3. Befehlssätze

- **Maschinensprache**
  - von der Struktur her einfacher als high-level-Sprachen
    - z.B. kein anspruchsvoller Kontroll-Fluss (wie while, for, try and catch)
  - auf die Hardware der Maschine bezogen
    - keine Variablen, nur Register und Hauptspeicheradressen
    - nur einfache logische und arithmetische Operationen (z.B. kein tanh)
- **MIPS Instruction Set Architecture**
  - ursprünglich Abkürzung für **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
    - HW fehlte, um Pipeline anzuhalten, Compiler war verantwortlich, s.u.
  - benutzt von NEC, Nintendo, Silicon Graphics, Sony
  - ähnlich zu anderen Architekturen, die seit 1980 entwickelt wurden
    - sehr ähnlich zu ARM, RISC-V

MIPS ISA

- **Design Ziele**
  - maximale Performance
  - minimale Kosten
    - auch durch geringe Entwicklungszeit
- **MIPS Register-Satz**
  - 32 Register
    - 32 bit breit
    - Bezeichnung
      - $0 … $31
    - alternativ Namen, um Benutzungskonventionen darzustellen
      - z.B. $zero oder $ra oder $s0
      - Zuordnung siehe Tabelle nächste Folie

Register-Konventionen

– Konventionen zur Benutzung der Register

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

– Register $1
  - `$at`: reserviert für Assembler (s. u.)

– Register $26 - $27
  - `$k0-$k1`: reserviert für Betriebssystem

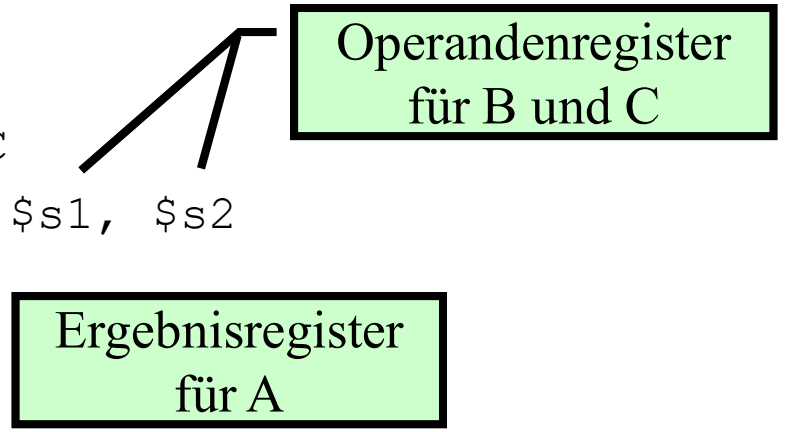
MIPS Arithmetik

- alle Operationen haben 3 Operanden
- die Operanden liegen in Registern
- Reihenfolge der Operanden ist fest (Ergebnis der Operation zuerst)

- Beispiel:

  C code: `A = B + C`

  MIPS code: `add $s0, $s1, $s2`



- Zuordnung der Variablen zu Registern
  - per Hand bei Assembler-Programmierung
  - automatisch durch Compiler

MIPS Arithmetik (2)

- **„Einfachheit durch Regularität“**
  - Regularität macht die Implementierung einfacher.
    - Alle Operationen haben dieselbe Struktur.
    - Alle Operationen haben genau drei Operanden (einschl. Ergebnis).
  - Das macht einige Dinge aber auch komplizierter.

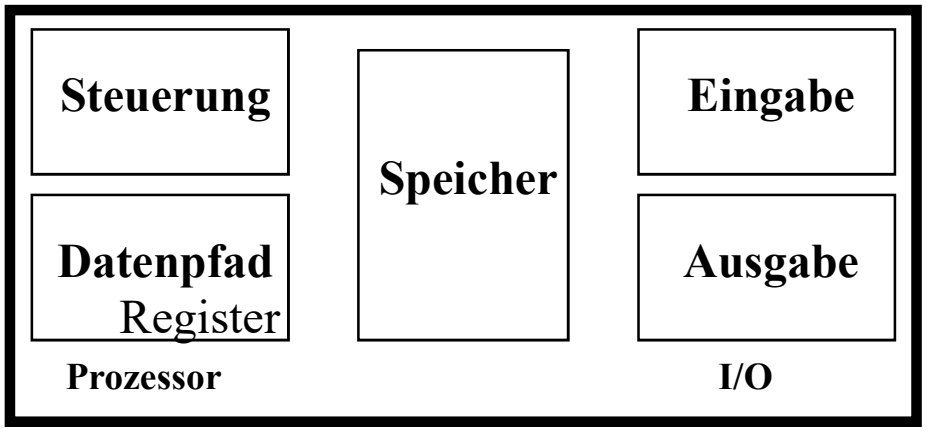
    C code:
    ```
    A = B + C + D;
    E = F - A;
    ```
    MIPS code:
    ```
    add $t0, $s1, $s2
    add $s0, $t0, $s3
    sub $s4, $s5, $s0
    ```
  - Argumente müssen in Registern liegen
    - Register für Zwischenergebnisse werden gebraucht
    - hier das `$t0`-Register (t für temporär)
  - Dafür sind in MIPS immerhin 32 Register vorhanden.

Weitere Designprinzipien

- **„Kleiner ist schneller“**
  - Größere Registersätze würden größere Verzögerungen durch längere Leitungen verursachen.
- **„Gutes Design erfordert gute Kompromisse“**
  - Das Optimum liegt oft irgendwo in der Mitte.

Register vs. Speicher

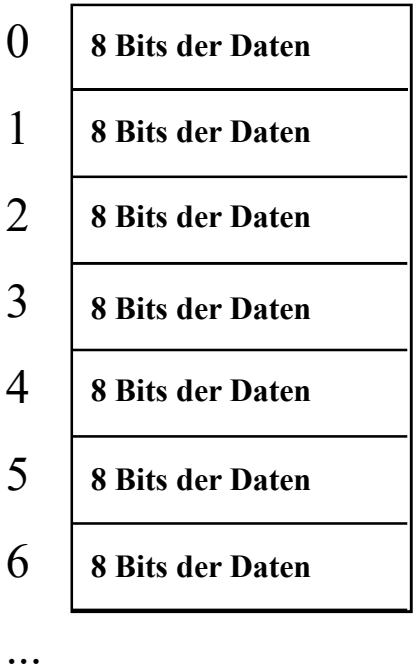
- Variablen liegen in Registern.
  - 32 Register
- Was ist mit Programmen mit einer großen Anzahl von Variablen?
  - Variablen müssen dann im Speicher liegen
  - werden nur bei Bedarf in die Register geladen
  - Verdrängungsstrategien
    - siehe Compilerbau



von Neumann Architektur

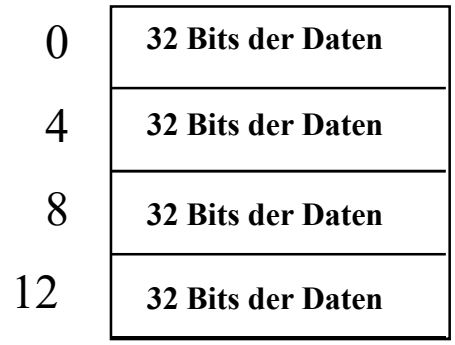
Speicher Organisation

- **Speicher**
  - Sieht wie ein großes eindimensionales Array aus.
  - Die Speicher-Adresse ist der Index für das Array.
  - "Byte Adressierung"
    - Index verweist auf einzelne Bytes des Speichers.



Speicher Organisation (2)

- Bytes sind zwar wichtig, doch werden für die meisten Datentypen größere "Worte" benutzt.
- Bei MIPS besteht ein Wort aus 32 Bits (4 Bytes).



Register speichern auch jeweils 32 Bits

  - Adressen auch 32 Bits: 2^{32} Bytes mit Byte-Adressen von 0 bis 2^{32}-1
  - 2^{30} Worte mit Byte-Adressen 0, 4, 8, ... 2^{32}-4
- Worte sind immer ausgerichtet (*aligned*).
  - Byte-Adressen von Worten sind immer ein Vielfaches von 4.
  - Was sind also die untersten 2 Bits einer Wort-Adresse?
  - Zugriff auf Worte im Speicher wird dadurch vereinfacht.

Instruktionen

- **load und store Instruktionen**
  - Beispiel: C code: `A[8] = h + A[8];`

    MIPS code:
    ```
    lw  $t0, 32($s3)
    add $t0, $s2, $t0
    sw  $t0, 32($s3)
    ```
  - Adresse im Hauptspeicher ist die Summe aus dem Wert in einem Register (hier `$s3`) + fixem Offset (hier: `32`)
  - store-word (`sw`) hat das Ziel der Operation auf der rechten Seite
    - widerspricht dem Prinzip, dass die Ergebnisse der Operation immer auf der linken Seite stehen sollten,
      - aber dadurch dieselbe Struktur wie `lw`-Instruktion
      - manchmal muss man halt Kompromisse eingehen
    - vereinfacht die Implementierung in der Hardware
  - Erinnerung
    - arithmetische Operanden liegen immer in Registern, nie im Speicher!

Zusammenfassung

- **MIPS**
  - lädt Worte, aber adressiert Bytes
  - Arithmetik nur mit Registern

Instruktion	Bedeutung
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Maschinen Sprache

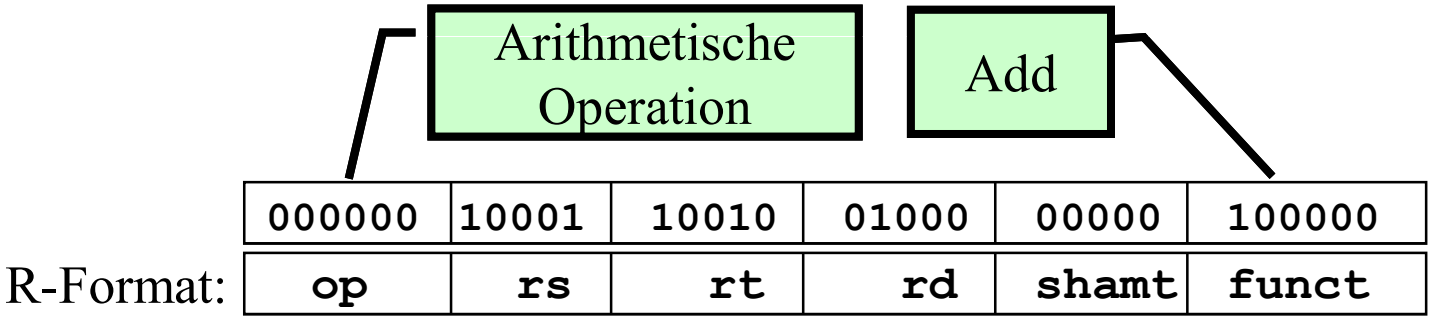
- Instruktionen sind genau wie Register und Daten-Worte 32 bit lang
  - Beispiel

    ```
    add $t0, $s1, $s2
    ```

  - Register sind in Wahrheit nummeriert (0 … 31)

    ```
    $t0=8, $s1=17, $s2=18
    ```

- Instruktions-Format:



- **Welche Bedeutung haben die Feldnamen?**

Maschinen Sprache (2)

- **Feldnamen der Instruktionen**
  - `op` (6 bit): Opcode, kodiert die grundlegende Instruktion
  - `rs` (5 bit): erstes Register, 1. *source* Operand
  - `rt` (5 bit): zweites Register, 2. *source* Operand
  - `rd` (5 bit): Ergebnis-Register, *destination*
  - `shamt` (5 bit): *shift amount*, s.u. shift Operationen (enthält Nullen, falls nicht geshifted werden soll)
  - `funct` (6 bit): *function*, wählt eine spezielle Variante der Operation, die durch den Opcode bestimmt wurde (*function code*)

Machinen Sprache (3)

- **Betrachte die load-word und store-word Instruktionen**
  - Was würde uns das Regularitäts-Prinzip auferlegen?
- **„Gutes Design erfordert gute Kompromisse“**
  - Führe ein neues Instruktions-Format ein
    - I-Format für Daten Transfer Instruktionen (I wie *Immediate*, s.u.)
    - das schon bekannte Format heißt R-Format für Registeroperationen (R wie Register)
- **Beispiel:** `lw $t0, 32($s2)`

	35	**18**	**8**	**32**
I-Format:	**op**	**rs**	**rt**	**16 bit Zahl**

- **Wo ist der Kompromiss?**

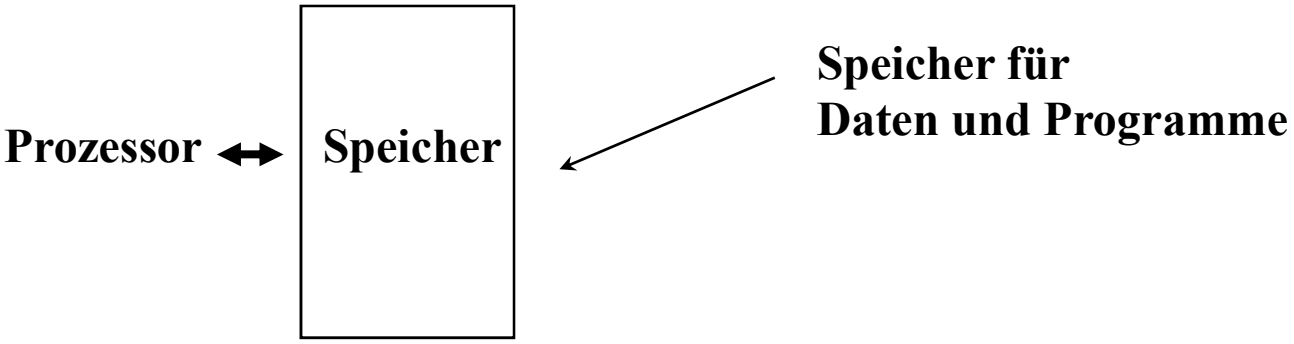
Logische Operationen

- Wir brauchen eine Möglichkeit, auf Bits in einem Wort zugreifen zu können.
- Dazu kann man logische Operationen benutzen.
  - Diese wirken paarweise auf alle 32 bit in Registern.
  - Logische Schiebeoperationen `sll` und `srl` schieben den Bitvektor um `shamt` bits in die gegebene Richtung (0 bis 31).
    - Anzahl der Bits: shamt-Feld im R-Format

Instruktion	Bedeutung
`and $s1, $s2, $s3`	`$s1 = $s2 & $s3`
`or  $s1, $s2, $s3`	`$s1 = $s2 | $s3`
`nor $s1, $s2, $s3`	`$s1 = ~($s2 | $s3)`
`xor $s1, $s2, $s3`	`$s1 = $s2 ^ $s3`
`sll $s1, $s2, 21`	`$s1 = $s2 << 21`
`srl $s1, $s2, 21`	`$s1 = $s2 >> 21`

Gespeichertes Programm

- **Instruktionen sind Bits**
- **Programme werden im Speicher abgelegt**
  - sie werden wie Daten gespeichert und gelesen



Fetch & Execute Cycle

- **Grundlegender Instruktionsverarbeitungszyklus**
  - PC (*program counter*, besser wäre *instruction address register*) enthält die Adresse der nächsten auszuführenden Instruktion
  - Instruktion wird in ein spezielles Register (IR, *instruction register*) geladen (*fetch*)
  - der PC wird um 4 erhöht, um auf die nachfolgende Instruktion im Speicher zu zeigen
  - die Bits im IR steuern (*control*) die nachfolgenden Aktionen (*execute*), also die Ausführung der Instruktion (dabei kann auch der Inhalt des PC verändert werden)
  - danach wird die nächste Instruktion von der Adresse aus dem PC geholt, und so weiter

Kontroll-Fluss: Branches

- **Instruktionen, die Entscheidungen fällen**
  - ändern Kontroll-Fluss, d.h. sie ändern die Adresse von der die nächste Instruktion geholt werden soll
    - indem die neue Adresse in den PC geschrieben wird
- **MIPS conditional branch instructions**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

} passt ins I-Format (s.u.)

Beispiel:

```
if (i == j) h = i + j;
```

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

Kontroll-Fluss: Branches (2)

- **Kein neues Instruktionsformat notwendig**
  - I-Format bietet Platz für zwei Register und einen 16 bit Wert
  - Der 16 bit Wert wird als Offset *x* interpretiert (Zweierkomplement).
    - Springe um *x* Befehle weiter, wenn die Verzweigung ausgeführt werden soll.
    - *x* kann auch negativ sein (Sprünge zurück).
      - *x* = 0 bedeutet: springe zum nachfolgenden Befehl (kein Sprung)
      - *x* = 1 bedeutet: überspringe den nächsten Befehl
      - *x* = -1 bedeutet: springe zurück zum Sprungbefehl selbst
      - etc.
  - Beispiel:

```
bne $s0, $s1, Label
```

	5	**16**	**17**	**1**
I-Format:	**op**	**rs**	**rt**	**16 bit Zahl**

Kontroll-Fluss: Jump

- **MIPS unconditional jump instruction**

```
j  label
```

- **Beispiel:**

```
if (i!=j)                      beq $s4, $s5, Lab1
    h=i+j;                     add $s3, $s4, $s5
else                           j Lab2
    h=i-j;             Lab1:   sub $s3, $s4, $s5
                       Lab2:   ...
```

- **Instruktion braucht keine Register**
  - daher mehr Platz für Zieladresse
  - neues Instruktionsformat sinnvoll

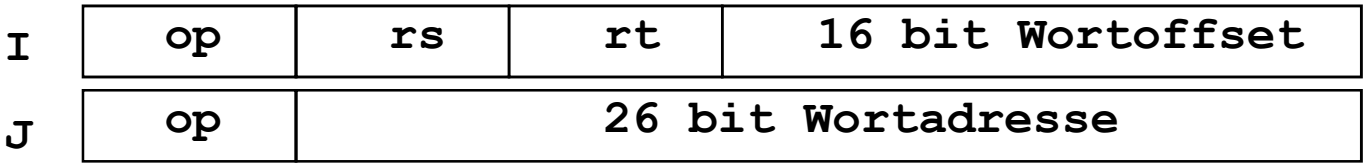
J-Format:

Adressen in Branches und Jumps

- **Instruktionen**

`bne $t4,$t5,Label`	Nächste Instr. ist am Label, falls `$t4≠$t5`
`beq $t4,$t5,Label`	Nächste Instr. ist am Label, falls `$t4=$t5`
`j Label`	Nächste Instruktion ist am Label

- **Formate**



Adressen sind also *nicht* 32 bit

- Wie erhalten wir die komplette 32 bit Adresse?
- beachte: Instruktionen starten immer an Wortgrenzen
- benutze Wortadressen als Sprungadressen

Adressen in Branches

- **Instruktionen**

  `bne $t4,$t5,Label` Nächste Instruktion ist am Label, falls $t4≠$t5
  `beq $t4,$t5,Label` Nächste Instruktion ist am Label, falls $t4=$t5

- **Format**

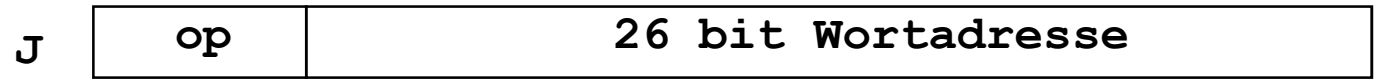
- **Lösung**
  - Die meisten Verzweigungen (*branches*) sind lokal, d.h. die Sprungweiten sind relativ klein (Prinzip der Lokalität).
  - Wir können ein Register benutzen (wie bei `lw` und `sw`), zu dem der Offset hinzuaddiert wird, um zur Zieladresse zu gelangen.
    - Instruction Address Register (PC = program counter)
    - Zu dem Zeitpunkt zeigte der PC bereits auf den nachfolgenden Befehl
    - Zieladresse = PC + 4*(16 bit Wortoffset)

Adressen in jumps

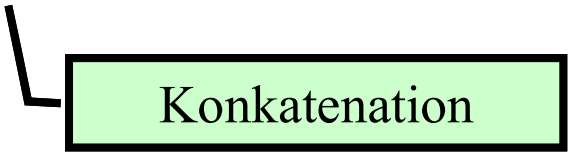
- **Instruktion**

  `j Label` Nächste Instruktion ist am Label

- **Format**



- **Lösung**
  - Es müssen manchmal auch große Sprünge durchgeführt werden.
  - Jump Instruktionen benutzen einfach die 4 höchstwertigen Bits des PC.
    - Adressgrenzen von 256 MB (= 2^{28} Bytes) können mit jump nicht übersprungen werden.
    - Zieladresse = PC(31..28) : 4 * Wortadresse



Zusammenfassung: Instrukionsformate

Instruktion	Bedeutung	
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	R-Format
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	
`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	
`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	
`sla $s1,$s2,21`	`$s1 = $s2 << 21`	
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`	I-Format
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`	
`bne $s4,$s5,L`	Nächste Instr. am Label L falls $s4 ≠ $s5	
`beq $s4,$s5,L`	Nächste Instr. am Label L falls $s4 = $s5	
`j L`	Nächste Instr. am Label L	J-Format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Kontroll-Fluss

- wir haben: beq, bne, was ist mit Branch-on-less-than?
  - wir werden sehen, dass dieser Befehl den Rechner ausbremsen würde
  - stattdessen andere Lösung:
- neue Instruktion

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
```

- kann benutzt werden, um

```
blt $s1, $s2, Label
```

  zu konstruieren
- wir können nun allgemeine Kontroll-Strukturen (Schleifen, Sprünge) realisieren (etwas fehlt aber noch, was?)
- beachte: der Assembler benötigt dafür ein Zwischenregister
  - es gibt Konventionen für die Benutzung der Register

Konstanten

- kleine Konstanten werden sehr häufig benötigt (ca. 50% der Operanden in typischen Programmen)

  z. B. A = A + 4; B = 3; C = C-1;

- mögliche Lösungen (wieso sollte man es nicht tun?)
  - lege 'typische Konstanten' im Speicher ab und lade sie
  - konstruiere fest-verdrahtete Register (z.B. $one) für Konstanten (wie $zero für 0)
- unsere Lösung: MIPS Instruktionen mit immediate Adressierung

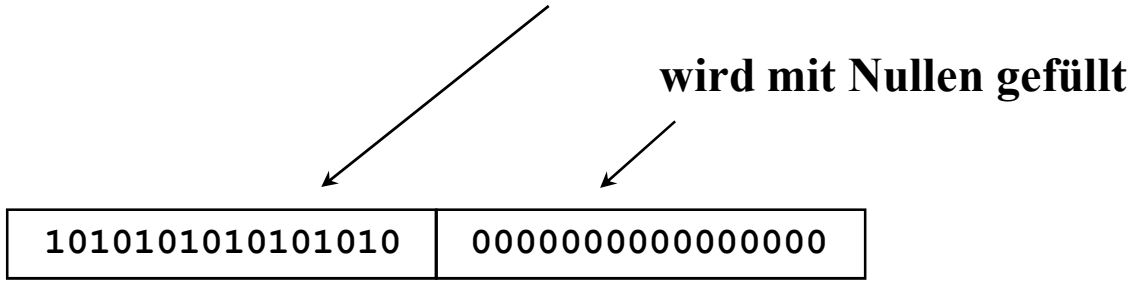
```
addi $29, $29,  4
slti  $8, $18, 10
andi $29, $29,  6
ori  $29, $29,  4
nori $29, $29, 42
xori $29, $29, 21
```

passt ins I-Format
(16 bit Konstanten)

Größere Konstanten

- wir möchten auch 32 bit Konstanten in ein Register laden können
- wir benötigen zwei Instruktionen
  - neue Instruktion: "load upper immediate"

```
lui $t0, 0xAAAA
```



  - 0x als Präfix bedeutet: hexadezimale Zahl
  - default ist: Dezimalzahl

Größere Konstanten (2)

- dann müssen die unteren 16 Bit dazu geladen werden
  - z.B. mit or immediate:

```
ori $t0, $t0, 0xBBBB
```

	1010101010101010	0000000000000000
ori	0000000000000000	1011101110111011
	1010101010101010	1011101110111011

Assembler vs. Maschinen-Sprache

- Assembler stellt bequeme symbolische Repräsentation zur Verfügung
  - viel einfacher, als Zahlen aufzuschreiben
  - Label statt Speicheradressen
  - einfach zu merkende Regeln
- Maschinen Sprache ist die zugrunde liegende Realität
  - also die Bits, die der Prozessor tatsächlich versteht
- Assembler stellt 'Pseudoinstruktionen' zur Verfügung
  - Einzelne Assembler-Instruktionen werden manchmal in mehrere Maschinen-Instruktionen umgesetzt (s.u.).
- Wenn die Performance ermittelt wird, muss man die echten Maschinen-Instruktionen berücksichtigen.

Pseudoinstruktionen

- sehen nur so aus wie Maschinen-Instruktionen (die durch Hardware implementiert sind)
- werden durch eine oder mehrere Maschinen-Instruktionen ersetzt
  - Dafür wird dann manchmal das reservierte Register `$at` benötigt.
- Beispiele
  - schiebe Inhalt von einem Register in ein anderes:

```
move $t0, $t1        →        add $t0,$t1,$zero
```

  - branch on less than:

```
blt $t1, $t2, L      →        slt $at, $t1, $t2
                              bne $at, $zero, L
```

  - branch on not equal für weit entfernte Adresse:

```
bne $t1, $t2, LFar →          beq $t1, $t2, LNear
                              j LFar
                       LNear: …
```

Pseudoinstruktionen (2)

- Weitere Beispiele
  - load immediate

```
li $t1, imm   →  ori $t1, $zero, imm
```

  - load address (die Adresse selbst, nicht den Inhalt der Adresse)

```
la $t1, addr  →  lui $t1, highhalf(addr)
                 ori $t1, $t1, lowhalf(addr)
```

  - Belege den Speicher mit Daten (der Loader wird beim Laden des Programms dazu veranlasst, die Daten in ein eigenes Datensegment des Speichers zu laden).

```
        .data
addr:   .asciiz "\nNUL terminated String\n"
```

Die Startadresse des Strings ist dann das Label `addr`

Unterprogramme

- **Prozeduren und Funktionen**
  - dienen dazu Programme zu strukturieren
  - führen abstrakte Aufgaben aus
  - dürfen keine Annahmen über das aufrufende Programm machen
  - dürfen keine Spuren hinterlassen
- **durchzuführende Schritte**
  - lege Parameter an einer Stelle ab, an die das Unterprogramm herankommt
  - übertrage die Kontrolle an das Unterprogramm
  - sorge für die notwendigen Speicherressourcen (Zwischenergebnisse)
  - führe die gewünschte Aufgabe durch
  - lege die Ergebnisse an eine Stelle ab, an die das aufrufende Programm herankommt
  - gib die Speicherressourcen wieder frei
  - übertrage die Kontrolle an die Stelle, von der aus das Unterprogramm aufgerufen wurde

Unterprogramme (2)

- **Register**
  - schnellste Möglichkeit, Daten zu speichern
  - daher Parameter- und Ergebnisübergabe möglichst in Registern
  - Konventionen
    - `$a0-$a3`: Register für vier Parameter (`a` wie ***a****rgument*)
    - `$v0-$v1`: Register für 2 Ergebniswerte (`v` wie ***v****alue*)
    - `$ra`: Register zur Speicherung der Rücksprungadresse
- **Unterprogrammaufruf**
  - *jump and link* Instruktion: `jal Address`
    - verzweigt zu der angegebenen Adresse (*jump*)
    - die Adresse des nächsten Befehls (PC+4) wird in das Register `$ra` geschrieben, um den Rücksprung zu ermöglichen (*link*)
- **Rücksprung ins aufrufende Programm**
  - *jump register* Instruktion: `jr $ra`
    - verzweigt an die Adresse, die in dem angegebenen Register steht
    - dient auch zur Realisierung von switch-statements

Unterprogramme (3)

- **Stack**
  - Werden mehr Register für die Argumentübergabe benötigt als dafür vorhanden, müssen Registerinhalte in den Speicher verschoben werden.
  - Ideale Datenstruktur: Stack (Operationen: *push* und *pop*)
  - Konventionen
    - `$sp` dient als Stack-Pointer
      - zeigt auf TOS (*top of stack*)
      - Stack wächst von höheren zu niedrigeren Adressen (per Konvention)
    - Register `$s0-$s7` dürfen vom Unterprogramm nicht verändert werden
      - falls das Unterprogramm diese Register benötigt, müssen die Inhalte auf dem Stack zwischengespeichert werden (`s` wie ***s**aved*)
    - Register `$t0-$t9` brauchen durch ein Unterprogramm nicht gesichert zu werden (`t` wie ***t**emporary*)
      - das aufrufende Programm speichert dort keine Daten, die nach einem Unterprogrammaufruf noch benötigt werden
    - diese Konventionen sparen viele *push*- und *pop*-Instruktionen

Unterprogramme (4)

- **Verschachtelte Unterprogramme**
  - Unterprogramme rufen wieder Unterprogramme auf.
    - Konflikte mit Argument-, Rückgabe- und Rücksprungadress-Registern
  - Aufrufendes Programm pusht Argumentregister, temporäre Reg. die nach dem Aufruf noch benötigt werden, und seine eigene Rücksprungadresse auf den Stack.
  - Das aufgerufene Programm pusht alten Frame-Pointer (s.u.), alle sicheren Register, die es selbst benötigt.

Unterprogramme (5)

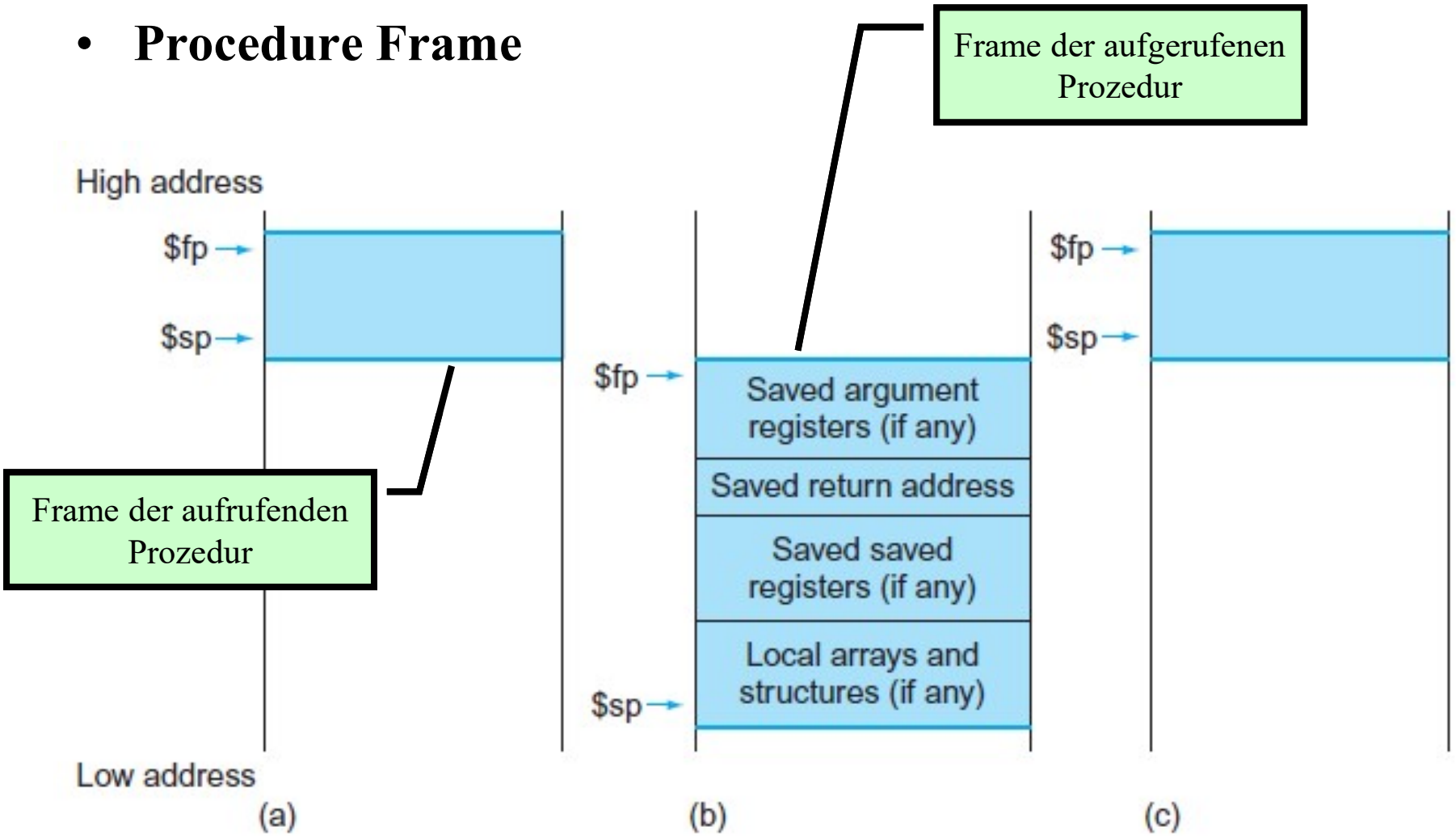
- **Lokale Variablen**
  - Unterprogramme benötigen Platz für lokale Variablen.
  - Dafür kann auch der Stack benutzt werden.
  - Das ganze nennt man dann auch einen Unterprogramm-Rahmen (*procedure frame*).
  - Da der Stack-Pointer dauernd woanders hinzeigt, ist es einfacher, einen fixen Frame-Pointer (`$fp`) zu verwenden, um auf die lokalen Variablen zuzugreifen.
    - Ansonsten wäre der Assembler-Code kaum noch zu verstehen.

Unterprogramme (6)

- **Procedure Frame**
  - auf dem Stack liegen also
    - vom aufrufenden Programm gespeichert
      - gesicherte eigene Argumente
      - gesicherte eigene Rücksprungadresse
      - gesicherte temporäre Register ($t), falls notwendig
      - weitere Argumente (falls Unterprogramm mehr als 4 Argumente benötigt)
    - vom aufgerufenen Unterprogramm gespeichert (Procedure Frame)
      - gesicherter vorheriger Frame-Pointer
      - gesicherte sichere Register ($s), falls notwendig
      - lokale Variablen, die nicht in temporäre Register passen
        - » Arrays
        - » Strukturen

Unterprogramme (7)

- **Procedure Frame**



Systemaufrufe

- System call code muss sich in $v0 befinden
- `syscall` führt den System Call dann durch.

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Weitere Befehle

- **load byte (`lb`) und store byte (`sb`)**
  - Wie `lw` und `sw`, nur wird kein ganzes Wort transferiert, sondern nur das Byte an der gegebenen Adresse.
  - Bei `lb` werden die höherwertigen Bits mit dem Vorzeichen aufgefüllt (Zweierkomplementzahlen, *sign extend*).

Instruktion	Bedeutung
`lb $s1,100($s2)`	`$s1 = Memory[$s2+100]` (nur ein Byte, signed)
`sb $s1,100($s2)`	`Memory[$s2+100] = $s1`

  - Bei `lbu` werden die höherwertigen Bits mit 0 aufgefüllt.
  - Entsprechend gibt es noch
    - `lh`, `sh`, `lhu`, `shu`: load/store half word signed/unsigned (also 16 bit)

Zusammenfassung Befehlssatz

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

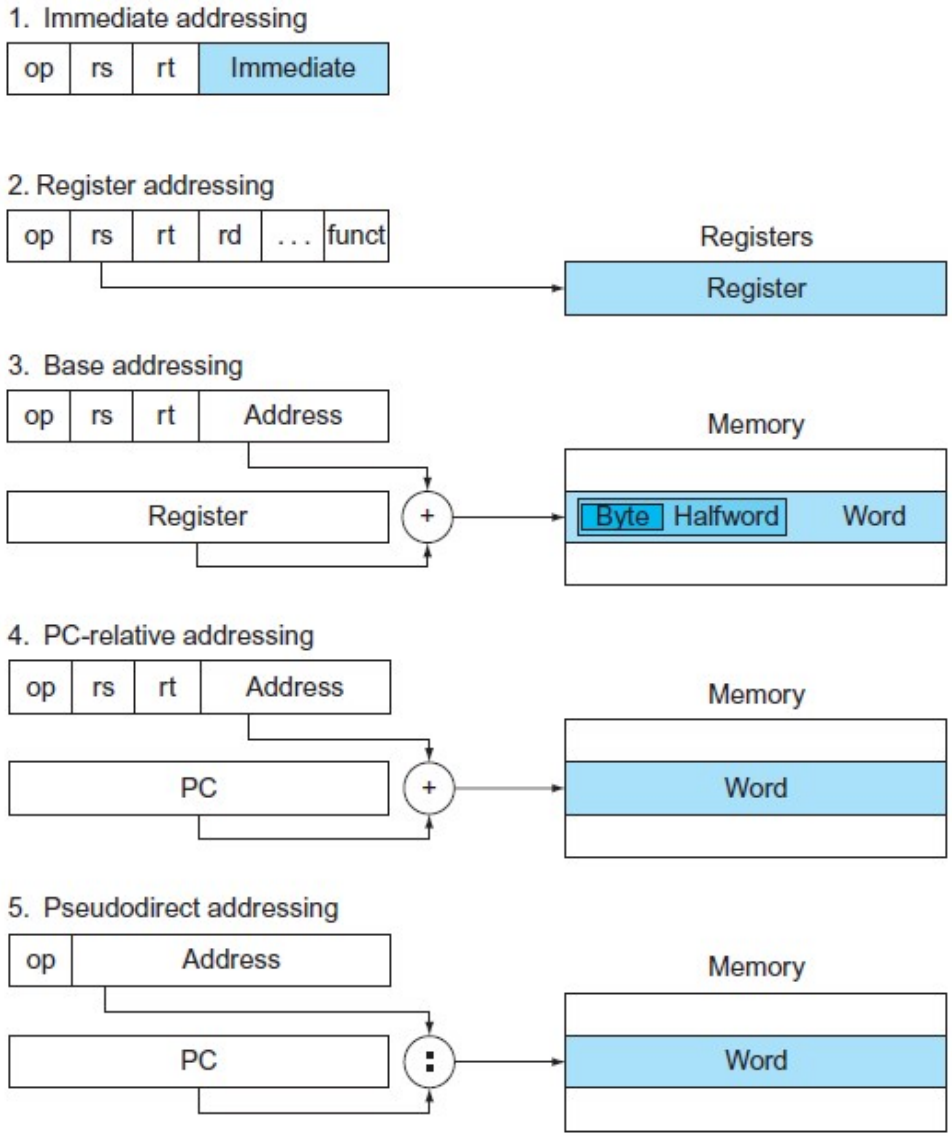
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Word from memory to register
	store word	`sw $s1,20($s2)`	`Memory[$s2 + 20] = $s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Halfword memory to register
	store half	`sh $s1,20($s2)`	`Memory[$s2 + 20] = $s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Byte from memory to register
	store byte	`sb $s1,20($s2)`	`Memory[$s2 + 20] = $s1`	Byte from register to memory
	load upper immed.	`lui $s1,20`	`$s1 = 20 * 2^16`	Loads constant in upper 16 bits

Zusammenfassung Befehlssatz (2)

Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional 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

Adressierungsarten bei MIPS



Andere Themen

- Dinge, die wir hier nicht im Einzelnen besprochen haben
  - Linker, Loader, Speicher-Layout
  - String-Manipulationen und Zeiger
  - Interrupts und Exceptions
  - Floating point Operationen
- Einige der Themen werden später noch behandelt.
- Wir haben uns auf Architektur-Themen konzentriert.
  - Grundlagen der MIPS Assembler Sprache und Maschinen Code
  - Wir werden einen Prozessor bauen, der diese Instruktionen ausführt.

Alternative Architekturen

- MIPS
  - ist ein RISC Prozessor (wenige und reguläre Instruktionen, aber sehr schnell, s.u.).
- Design Alternative
  - CISC Prozessor: mächtigere Instruktionen
  - Anzahl der Instruktionen für ein gegebenes Programm ist dann geringer.
  - Gefahr ist eine höhere Zykluszeit und/oder eine höhere CPI
- “RISC vs. CISC”
  - Praktisch alle neuen ISA's seit 1982 sind RISC-Architekturen.
  - VAX (typischer CISC Prozessor)
    - sehr mächtige Instruktionen: von 1 bis 54 Bytes Länge!
  - Pentium (und Nachfolger)
    - einziger CISC Prozessor, der sich am Markt hält
    - besitzt RISC Kern (interne HW zur Übersetzung in RISC Befehle)
    - Aufwand lohnt sich nur wegen der großen Stückzahlen
      - » und weil so viel Software existiert, die die Kunden unverändert weiter benutzen möchten

PowerPC

- **Indizierte Adressierung**
  - Beispiel

    ```
    lw $t1,$a0+$s3  # $t1=Memory[$a0+$s3]
    ```

  - Was müssen wir beim MIPS dafür tun?
- **Update Adressierung**
  - aktualisiere ein Register als Teil der Ladeoperation (um durch Arrays zu wandern)
  - Beispiel

    ```
    lwu $t0,4($s3) # $t0=Memory[$s3+4]; $s3=$s3+4
    ```

  - Was müssen wir beim MIPS dafür tun?
- **Andere Instruktionen**
  - load multiple/store multiple
    - mehrere Register mit einer Instruktion laden oder speichern
  - ein spezielles Zähler-Register für Schleifen

    "`bc Loop`"

    decrement counter, if not 0 goto loop

80x86

1978:	Intel 8086 angekündigt, 16 bit Architektur
1980:	8087 floating point coprocessor
1982:	80286 vergrößert Adressraum auf 24 bit, neue Instruktionen
1985:	80386 Erweiterung auf 32 bit, neue Adressierungsarten
1989-1995:	80486, Pentium, Pentium Pro integrierte FPU, Cache auf dem Chip, ein paar neue Instruktionen (zumeist, um höhere Performance zu erzielen)
1997:	Pentium II, MMX*) hinzugefügt (Integer-SIMD)
1999:	Pentium III, SSE**) hinzugefügt (fp-SIMD)
2001:	Pentium IV, Hyper-Threading, Hyper-Pipeline, SSE2
2005:	Pentium D, zwei Pentium IV Kerne, kein Hyper-Threading
2006:	Core2 Duo, neue interne Architektur (Core2), zwei Prozessoren in einem Gehäuse, SSSE3***), höhere Leistung bei niedrigerem Energieverbrauch

*) Multi-Media Extensions **) Streaming-SIMD-Extensions ***) supplemental SSE

Eine gewachsene Architektur: 80x86

- **Komplexität**
  - Instruktionen von 1 bis 17 Bytes Länge
  - ein Operand ist sowohl Quelle als auch Ziel
  - ein Operand kann aus dem Speicher kommen
  - komplexe Adressierungsarten, z.B.

    "base or scaled index with 8 or 32 bit displacement"
- **Kompatibilität**
  - Neuere Prozessorgeneration sollte immer die Programme der früheren Generationen verarbeiten können.
  - Dadurch entstand im Laufe der Zeit eine sehr komplexe Architektur.
- **Fairerweise sollte man sagen:**
  - Die am häufigsten benutzten Instruktionen sind nicht zu schwierig zu bauen und werden schnell ausgeführt.
  - Compiler vermeiden die Teile der ISA, die langsam sind.

Zusammenfassung

- **Instruction Set Architecture (ISA)**
  - eine sehr wichtige Abstraktion!
- **Instruktions-Komplexität ist eine Schraube, an der man beim Design drehen kann**
  - geringere Anzahl von Instruktionen
    vs.
    höhere CPI / niedrigere Taktrate
- **Design Prinzipien**
  - Einfachheit durch Regularität
  - Kleiner ist schneller
  - Gutes Design erfordert gute Kompromisse
  - Mache den Normalfall schnell