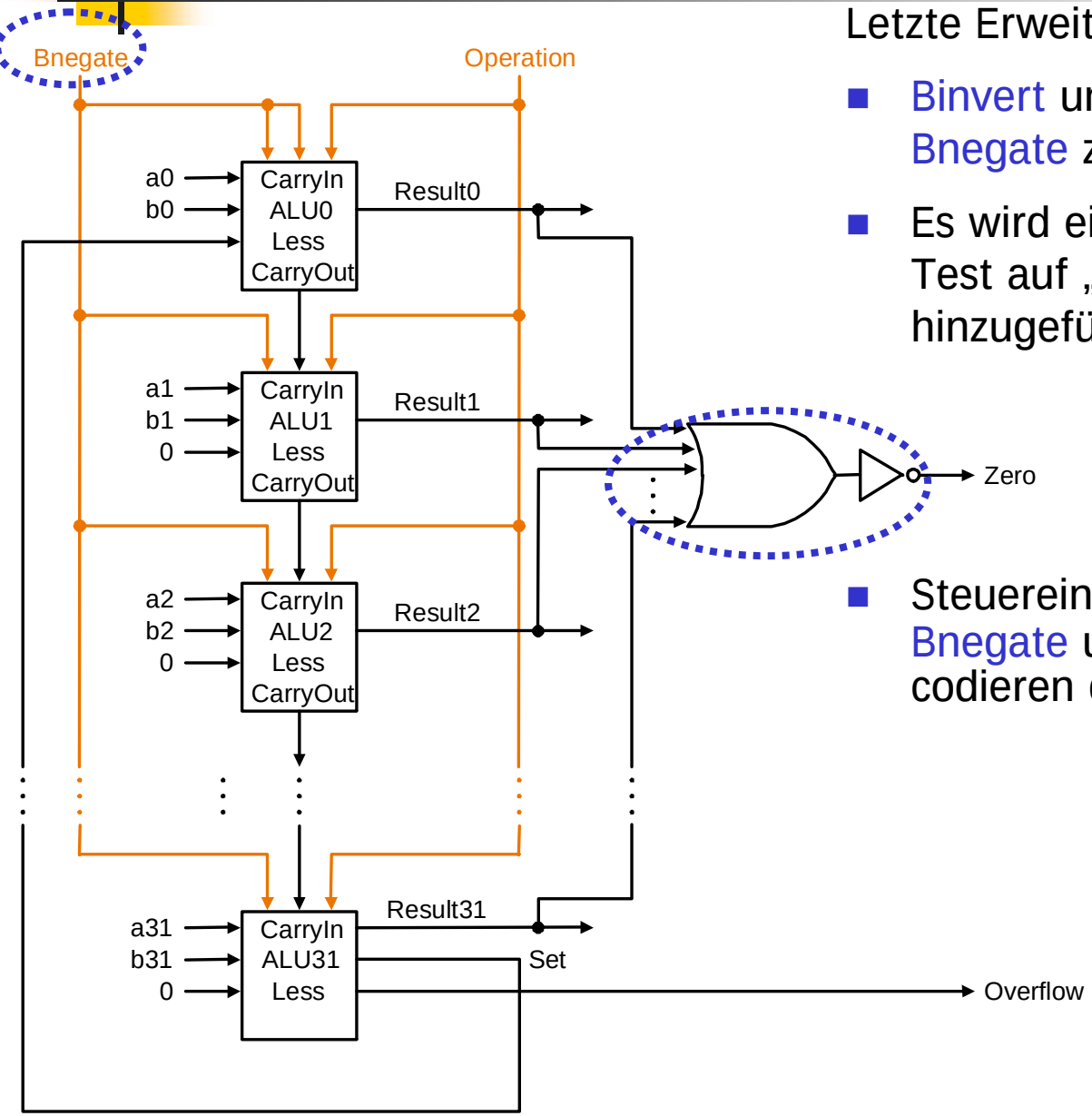
- Das niederwertigste Bit muß abhängig vom Vergleich von a und b auf „0“ oder „1“ gesetzt werden.
- Es gilt:
  `a < b` ⇔ `a-b < 0`
  `a < b` ⇔ `a31.Result = 1`
- Das Resultat-Bit der höchstwertigen Stelle wird also als Set-Ausgang herausgeführt und an den Less-Eingang des niederwertigsten Bits angelegt.
- Zusätzlich erhält das höchstwertige Bit eine Overflow-Detektion.

