Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

HARDWARE / SOFTWARE-SCHNITTSTELLEN

Prozessor-Design

23. April 2014

Robert Kaiser

Technische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**

GRUNDSÄTZLICHES

ENDLICHER AUTOMAT

Abstrakt betrachtet ist ein Computer ein *endlicher Automat*:

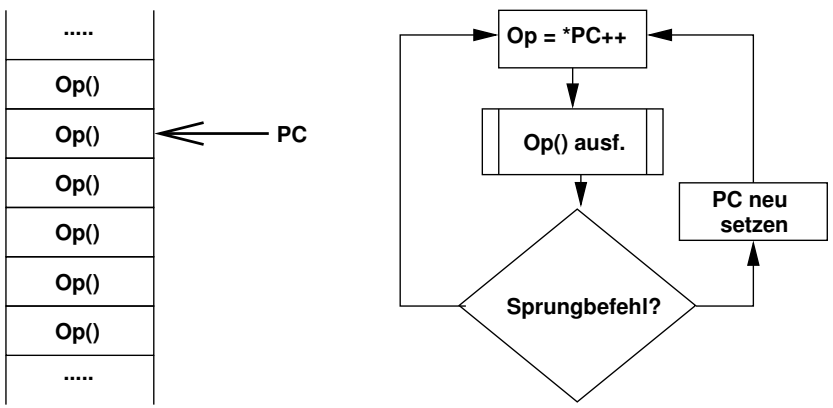
- → $Neuerzustand = Operation(Alterzustand)$
- → **Digital**rechner: Zustände werden durch Bitmuster dargestellt
- → Anzahl der Zustände: $2^{AnzahlBits}$
- → Beispiel: 1GB ($= 2^{30}$ Byte) Speicher $\Rightarrow 2^{(8 \cdot 2^{30})}$ Zustände
  $\Rightarrow$ Ziemlich (aber doch endlich) viele Zustände,
- → *Operation*() ist eine Boolesche Funktion (vgl. TechInfo: „Schaltnetz“)
- → Auswahl der *Operation*() erfolgt durch den *Programmzähler*

PROGRAMMZÄHLER

Ein Teil des Zustandes wird durch **Register** repräsentiert

→ Der **Programmzähler** (*PC*, *IP*, *EIP*, ...) ist das erste und wichtigste davon

→ Zeigt auf die nächste auszuführende Operation im Speicher



→ Verzweigungsoperationen setzen *PC* = *Sprungziel*

OPERATIONEN

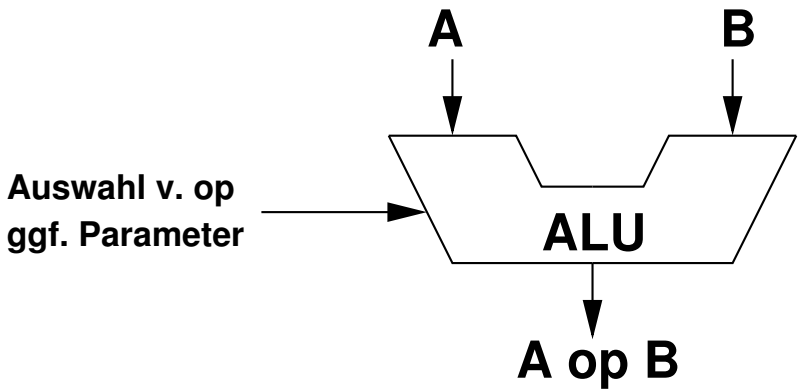
Die verfügbaren Operationen können klassifiziert werden

- → Arithmetisch-Logische Operationen
  - (+, −, ·, :, *AND*, *OR*, *XOR*, ...)
- → Datentransport
  - Zugriff auf E/A oder Speicher
- → Kontrollfluss
  - (bedingte) Sprünge, Unterprogrammaufruf und -Rückkehr
- → Steuerung und Konfiguration der Maschine
  - Interrupt sperren, Ausnahmebehandlung, etc.

ARITHMETISCH-LOGISCHE OPERATIONEN

Implementiert als **Schaltnetz** (ALU: Arithmetic Logic Unit)

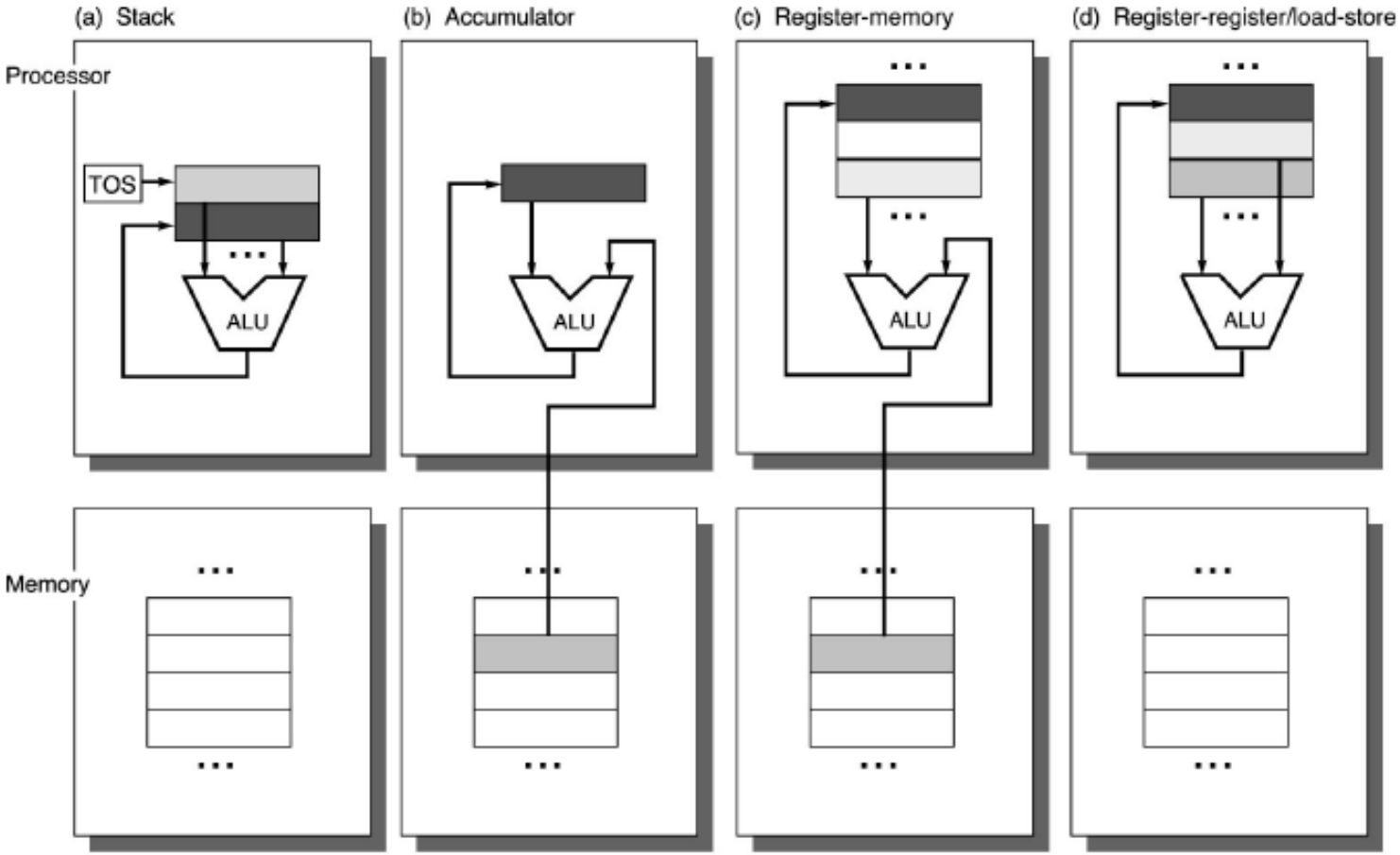
→ In der Regel[1] zwei Eingänge für Operanden
→ Ein Ausgang für Ergebnis
→ Steuereingang wählt Operation aus und enthält ggf weitere Parameter (z.B. *shift amount*)



→ Evtl. nur Strichrechnung und Schiebeoperationen
→ Multiplikation und Division algorithmisch (Mikroprogramme)

[1] aber nicht zwingend, vgl. Signalprozessoren

BEFEHLSSATZARCHITEKTUREN



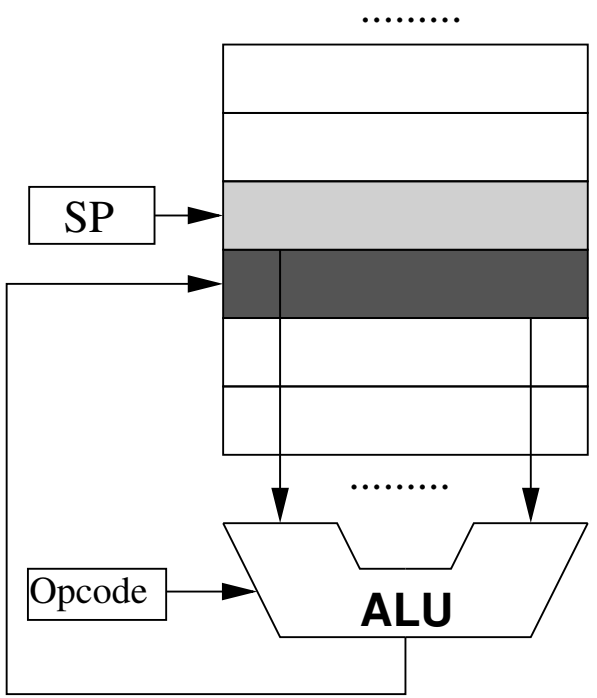
STACK-ARCHITEKTUR

Ein weiteres Register (SP = Stack Pointer)

- → Opcode bestimmt Operation
- → Operanden liegen „oben“ auf dem Stack
- → Ergebnis wird auf Stack abgelegt

Beispiel: berechne $a = (x + y) \cdot c$:

```
1     push x
2     push y
3     add
4     push c
5     mult
6     pop a
```



STACK-ARCHITEKTUR

Vorteile der Stack-Architektur

- → Minimaler Prozessorzustand (PC + SP)
- → Sehr kompakter Code, da keine Adressen enthalten (daher auch **Null-Adress**-Maschine)
- → Einfache Compiler/Interpreter

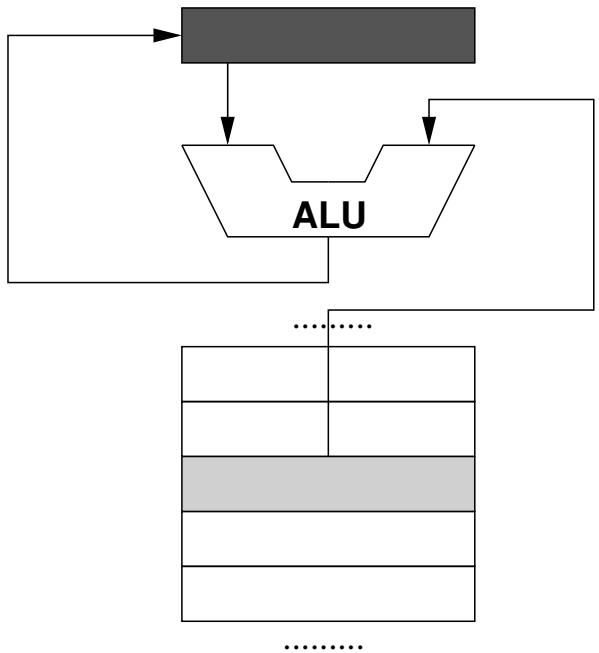
Haupt-Nachteil: viele (teure) Speicherzugriffe

- → Praktikabel, als Speicherzugriffszeiten noch eine untergeordnete Rolle spielten (z.B. HP 3000, 1972)
- → Heute nur noch als **virtuelle Maschine** (JVM, UCSD p-machine, FORTH)

AKKUMULATOR-ARCHITEKTUR

Ein Register (*Akkumulator*) dient als Operand und als Ergebnisspeicher

→ Zweiter Operand wird aus dem Speicher bezogen



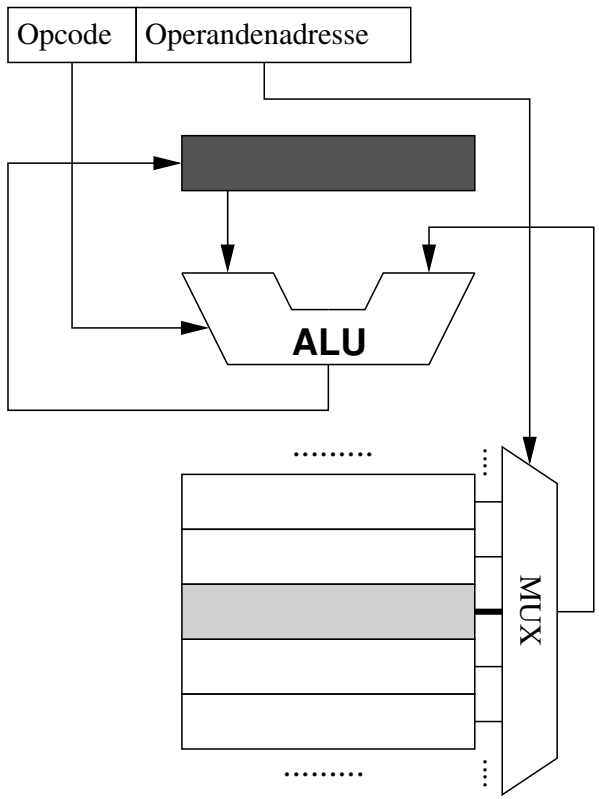
Beispiel: berechne $a = (x + y) \cdot c$:

```
1    lda x
2    add y
3    mult c
4    sta a
```

AKKUMULATOR-ARCHITEKTUR

Ein Register (*Akkumulator*) dient als Operand und als Ergebnisspeicher

- → Zweiter Operand wird aus dem Speicher bezogen
- → Befehl muss Operation + Speicheradresse des 2. Operanden enthalten (⇒**Ein-Adress**-Maschine)



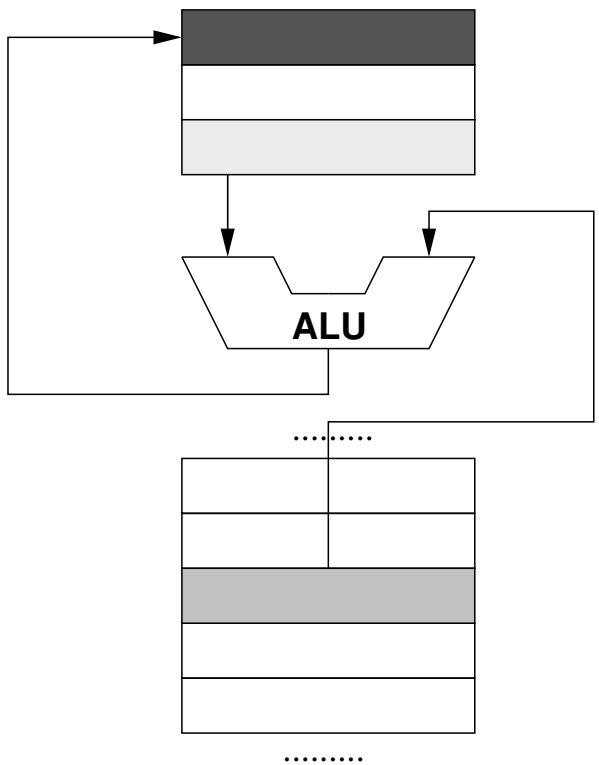
Beispiel: berechne $a = (x + y) \cdot c$:

```
1    lda x
2    add y
3    mult c
4    sta a
```

REGISTER-MEMORY-ARCHITEKTUR

Mehrere Register, von denen eines als Operand und Ergebnisspeicher dient

→ Zweiter Operand wieder aus dem Speicher



Beispiel: berechne $a = (x + y) \cdot c$:

```
1     ld   r1,x
2     add  r1,y
3     mult r1,c
4     st   r1,a
```

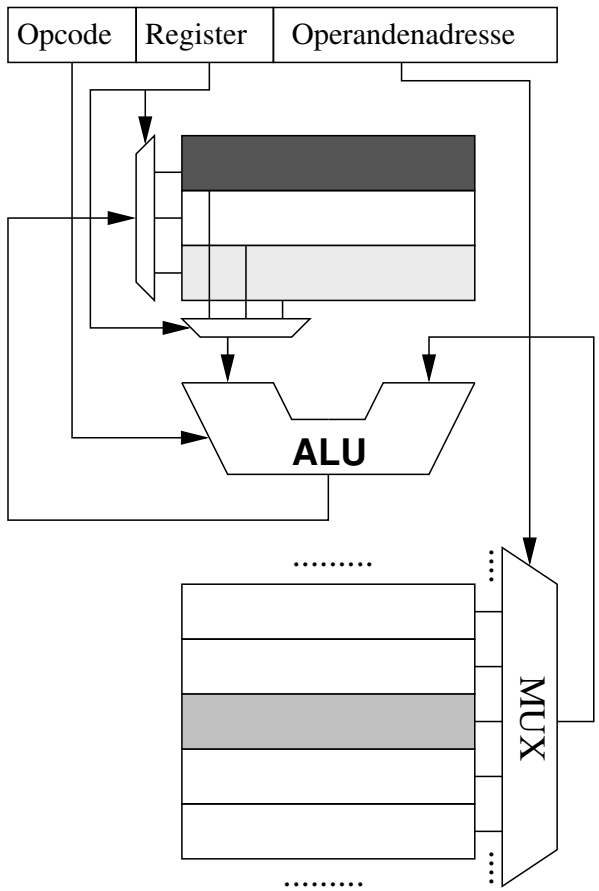
REGISTER-MEMORY-ARCHITEKTUR

Mehrere Register, von denen eines als Operand und Ergebnisspeicher dient

- → Zweiter Operand wieder aus dem Speicher
- → Befehl enthält Operation + Register + Adresse (⇒**Zwei-Adress**-Maschine)

Beispiel: berechne $a = (x + y) \cdot c$:

```
1    ld   r1,x
2    add  r1,y
3    mult r1,c
4    st   r1,a
```



AKKUMULATOR- UND REGISTER-MEMORY-ARCHITEKTUR

Vorteile

- → Universell: Es kann direkt auf Speichervariablen gerechnet werden
- → Kompakter Code

Nachteil: Operation bedingen i.d.R. einen Speicherzugriff

- → Teuer
- → Adresse (z.B.: 32-bit) im Befehl enthalten
  ⇒ Befehle sind größer als ein Maschinenwort
  ⇒ **Skalarität** ist so nicht umsetzbar

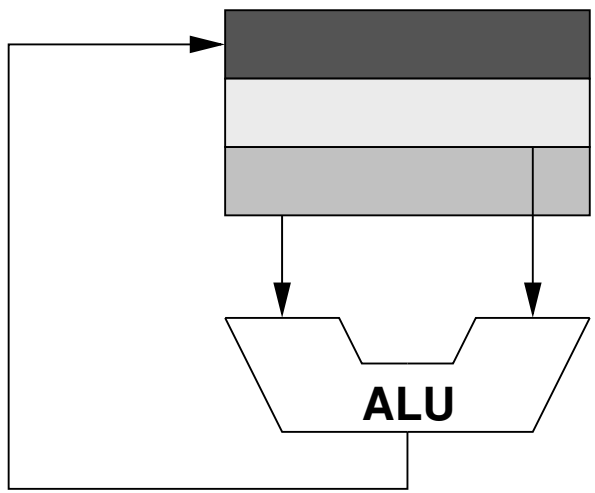
REGISTER-REGISTER- (LOAD/STORE-)ARCHITEKTUR

Arithmetikbefehle arbeiten **nur** auf Registern
Speicherzugriff nur über explizite Load- und Store-Befehle

→ Alle Operanden sind Register

Beispiel: berechne $a = (x + y) \cdot c$:

```
1     ld   r1,x
2     ld   r2,y
3     add  r1,r2,r1
4     ld r2,c
5     mult r2,r2,r1
6     st r2,a
```



REGISTER-REGISTER- (LOAD/STORE-)ARCHITEKTUR

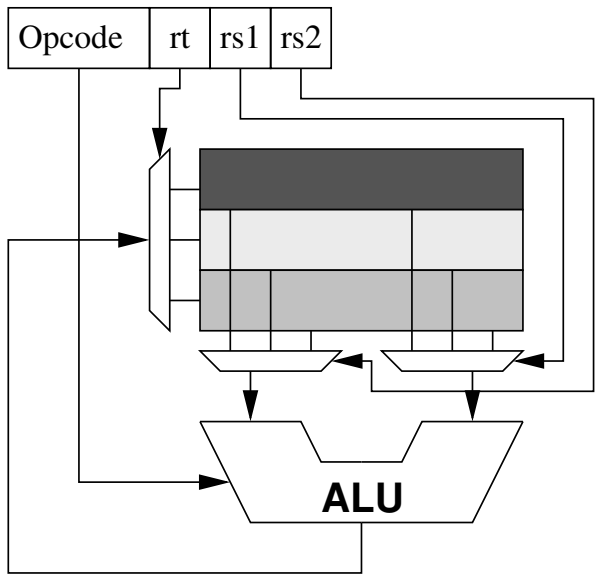
Arithmetikbefehle arbeiten **nur** auf Registern
Speicherzugriff nur über explizite Load- und Store-Befehle

→ Alle Operanden sind Register

→ Befehl enthält Operation + 3 Registernummern (⇒**Drei-Adress**-Maschine)

Beispiel: berechne $a = (x + y) \cdot c$:

```
1     ld   r1,x
2     ld   r2,y
3     add  r1,r2,r1
4     ld r2,c
5     mult r2,r2,r1
6     st r2,a
```



REGISTER-REGISTER- (LOAD/STORE-)ARCHITEKTUR

Vorteile

- → Arithmetik ohne Speicherzugriffe
  (solange genügend Register vorhanden)
- → Drei-Adress-Befehle „passen“ in ein Maschinenwort
  → Skalarität möglich

Nachteil: Weniger kompakter Code
(d.h. für die gleiche Funktion sind mehr Befehle nötig)

- → Programme werden größer (mehr Programmspeicher erforderlich)
- → Mehr Befehlszyklen (Opcode fetches)
  → kann durch Befehlscache kompensiert werden

Typisch für RISC-Architekturen.

WIMP-BEFEHLSSATZ

WIESBADEN MIKROPROZESSOR

WIMP: eine Untermenge des MIPS32 Befehlssatzes.

- → RISC-Architektur
- → 32-bit Big Endian
- → 32 Register (→ Registeradressen haben 5 Bit)
- → Load/Store-Architektur

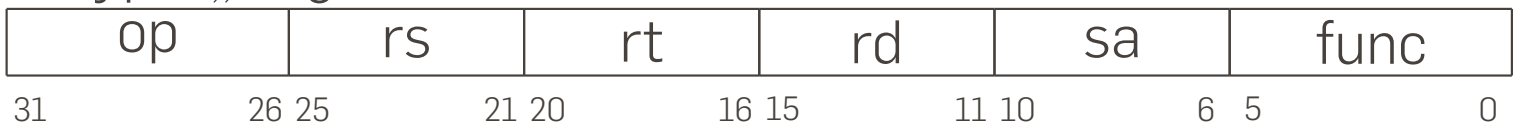
Vereinfachungen:

- → Keine Gleitkommaarithmetik, Multiplikations-/Divisionsbefehle
- → Kein TLB / keine MMU
- → Ausschließlich Word-(32-Bit-) Load/Store
- → (Noch) keine Exceptions / Interrupts
- → (Noch) keine Pipeline

BEFEHLSFORMATE

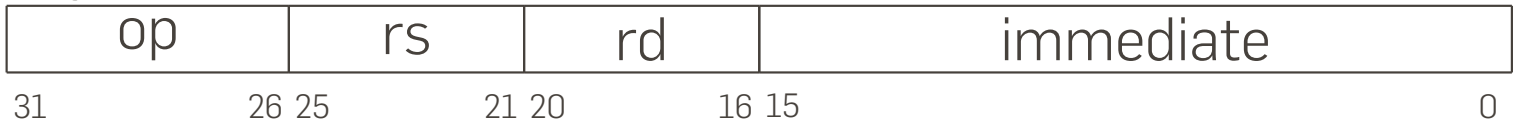
Feste Wortlänge: Alle Befehle sind 32-Bit groß. Drei Formate:

1. R-Type: „Register“-Befehle:



2. J-Type: „Jump“-Befehle:

3. I-Type: „Immediate“-Befehle:

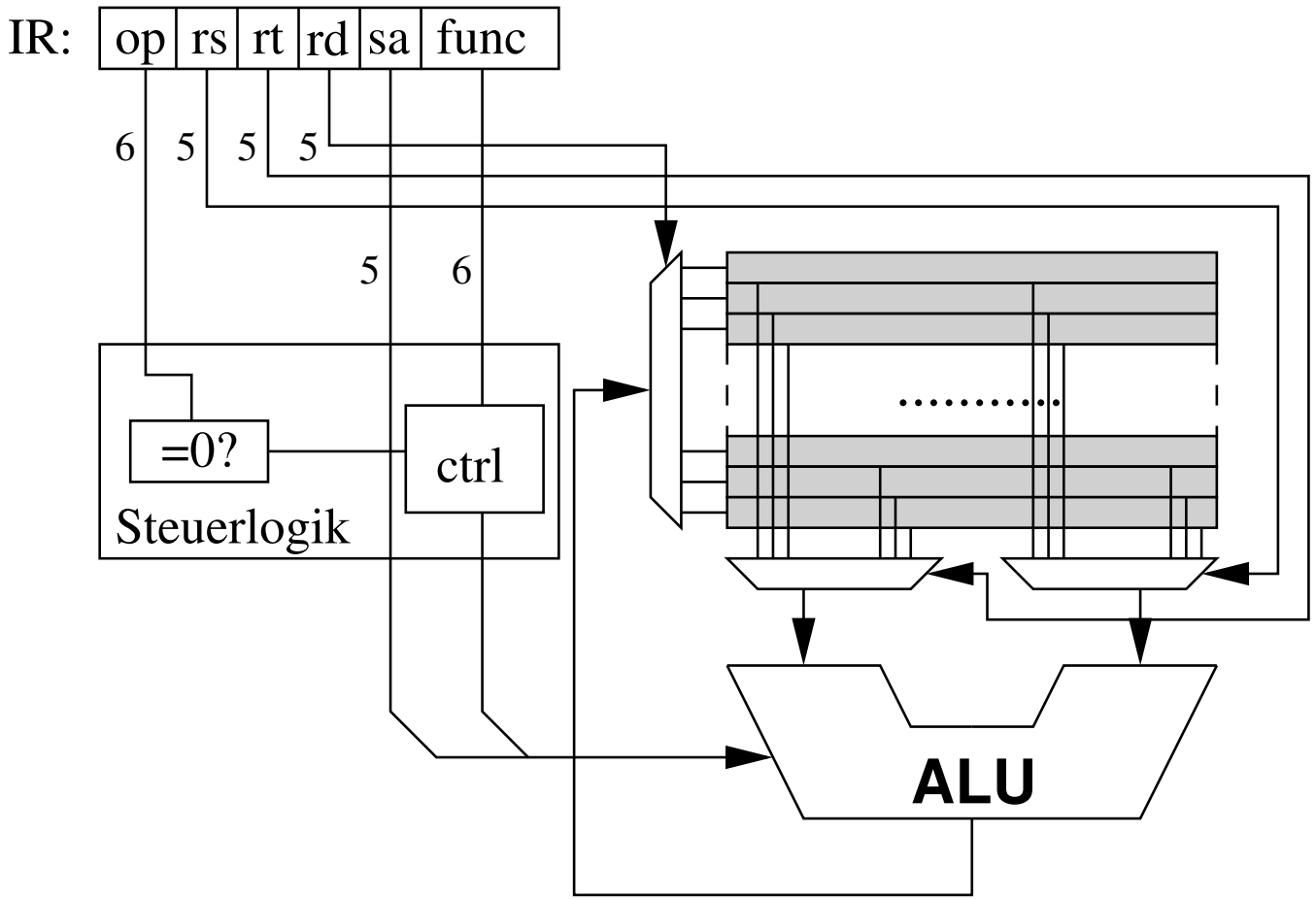


Bits 31-26 (*op*) gibt das Befehlsformat an:

→ $op = 0 \Rightarrow$ R-Type
→ $op = 2$ oder $3 \Rightarrow$ J-Type
→ sonst: $\Rightarrow$ I-Type

R-TYPE BEFEHL: IMPLEMENTIERUNG

Falls $op = 0$: $rd = func(rs, rt, sa)$
sa: „shift amount“ nur bei Schiebe-Befehlen (`sll`, `srl`,`sra`)
(sonst = 0)



R-TYPE BEFEHLE

Die meisten R-Type Befehle führen arithmetische Operationen nach dem o.g. Schema $rd = func(rs, rt, sa)$ aus.

- → Schieben nach links / rechts, arithmetisch / logisch, um variabel / konstant viele Bits: `sll`, `srl`, `sra`, `sllv`, `srlv`, `srav`
- → Addieren / Subtrahieren mit / ohne Trap[2] bei Überlauf: `add`, `addu`, `sub`, `subu`
- → Bitweise logische Operationen: `and`, `or`, `xor`, `nor`
- → Vergleichs-Operationen: `slt`, `sltu`

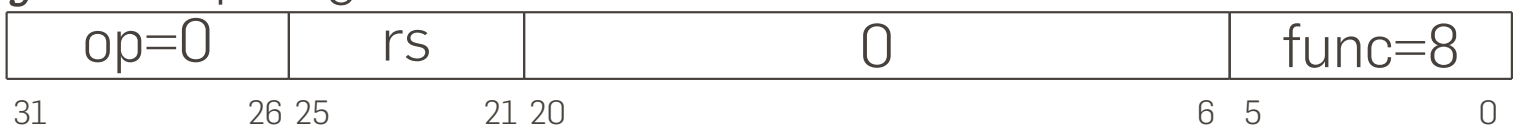
Details: siehe Prozessorhandbuch
Einige Befehle sind jedoch speziell ...

[2] WIMP kennt derzeit keine Traps

SPEZIELLE R-TYPE BEFEHLE (1)

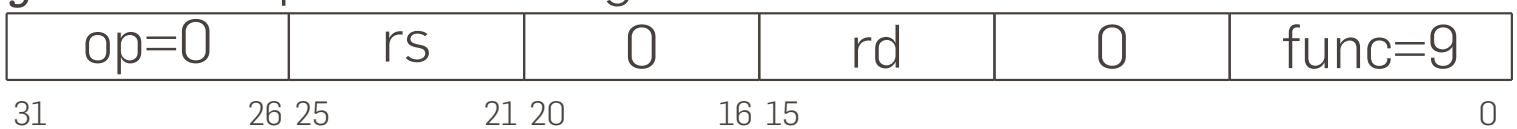
R-Type Sprungbefehle:

→ `jr`: Jump Register



Operation: Inhalt von *rs* in Programmzähler laden ($PC = rs$)

→ `jalr`: Jump and link Register

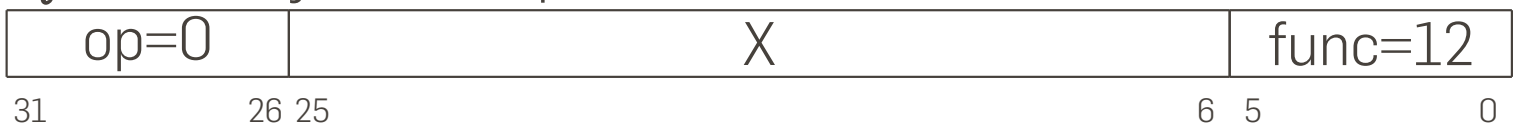


Operation: Inhalt von Programmzähler + 4 in *rd* kopieren,
Inhalt von *rs* in Programmzähler laden ($rd = PC + 4$; $PC = rs$)
In der Regel ist $rd = 31$ (s.u.)

SPEZIELLE R-TYPE BEFEHLE (2)

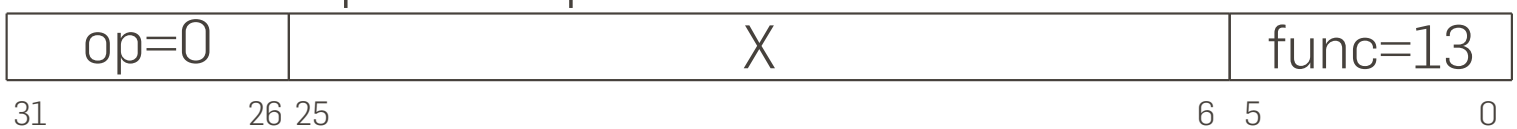
R-Type Trap-Befehle:

→ `syscall`: Syscall Trap auslösen



Operation: Synchrone Exception auslösen, wird gewöhnlich für Systemaufrufe verwendet

→ `break`: Breakpoint Trap auslösen



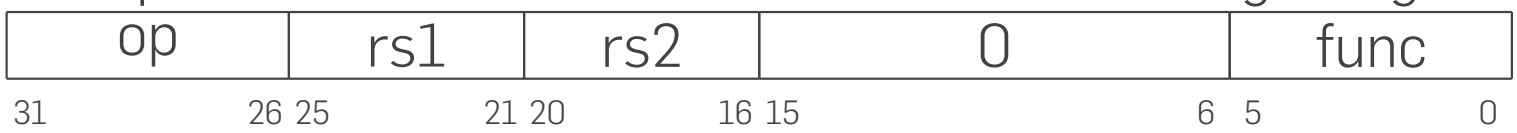
Operation: Synchrone Exception auslösen, wird gewöhnlich für Debug-Breakpoints verwendet

Die Bits 25-6 (Feld „X“) sind don't care, werden aber konventionell zur Kodierung von -z.B.- Trap-Nummern verwendet. WIMP kennt derzeit noch keine Exceptions, d.h. diese Befehle werden nicht unterstützt. WIE (s.u.) verwendet sie aber.

SPEZIELLE R-TYPE BEFEHLE (3)

Multiplikations- und Divisions-Befehle:

- → Der MIPS-Prozessor besitzt eine 32x32 bit Multiplikationseinheit **(WIMP nicht)**
- → Ergebnisse von Multiplikationen (bis zu 64 Bit) werden in zwei speziellen Registern *lo* und *hi* abgelegt
- → Multiplikations- und Divisionsbefehle sind wie folgt aufgebaut



dabei ist:

- → $func = 24 \Rightarrow$ `mult`: signed multiply
- → $func = 25 \Rightarrow$ `multu`: unsigned multiply
- → $func = 26 \Rightarrow$ `div`: signed divide
- → $func = 27 \Rightarrow$ `divu`: unsigned divide

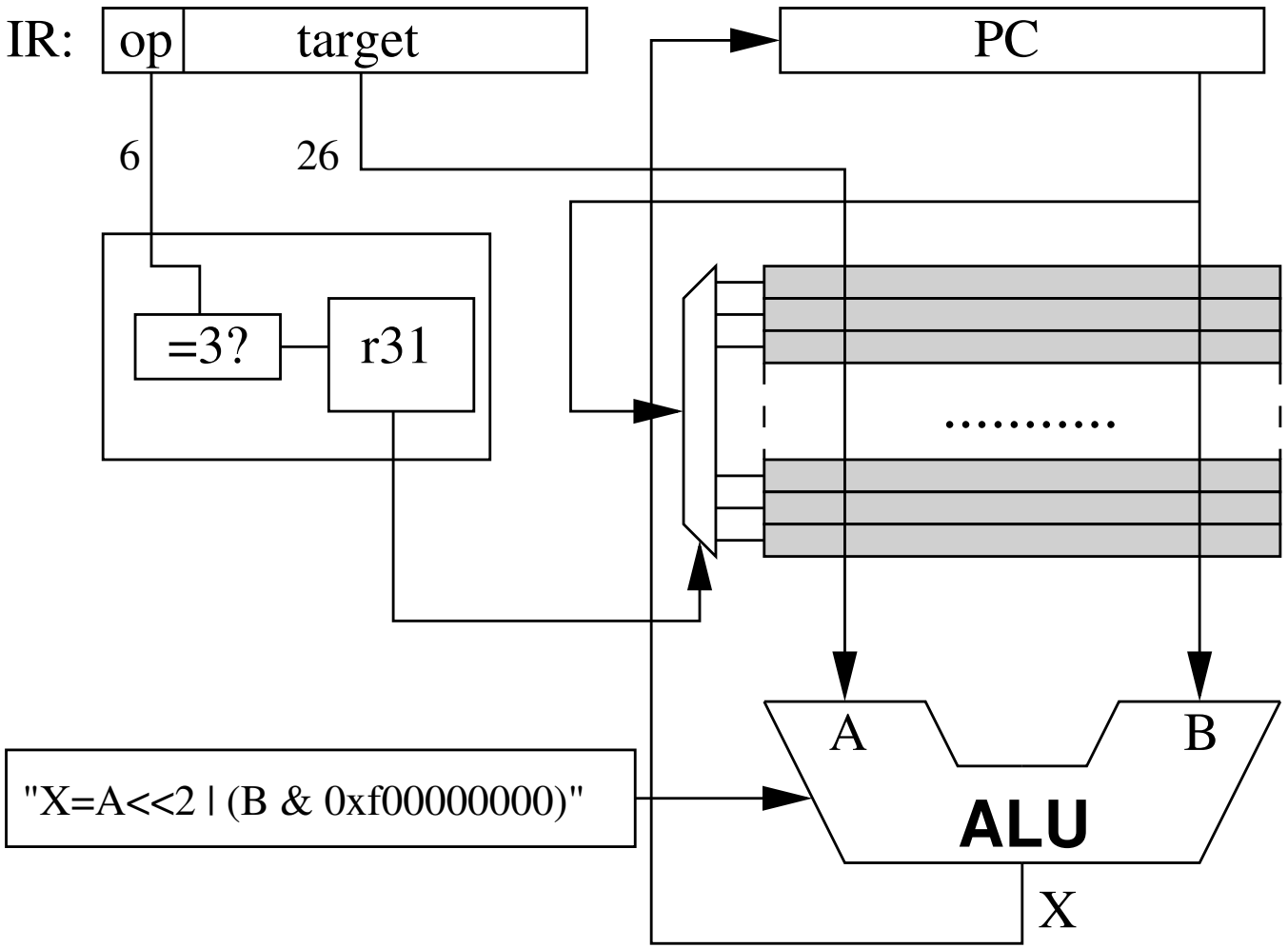
Darüber hinaus gibt es R-Type Befehle zum Datentransport zwischen den Registern *lo* und *hi* und den „normalen“ Registern:

`mflo`, `mtlo`, `mfhi`, `mthi`

J-TYPE BEFEHL: IMPLEMENTIERUNG

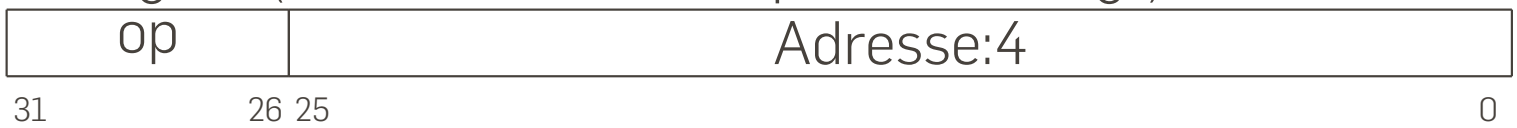
Falls $op = 3$: $r31 = PC$

$PC = (PC \& 0xf0000000) | (Adresse << 4)$



J-TYPE BEFEHLE: ZIELADRESSE

→ Für die Ziel-Sprungadresse sind nur 26 Bit des Befehls verfügbar (6 werden für den Opcode benötigt)



→ Wegen der festen Befehlswortlänge von 32 Bit (=4 Byte) **muss** die Zieladresse durch 4 teilbar sein → ihre beiden niederwertigsten Bits sind immer 0

→ Im Befehl wird die durch vier dividierte Adresse codiert

→ Die 4 noch fehlenden Bits werden vom aktuellen PC übernommen

→ Sprünge mit *j* oder *jal* sind immer nur innerhalb eines 1GB Segmentes möglich

→ Für größere Sprungdistanzen: R-Type Befehl *jr* verwenden (s.o.)

J-TYPE BEFEHLE: JUMP AND LINK

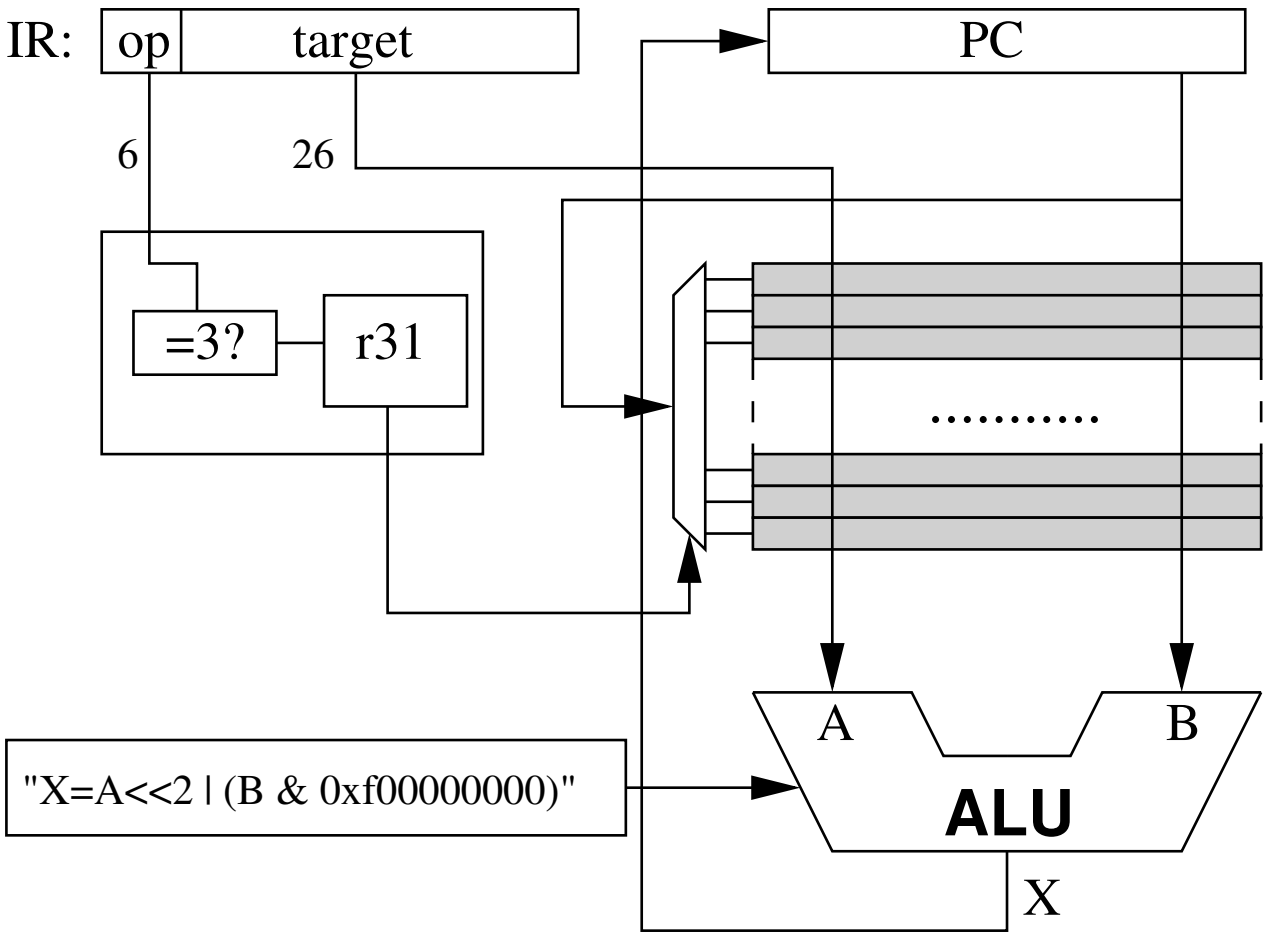
Bei Prozeduraufrufen mittels *jal*-Befehl muss die Returnadresse gerettet werden. CISC-Prozessoren kopieren diese gewöhlich auf den Stack. WIMP kennt aber von Hause aus **keinen Stack**

- → Zum Speichern der Returnadresse wird per Konvention das Register Nummer 31 verwendet (s.u.)
- → Der *jal*-Befehl ($op = 3$) speichert den **nach dem Opcode Fetch bereits inkrementierten** Programmzähler in $r31$
- → Falls die aufgerufene Prozedur weitere Prozeduraufrufe tätigt, muss sie selbst den Inhalt von $r31$ retten und dazu z.B. einen Stack in Software implementieren

I-TYPE BEFEHL: IMPLEMENTIERUNG

$rd = func(rs, immediate)$

immediate wird auf 32 Bit erweitert. Abhängig von *op* erfolgt die Erweiterung vorzeichenrichtig oder ohne vorzeichen.



I-TYPE BEFEHLE

Die meisten I-Type Befehle führen arithmetische Operationen nach dem o.g. Schema $rd = func(rs, immediate)$ aus.

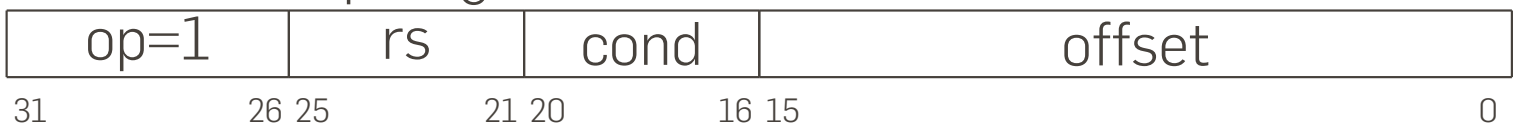
- → Addieren[3] mit / ohne Trap[4] bei Überlauf: `addi`, `addiu`
- → Bitweise logische Operationen: `andi`, `ori`, `xori`
- → Vergleichs-Operationen: `slti`, `sltiu`

Details: siehe Prozessorhandbuch
Einige Befehle sind wieder speziell ...

[3] Subtrahieren durch negatives *immediate*
[4] WIMP kennt derzeit keine Traps

SPEZIELLE I-TYPE BEFEHLE (1)

- PC-relative Sprungbefehle:



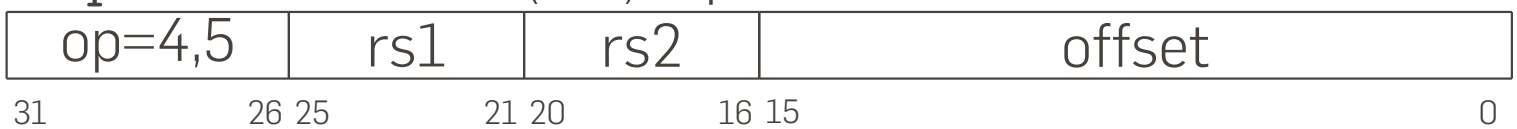
  dabei ist:
  - *cond* = 0 ⇒ `bltz`: branch on less than zero
  - *cond* = 1 ⇒ `bgez`: branch on greater or equal to zero
  - *cond* = 16 ⇒ `bltzal`: branch and link on less than zero
  - *cond* = 17 ⇒ `bgezal`: branch and link on gt. or eq. to zero
- Wenn *rs* die Bedingung erfüllt, addiere *offset* · 4 (vorzeichenerweitert) zu PC
- PC-relativ: Code ist positionsunabhängig
- Maximale Sprungdistanz ±128 K
- „and link“-Varianten speichern Returnadresse in *r*31, **auch wenn nicht gesprungen wird**

Unbedingte Sprünge durch Wahl von Register 0 als *rs* (s.u.)

SPEZIELLE I-TYPE BEFEHLE (2)

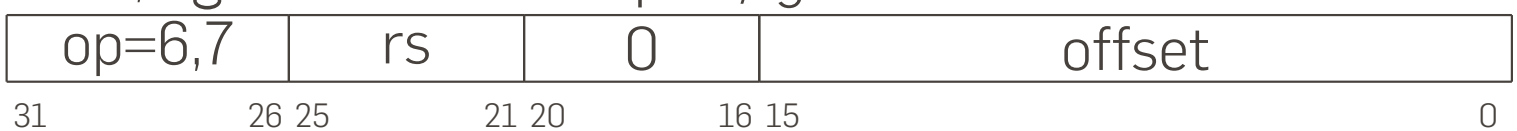
PC-relative Vergleichs-und-Sprungbefehle:

→ **`beq`**, **`bne`**: Branch on (not) equal



Springen wenn $rs1 = rs2$ bzw. $rs1 \neq rs2$

→ **`blez`**, **`bgtz`**: Branch on equal / greater than zero



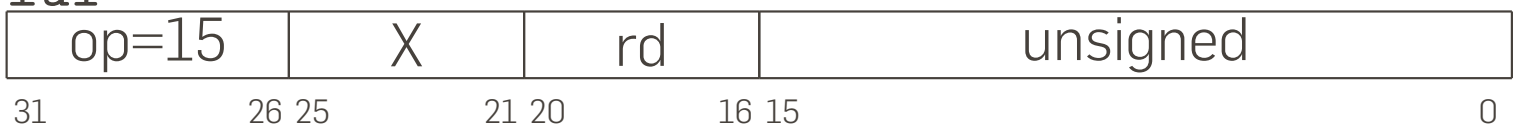
Springen wenn $rs = 0$ bzw. $rs > 0$

→ Wenn Bedingung erfüllt, addiere $offset \cdot 4$ (vorzeichenerweitert) zu PC

SPEZIELLE I-TYPE BEFEHLE (3)

Load upper immediate:

→ `lui`



→ Funktion: $rd = unsigned \cdot 2^{16}$

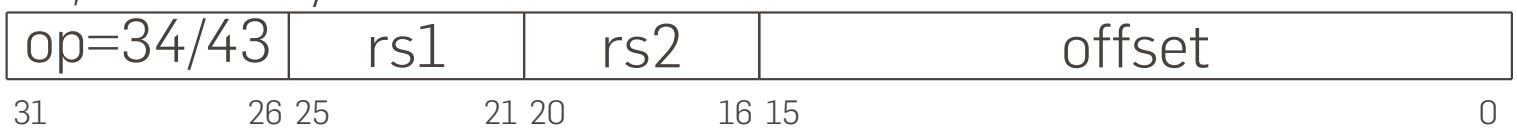
Das Laden einer beliebigen 32-Bit Konstante in ein Register ist wegen der festen Befehlswortgröße nicht möglich
(Die Konstante würde alleine schon 32 Bit benötigen).
Das Laden solcher Konstanten geschieht in zwei Schritten:

```
1    lui rd,konstante>>16
2    ori rd,rd,konstante&0xffff
```

SPEZIELLE I-TYPE BEFEHLE (4)

Load/Store: Speicherzugriff

→ `lw`, `sw`: Load/Store Word



→ Word- (d.h. 32-bit-) Speicherzugriff auf Adresse *rs* + *offset*

→ Die effektive Adresse muss durch 4 teilbar sein (sonst → Exception)

Der MIPS-Prozessor unterstützt darüber hinaus diverse Befehle für Byte- und Halbwortzugriffe, **WIMP** jedoch **nicht**. Der Assembler WAS (s.u.) synthetisiert passende Befehlssequenzen für `lb` und `sb`, die allerdings mit Vorsicht zu genießen sind.

REGISTERMODELL

REGISTER

WIMP (MIPS) hat 32 Register: $r0 \ldots r31$.
Zwei davon haben spezielle Funktionen:

- → $r0$ ist die hartverdrahtete Konstante Null:
  - → Lesen liefert stets 0
  - → Schreiben ist ohne Wirkung

  32-Bit Konstanten im Bereich $\pm 2^{15}$ können mit nur einem Befehl generiert werden (`addi rd,r0,<zahl>`)
- → $r31$ dient Instruktionen wie `jal` als „Return Address“ Register

Darüber hinaus gibt es **Konventionen** zur Verwendung der übrigen Register, die aber keine Sonderstellung der Register innerhalb der Hardware-Achitektur erfordern.

REGISTERKONVENTIONEN (1)

Neben Registernummern (*r*0 . . . *r*31) können Register auch durch **Namen** referenziert werden, die sich an der üblichen Verwendung dieser Register orientiert:

Nummer	Name	Funktion
*r*0	*zero*	Hardverdrahtete Null (s.o)
*r*1	*at*	Assembler Temporary Register
*r*2 . . . *r*3	*v*0 . . . *v*1	Returnwert („Value“) für Prozeduren
*r*4 . . . *r*7	*a*0 . . . *a*3	Erste vier Parameter für Prozeduren
*r*8 . . . *r*15	*t*0 . . . *t*7	Temporäre Register („Caller Saved“)
*r*16 . . . *r*23	*s*0 . . . *s*7	Statische Register („Callee Saved“)
*r*24 . . . *r*25	*t*8 . . . *t*9	Temporäre Register („Caller Saved“)
*r*26 . . . *r*27	*k*0 . . . *k*1	Kernel Register
*r*28	*gp*	Global Pointer
*r*29	*sp*	Stack Pointer
*r*30	*fp* oder *s*8	Frame Pointer („Callee Saved“)
*r*31	*ra*	Return Address Register

REGISTERKONVENTIONEN (2)

... im Detail:

- → *at*: Kann vom Assembler als Scratchregister verwendet werden → nicht verwenden
- → *t*0 . . . *t*9: für temporäre Variablen: dürfen von Prozeduren überschrieben werden
- → *s*0 . . . *s*9: für statische Variablen: dürfen **nicht** von Prozeduren überschrieben werden
- → *k*0 . . . *k*1: zur Verwendung durch das Betriebssystem: können sich **jederzeit** (z. B. durch Interruptbehandlung) ändern → nicht verwenden
- → *gp*: Basisadresse globaler Daten (für positionsunabhängige Daten)

SYNTHETISCHE BEFEHLE

SYNTHETISCHE BEFEHLE

Assembler unterstützen in der Regel einige Befehle, die im Befehlssatz nicht vorgesehen sind, weil sie:

- → redundant sind, d.h. weil sie durch existierende Befehle bei identischer Funktion realisiert werden können, oder weil sie
- → unter Umständen nicht in einem Befehlswort dargestellt werden können.

Der Assembler **synthetisiert** diese Befehle, wobei er ggf. auf das *at* Register als Zwischenvariable zurückgreift.

SYNTHETISCHE ARITHMETIK-BEFEHLE

Befehl	Synthetisiert als	Beschreibung
`nop`	`sll zero,zero,0`	Einen Zyklus lang nichts tun
`move rt,rs`	`addu rt,rs,zero`	Kopiere *rs* nach *rt*
`not rt`	`nor rt,rt,zero`	*rt* Bit-weise invertieren
`neg rt`	`nor rt,rt,zero`	*rt* negieren (2er-Kompl.)
	`addiu rt, rt, 1`	
`subi rs,rt,imm`	`addi rs,rt,-imm`	Subtrahiere: $rt = rs - imm$
`subiu rs,rt,imm`	`addiu rs,rt,-imm`	Subtrahiere: $rt = rs - imm$

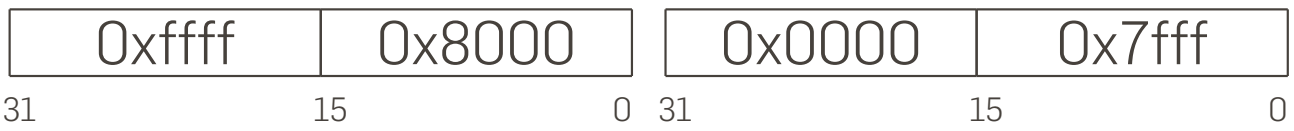
SYNTHETISCHE SPRUNG-BEFEHLE

Befehl	Synthetisiert als	Beschreibung
`b label`	`beq zero,zero,label`	Unbedingter Sprung
`bal label`	`bgezal zero,label`	Unterprogrammaufruf
`blt rs,rt,label`	`slt at,rs,rt`	Springe wenn $rs < rt$
	`bne at,zero,label`	
`bgt rs,rt,label`	`slt at,rt,rs`	Springe wenn $rs > rt$
	`bne at,zero,label`	
`ble rs,rt, label`	`slt at,rt,rs`	Springe wenn $rs \leq rt$
	`beq at,zero,label`	
`bge rs,rt,label`	`slt at,rs,rt`	Springe wenn $rs \geq rt$
	`beq at,zero,label`	
`bgtu rs,rt,label`	`sltu at,rt,rs`	Springe wenn $rs \geq rt$
	`bne at,zero,label`	(unsigned)
`beqz rs,label`	`beq rs,zero,label`	Springe wenn $rs = 0$

Alle Sprünge sind **PC-relativ**

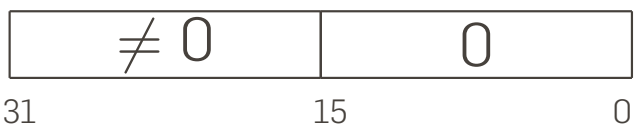
SYNTHETISCHE LADE-BEFEHLE: `LI` UND `LA`

`li rt,konstante` („load immediate“) und
`la rt,konstante` („load address“) laden eine 32-Bit Konstante in Register *rt*. Mit einem einzigen Befehl geht das nur wenn die Konstante zwischen -2^{15} und $+2^{15}-1$ liegt … :



→ dann: **`addiu rt,zero,konstante`**

→ … oder wenn sie Vielfaches von 2^{16} ist:



dann: **`lui rt,konstante>>16`**

ansonsten zwei Befehle:

```
1    lui rd,konstante>>16
2    ori rd,rd,konstante&0xffff
```

SYNTHETISCHE LOAD/STORE-BEFEHLE

- → (S.o.) WIMP kann ausschließlich 32-Bit-weise auf Speicher zugreifen
- → Es wäre Sache eines Compilers, Bytezugriffe zu realisieren
- → Luxuriöserweise bietet der Assembler (WAS, s.u.) synthetisierte `lb`- und `sb`-Befehle an
- → Diese sind komplex und benötigen viele Befehle:
  - → `lb`: 11 Befehle
  - → `sb`: 24 Befehle
- → beide zerstören das *at*-Register
- → Der synthetisierte `sb`-Befehl benötigt zudem 12 Byte Stack, um vorübergehend Register zu retten. D.h. er funktioniert nur, wenn der Stackpointer korrekt initialisiert ist.