ulm university universität ulm

Grundlagen der Betriebssysteme

[CS2100]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 7

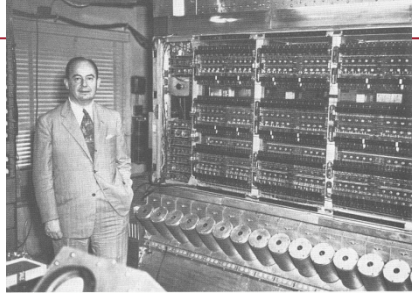
Einführung in MIPS-Assembler

Inhalte der Vorlesung

Inhalte des Kapitels

7. Einführung in MIPS-Assembler

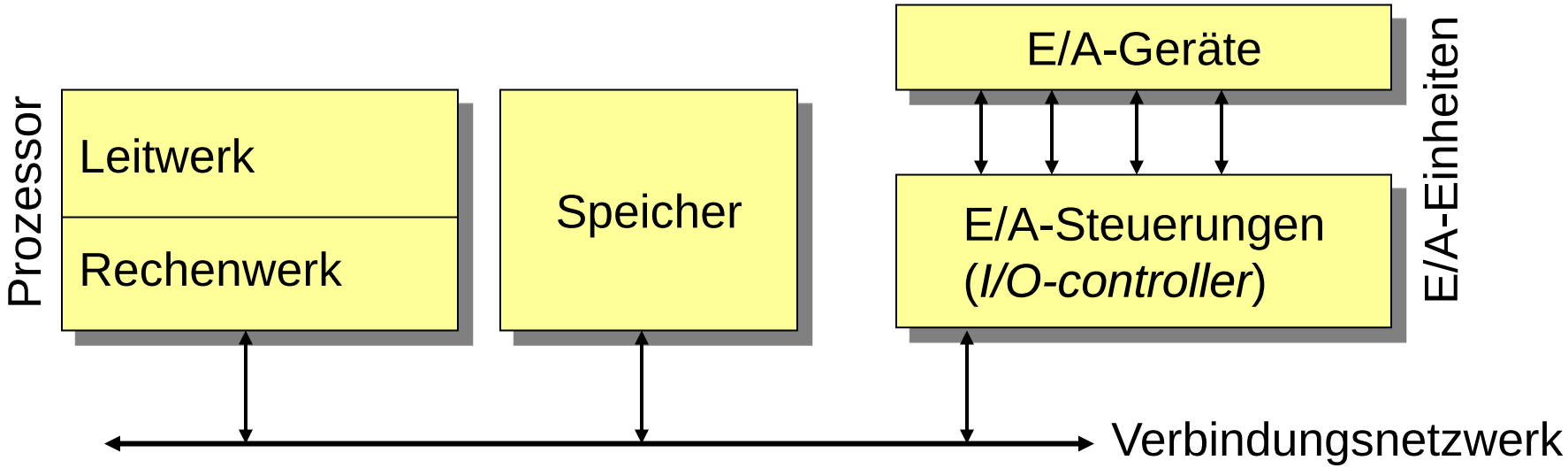
- Einführung
  - Von-Neumann Rechnerarchitektur
  - MIPS Architekturskizze
  - Abstraktionsebenen von Programmiersprachen
  - Werkzeuge zur Code-Erzeugung
- Exemplarische Betrachtung der MIPS Assemblersprache
  - Befehlsformate
  - MIPS Maschinenbefehle und deren Bedeutung
- Übersetzung von hochsprachlichen Konstrukten
  - Kontrollstrukturen
  - Datenzugriffe
- Beispiel: *bubble sort*

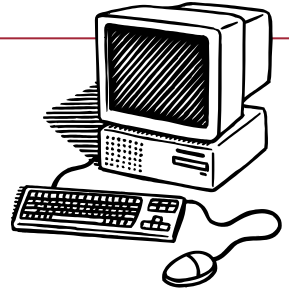


Das Von Neumann-Modell (1)

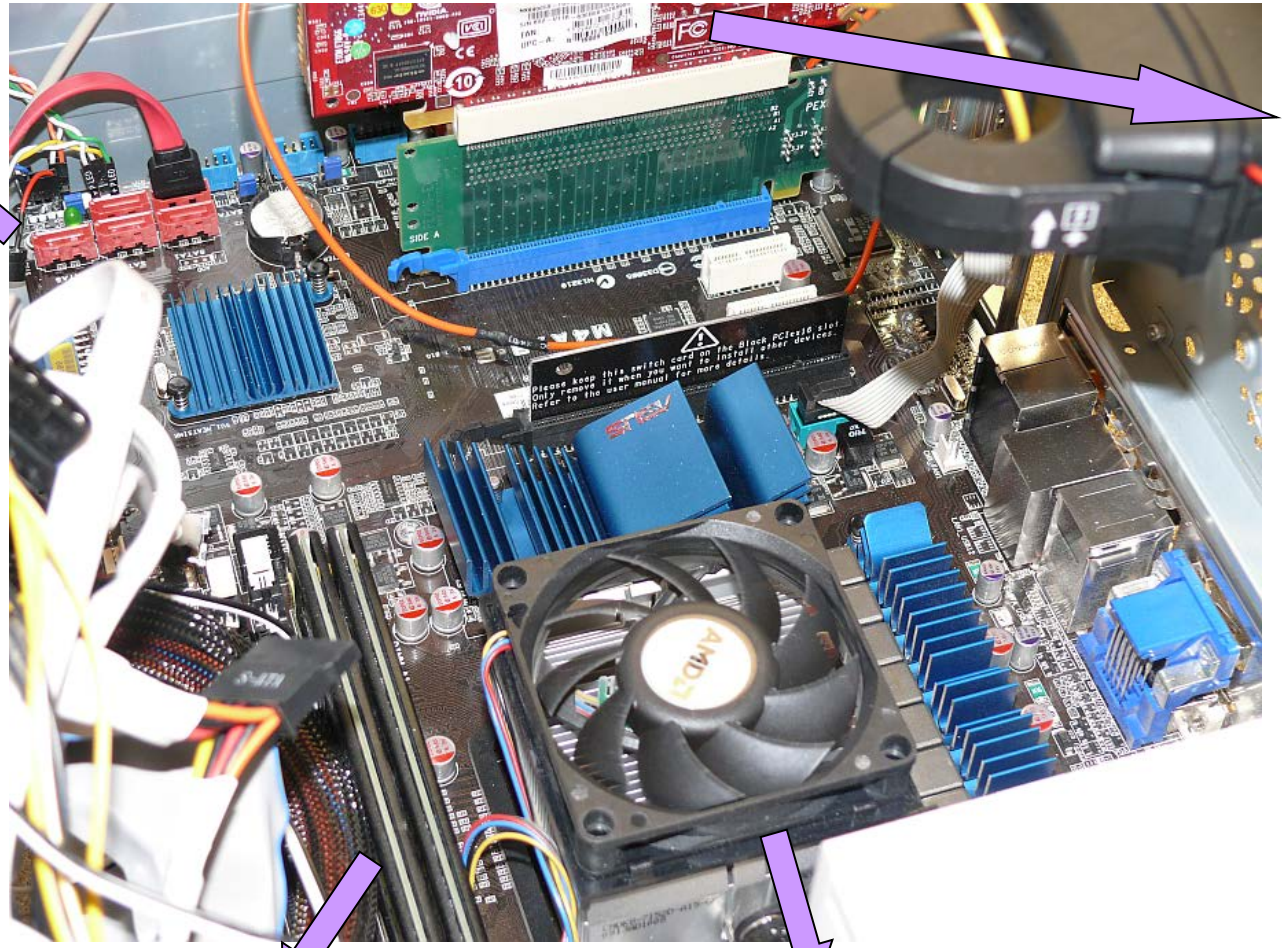
Fast alle der heute üblichen Rechner gehen auf den Von Neumann-Rechner mit folgenden Eigenschaften zurück:

1. Die Rechenanlage besteht aus den Funktionseinheiten **Speicher**, **Leitwerk** (bzw. Steuerwerk, engl. *controller*), dem **Rechenwerk** (engl. *data path*) und **Ein-/Ausgabe-Einheiten**.





Wo sind diese Komponenten auf PC-*Boards*?



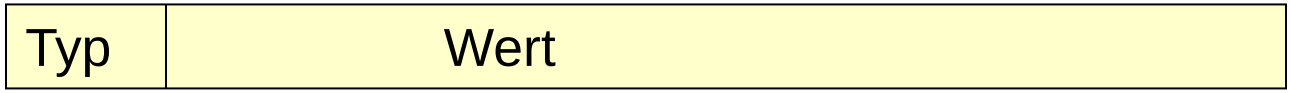
Das Von Neumann-Modell (2)

2. Die Struktur der Anlage ist unabhängig vom bearbeiteten Problem. Die Anlage ist **speicherprogrammierbar**.
3. **Anweisungen und Operanden** (einschl. Zwischenergebnissen) werden **in demselben physikalischen Speicher** gespeichert.
4. Der Speicher wird in **Zellen gleicher Größe** geteilt. Die Zellnummern heißen **Adressen**.
5. Das Programm besteht aus einer Folge von elementaren **Befehle**n, die **in der Reihenfolge der Speicherung bearbeitet** werden.
6. Abweichungen von der Reihenfolge sind mit (bedingten oder unbedingten) **Sprungbefehlen** möglich.

Das Von Neumann-Modell (3)

7. Es werden **Folgen von Binärzeichen** (nachfolgend Bitvektoren genannt) verwendet, um alle Größen darzustellen.
8. Die Bitvektoren erlauben **keine explizite Angabe des repräsentierten Typs**. Aus dem Kontext heraus muss stets klar sein, wie die Bitvektoren zu interpretieren sind.

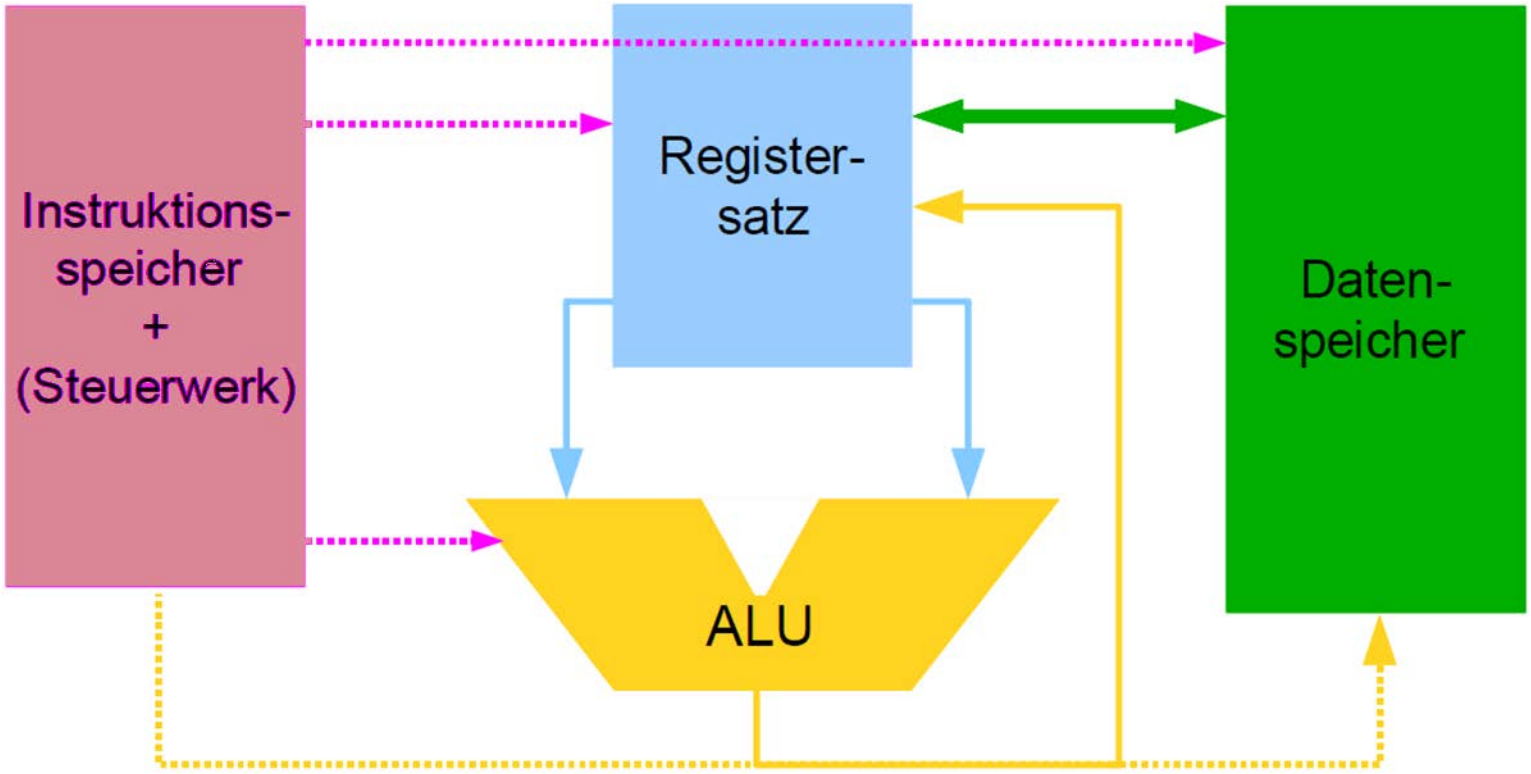
   Alternative:



MIPS Architekturskizze

Vereinfachte Sicht einer MIPS

- Informationsflüsse
  - Steuersignale, Speicherdaten, Registerdaten, Adressen, Instruktionen



Der MIPS Befehlssatz

Beispiel: MIPS *(~ Machine with no Interlocked Pipe Stages)*

≠ MIPS *(million instructions per second)*

Entwurf Anfang der 80er Jahre

Warum MIPS?

- Weitgehend sauberer und klarer Befehlssatz
- Kein historischer Ballast
- Basis der richtungsweisenden Bücher von Hennessy/Patterson
- Simulator verfügbar
- MIPS außerhalb von PCs weit verbreitet
  (bei Druckern, Routern, Handys)

Begriffe

- **MIPS-Befehle** sind elementare Anweisungen an die MIPS-Maschine
- Ein **MIPS-Maschinenprogramm** ist eine konkrete Folge von MIPS-Befehlen
- Die **MIPS-Maschinensprache** ist die Menge möglicher MIPS-Maschinenprogramme

- Entsprechendes gilt für die Assemblerprogramm-Ebene
- Vielfach keine Unterscheidung zwischen beiden Ebenen

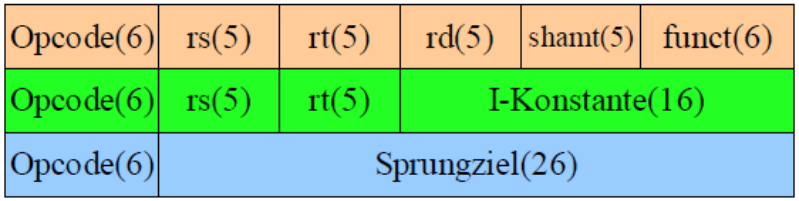
MIPS Instruktionsformate

Generell

- 32-Bit Wörter
- 32 Operandenregister
- 6 Bit für Operationscode
- je 5 Bit Registerindex für Operand 1, Operand 2 und Resultat
- 5 Bit Verschiebung *(shift amount)*
- 6 Bit Funktionscode

Instruktionsformate

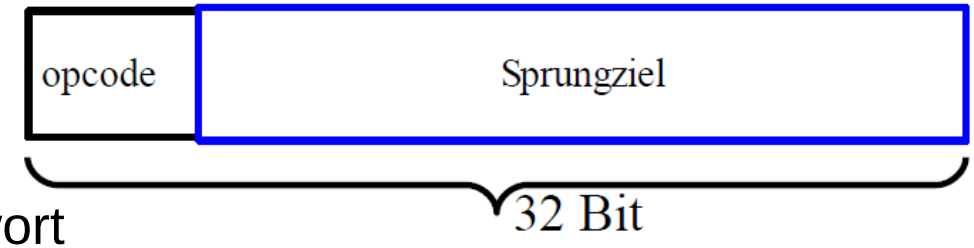
- R-Format für Register/ALU-Operationen
- I-Format für *Immediate*-Operanden
- J-Format für *Jump*-Befehle



J-Format Instruktion

Zuordnung der Bitfelder

- Opcode = IR[31-26]
- Sprungziel = IR[25-0]



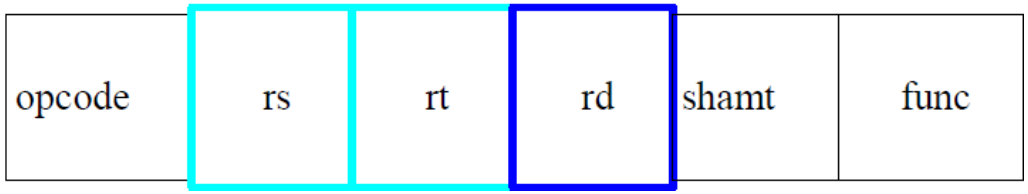
Semantik der *Jump*-Instruktion

- PZ = IR[25-0] * 4
- Direkter Sprung auf ein Speicherwort
- Sprungziel jeweils an einer durch 4 teilbaren Adresse *(alignment)*
- Maximale Adresse für das Sprungziel: 0x10000000 = 2^{28}

R-Format Instruktion

Für Registeroperanden

- Zwei Register lesen, eines schreiben
- Gelesene Register weiter zur ALU
- Drei Instruktionsfelder à 5 Bit
- Resultat zurück von ALU



Input-Register für die ALU

- rt = IR[20-16] wählt Register[rt] zur ALU
- rs = IR[25-21] wählt Register[rs] zur ALU

Zielregister für Resultat von ALU

- rd = IR[15-11] wählt Register[rd] für Resultat

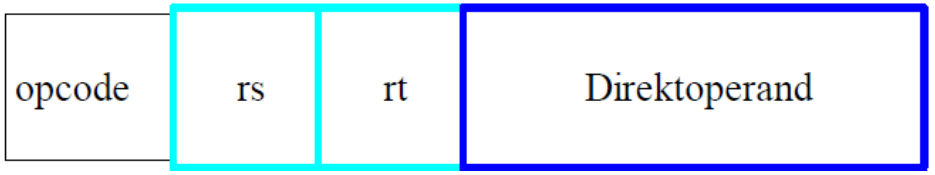
Kennzeichnung der ALU-Funktion

- 6 Bit Opcode
- 5 Bit Verschiebungsbetrag
- 6 Bit Funktionscode für ALU

I-Format Instruktion

Immediate-Operand Instruktion

- Hauptoperand findet sich direkt in der 32 Bit Instruktion
- entweder Konstanten-Wert
- oder Speicheradresse



Zuordnung der Bitfelder

- Basis-/Indexregister: rs = IR[25-21]
- Ziel-/Quellregister: rt = IR[20-16]
- Direktoperand: imm = IR[15-0]

Übertragungsrichtungen für rt

- Quellregister bei *Store*-Instruktion, Zielregister bei *Load*-Instruktion

Direktoperand

- Platz für max. 16 Bit in der Instruktion
- Vorzeichenerweiterung 16 → 32 Bit
- Größere Konstanten zusammensetzen

Binäres Maschinenprogramm

Instruktionen im Programmspeicher

– Instruktionen und Programmausführung ab Adresse 0x00400000

```
0x00400000: 00000000 00011111 00001000 00100000
0x00400004: 00000011 11100000 00001000 00100010
0x00400008: 00000000 00000001 11111000 00101010
0x0040000C: 00100000 00000111 10101111 11111110
0x00400010: 10101100 01100111 10101111 11110000
0x00400014: 00001000 00010000 00000000 00000000
```

Legende

– Hex-Adresse
– Opcode
– rs-Registerindex
– rt-Registerindex
– rd-Registerindex
– *shift amount*
– Funktionscode
– *Immediate* Operand
– Sprungziel

☞ Programmerstellung direkt als Binärcode oder Hexcode fehleranfällig
Nur im äußersten Notfall in Betracht zu ziehen → Assemblersprache

Binäre Maschinenbefehle (1)

Binäre Notation ist nicht mnemonisch und schlecht lesbar

- Manuelle Adressrechnungen und Anpassungen
- Unsystematische binäre Befehlscodierungen
- Manuelle Anpassung von Sprungdistanzen
- Kein Bezug auf externe Symbole

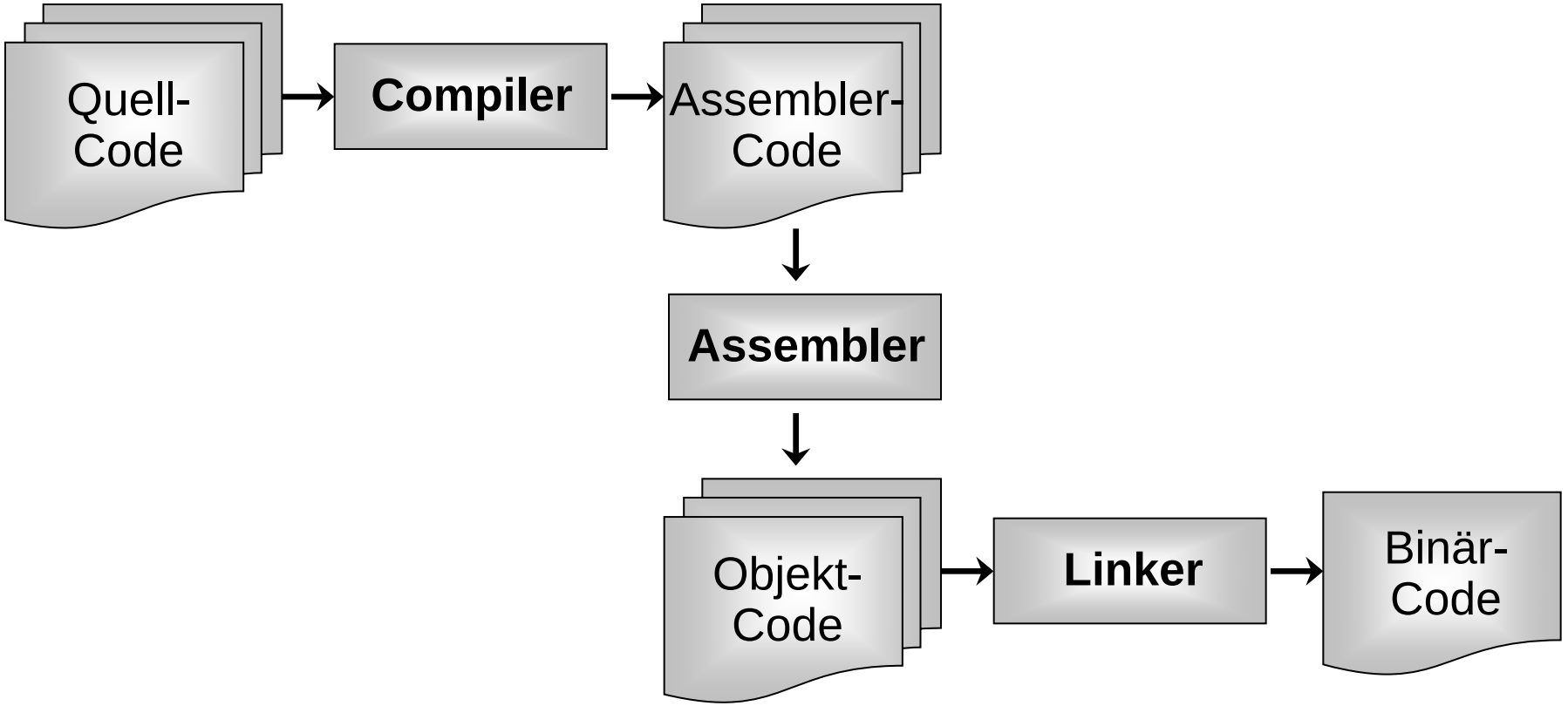
```
0x00400000: 00000000 00011111 00001000 00100000
0x00400004: 00000011 11100000 00001000 00100010
0x00400008: 00000000 00000001 11111000 00101010
0x0040000C: 00100000 00000111 10101111 11111110
0x00400010: 10101100 01100111 10101111 11110000
0x00400014: 00001000 00010000 00000000 00000000
```

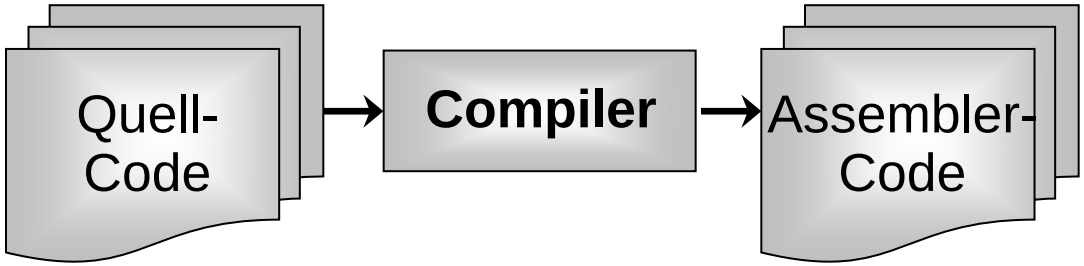
Binäre Maschinenbefehle (2)

Auch hexadezimale Notation ist ohne Kommentar äußerst unpraktisch

```
0x00400000: 00 1F 08 20     # add   $1,$0,$31
0x00400004: 03 E0 08 22     # sub   $1,$31,$0
0x00400008: 00 01 F8 2A     # slt   $31,$0,$1
0x0040000C: 20 07 AF FE     # addi  $7,$0,0xaffe
0x00400010: AC 67 AF F0     # store $7,0xaff0($3)
0x00400014: 08 10 00 00     # jump  0x400000
```

Werkzeuge zur Code-Generierung





Quellcode

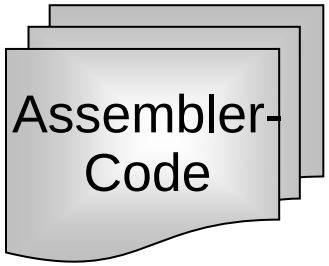
- Von Menschen les- / verstehbare Programmiersprache
- Hochsprachliche Konstrukte: Klassen, Prozeduren, Schleifen, Variablen
- Hohes Abstraktionsniveau: Maschinenunabhängige Algorithmen

Assemblercode

- Symbolischer Maschinencode
- Für Menschen eingeschränkt les- / verstehbar
- Maschinensprachen-Konstrukte: ALU-Befehle, Register, …
- Niedriges Abstraktionsniveau: Maschinenabhängige Darstellung

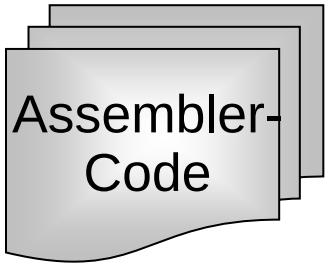
Compiler

- Übersetzt Quell- in Assemblercode
- Optimiert Code (z.B. bzgl. Laufzeit, Codegröße, Energieeffizienz, …)



- Lesbare Textdarstellung
- Keine / wenige reale Adressen
- Statt dessen: Symbolische Adressen
  z.B. **`init`**, **`loop`**, **`arr`**

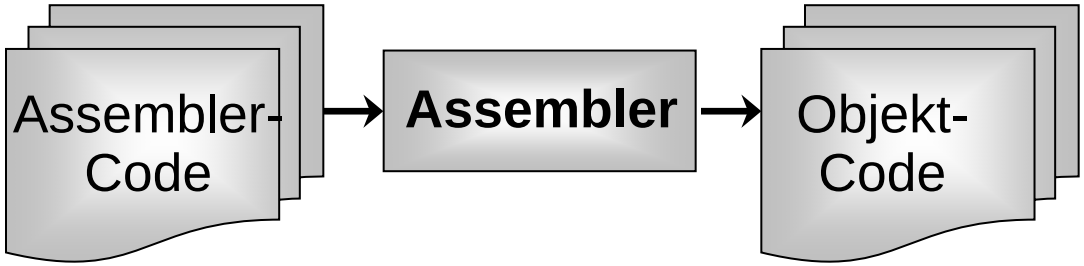
```
        .data
arr:    .word  0:10          # Reserviere 10 Array-Elemente
        .text
init:   la $7,arr($0)        # Lade Adresse von array arr nach $7
        addi $8,$7,40        # Setze End-Index
        addi $9,$0,0x41      # Lade ‘A‘ nach $9
loop:   sw $9,0($7)          # Speichere ‘A‘ in arr
        addi $7,$7,4         # Setze Index auf nächstes array-Element
        beq $7,$8,end        # Schleifen-Abbruch?
        j loop               # Schleifen-Rücksprung
end:    addi $2,$0,10        # Systemaufruf zum Beenden
        syscall
```



Zeilenorientierte Assemblerbefehle

- Prinzipieller Aufbau einer Zeile
  - [<*Label*>] <Mnemonic> [<Operanden>] [<Kommentar>]

- *Label*: symbolische Marke, die (meist) mit der aktuellen Speicheradresse verbunden wird (manchmal wird auch ‘**:**‘ hinter *Label* erwartet)
- Mnemonic: symbolischer Maschinenbefehl oder Assemblerdirektive (Steuerungsanweisung für den Assembler)
- Operanden: z.B. Angaben zur Adressierungsart, Adressen
- Kommentar: meist bis zum Ende der Zeile (oft eingeleitet mit speziellem Zeichen, z.B. ‘#‘ oder ‘**;**‘)

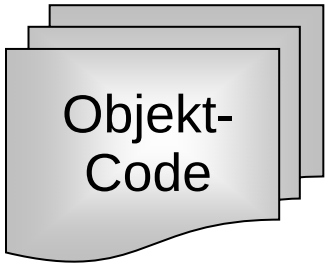


Objektcode

– Binärdarstellung von Assemblercode, nicht mehr lesbar
– Keine Klartext-Mnemonics, statt dessen 0/1-Sequenzen
– *Wenn möglich*, symbolische Adressen durch reale ersetzt

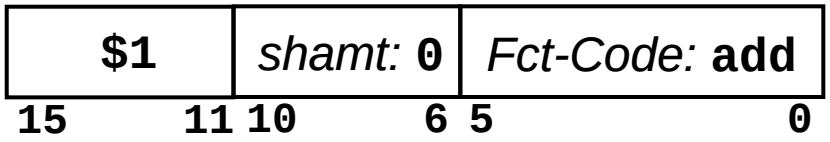
Assembler

– Zeilenweise Übersetzung
  Assembler-Befehle → Maschinen-Befehle
– Innerhalb eines Assembler-Files: Adress-Auflösung

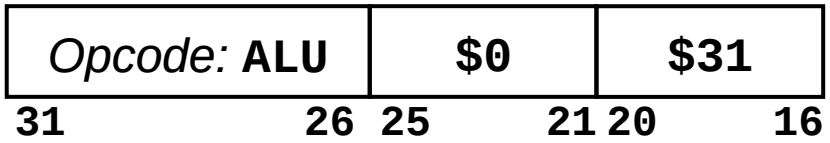


Übersetzung

`add $1,$0,$31` →



`000000 00000 11111 00001 00000 100000` ←



Adress-Auflösung

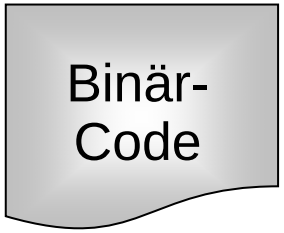
- Symbolische Adresse **`arr`** in gleichem Assembler-File deklariert:
  - Symbol **`arr`** ist Assembler bekannt
  - Ersetzung von **`arr`** durch relative Adresse, relativ in dem Objekt-File
- **`arr`** ist Assembler unbekannt:
  - Adress-Auflösung erfolgt später

Binärcode

– Ausführbare Programm-Darstellung
– *Alle* symbolischen Adressen durch reale ersetzt
– Niedrigstes Abstraktionsniveau

Linker

– Vereinigung vieler Objektcodes und Bibliotheken zu einem ausführbaren Programm
– Symbol-Auflösung mit Hilfe von Objektcode-Bibliotheken
– Code-Anordnung im Speicher

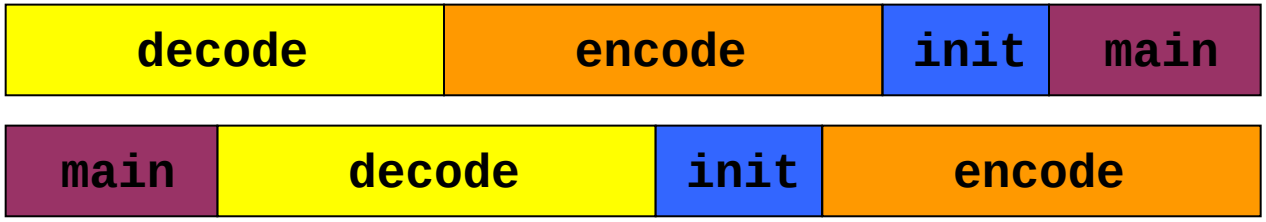


Beispiel Symbol-Auflösung

- Objektcode enthält Aufruf einer externen Funktion: **`jal init`**
- Suche **`init`** in allen anderen Objektcodes & Bibliotheken
- Füge Code von **`init`** dem Binärcode zu

Beispiel Speicher-Layout des Binärcodes

- Binärcode besteht aus Funktionen **`init`**, **`decode`**, **`encode`**, **`main`**



- Speicher-Anordnung definiert abschließend reale Adressen

Bestandteile eines Binär-Programms

- Header: Identifikator *(Magic Number)*, Längenangaben, Datum, …
- Codesegment / Textsegment: Maschinenbefehle des Programms, Konstanten, *Strings*, Sprungtabellen, …
- Datensegment:
  - (initialisierte) Variablen
  - nichtinitialisierte Variablen *(Block Storage Segment, BSS)*
- Symboltabelle: *Labels* im-/exportierter Variablen & Einsprungpunkte, …
- *Debugging*-Information:
  Zeilennummern, Variablen, Datenstrukturen & Prozedurnamen
- Relokationsinformationen
  - Adressen, welche bei einer Verschiebung des Programms angepasst werden müssen

Dateiformate für Binär-Programme

a.out
- Klassisches UNIX-Format
- Details sind plattformabhängig

COFF *(Common Object File Format)*
- Idee des einheitlichen Formats über alle Plattformen
- Headervariable spezifiziert Prozessorplattform
- Variante PE-COFF *(Portable Executable …)*: Windows-Welt

ELF *(Executable and Linkable Format)*
- Einheitliches Format für UNIX-Systeme
- Definiert auch Systemaufrufe und deren Semantik (System-API)

Was fehlt noch?

Der Lader *(Loader)*

- Laden eines Binär-Programms an die gewünschte Stelle im Hauptspeicher
- „Relozierung“ bedeutet, Programme im Hauptspeicher zu verschieben und Referenzen auf Speicheradressen anzupassen
- Verwaltung und Berücksichtigung von Relokationsinformationen

Roter Faden

7. Einführung in MIPS-Assembler

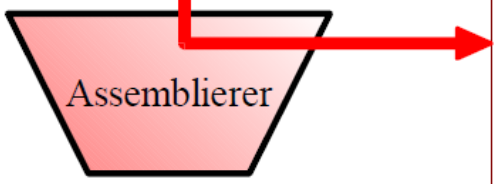
- Einführung
  - Von-Neumann Rechnerarchitektur
  - MIPS Architekturskizze
  - Abstraktionsebenen von Programmiersprachen
  - Werkzeuge zur Code-Erzeugung
- Exemplarische Betrachtung der MIPS Assemblersprache
- Übersetzung von hochsprachlichen Konstrukten
- Beispiel: *bubble sort*

Assemblersprache

Programmiersprache für maschinennahe Programmierung

- Mnemonische Kürzel für die einzelnen Maschinenbefehle
- Symbolische Namen für Programm- und Datenadressen
- Notation für die Adressierungsarten und Konstanten
- Normalerweise eine Zeile pro Maschinenbefehl

```
        .text
Main:   add     $1,  $0, $31        # add register[0] and r[31] into r[1]
        sub     $1, $31, $0         # subtract r[0] from r[31] into r[1]
        slt     $31, $0, $1         # set r[31] if r[0] < r[1]
        addi    $7, $0, -20482      # add immediate+r[0] into r[7]
        sw      $7, -20496($3)      # store word from r[7] to Mem[$3+0xaff0]
        j       Main                # jump to instruction at Mem[Main]
```




```
0x00400000: 00000000  00011111  00001000  00100000
0x00400004: 00000011  11100000  00001000  00100010
0x00400008: 00000000  00000001  11111000  00101010
0x0040000c: 00100000  00000111  10101111  11111110
0x00400010: 10101100  01100111  10101111  11110000
0x00400014: 00001000  00010000  00000000  00000000
```

MARS-IDE

IDE = *Integrated Development Environment*

– Texteditor, Assembler, Linker, Debugger, …

MARS = *MIPS Assembler and Run-Time Simulator*

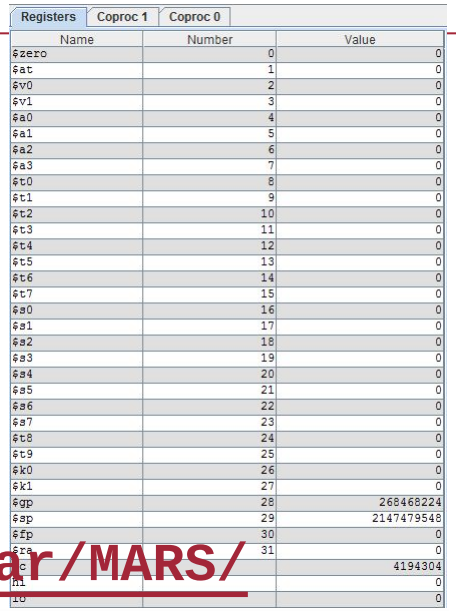
– http://courses.missouristate.edu/KenVollmar/MARS/

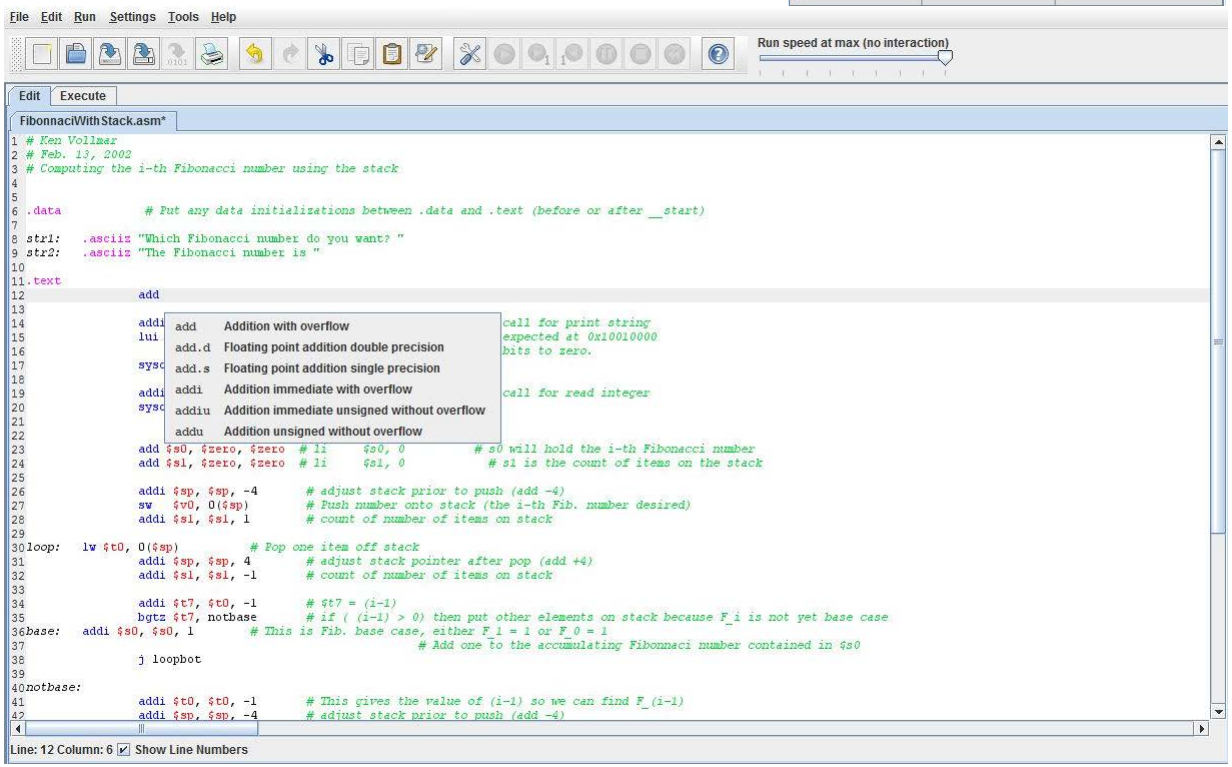
– Anzeige der Namen *(Labels)* für Sprungziele und Variablen

– Anzeige der Instruktionen im Speicher

– Anzeige der Prozessorregister

– Anzeige des Datenspeichers





(Höhere) Programmiersprachen im Vergleich (1)

Wesentliche Erleichterung für die Programmierarbeit

- Quelltext zum Beispiel in Java/C++/C
- Typenkonzept für Zuweisungen und Funktionsaufrufe
- Mehr als ein Maschinenbefehl pro Statement
- Compiler verwendet u.U. nicht alle Befehle
- Geschwindigkeitsnachteil
- Bessere Produktivität

```
...
max = 40;
val = ‘A‘;
do {
  arr[inx++] = val;
}
while ( inx < max );
...
```

Entsprechendes Programm in Assembler

- Auf die MIPS-CPU bezogene Befehlskürzel
- Gute Auslastung der vorhandenen Register
- Pro Zeile ein Maschinenbefehl
- Kein Typenkonzept

```
init: la   $7,arr($0)
      addi $8,$7,40
      addi $9,$0,‘A‘
loop: sw   $9,0($7)
      addi $7,$7,4
      beq  $7,$8,end
      j    loop
```

(Höhere) Programmiersprachen im Vergleich (2)

Entsprechendes Programm in Assembler

- Auf die MIPS-CPU bezogene Befehlskürzel
- Gute Auslastung der vorhandenen Register
- Pro Zeile ein Maschinenbefehl
- Kein Typenkonzept

```
init: la   $7,arr($0)
      addi $8,$7,40
      addi $9,$0,'A'
loop: sw   $9,0($7)
      addi $7,$7,4
      beq  $7,$8,end
      j    loop
```

Entsprechender Maschinencode

- Daten & Code im Speicher festgelegt
- Daten bei 0x10010000
- Besondere Debugtechniken
- Keine Mnemonics

```
0x400000:  0x3c011001
0x400004:  0x34210000
0x400008:  0x00013820
0x40000c:  0x20e80028
0x400010:  0x20090041
0x400014:  0xace90000
0x400018:  0x20e70004
0x40001c:  0x10e80001
0x400020:  0x08100005
```

Elemente der MIPS Assemblersprache (1)

Zeilenorientierte Assemblerbefehle

- Befehlszeilen werden in eine Maschineninstruktion übersetzt
- Assemblerdirektiven bzw. Pseudobefehle steuern die Übersetzung

Prinzipieller Aufbau einer Befehlszeile

- **`[Label:] Operationscode Operanden [Kommentar]`**

Labels

- Programmausführung beginnt beim obligatorischen Label **`main`**
- *Labels* dienen als Bezeichner und beginnen nicht mit einer Ziffer
- Symbolische Marke im Text, abgeschlossen durch ‘**`:`**‘
- Wird mit aktueller Speicheradresse verbunden
- Delegiert Adressberechnung an den Assembler

Elemente der MIPS Assemblersprache (2)

Bedeutung der Operationscodes

- Befehle mit einem *Immediate*-Operanden (*load*, *store*, *add-immediate*, …)
- Sprungbefehle (*jump absolute*, *branch on equal*, …)
- Registerbefehle (*add*, *shift*, *set less than*, …)

☞ Siehe folgende Folien

☞ Siehe Referenzkarte zur MIPS-ISA
http://refcards.com/docs/waetzigj/mips/mipsref.pdf

Operanden

- Registernamen, Adressen, Werte, Konstanten, Zeichenketten
- Angaben zur Adressierungsart
- z.B.

```
addi  $8,$0,'A'
sw    $8,capitalA($0)
```

Elemente der MIPS Assemblersprache (3)

Kommentare

– Dienen der Erläuterung durch den menschlichen Leser
– Eingeleitet durch ein ‘#‘
– Gehen bis zum Ende der Zeile

Zahlendarstellung

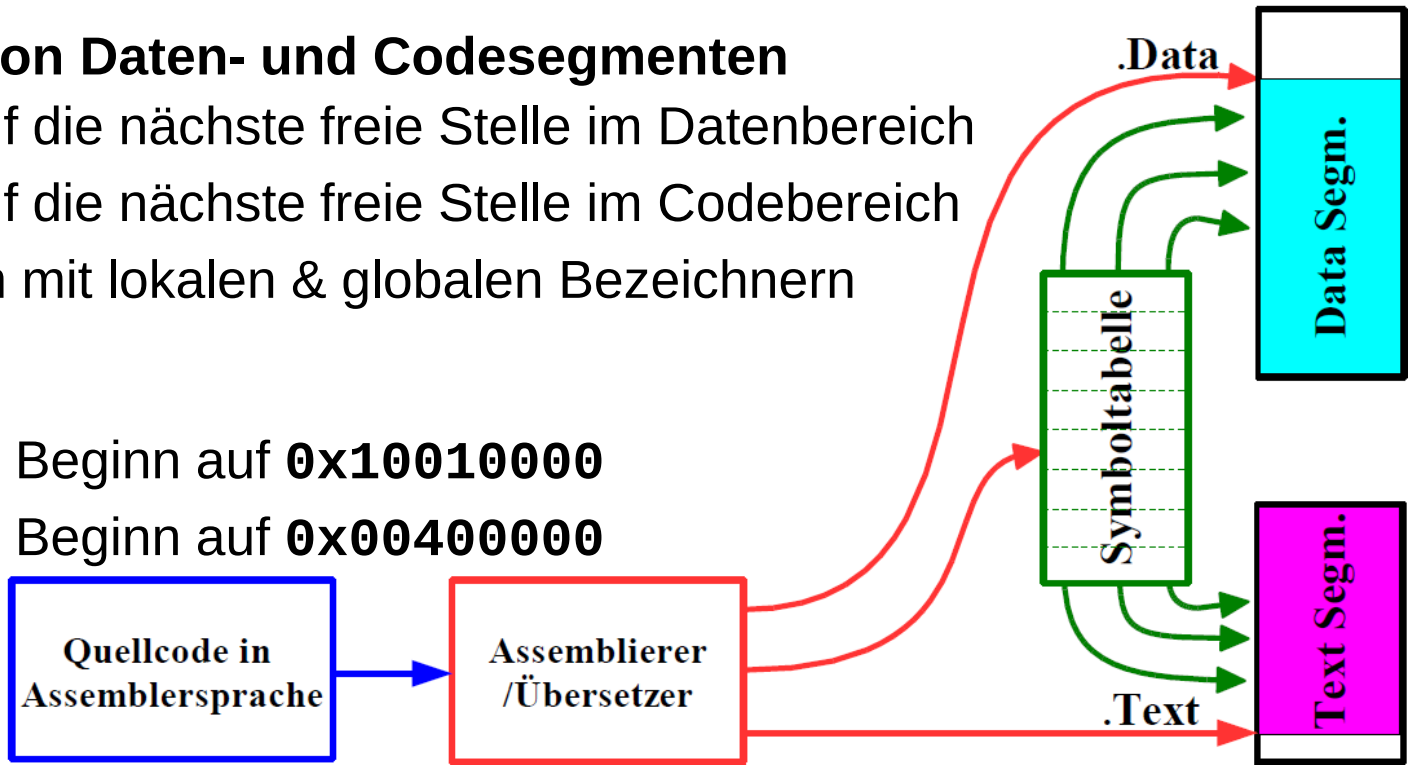
– Dezimalzahlen: wie gewohnt, z.B. **`3455`**
– Hexadezimalzahlen: beginnen mit Präfix **`0x`**, z.B. **`0xAFFE4711`**
– Oktalzahlen: beginnen mit einer Null, z.B. **`0030701`**
(Vorsicht!)

Assemblerdirektiven

Assemblerdirektiven steuern den Übersetzungsvorgang
- Vereinbarung von Konstanten und Variablen
- Platzierung von Daten- und Codebereichen
- Vereinbarung von Sprungzielen, …

Assemblierung von Daten- und Codesegmenten
- **`.data`** zeigt auf die nächste freie Stelle im Datenbereich
- **`.text`** zeigt auf die nächste freie Stelle im Codebereich
- Symboltabellen mit lokalen & globalen Bezeichnern

Initialwerte
- **`.data`** zeigt zu Beginn auf **`0x10010000`**
- **`.text`** zeigt zu Beginn auf **`0x00400000`**



Assemblerdirektiven für die Segmentierung (1)

`.text [address]`

– Nachfolgende Assemblierung in das Textsegment (normalerweise Code)
– Von einer Modifikation des Codesegments wird abgeraten…
– Optionale Adresse bezeichnet die neue Adresse

`.data [address]`

– Nachfolgende Assemblierung in das Datensegment
– Optional Datenadresse neu setzen
– Ausführung von Werten aus dem Datensegment als Instruktionen ist nicht empfohlen…

`.extern [symbSize]`

– Das nachfolgende *Label* liegt nicht im eigenen Modul
– Das *Label* hat die Größe **`symbSize`** (optional)

Assemblerdirektiven für die Segmentierung (2)

`.globl symb`

- Später geladene Module können das *Label* als externe Referenz benutzen (z.B. bei Funktionsaufrufen oder Variablenzugriffen)
- Damit können separat übersetzte Module miteinander kommunizieren
- Es gibt kein Typenkonzept!

Speicherreservierung und Variablendeklaration (1)

`.byte byte1,byte2,...`

– Bytes fortlaufend im Speicher ablegen

`.word word1,word2,...`

– 32-Bit Wörter fortlaufend im Speicher ablegen

`.float flt1,flt2,...`

– Gleitkommazahlen einfacher Präzision assemblieren

`.double dbl1,dbl2,...`

– Gleitkommazahlen doppelter Präzision assemblieren

`.ascii “string1“,...`

– Nachfolgende Zeichenketten fortlaufend im Speicher ablegen

Speicherreservierung und Variablendeklaration (2)

`.asciiz “string1“,...`

– Nachfolgende Zeichenketten als Null-terminierte C-Strings fortlaufend im Speicher ablegen

`.align n`

– Die nächste Adresse für den Assembler hat die untersten **n** Bits auf Null gesetzt

`.space size`

– **`size`** Bytes leeren Speicherplatz freilassen

Speicherreservierung und Variablendeklaration (3)

– Hat eine Zeile ein *Label*, so kann sie als Variable referenziert werden

```
          .data
wordVar:  .word  3536
byteVar:  .byte  0x35
          .align 2
char:     .ascii “z“
          .text
          lw $9,wordVar($0)     # Variable nach Register 9
```

– MIPS-Befehle brauchen zwingend ein *Alignment* auf 4-Byte-Grenzen!

– Reservierung eines ganzen Speicherbereichs (DLX-Assembler)
z.B. **`array: .space 100*4`** reserviert 100 aufeinander folgende Speicherwörter ☞ Assemblierung fährt fort bei Adresse **`array`** + 400

Semantik von Assemblerbefehlen

Register-Transfer-Notation

– Argumente oder Ziele: Register oder Speicher, z.B.

  `Reg[3]`, **`PC`**, **`Speicher[4]`**

– Zuweisungen: mittels **`:=`**, z.B.

  `PC := Reg[31]`

– Konstante Bitvektoren: Einschluss in **“**, z.B.

  “0101010011“

– Selektion einzelner Bits: Punkt + runde Klammern, z.B.

  `PC.(15:0)`

– Konkatenation (Aneinanderreihung) mit **`&`**, z.B.

  `(Hi & Lo)`, **`PC := PC.(31:28) & I(25:0) & “00“`**

Semantik des Additionsbefehls

Beispiel

```
add $3,$2,$1    # Reg[3] := Reg[2] + Reg[1]
```

Vorzeichenbehandlung?

- Muss der **add**-Befehl höchstwertigstes Bit als Vorzeichen betrachten?
- Nein: Addition von Zahlen im Zweierkomplement klappen stets, unabhängig ob die Bitfolgen vorzeichenbehaftet oder vorzeichenlos sind.

☞ Es reicht ein **add**-Befehl aus, der sowohl vorzeichenbehaftete als auch vorzeichenlose Zahlen addieren kann!

- Allerdings Unterschiede zur Behandlung von Bereichsüberschreitungen
- MIPS bietet zwei Additionsbefehle:
  - **add** (für vorzeichenbehaftete Zahlen): signalisiert Bereichsüberschreitungen
  - **addu** (für *unsigned* Zahlen): ignoriert Bereichsüberschreitungen

Der `load word`-Befehl

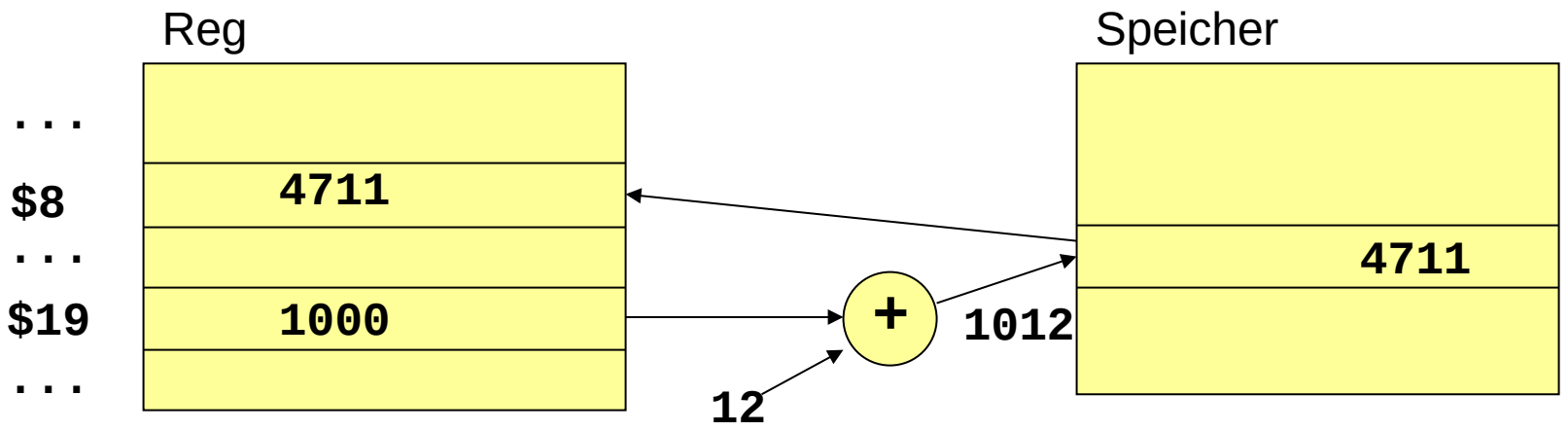
Allgemeine Form

```
lw ziel,offset(reg)
```

Mit **`ziel`**, **`reg`** ∈ **`$0`**, ..., **`$31`**, **`offset`**: Konstante ∈ -2^{15}, …, $2^{15} - 1$ bzw. deren Bezeichner, deren Wert beim Laden des Programms bekannt sein muss

Beispiel

```
lw $8, 12($19)        # Reg[8] := Speicher[12+Reg[19]]
```



Der `store word`-Befehl

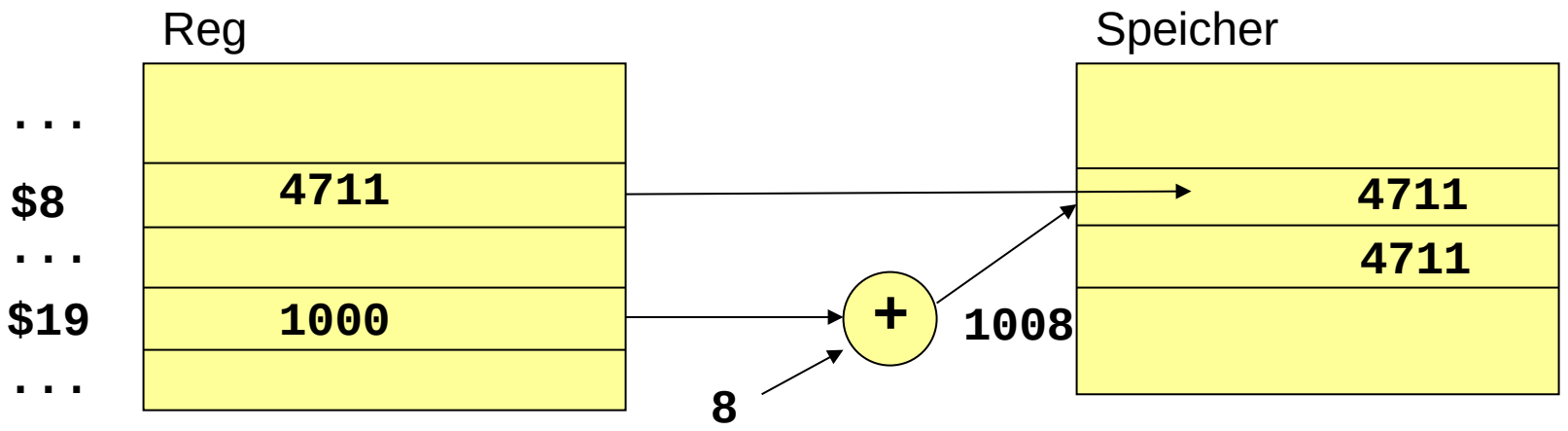
Allgemeine Form

```
sw quel,offset(reg)
```

Mit **`quel`**, **`reg`** ∈ **`$0`**, ..., **`$31`**, **`offset`**: Konstante ∈ -2^{15}, …, $2^{15} - 1$ bzw. deren Bezeichner, deren Wert beim Laden des Programms bekannt sein muss

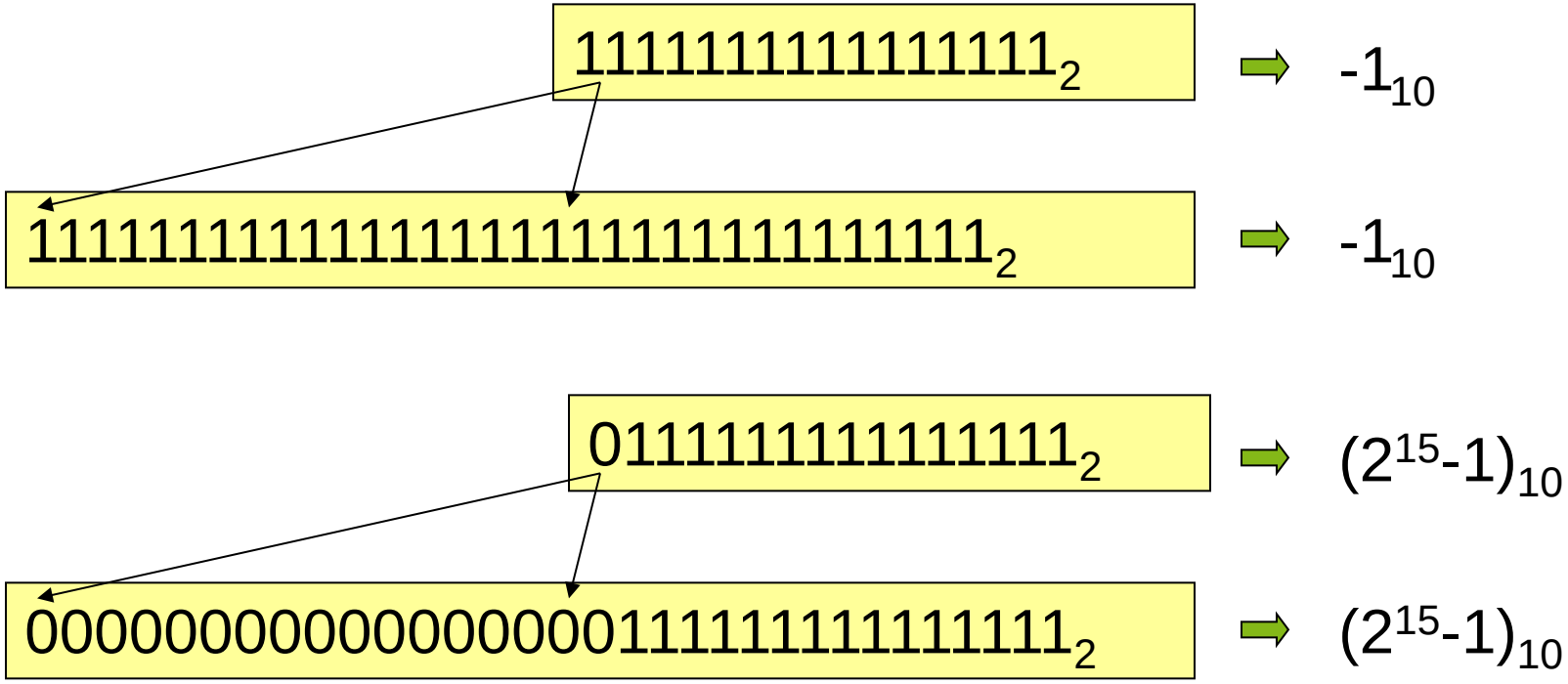
Beispiel

```
sw $8, 8($19)        # Speicher[8+Reg[19]] := Reg[8]
```



Nutzung des 16-Bit Offsets in 32-Bit Arithmetik

Replizierung des Vorzeichenbits liefert 32-Bit Zweierkomplement-Zahl



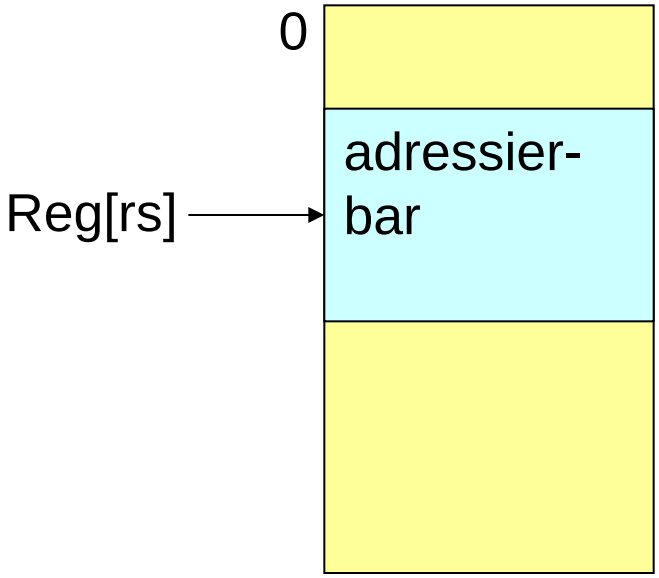
`sign_ext(a, m)`: Funktion, die Bitvektor **`a`** auf **`m`** Bits erweitert

Auswirkung der Nutzung von `sign_ext` bei Adressierung

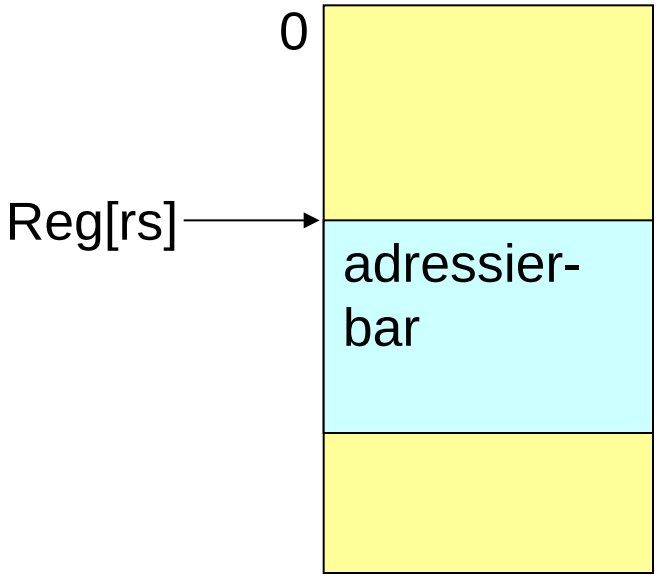
Präzisere Beschreibung der Bedeutung des **sw**-Befehls

```
sw rt,offset(rs)      # Speicher[Reg[rs]+
                        sign_ext(offset,32)] :=
                      Reg[rt]
```

mit Vorzeichenerweiterung



mit *zero-extend* (00000 & ...)



Vorzeichenerweiterung bei I-Format Instruktionen

– **Beschreibung der Bedeutung des *add immediate*-Befehls***

```
addi rt,rs,const        # I-Format
  # Reg[rt] := Reg[rs] + sign_ext(const,32)
```

– ***add immediate unsigned*-Befehl**

```
addiu rt,rs,const       # I-Format
```

Wie **`addi`**, nur ohne Erzeugung von Überläufen (!)

* Der MIPS-Assembler erzeugt vielfach automatisch *immediate*-Befehle, wenn statt eines Registers direkt eine Konstante angegeben ist, z.B. bei **`add $2,$2,3`**

Subtraktionsbefehle (R-Format)

- **`sub $4,$3,$2`** **`# Reg[4] := Reg[3] – Reg[2]`**
  Subtraktion, Ausnahmen möglich

- **`subu $4,$3,$2`** **`# Reg[4] := Reg[3] – Reg[2]`**
  Subtraktion, keine Ausnahmen signalisiert

Multiplikationsbefehle

– Die Multiplikation liefert doppelt lange Ergebnisse

  Beispiel: $-2^{31} * -2^{31} = 2^{62}$

  2^{62} benötigt zur Darstellung einen 64-Bit-Vektor

– Wo soll man ein solches Ergebnis abspeichern?

☞ MIPS-Lösung: 2 spezielle Register **`Hi`** und **`Lo`**:

```
mult $2,$3        # Hi & Lo := Reg[2] * Reg[3]
```

Höherwertiger Teil des Ergebnisses

Niederwertiger Teil des Ergebnisses

Konkatenation (Aneinanderreihung)

Transport in allgemeine Register:

```
mfhi $3 # Reg[3] := Hi
mflo $3 # Reg[3] := Lo
```

Varianten des Multiplikationsbefehls

- **`mult $2,$3`** **`# Hi & Lo := Reg[2] * Reg[3]`**
  für ganze Zahlen in 2k-Darstellung
- **`multu $2,$3`** **`# Hi & Lo :=`** $\mathbf{Reg[2] *_u Reg[3]}$
  für natürliche Zahlen *(unsigned int)*
- **`mul $4,$3,$2`** `#` besteht aus **`mult`** und **`mflo`**
  `# Hi & Lo := Reg[3] * Reg[2]; Reg[4] := Lo`
  für 2k-Zahlen, niederwertiger Teil hinterher im allgemeinen Register
- **`mulo $4,$3,$2`**
  `# Hi & Lo := Reg[3] * Reg[2]; Reg[4] := Lo`
  wie **`mul`**, nur inkl. Überlauftest
- **`mulou $4,$3,$2`**
  `# Hi & Lo :=` $\mathbf{Reg[3] *_u Reg[2]; Reg[4] := Lo}$
  wie **`mulo`**, nur für *unsigned integers*

Divisionsbefehle

Problem

– Man möchte gerne sowohl den Quotienten als auch den Rest der Division speichern
– Passt nicht zum Konzept eines Ergebnisregisters

MIPS-Lösung: Verwendung von `Hi` und `Lo`

– `div $2,$3` # für ganze Zahlen in 2k-Darstellung
  `# Lo := Reg[2] / Reg[3]; Hi := Reg[2] mod Reg[3]`
– `divu $2,$3` # für natürliche Zahlen *(unsigned integers)*
  `# Lo := Reg[2] /u Reg[3]; Hi := Reg[2] mod Reg[3]`

Logische Befehle

Beispiel	Bedeutung	Kommentar
`and $4,$3,$2`	Reg[4] := Reg[3] ∧ Reg[2]	und
`or $4,$3,$2`	Reg[4] := Reg[3] ∨ Reg[2]	oder
`andi $4,$3,100`	Reg[4] := Reg[3] ∧ 100	und mit Konstanten
`sll $4,$3,10`	Reg[4] := Reg[3] << 10	Schiebe nach links logisch
`srl $4,$3,10`	Reg[4] := Reg[3] $>>_l$ 10	Rechts-Schieben logisch
`sra $4,$3,10`	Reg[4] := Reg[3] $>>_a$ 10	Rechts-Schieben arithmetisch

zero_ext()

Logisches vs. Arithmetisches Rechts-Schieben

Logisches Rechts-Schieben

– Das höchstwertige Bit *(most-significant bit, MSB)* wird stets mit ‘0‘ gefüllt.

☞ Verschiebt man eine vorzeichenbehaftete Zahl logisch, geht u.U. das Vorzeichenbit verloren!

☞ Aus negativen Zahlen können plötzlich positive Zahlen werden:

$$\texttt{-8} >>_l \texttt{1} = \texttt{1000} >>_l \texttt{1} = \texttt{0100} = \texttt{4}$$

Arithmetisches Rechts-Schieben

– Das MSB wird stets mit dem alten MSB gefüllt.

☞ Das Vorzeichenbit bleibt beim arithmetischen Schieben erhalten:

$$\texttt{-8} >>_a \texttt{1} = \texttt{1000} >>_a \texttt{1} = \texttt{1100} = \texttt{-4}$$

Laden von Konstanten (1)

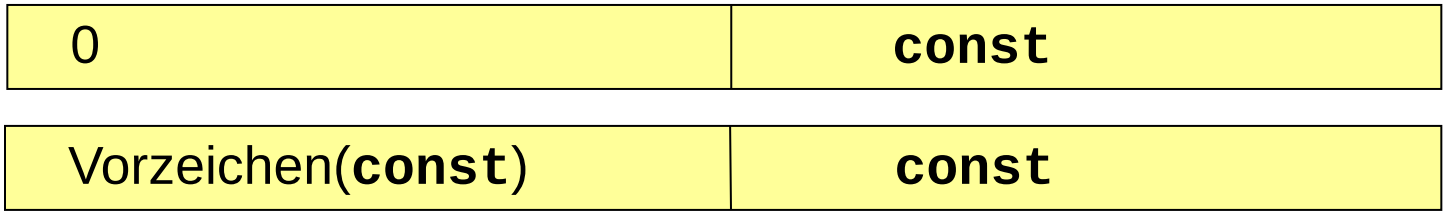
Wie kann man 32-Bit-Konstanten in Register laden?

– Direktoperanden für das untere Halbwort:

```
ori r,s,const          # Reg[r] := Reg[s] ∨
                                 (0000_16 & const)
addiu r,s,const        # Reg[r] := Reg[s] +
                                 sign_ext(const,32)
```

– Für den Sonderfall **`s = $0`**:

```
ori r,$0,const         # Reg[r] := 0 ∨
                                 zero_ext(const,32)
addiu r,$0,const       # Reg[r] := sign_ext(const,32)
```



Laden von Konstanten (2)

– Für Konstanten mit unterem Halbwort = 0

`lui r,const` **`# Reg[r] := const & 0000`**$_{16}$

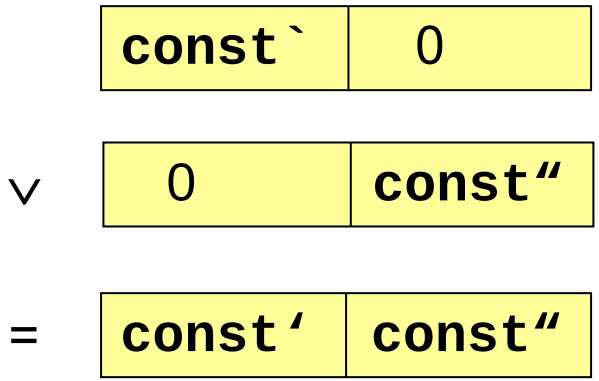
(load upper immediate)

`const`	0

Laden von Konstanten (3)

– Für andere Konstanten

```
lui $1,const‘
ori r,$1,const‘‘
```



– Obige Sequenz wird vom Assembler für **`li`** *(load immediate)* erzeugt

☞ Register **`$1`** ist immer für den Assembler freizuhalten!

Der *load address*-Befehl `la`

In vielen Fällen muss die Adresse einer Speicherzelle in einem Register bereit gestellt werden

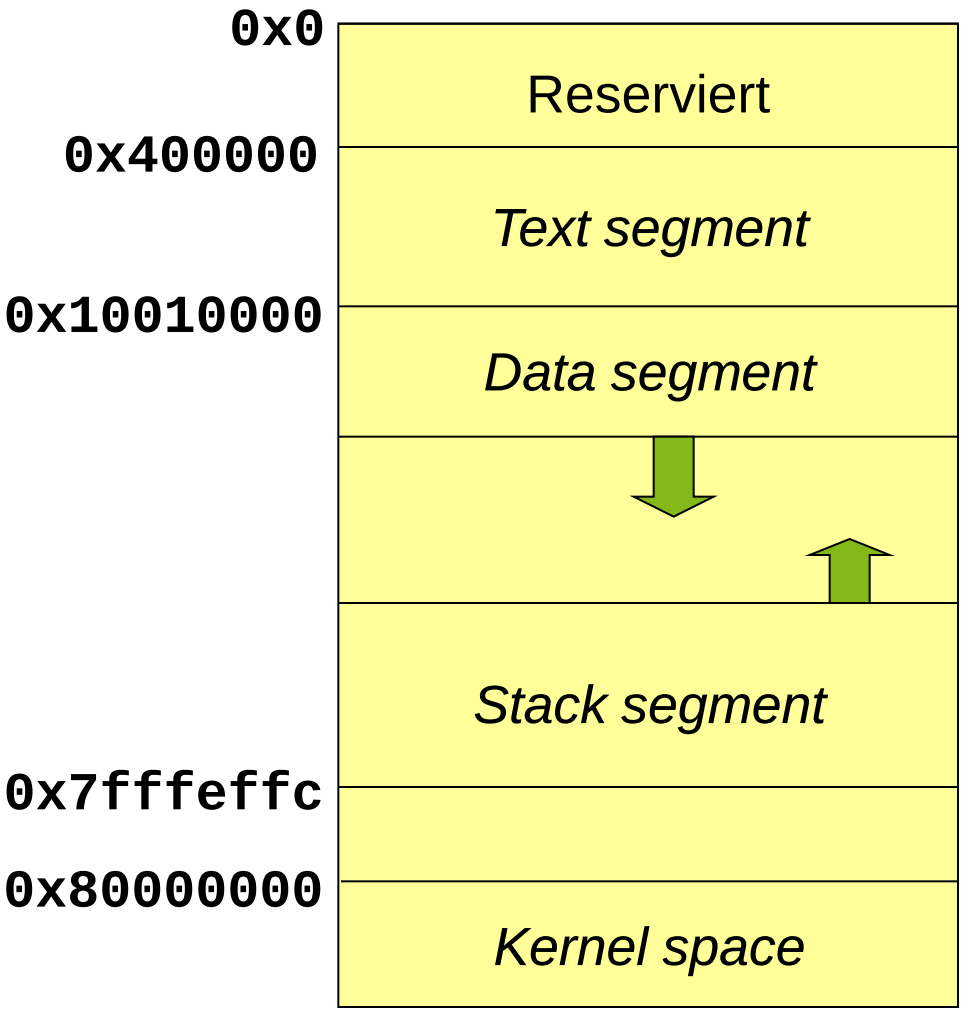
– Dies ist mit den bislang vorgestellten Befehlen zwar möglich, der Lesbarkeit wegen wird aber ein eigener Befehl eingeführt.
– Der Befehl **`la`** entspricht dem **`lw`**-Befehl, wobei der Speicherzugriff unterbleibt.

Beispiel

– `la $2,0x20($3)  # Reg[2] := 0x20 + Reg[3]`
– **`la`** kann über eine Sequenz aus **`li`** und **`add`** erzeugt werden

Physikalischer Adressraum der MIPS

Diese MIPS unterstützt keinen virtuellen Speicher, daher nur ein physikalischer Adressraum mit statischer Zuteilung für das Betriebs-system!



Befehle

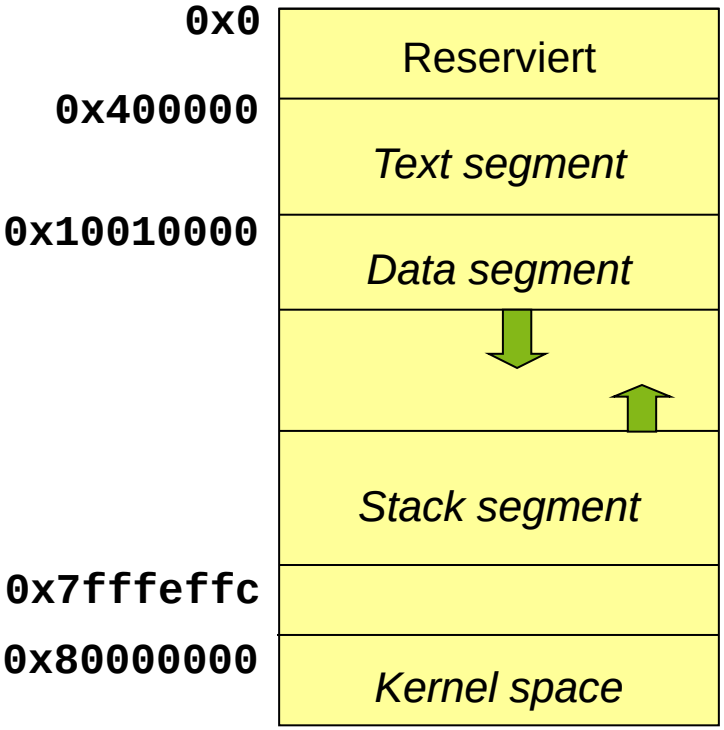
Daten bekannter Größe
+ weitere Daten

Dynamisch erzeugte Daten,
Auf- und Abbau bei Aufruf von
Prozeduren/Unterprogrammen

Adressen Ein- / Ausgabegeräte

Betriebssystem

Problem dieser *Memory Map*



Daten- und Stack-Segmente wachsen aufeinander zu. Liegen beide Segmente im gleichen physikalischen Speicher, können sie sich u.U. gegenseitig überschreiben!

Ausweg:

– Umrechnung von virtuellen in reale Adressen

Beispiel zur Benutzung der Speicherbereiche

Beispielprogramm

```
        .globl main              # Globales Symbol
  main: lw $2,0x10010000($0)     # Anfang Datenbereich
        lw $3,0x10010004($0)
        add $3,$2,$3
        sw $3,0x10010008($0)
```

Problem: zu große Adress-Offsets in `lw`/`sw`-Befehlen

- **`lw $z,d($b)`** wird bei zu großem Offset **`d`** von MARS übersetzt in:

  `lui $1,`<*obere 16 Bit von* **`d`**>

  `addu $1,$1,$b` **`# Ohne Overflow-Prüfung`**

  `lw $z,`<*untere 16 Bit von* **`d`**>**`($1)`**
- **`addu`** entfällt, falls **`($b)`** nicht vorhanden ist, aber nicht bei **`b`** = **`$0`**!
- 3 MIPS-Befehle für nur einen **`lw`** ☞ extrem ineffizient

Geschicktere Verwendung des Adressierungsmodus

Ursprüngliche Version des Additionsprogramms

```
        .globl main             # Globales Symbol
main:   lw $2,0x10010000($0)    # Anfang Datenbereich
        lw $3,0x10010004($0)
        add $3,$2,$3
        sw $3,0x10010008($0)
```

Geschicktere Version

```
        .globl main
main:   li $4,0x10010000
        lw $2,0($4)
        lw $3,4($4)
        add $3,$2,$3
        sw $3,8($4)
```

Sprungbefehle

Problem

– Mit dem bislang erklärten Befehlssatz: keine Abfragen möglich

Lösung: (bedingte) Sprungbefehle *(conditional branches)*

Elementare MIPS-Befehle:

– **`beq rega,regb,Sprungziel`** *(branch if equal)*
mit **`rega`**, **`regb`** ∈ **`$0`**, ..., **`$31`** und
`Sprungziel`: 16-Bit *integer* (oder Bezeichner dafür)

– **`bne rega,regb,Sprungziel`** *(branch if not equal)*
mit **`rega`**, **`regb`** ∈ **`$0`**, ..., **`$31`** und
`Sprungziel`: 16-Bit *integer* (oder Bezeichner dafür)

– **`j Sprungziel`** *(unconditional branch)*
mit **`Sprungziel`**: 26-Bit *integer* (oder Bezeichner dafür)

Tests auf <, ≤, >, ≥

MIPS-Lösung: `slt`-Befehl *(set if less than)*

```
slt ra,rb,rc
  # Reg[ra] := if Reg[rb] < Reg[rc] then 1 else 0
```

– Tests werden vom Assembler aus **`slt`**, **`bne`** und **`beq`**-Befehlen zusammengesetzt

– Beispiel:
aus **`blt $2,$3,L`** wird

```
slt $1,$2,$3
bne $1,$0,L
```

– Erinnerung: **`$1`** ist per Konvention dem Assembler vorbehalten!

Weitere arithmetische Tests

Weitere Befehle

- **`slti`** (Vergleich mit Direktoperanden)
- **`sltu`** (Vergleich für Betragszahlen)
- **`sltui`** (Vergleich für Betragszahlen als Direktoperand)
- **`ble`** (Verzweige für *less or equal*)
- **`blt`** (Verzweige für *less than*)
- **`bgt`** (Verzweige für *greater than*)
- **`bge`** (Verzweige für *greater or equal*)

} Pseudo-befehle

Realisierung von berechneten Sprüngen

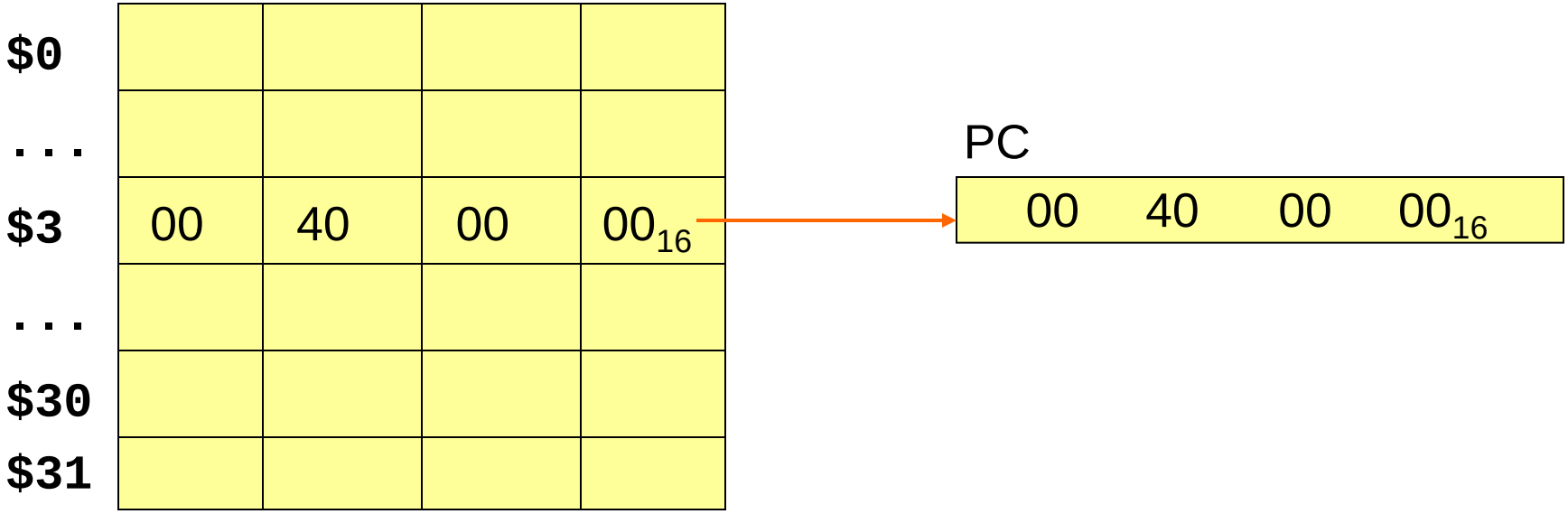
Der `jr`-Befehl *(jump register)*

```
jr reg              # PC := Reg[reg]
```

mit **`reg`** $\in$ **`$0`**, ..., **`$31`**

Beispiel: **`jr $3`**

Registerspeicher (**Reg**)



Realisierung von Prozeduraufrufen

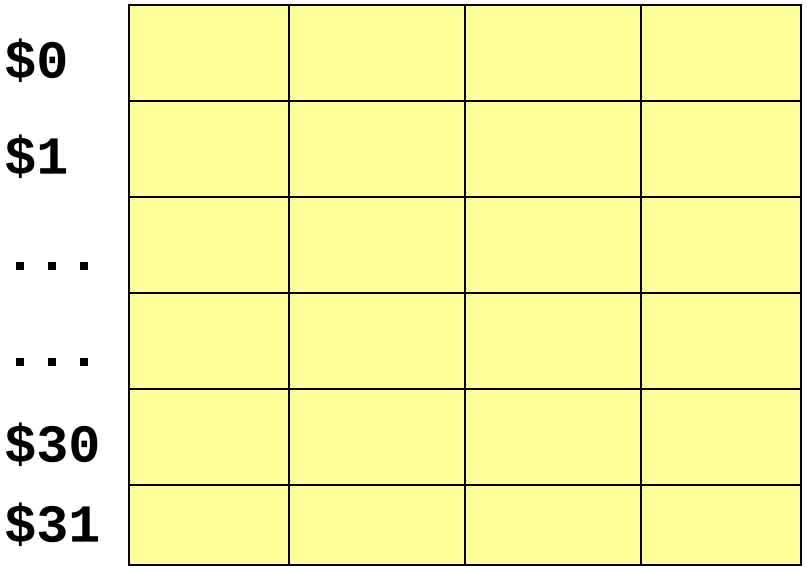
Der `jal`-Befehl *(jump and link)*

```
jal adresse                 # Reg[31] := PC+4; PC := adresse
```

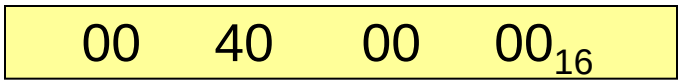
wobei **`adresse`** im aktuellen 256 MB-Block liegt

Beispiel: `jal 0x410000`

Registerspeicher (**Reg**)



PC



Vor der Ausführung von **`jal`** an Adresse 400000_{16}

Realisierung von Prozeduraufrufen

Der `jal`-Befehl *(jump and link)*

```
jal adresse                 # Reg[31] := PC+4; PC := adresse
```

wobei **`adresse`** im aktuellen 256 MB-Block liegt

Beispiel: `jal 0x410000`

Registerspeicher (**Reg**)

$0				
$1				
...				
...				
$30				
$31	00	40	00	04_{16}

PC

00	41	00	00_{16}

Nach der Ausführung von **`jal`** an Adresse 400000_{16}

Roter Faden

7. Einführung in MIPS-Assembler

- Einführung
- Exemplarische Betrachtung der MIPS Assemblersprache
  - Befehlsformate
  - MIPS Maschinenbefehle und deren Bedeutung
- Übersetzung von hochsprachlichen Konstrukten
- Beispiel: *bubble sort*

Hochsprachliche Kontrollkonstrukte: *if-Statements* (1)

Übersetzung eines einfachen *if-Statements*

```
    if ( i == j ) goto L1;
    f = g + h;
L1: f = f - i;
```

in

```
    beq $19,$20,L1      # nach L1, falls i == j
    add $16,$17,$18     # f = g + h
L1: sub $16,$16,$19     # immer ausgeführt
```

Assembler rechnet **`L1`** in die im Maschinencode zu speichernde Konstante um

Label: Symbolische Bezeichnung für eine Befehlsadresse

Hochsprachliche Kontrollkonstrukte: *if-Statements* (2)

Übersetzung eines (normalen) *if-Statements*

```
    if ( i == j )
      f = g + h;
    f = f – i;
```

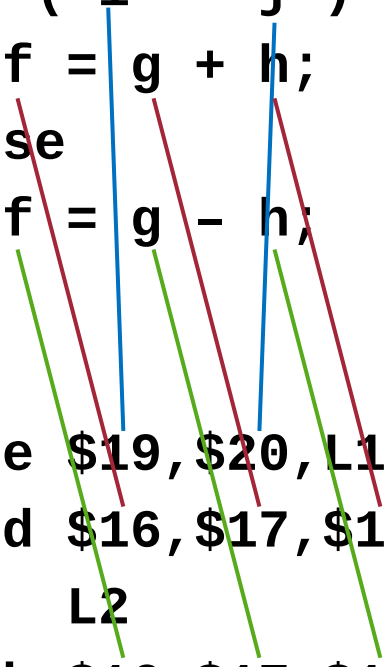
in

```
    bne $19,$20,L1          # nach L1, falls i ≠ j!
    add $16,$17,$18         # f = g + h
L1: sub $16,$16,$19         # immer ausgeführt
```

Hochsprachliche Kontrollkonstrukte: *if-else-Statements*

Übersetzung einer vollständigen Fallunterscheidung

```
    if ( i == j )
      f = g + h;
    else
      f = g - h;
```



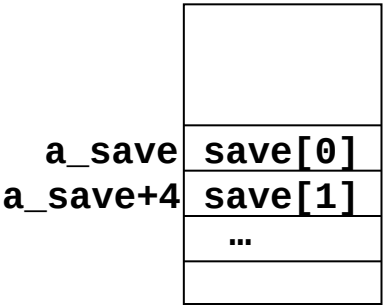
in

```
    bne $19,$20,L1          # nach L1, falls i ≠ j!
    add $16,$17,$18         # then-Zweig: f = g + h
    j   L2                  # Umgehe else-Zweig stets
L1: sub $16,$17,$18         # else-Zweig: f = g - h
L2: ...                     # Ende des if-else
```

Hochsprachliche Kontrollkonstrukte: *Array*-Zugriffe

Situation in der Programmiersprache C

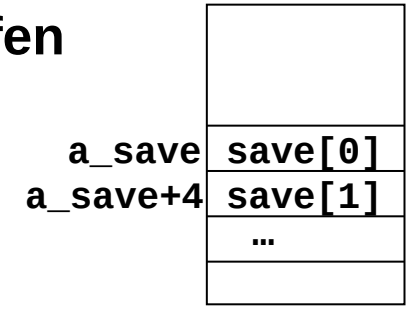
- In C beginnen *Arrays* mit dem Index 0
- Die Adresse des *Arrays* ist gleich der Adresse des *Array*-Elements 0
- Wenn jedes *Array*-Element ein Wort belegt, dann belegt Element *i* das Wort *i* des *Arrays*
- Wenn **a_save** die Anfangsadresse eines *Arrays* ist, dann ist (**a_save** + *i* * *c*) die Adresse von Element *i*
- *c* ist die Anzahl der adressierbaren Speicherzellen, die pro Element des *Arrays* belegt werden (d.h. *c* = 4 bei 32-Bit Integer-Elementen auf der MIPS-Maschine)



Hochsprachliche Kontrollkonstrukte: Schleifen & *Arrays*

Übersetzung einer einfachen Schleife mit *Array*-Zugriffen

```
    while ( save[i] == k )
      i = i + j;
```



in

```
    li  $10,4                 # Lade $10 mit 4
L1: mul $9,$19,$10            # Berechne i * 4
    lw  $8,a_save($9)         # Lade save[i] nach $8
    bne $8,$21,L2             # Ist save[i] == k?
    add $19,$19,$20           # Nein: i = i + j
    j   L1                    # Schleifen-Rücksprung
L2: ...                       # Ende der Schleife
```

Hochsprachliche Kontrollkonstrukte: *for*-Schleifen

Übersetzung einer einfachen *for*-Schleife

```
    j = 0;
    for ( i = 0; i < n; i++ )
      j = j + i;
```

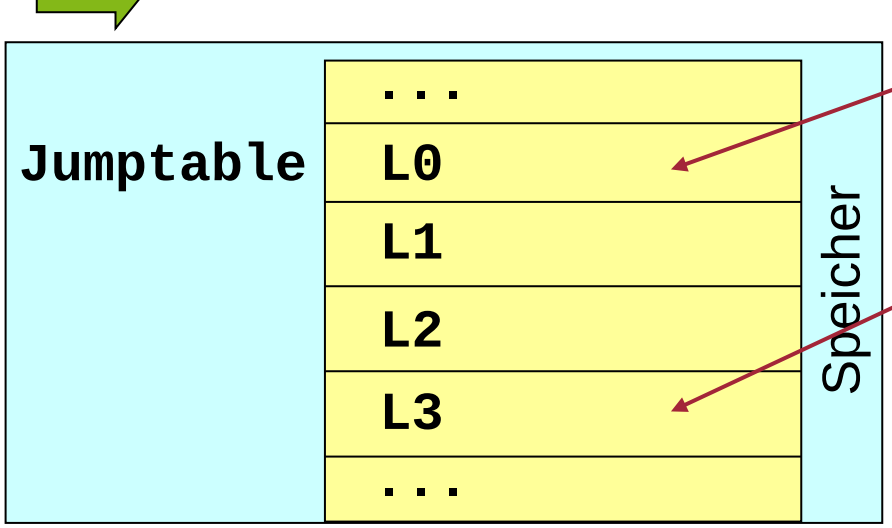
in

```
    li   $2,0                  # j = 0
    li   $3,0                  # i = 0
L1: bge  $3,$4,L2              # Ist i >= n?
    add  $2,$2,$3              # j = j + i
    addi $3,$3,1               # i++
    j    L1                    # Schleifen-Rücksprung
L2: ...                        # Ende der Schleife
```

Hochsprachliche Kontrollkonstrukte: *switch*-Anweisungen

Übersetzung mit Hilfe einer Sprungtabelle *(jump table)* *Annahme:* **k** = *2*

```
switch ( k ) {
    case 0: f = i + j; break;          /* k = 0 */
    case 1: f = g + h; break;          /* k = 1 */
    case 2: f = g - h; break;          /* k = 2 */
    case 3: f = i - j; break;          /* k = 3 */
}
```



Adresse des ersten Befehls für **`case 0`**

Adresse des ersten Befehls für **`case 3`**

```
.data
Jumptable: .word L0,L1,L2,L3, …
```

Hochsprachliche Kontrollkonstrukte: *switch*-Anweisungen

Realisierung mittels des `jr`-Befehls

```
switch ( k ) {
  case 0: f = i + j; break;
  case 1: f = g + h; break;
  case 2: f = g - h; break;
  case 3: f = i - j; break; }
```

```
      li   $10,4
swit: mul  $9,$10,$21        # Reg[9] := k * 4
      lw   $8,Jumptable($9)  # Lade k. Tabelleneintrag
      jr   $8                # Springe dorthin
L0:   add  $16,$19,$20       # Code von Case 0
      j    Exit              # entspricht break
L1:   add  $16,$17,$18       # Code von Case 1
      j    Exit              # ...
L2:   sub  $16,$17,$18
      j    Exit
L3:   sub  $16,$19,$20
Exit: ...                    # Ende des switch/case
```

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (1)

```
void C()
{
  ...
}
void B()
{
  C();
}
void A()
{
  B();
}
void main(){ A(); }
```

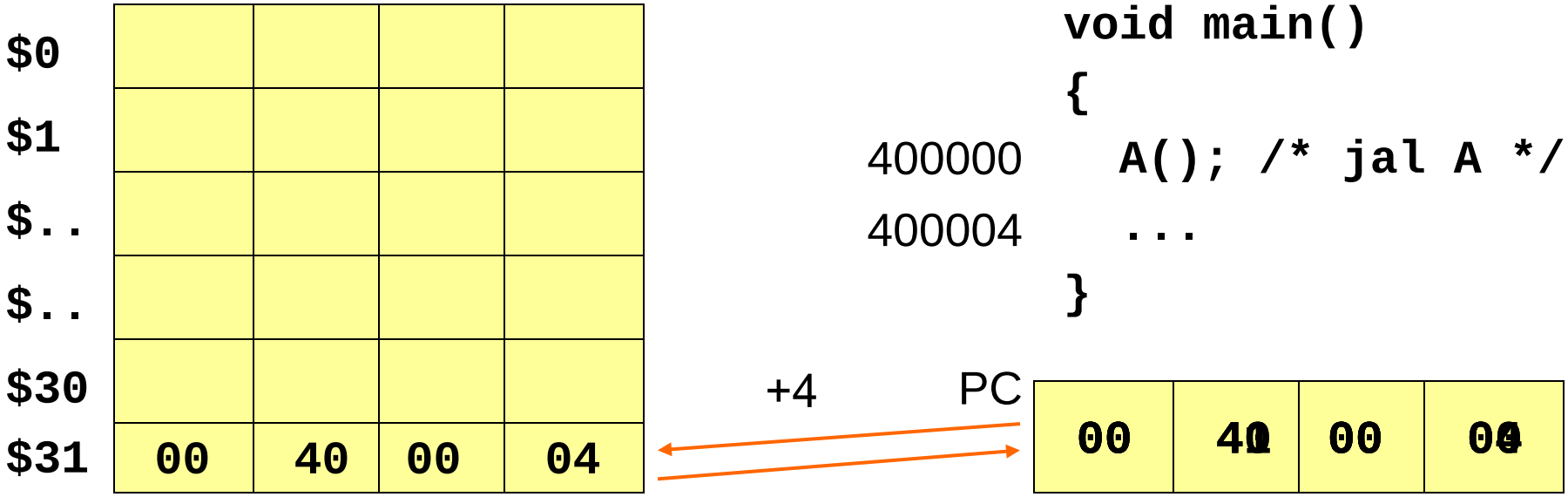
Man muss

- sich merken können, welches der auf den aktuellen Befehl im Speicher folgende ist (d.h. man muss sich PC+4 merken), und
- an eine (Befehls-) Adresse springen.

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (2)

Realisierung nicht verschachtelter Aufrufe mit `jal` und `jr`

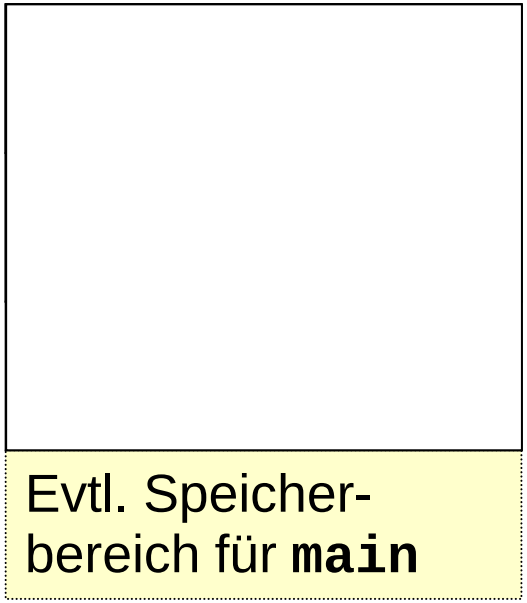
```
                 void A()
                 {
410000             ...
410004             /* jr $31 */
                 }
                 void main()
                 {
400000             A(); /* jal A */
400004             ...
                 }
```



Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (3)

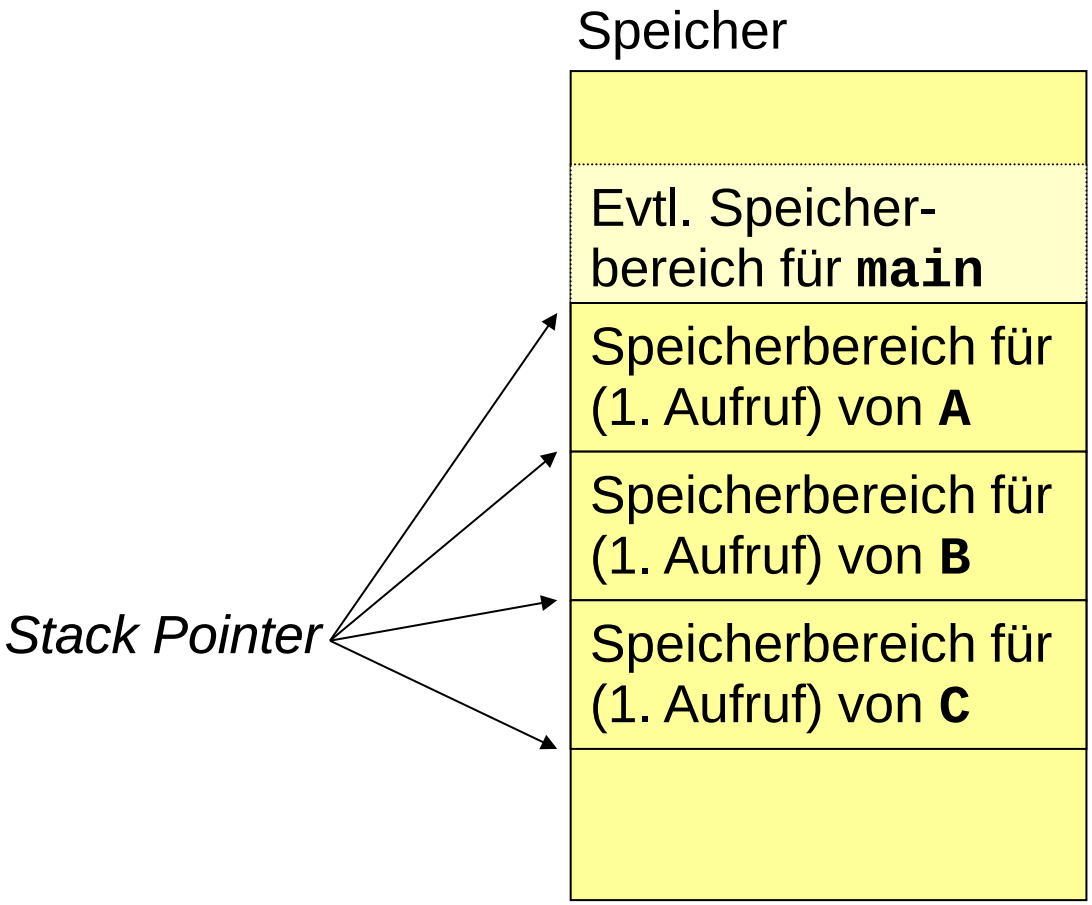
```
void C()
{
  ...
}
void B()
{
  C();
}
void A()
{
  B();
}
void main(){ A(); }
```

Das Stapelprinzip



Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (4)

Realisierung eines Stapels im Speicher



Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (5)

Stack / Stapelspeicher / Kellerspeicher

– Hauptspeicherbereich, dessen eines Ende dynamisch wächst bzw. schrumpft

Verwendung des _Stacks_ für

– Übergabe von Parametern an Funktionen
– Rücksprungadressen bei Funktionsaufrufen
– Zurückschneiden des *Stacks* nach Funktionsaufrufen
– Sicherung von Registerinhalten
– Lokale Variablen in Funktionen

Konzept des _Stacks_

– Bereits bekannt (☞ Kapitel 3, *Interrupts*)
– *Stack* für Prozeduraufrufe ist derselbe wie der zur *Interrupt*-Behandlung

☞ Jeder Rechner hat systemweit *einen zentralen Stack*

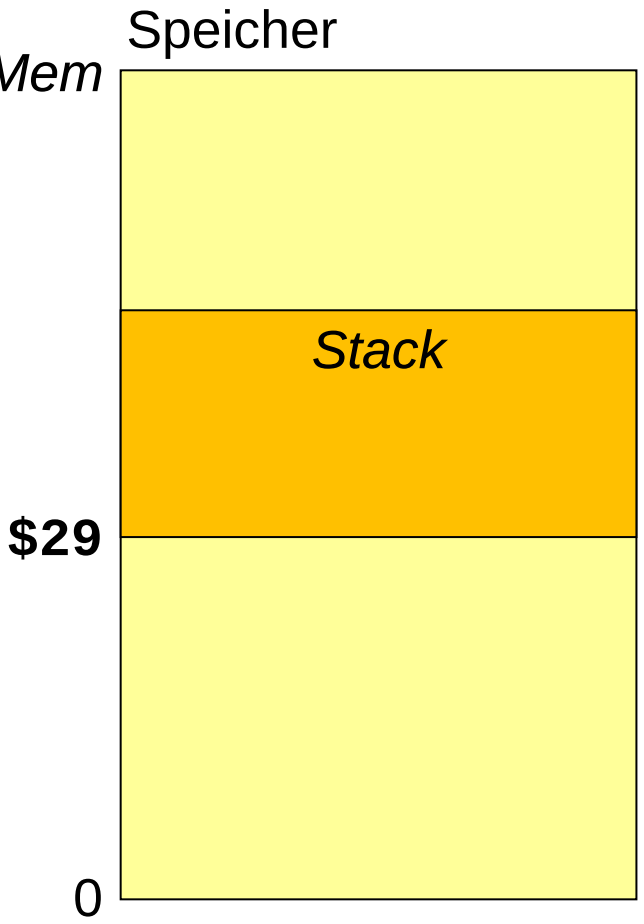
Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (6)

MIPS-Registerkonvention

- Register **$29** dient als *Stack Pointer*
- Register **$30** zeigt auf die lokalen Variablen *(Frame Pointer)*
- Register **$31** enthält Rücksprungadresse

Wachstumsrichtung des *Stacks*

- *Stack* wächst in Richtung der niedrigeren Adressen („von oben nach unten“)
- *Stack Pointer* zeigt auf zuletzt eingefügtes Byte bzw. Wort
- d.h. niedrigste gültige Adresse im *Stack*
- d.h. „oberstes“ *Stack*-Element



Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (7)

Statisch

```
void C()
{ ... } /* jr */
void B()
{       /* $31 -> stack */
  C();
}       /* stack -> $31, jr */
void A()
{       /* $31 -> stack */
  B();
}       /* stack -> $31, jr */
void main(){ A(); }
```

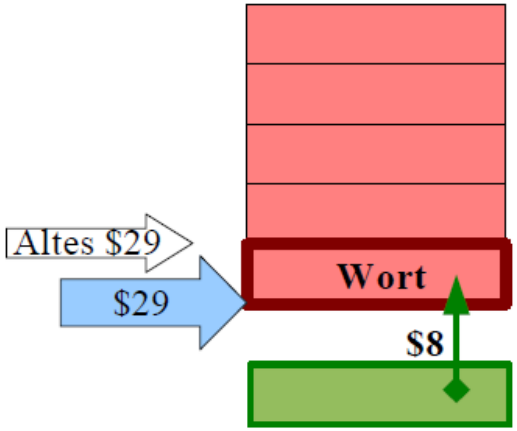
Dynamisch

Aktiv	Befehl
main	jal A
A	$31 -> *Stack*
A	jal B
B	$31 -> *Stack*
B	jal C
C	jr $31
B	Rückschreiben von $31
B	jr $31
A	Rückschreiben von $31
A	jr $31
main	

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (8)

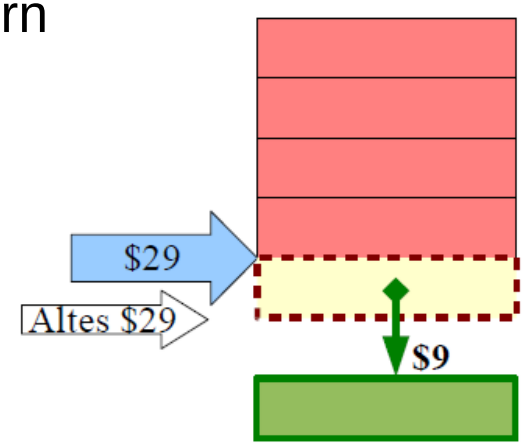
Daten zum *Stack* hinzufügen

- Indizierte Adressierung
- Dekrementieren des *Stack Pointers*
- „Oben“ auf den Keller
- Kellerüberlauf?



***Push*-Operation**

- **`$29`** um 4 dekrementieren, 1 Wort auf *Stack* einkellern
-
```
addi $29,$29,-4
sw   $8,0($29)
```



***Pop*-Operation**

- 1 Wort vom *Stack* holen, **`$29`** um 4 inkrementieren
-
```
lw   $9,0($sp)
addi $sp,$sp,4
```

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (9)

Sichern von Registerinhalten

- Einige Register sollten von jeder Prozedur genutzt werden dürfen
- Unabhängig davon, welche Prozeduren sie aufrufen und von welchen Prozeduren sie gerufen werden
- Die Registerinhalte müssen beim Prozeduraufruf gerettet und nach dem Aufruf zurückgeschrieben werden

2 Methoden

- Aufrufende Prozedur rettet vor Unterprogrammaufruf Registerinhalte *(caller save)* und kopiert sie danach zurück.
- Gerufene Prozedur rettet nach dem Unterprogrammaufruf diese Registerinhalte und kopiert sie vor dem Rücksprung zurück *(callee save)*.

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (10)

caller save

`caller:` Retten der Register auf *Stack*
`jal callee`

`callee:` Retten von **`$31`**
Befehle für Rumpf
Rückschreiben von **`$31`**
`jr $31`

`caller:` Rückschreiben der Register

callee save

`caller: jal callee`

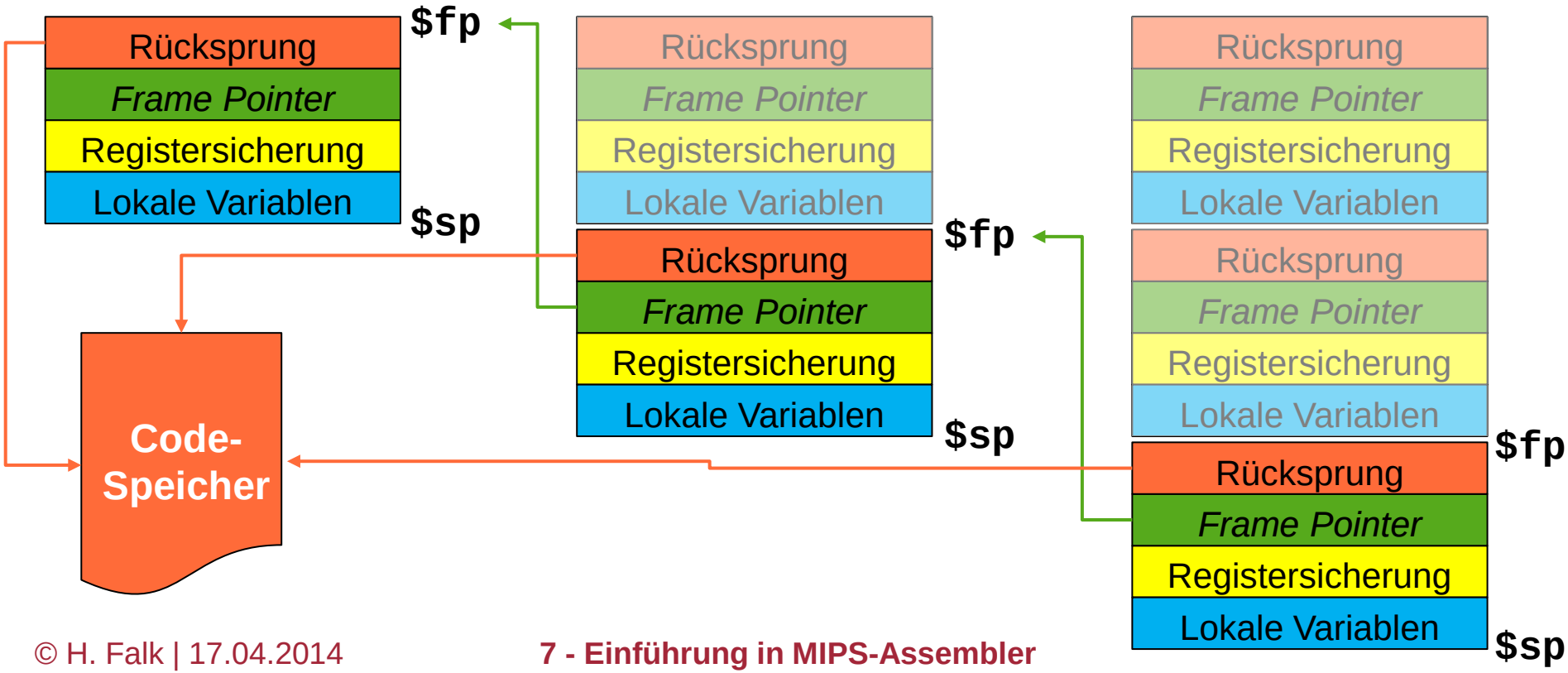
`callee:` Retten der Register auf *Stack*
Retten von **`$31`**
Befehle für Rumpf
Rückschreiben von **`$31`**
Rückschreiben der Register
`jr $31`

`caller:`

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (11)

Komplexer *Stack*-Aufbau: 3 *Stack Frames* auf dem *Stack*

- Rücksprungadresse zeigt in den Codebereich
- *Frame Pointer* **`$30`** zeigt in den *Stack*
- „mal höher, mal niedriger“



Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (12)

Prozeduren mit Parametern

```
int hour2sec( int hour )
{
  return hour * 60 * 60;
}
```

Wo findet **hour2sec** den Eingabeparameter?

```
hour2sec( 5 );
```

Konflikt

– Parameter möglichst in Registern übergeben (☞ schnell)
– Man muss eine beliebige Anzahl von Parametern erlauben

MIPS-Konvention

– Die ersten 4 Parameter einer Prozedur werden in Registern **$4**, **$5**, **$6**, **$7** übergeben, alle weiteren im *Stack*

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (13)

Benutzung der MIPS-Register

Register	Verwendung
`$0`	0
`$1`	Assembler
`$2,$3`	Funktionsergebnis
`$4-$7`	Parameter
`$8-$15`	Hilfsvariable, nicht gesichert
`$16-$23`	Hilfsvariable, gesichert
`$24-$25`	Hilfsvariable, nicht gesichert
`$26-$27`	Für Betriebssystem reserviert
`$28`	Zeiger auf globalen Bereich
`$29`	*Stack Pointer*
`$30`	*Frame Pointer*
`$31`	Rückkehradresse

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (14)

An Prozeduren Parameterwert oder dessen Adresse übergeben?

Parameterwert selbst *(call by value)*
- Gerufener Prozedur ist nur der Wert bekannt
- Die Speicherstelle, an der er gespeichert ist, kann nicht verändert werden

Adresse des Parameters *(call by reference)*
- Erlaubt Ausgabeparameter
- Änderung der Werte im Speicher durch die gerufene Prozedur möglich
- Bei großen Strukturen *(Arrays)* effizient

ANSI-C
- Bei skalaren Datentypen (int, Zeiger, usw.): *call by value*
- Bei komplexen Datentypen *(Arrays)*: *call by reference*

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (15)

Prozeduraufruf

– Parameter in Register **`$4`** bis **`$7`** legen
– Prozeduraufruf mit **`jal`**-Instruktion

Prozedur-Prolog

– *Stack Pointer* **`$29`** dekrementieren
– Rücksprungadresse aus **`$31`** auf *Stack* sichern
– Alten *Frame Pointer* **`$30`** auf *Stack* sichern
– Register sichern, die durch aufgerufene Prozedur zerstört werden

Prozedur-Epilog

– Zerstörte Arbeitsregister wiederherstellen
– Alten *Frame Pointer* **`$30`** wiederherstellen
– Rücksprungadresse in Register **`$31`** bereitstellen
– *Stack Pointer* **`$29`** inkrementieren
– Rücksprung mit **`jr`**-Instruktion

Hochsprachliche Kontrollkonstrukte: Prozeduraufrufe (16)

Konventionen zur Registersicherung

- Die aufgerufene Prozedur sichert die Register **$16** bis **$23** falls diese verändert werden
- Zurückgegebener Funktionswert liegt in Registern **$2** und **$3**
- Aktuelle Parameter liegen in Register **$4** bis **$7**

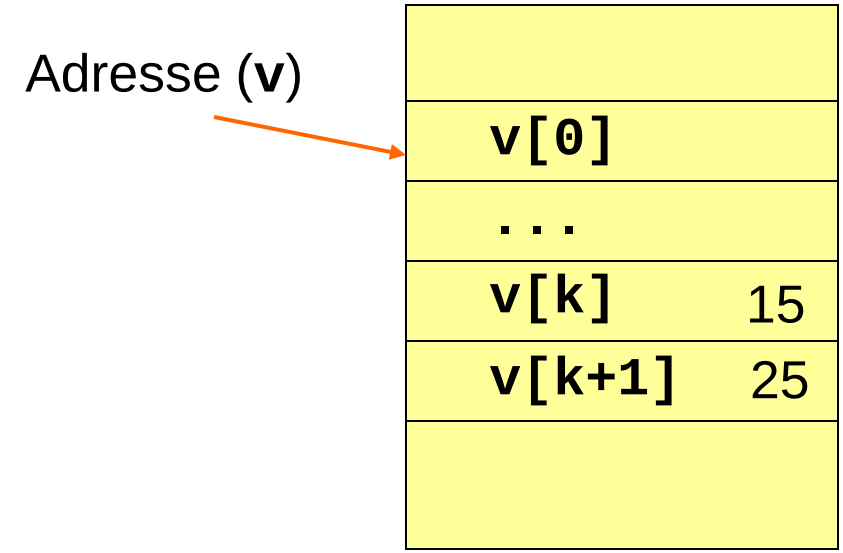
Roter Faden

7. Einführung in MIPS-Assembler

- Einführung
- Exemplarische Betrachtung der MIPS Assemblersprache
- Übersetzung von hochsprachlichen Konstrukten
  - Kontrollstrukturen
  - Datenzugriffe
- Beispiel: *bubble sort*

Prozedur swap, Prinzip der Übersetzung in Assembler

```
swap( int v[], int k )
{
  int temp;
  temp = v[k];
  v[k] = v[k + 1];
  v[k + 1] = temp;
}
```



Schritte der Übersetzung in Assembler

1. Zuordnung von Speicherzellen zu Variablen
2. Übersetzung des Prozedurrumpfs in Assembler
3. Sichern und Rückschreiben der Register

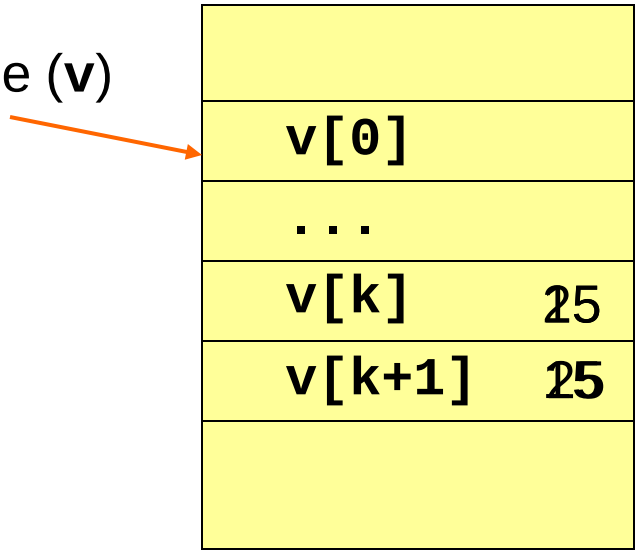
Zuordnung von Registern

Variable	Speicherzelle	Kommentar
Adresse von **v**	**$4**	Aufgrund der MIPS-Konvention für den 1. Parameter
k	**$5**	Aufgrund der MIPS-Konvention für den 2. Parameter
temp	**$15**	(irgendein freies Register)
Interne Hilfsvariable	**$16**	(irgendein freies Register)

Realisierung des Rumpfes

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

$4 = Adresse (**v**)



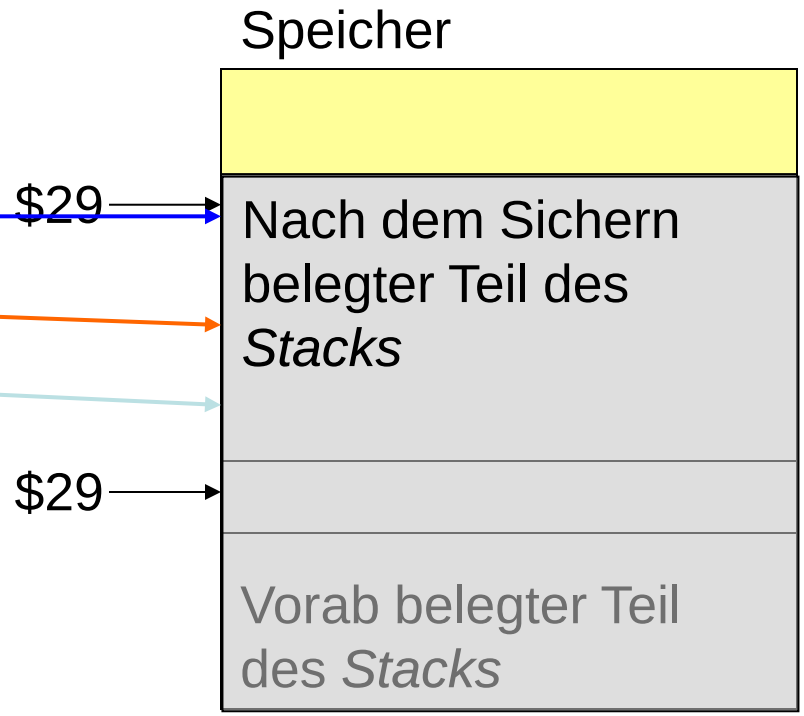

```
li  $2,4
mul $2,$2,$5      # Reg[2] := 4*k
add $2,$4,$2      # Adresse von v[k]
lw $15,0($2)      # Lade v[k]
lw $16,4($2)      # Lade v[k + 1]
sw $16,0($2)      # v[k] := v[k + 1]
sw $15,4($2)      # v[k + 1] := temp
```

Sichern der Register

Prolog: Sichern der in swap überschriebenen Register:

- Prolog
- Prozedurrumpf
- Epilog

```
addi $29,$29,-12
sw   $2,0($29)
sw   $15,4($29)
sw   $16,8($29)
```



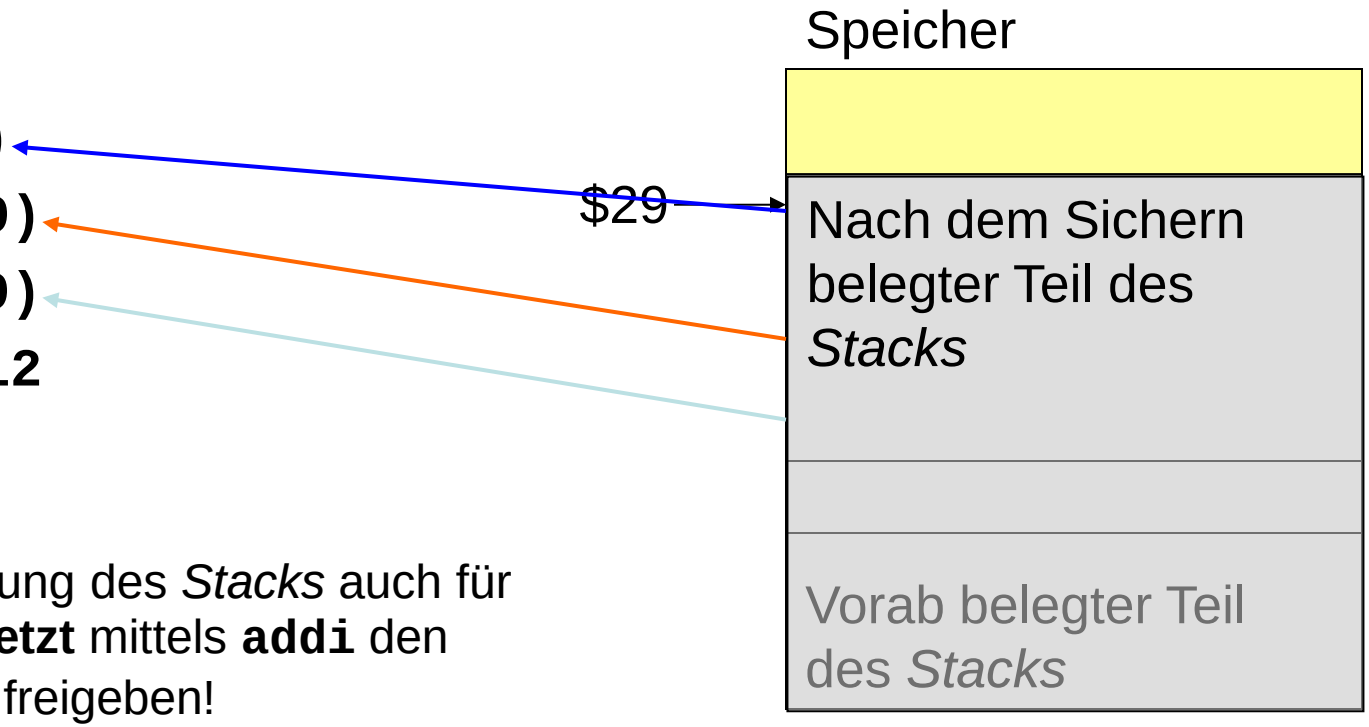
Wegen der Verwendung des *Stacks* auch für *Interrupts* immer **zuerst** mittels **`addi`** den Platz auf dem *Stack* reservieren!

Rückschreiben der Register

Epilog: Rückschreiben der in swap überschriebenen Register:

- Prolog
- Prozedurrumpf
- Epilog 

```
lw   $2,0($29)
lw   $15,4($29)
lw   $16,8($29)
addi $29,$29,12
```



Wegen der Verwendung des *Stacks* auch für *Interrupts* immer **zuletzt** mittels **`addi`** den Platz auf dem *Stack* freigeben!

Übersetzung von *bubble sort*

```
int v[10000];
sort( int v[], int n )
{
  for ( i = 0; i < n; i++ )
    for ( j = i – 1; j >= 0 && v[j] > v[j + 1]; j-- )
      swap( v, j );
}
```

Schritte der Übersetzung in Assembler

1. Zuordnung von Speicherzellen zu Variablen
2. Übersetzung des Prozedurrumpfs in Assembler
3. Sichern und Rückschreiben der Register

Zuordnung von Registern

Variable	Speicherzelle	Kommentar
Adresse von **v**	**$4**	Lt. MIPS-Konvention
n	**$5**	Lt. MIPS-Konvention
j	**$17**	irgendein freies Register
Kopie von **$4**	**$18**	Sichern der Parameter zur Vorbereitung des Aufrufs von **swap**
i	**$19**	irgendein freies Register
Kopie von **$5**	**$20**	wie **$4**
Hilfsvariablen	**$15**, **$16**, **$24**, **$25**	irgendwelche freien Register
Hilfsvariable	**$8**	nicht gesichert

Übersetzung des Rumpfes

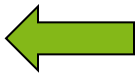
```
for(i=0;i<n;i++)
  for(j=i-1;
      j>=0 &&
      v[j]>v[j+1];j--)
    swap(v, j);
```

Adr. von **v**	**$4**
n	**$5**
j	**$17**
Kopie von **$4**	**$18**
i	**$19**
Kopie von **$5**	**$20**
Hilfsvariable	**$15**,**$16**,**$24**,**$25**
Hilfsvar., nicht gesichert	**$8**

```
        add  $18,$4,$0
        add  $20,$5,$0
        li   $19,0          # i=0
for1:   bge  $19,$20,ex1    # i>=n?
        addi $17,$19,-1     # j=i-1
for2:   slti $8,$17,0       # j<0?
        bne  $8,$0,ex2      # &&
        li   $8,4
        mul  $15,$17,$8     # j*4
        add  $16,$18,$15    # Adr(v[j])
        lw   $24,0($16)     # v[j]
        lw   $25,4($16)     # v[j+1]
        ble  $24,$25,ex2    # v[j]<=v[j+1]?
        add  $4,$18,$0      # 1. Parameter
        add  $5,$17,$0      # 2. Parameter
        jal  swap
        addi $17,$17,-1     # j--
        j    for2           # for-Ende
ex2:    addi $19,$19,1      # i++
        j    for1           # for-Ende
ex1:                        # ende
```

Sichern der Registerinhalte

Prolog: Sichern der überschriebenen Register:

- Prolog 
- Prozedurrumpf
- Epilog

```
addi $29,$29,-36
sw   $15,0($29)
sw   $16,4($29)
sw   $17,8($29)
sw   $18,12($29)
sw   $19,16($29)
sw   $20,20($29)
sw   $24,24($29)
sw   $25,28($29)
sw   $31,32($29)
```

Adr. von **v**	**$4**
n	**$5**
j	**$17**
Kopie von **$4**	**$18**
i	**$19**
Kopie von **$5**	**$20**
Hilfsvariable	**$15**,**$16**,**$24**,**$25**
Hilfsvar., nicht gesichert	**$8**

Rückschreiben der Registerinhalte

Epilog: Rückschreiben der Registerinhalte:

Prolog

Prozedurrumpf

Epilog 

Adr. von **v**	**$4**
n	**$5**
j	**$17**
Kopie von **$4**	**$18**
i	**$19**
Kopie von **$5**	**$20**
Hilfsvariable	**$15**,**$16**,**$24**,**$25**
Hilfsvar., nicht gesichert	**$8**

```
lw   $15,0($29)
lw   $16,4($29)
lw   $17,8($29)
lw   $18,12($29)
lw   $19,16($29)
lw   $20,20($29)
lw   $24,24($29)
lw   $25,28($29)
lw   $31,32($29)
addi $29,$29,36
```

Zusammenfassung (1)

Schichtenmodell

- Programmiersprache
- Assemblersprache
- Maschinenprogramme
- Register-Transfer Strukturen
- Funktion des Assemblers

Exemplarische Betrachtung der MIPS Assemblersprache

- **add**, **lw**, **sw**-Befehle
- Register-Transfer Semantik
- Darstellung von Maschinenbefehlen
- Sequentielle Befehlsbearbeitung
- Laden von Konstanten
- Vorzeichenerweiterung

Zusammenfassung (2)

Exemplarische Betrachtung der MIPS Assemblersprache

– Weitere arithmetische/logische Befehle
– Übliche Speichereinteilung
– Sprungbefehle

Übersetzung von hochsprachlichen Konstrukten

– Fallunterscheidungen *(if-then, if-then-else)*
– *Array*-Zugriffe („*Base* + *Offset*“-Adressierung)
– Schleifen
– *switch-case* Ausdrücke (Sprungtabellen)
– Prozeduraufrufe (*Stack*-Prinzip, verschachtelte Prozeduren, Register-Konventionen, Übergabe- und Sicherungskonventionen für Parameter)

Beispiel *bubble sort*