Resultierende 32-Bit ALU

- Die vollständige 32-Bit Schaltung mit Hardware für die **`slt`**-Operation sieht also aus wie hier dargestellt.



(4. Schritt) Vollständige 32-Bit ALU



Letzte Erweiterungen:

- Binvert und CarryIn werden zu Bnegate zusammengefaßt.
- Es wird eine einfache Schaltung zum Test auf „0“ mit dem Ausgang Zero hinzugefügt.
- Steuereingänge: Die drei Leitungen Bnegate und Operation (2 Bit) codieren die ALU-Operationen:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

Was haben wir bis jetzt erreicht ?

- Eine 32-Bit ALU für MIPS-Befehlssatz läßt sich schrittweise entwerfen:
  - Multiplexer zur Auswahl des gewünschten Resultates verwenden.
  - Subtraktion kann durch Addition des Zweier-Komplements effizient realisiert werden.
  - Durch mehrfache Verwendung einer 1-Bit ALU kann 32-Bit ALU realisiert werden.
- Heute: Verwendung von Synthese-Tools
  - Eingabe: Hardware-Beschreibung, z.B. in Verilog
  - Ausgabe: optimierte ALU
  - Benutzer: Steuerung der Synthese über geeignete Beschreibungsformen, Modularisierungen, Verwendung von bestimmten Komponenten, Einstellung von Parametern.

Insgesamt: Ganzzahlarithmetik in MIPS

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants. Hi and Lo contain the results of multiply and divide.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

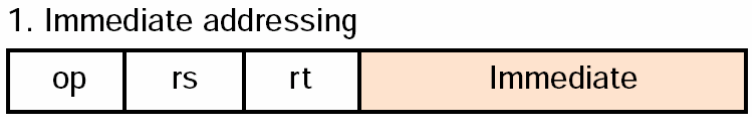
2 neue Register

MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + $s3	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

Zusammenfassung der Adressierungsarten

1. Immediate-Adressierung (Konstantendressierung):
   Operand ist Konstante im Befehl selbst.



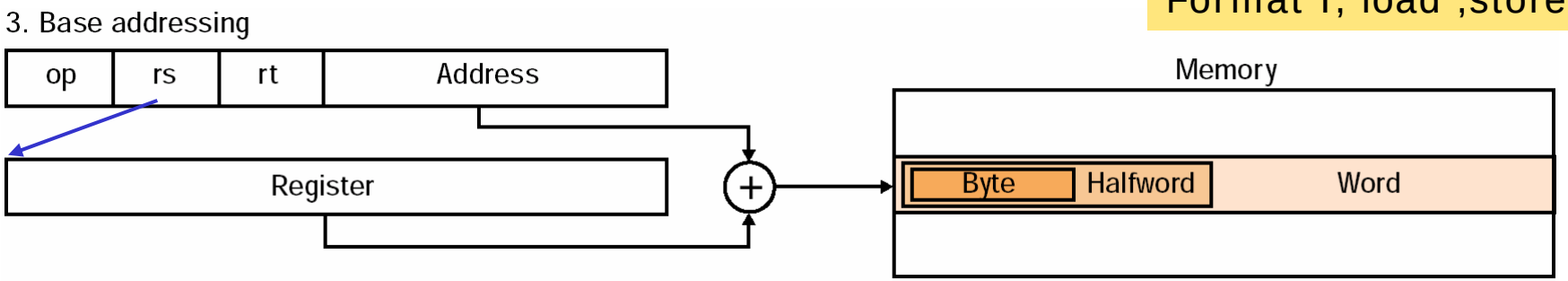
Format I: addi rs,rt, IMMEDIATE

2. Registeradressierung: Operand ist Registerinhalt

Format R

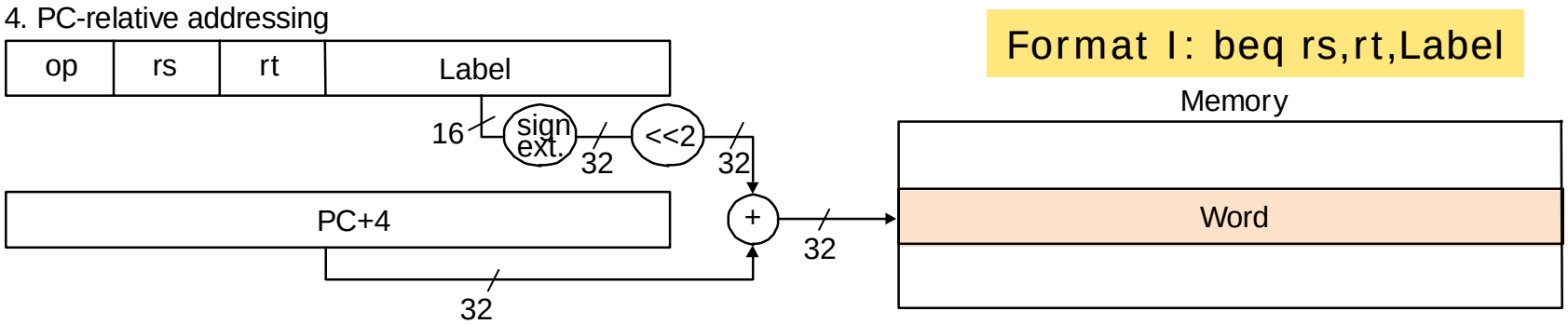
3. Basisadressierung: Operandenadresse ist Summe von Registerinhalt und Offset-Address im Befehl

Format I, load ,store

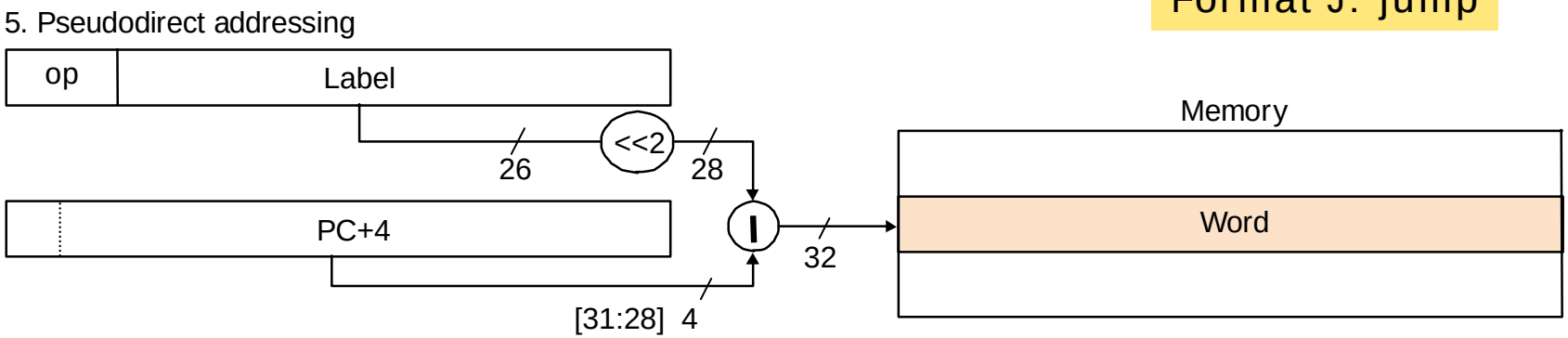


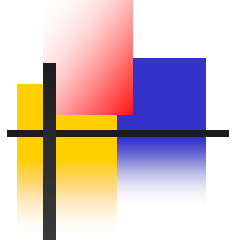
Zusammenfassung der Adressierungsarten

4. PC-relative Adressierung: Sprungzieladresse (branch target address) ist Summe von PC und Sprungadresse (Label) im Befehl, verschoben nach links um 2 Bitpositionen plus PC plus 4



5. Pseudodirekte Adressierung: Sprungzieladresse setzt sich aus 26 Bits im Befehl (Label), verschoben nach links um 2 Bitpositionen und den oberen 4 Bits des PC (vorher plus 4) zusammen





Gleitkomma-Arithmetik in MIPS

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	$f2 = $f4 – $f6	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	$f2 = $f4 × $f6	FP. multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	$f2 = $f4 – $f6	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Condi-tional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

Gleitkomma-Arithmetik in MIPS

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits