Rechnerarchitekturen 1*

Einführung

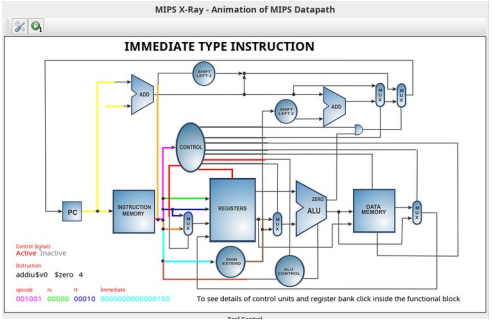
Prof. Dr. Alexander Auch

*Teilweise entnommen aus “Mikrocomputercomputertechnik 1” von Prof.Dr-Ing. Ralf Stiehler und Rechnerarchitekturen von Dr. Leonhard Stiegler

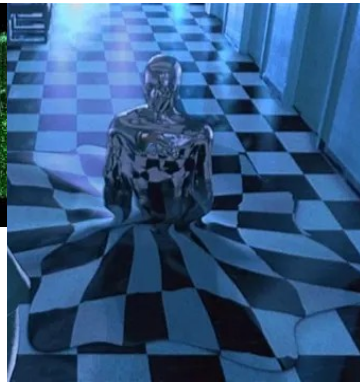
Ziele der Veranstaltung

- **Rechnerentwurf**:
  - Prozessor, Speicher, Ein-/Ausgabe
  - Entwurfs- und Optimierungsmöglichkeiten
- **Prozessorentwurf**:
  - Befehlsverarbeitung
  - Entwurfs- und Optimierungsmöglichkeiten
- **Assemblerprogrammierung**:
  - im MIPS-Simulator MARS

Warum MIPS?



- Der MIPS-Befehlssatz ist übersichtlich und einfach zu erlernen (RISC)
- Viele moderne Architekturen sind sehr ähnlich angelegt (z.B. ARM oder RISC V)
- Es gibt viele gute Emulatoren – MARS zeigt sogar den Datenpfad „live“
- War lange Zeit in vielen Systemen im Einsatz (von der Workstation bis zur Fritzbox)
- Z.B. SGI Indigo (1991), Onyx, Indy (1993), O2 (1996) und Octane (1997)
- In den 90ern wurden CGI-Effekte in Filmen auf SGI-Hardware (Indigo, MIPS) berechnet



Quellen: Wikipedia, Openwrt.org, Starring the Computer (Billion Dollar Code)
Siehe auch http://www.sgistuff.net/funstuff/hollywood/index.html

Instruction Set Architecture (Prozessorarchitektur) ISA

Einführung

⇒ ISA ist die Schnittstelle zwischen Hardware und unterster Softwareschicht
⇒ ISA ist der Befehlssatz des Prozessors, also alle Instruktionen, die der Prozessor „kennt“ und „versteht“

Instruktionen

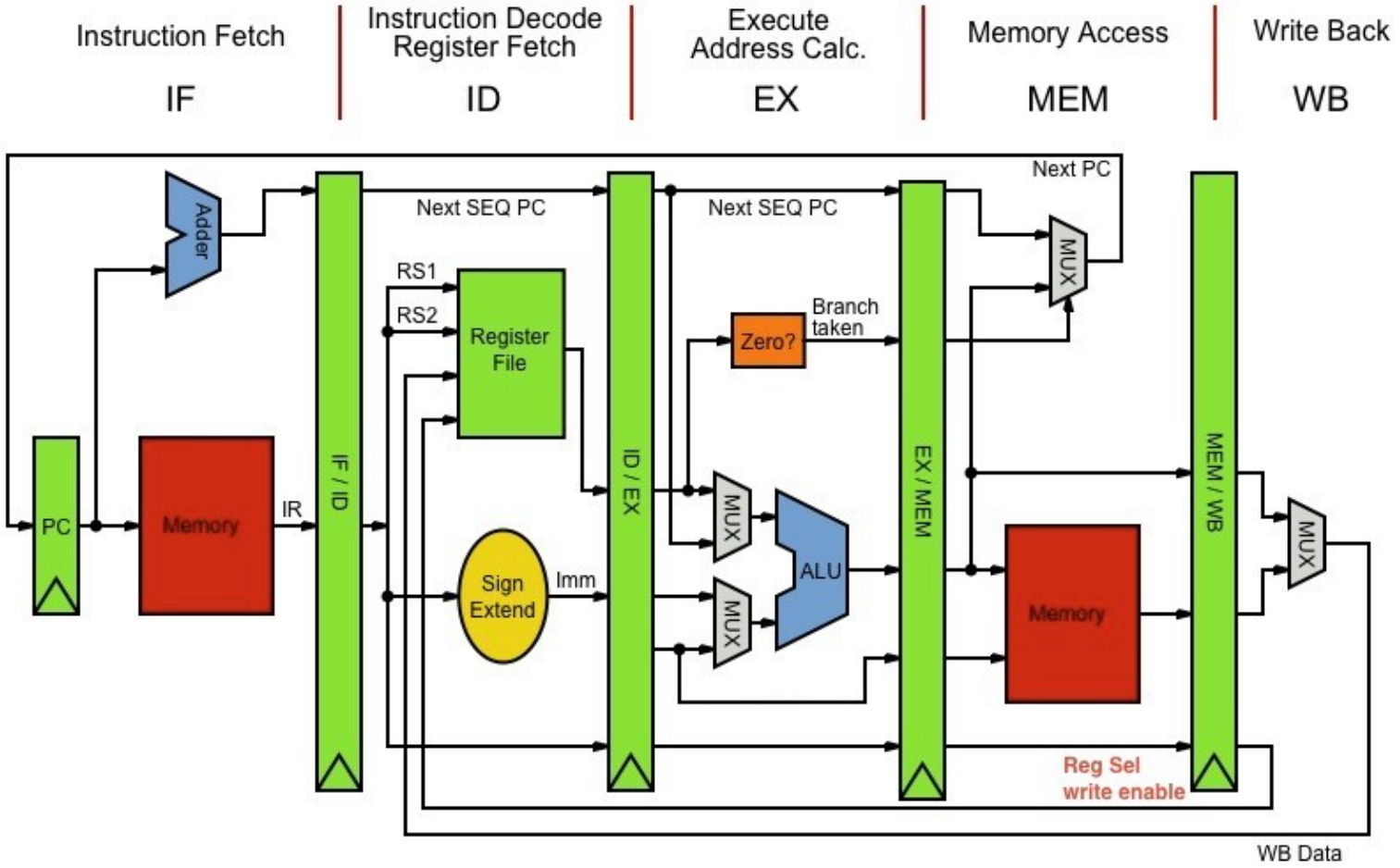
⇒ teilen dem Prozessor schrittweise mit, welche Teilaufgaben er abzuarbeiten hat
⇒ sind die Wörter einer Maschinensprache

Maschinensprache (Assemblersprache)

⇒ basiert auf Hardware, die dieselben Prinzipien hat (Neumann, Harvard)
⇒ ist deswegen sehr ähnlich für verschiedene Prozessoren
⇒ Assembler wird immer seltener zum Entwurf eingesetzt
⇒ meist Programmierung in Hochsprache
  ⇒ dennoch grundlegend zum Verständnis

MIPS: Prozessorarchitektur

- MIPS: Microprocessor without Interlocked Pipelined Stages
  - 1985: Erster Multi-Prozessor mit unabhängigem Pipeline-Phasen (keine Locks)
  - RISC-Architektur: moderne Prozessoren
  - Pipeline-Phasen:



MIPS: 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. MIPS register $zero always equals 0. Register $at is reserved for 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 words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

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

CPU-Eigenschaften

CPU-Befehlssatz

Aufbau eines Befehls in MIPS-Assembler

⇒ Ablauf innerhalb von Registern (= Speicherelemente im Inneren des Prozessors) in denen die Operanden gespeichert werden

⇒ Moderne RISC Architekturen haben typ. **32**, 64 oder 128 Register à **32**/64 Bit

⇒ Arithmetische MIPS-Intruktionen alle den gleichen Aufbau

⇒ Insgesamt gibt es bei MIPS nur 3 Formen (R,I,J –Typ)

⇒ **Design-Prinzip: Einfachheit durch Regularität**

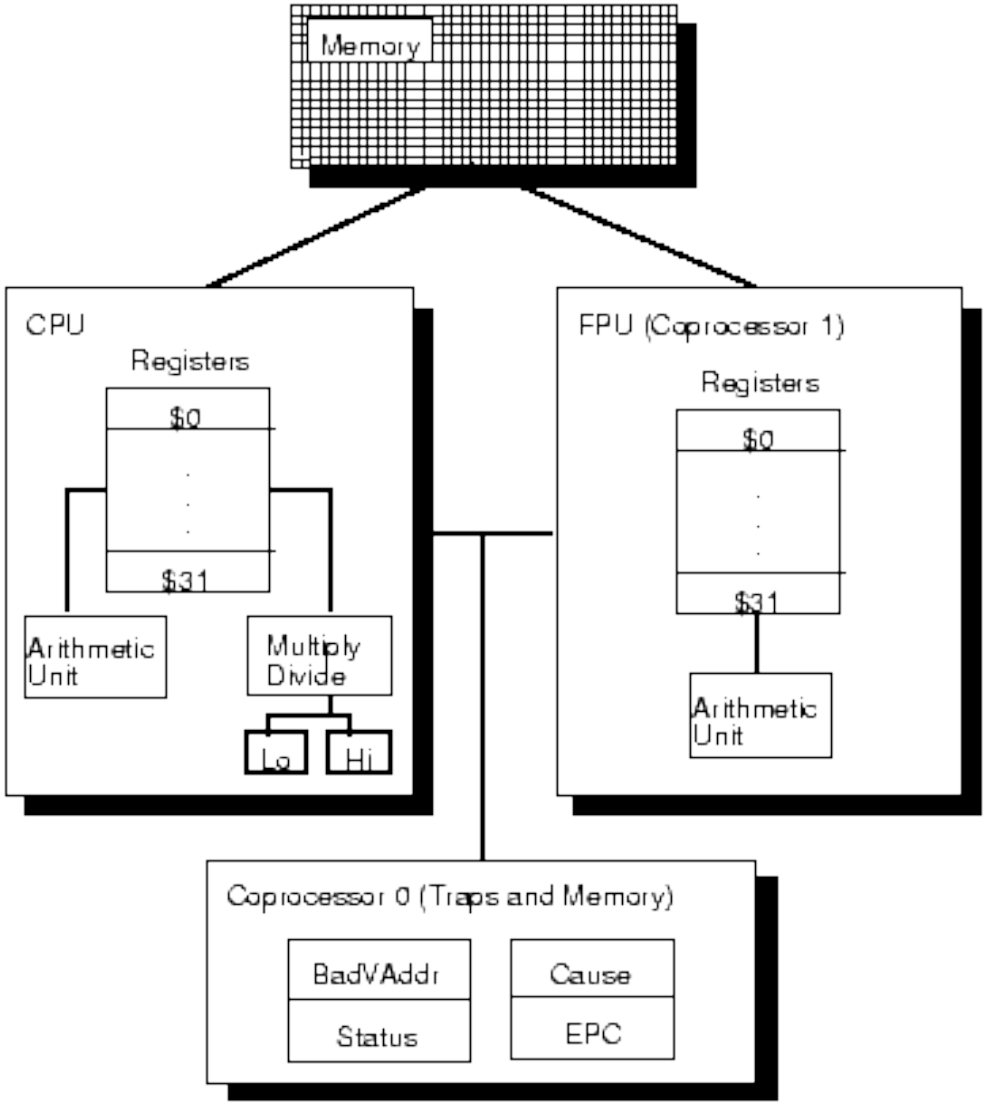
⇒ pro Instruktion wird nur eine Operation ausgeführt

⇒ längere, sog. „mehrwertige“ Operationen werden sequenzialisiert

⇒ **Beispiel a=b+c+d+e**

```
add a,b,c
add a,a,d
add a,a,e
```

Aufbau MIPS-CPU



MIPS: Verwendung von Registern

- ⇒ Daten (Variablen, Konstanten) werden im Prozessor in *Registern* gespeichert
- ⇒ Registergröße = Prozessorwortbreite = Breite der Datenpfade = 8, 16, **32**, 64, 128
- ⇒ Registeranzahl so klein wie möglich

  => höhere Zugriffsgeschwindigkeit und weniger Bits zur Adressierung nötig

  => viele Register führen zu langen Signalwegen und großen clock cycle times

  ⇒ **Design Prinzip: Kleiner ist schneller**

- ⇒ MIPS : 32 32-Bit Register im Inneren des Prozessors (32 Bit == 1 Wort)
- ⇒ Wie die Register verwendet werden, ist eine Konvention, d.h. Entwickler sollten sich daran halten, müssen aber nicht

Registerkonventionen bei MIPS

⇒ bei MIPS Arithmetik-Befehlen müssen die 3 Operanden in einem der 32 Register stehen. Wo was liegt, ist festgelegt (Software-Konvention)

⇒ $s0, $s1,… Register 16-23 für Variablen (etwa eines C-Programmes)

⇒ $t0, $t1,… Register 8-15, 24,25 für Zwischenergebnisse

⇒ Der Compiler assoziiert die Variablen eines Programms mit Registern :

Beispiel : f = (g + h) – (i + j)

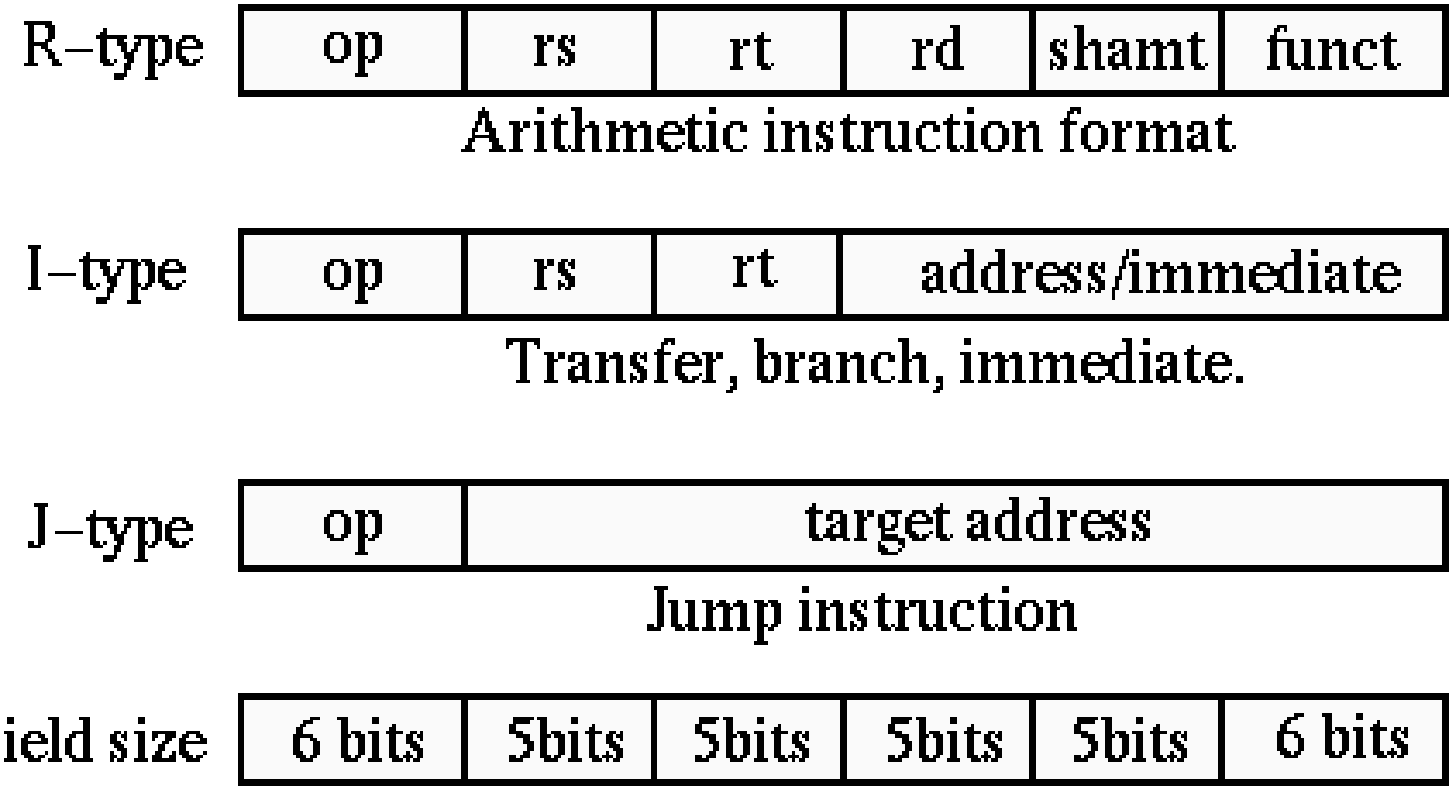
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

f, g, h, i , j seien assoziiert mit $s0, $s1, $s2, $s3,$s4

MIPS: Register-Konventionen

Bezeichnung	Register-Nr.	Verwendung	Preserve on call?
$zero	0	constant 0	n.a.
$at	1	reserved for assembler (macros like blt)	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for interrupts	
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr	yes

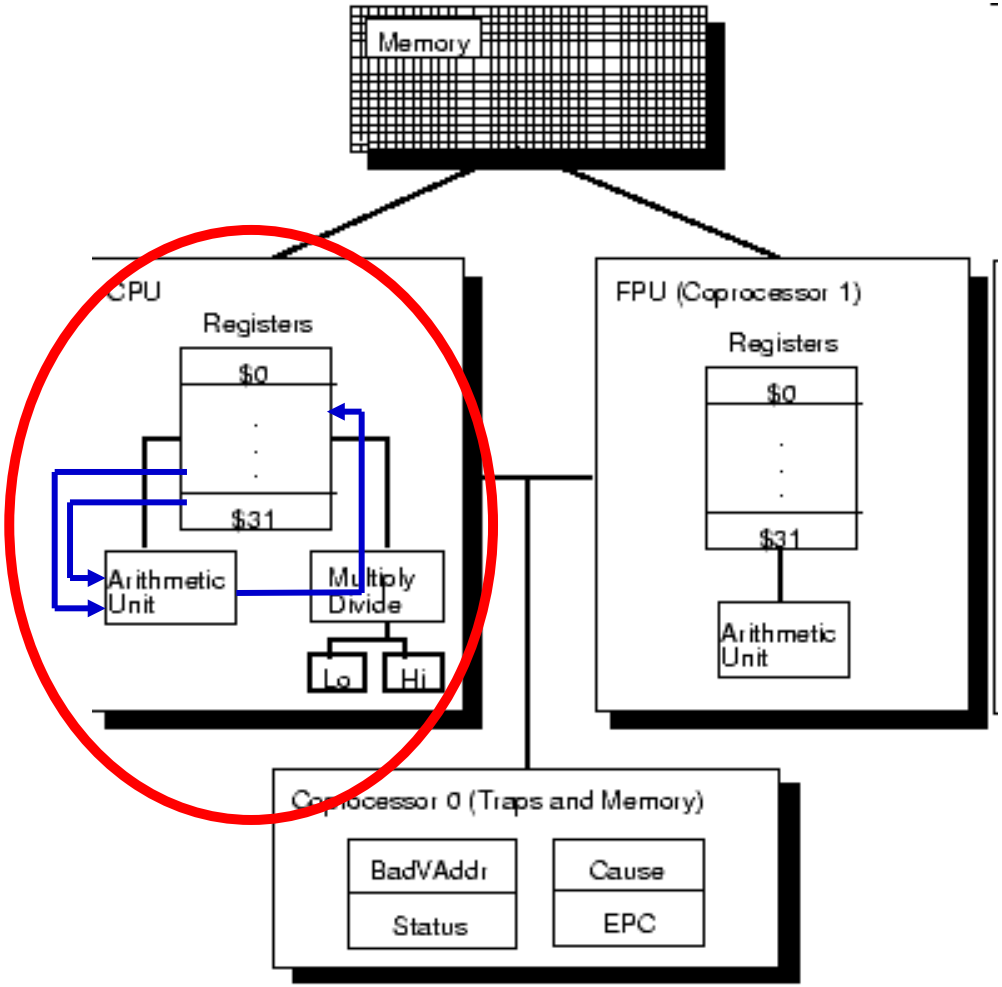
MIPS: Befehlstypen



R-type instructions

only work on registers

(= memory in the inner CPU)



MIPS R2000 CPU structure

Bitvektordarstellung von Instruktionen (Maschinensprache)

- ⇒ Prozessor versteht nur Bitvektoren (die aus dem Speicher eingelesen werden).
- ⇒ Assembler setzt die Assemblersprache in Maschinensprache um
- ⇒ Beispiel Addition: `add $t0, $s1, $s2 → $t0 = $s1 + $s2`

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

- ⇒ op: Operationcode des Befehls funct: Auswahl einer Variante des op-Feldes
- ⇒ OP und FUNCT zusammen definieren den Befehl (OP=000000 / FUNCT=100000)
- ⇒ rs, rt, rd: Source- und Destination-Register (5 Adressbits für 2^5= 32 Register)
- ⇒ shamt: shift amount („don‘t care“ …hier einfach Null …für Addition)

Bitvektordarstellung von Instruktionen (Maschinensprache)

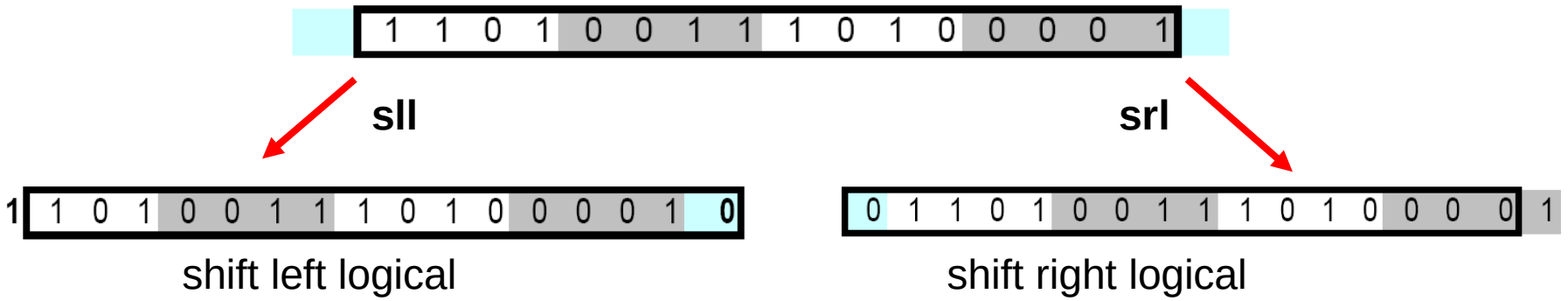
⇒ Weitere arithmetische und logische MIPS-Instruktionen

Instruktionsformat							
Instruktion	op 6 Bit	rs 5 Bit	rt 5 Bit	rd 5 Bit	shamt 5 Bit	funct 6 Bit	Bedeutung
add	000000	rs	rt	rd	xxxxx	100000	rd = rs + rt
sub	000000	rs	rt	rd	xxxxx	100010	rd = rs - rt
and	000000	rs	rt	rd	xxxxx	100100	rd = rs & rt
or	000000	rs	rt	rd	xxxxx	100101	rd = rs | rt
xor	000000	rs	rt	rd	xxxxx	100110	rd = rs **xor** rt
sll	000000	xxxxx	rt	rd	shamt	000000	rd = **shift** <<< rt
srl	000000	xxxxx	rt	rd	shamt	000010	rd = **shift** >>> rt
nop	000000	00000	00000	00000	00000	000000	no operation

Alle Befehle haben **Op**code=000000
Unterscheidung über **funct**

Bitvektordarstellung von Instruktionen (Maschinensprache)

⇒ Anmerkungen zu Shift Left Logical und Shift Right Logical



Bei **sll** geht das linke Bit verloren, rechts wird eine 0 nachgezogen
⇒ dies entspricht Multiplikation mit Faktor 2

Bei **srl** geht das rechte Bit verloren, links wird eine 0 nachgezogen
⇒ dies entspricht Division durch 2

shamt = shift amount spezifiert Anzahl der zu verschiebenden Stellen

nop = no operation wird umgesetzt als => sll mit shamt = 0

Alle Befehle seither sind gleich aufgebaut => **R-TYPE-Instruktionen**

Die Befehle load word und store word

I-Type Instruktionen

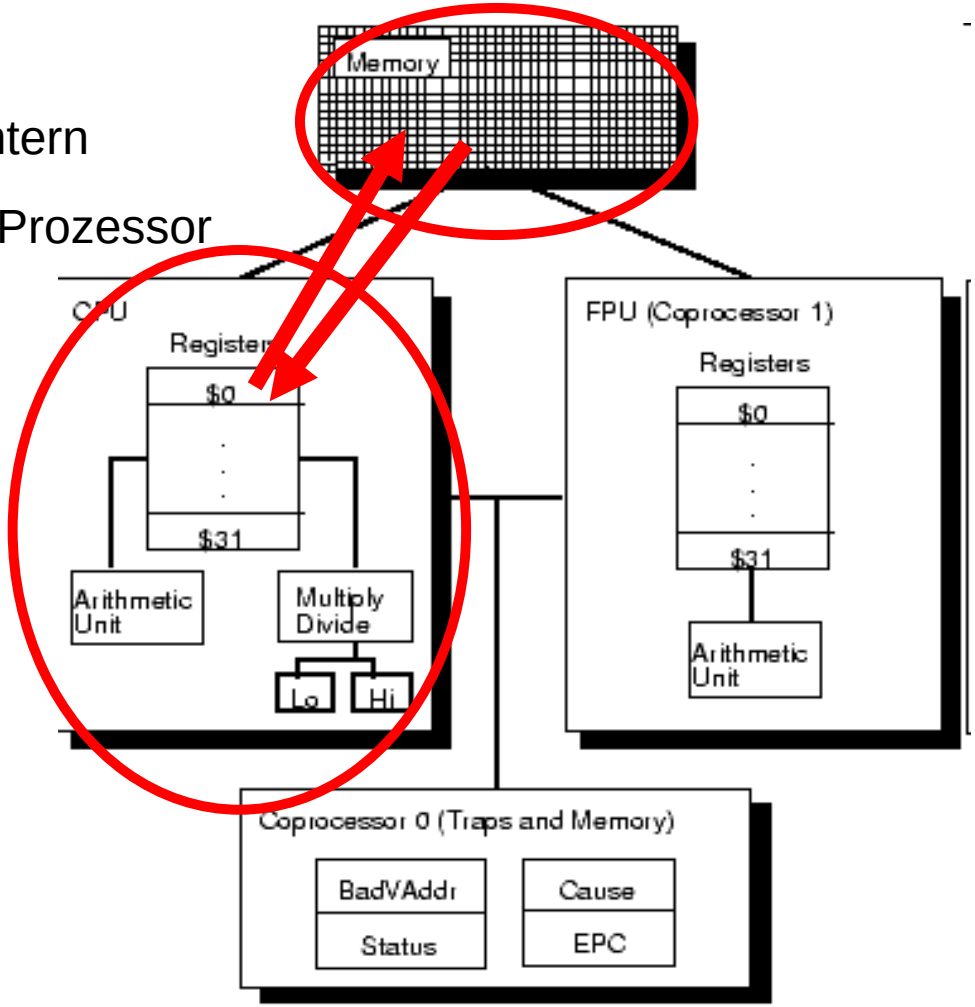
- ⇒ alle bisherigen Befehle waren CPU-intern
- ⇒ Bearbeitung von Registerinhalten im Prozessor
- ⇒ 32 Register reichen nicht aus…
- ⇒ …Speicher ist aber auch noch da !
- ⇒ Weitere Instruktionen nötig, um

  a) Daten vom Speicher in ein CPU-Register zu laden

  => Load word (lw)

  b) Daten von einem Register in den Arbeitsspeicher zu schreiben

  => Store word (sw)



Aufbau MIPS R2000

Format der Befehle load word und store word

Konstante: -2^{15} bis $+2^{15}-1$

Instruktionsformat					
Instruktion	op 6 Bit	rs 5 Bit	rt 5 Bit	immediate 16 Bit	Bedeutung
load word	lw	rs	rt	Offset	**lw rt, Offset(rs)**
		$t0	s$2	0	**lw $s2, 0($t0)**
	100011	01000	10010	0000 0000 0000 0000	
store word	sw	rs	rt	Offset	**sw rt, Offset(rs)**
		$t0	s$2	16	**sw $s2,16($t0)**
	101011	01000	10010	0000 0000 0001 0000	

`lw $s2, 0($t0)` $s2 : Register, in das der Speicherwert geladen wird
$t0 : Register, das die Speicheradresse enthält
0 : Offset zur Speicheradresse (sinnvoll für Arrays)

`sw $s2, 16($t0)` $s2 : Register, von dem geschrieben wird
$t0 : Register, das die Speicheradresse enthält
16 : Offset zur Speicheradresse (sinnvoll für Arrays)

Diesen Befehlsaufbau nennt man **I-TYPE-Instruktionen**

Hinweise zu lw und sw (1)

⇒ Befehlsstruktur von lw und sw unterscheidet sich von den bisher bekannten arithmetischen Befehlen, nur 2 Registeradressen, dafür aber zusätzlich ein Offset (um auf einfache Weise ein Element aus einem Array herauszupicken)

⇒ Wir haben nun neben dem R-Type einen I-Type (I=immediate).

⇒ Dies widerspricht dem 1. Designprinzip **Einfachheit durch Regularität,** für eine einfache Hardware wären gleiche Bitbreiten besser.

=> Kompromiss: Alle MIPS-Instruktionen haben gleiche Länge
zwar mit unterschiedlichem, aber ähnlichen Format

⇒ **Design Prinzip: Gute Designs brauchen gute Kompromisse**

Hinweise zu lw und sw (2)

- ⇒ Load word und store word sind die einzigen MIPS-Befehle, mit denen man auf den Speicher zugreifen kann
- ⇒ Alle anderen Instruktionen arbeiten ausschließlich auf den 32 CPU-internen Registern.
- ⇒ Diese Struktur nennt man LOAD/STORE-ARCHITEKTUR (typisch für RISC)
- ⇒ Viele andere Prozessortypen lassen Speicherzugriffe aber auch in anderen Instruktionen zu, z.B. Addieren eines Speicherinhalts mit einem Registerinhalt.
- ⇒ Problem : Speicherzugriffe dauern länger wie Registerzugriffe
  - ⇒ Befehle dauern nicht gleichlang
    - ⇒ steht Pipeline-Strukturen entgegen

Hinweise zu lw und sw (3)

- ⇒ Mit Load word und store word kann MIPS auch auf I/O-Geräte zugreifen
  - ⇒ Speicheradressen zeigen dann auf I/O anstatt auf den Speicher
    - ⇒ Vorteil : keine extra Befehle nötig

      ⇒ MEMORY MAPPED I/O

- ⇒ Die Alternative wäre ISOLATED I/O.

  - ⇒ Prozessor hat dann extra Signal /MIO, zur Auswahl von Memory oder I/O
    - ⇒ sämtliche anderen Module müssen dieses Signal auch auswerten

Hinweise zu lw und sw (4)

⇒ Höhere Programmiersprachen arbeiten oft mit Feldern (Arrays).
  ⇒ Die Dimension dieser Felder übersteigt in der Regel die Registeranzahl eines Prozessors

=> Arrays werden folglich im Speicher abgelegt

⇒ Aufbau der Befehle lw und sw vereinfacht den Zugriff auf Arrays durch Basisadresse (im Register) + Offset (16 Bit, im Befehl)

=> MIPS unterstützt byteweise Adressierung
(weil bei Speicherchips eine adressierbare Speicherzelle i.d.R. 1 Byte hat)

=> 4 Byte = 1 MIPS-Wort
=> Wortadressen sind Vielfache von 4 (0,4,8,12,16,20….)

⇒ Welches der 4 Byte ist die Wortadresse ? => big endian (wird noch behandelt)
  => Wortadresse = Adresse des höchstwertigsten Byte im Wort
  => Es ist das linksstehende Byte im Wort !

Laden von 32bit-Werten

- Für die meisten Zwecke reichen die 16bit im immediate-Feld
- 32bit-Konstanten müssen über 2 Befehle geladen werden:

statt

```
li $t0, 0xcafeaffe
```

Erst oberen Wert laden mit „load upper immediate“, dann via ori unteren Wert nachladen:

```
lui $t0, 0xcafe
ori $t0, 0xaffe
```

ca fe 00 00
ca fe af fe

→ wird vom Assembler automatisch umgesetzt mit Hilfsregister `$at`

Laden von 32bit-Werten: Adressen

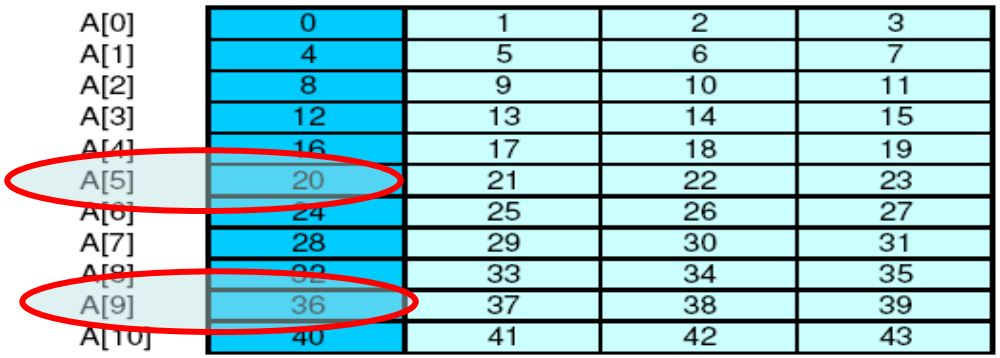
- Um Adressen (Pointer) in ein Register zu laden gibt es das Makro `la` (load address) Dieser wird vom Assembler ebenfalls in `lui` und `ori` übersetzt.

```
.data
text:
    .asciiz “Hello World“
.text
  la $a0, text
```

Der Assembler setzt hier automatisch für das Label „text“ die Adresse im Speicher ein (Teil des Datensegments).

Beispiel zur Offset-Adressierung

```
lw      $t0, 36($s3)
add     $t0, $s2, $t0
sw      $t0, 20($s3)
```



A[0]	0	1	2	3
A[1]	4	5	6	7
A[2]	8	9	10	11
A[3]	12	13	14	15
A[4]	16	17	18	19
A[5]	20	21	22	23
A[6]	24	25	26	27
A[7]	28	29	30	31
A[8]	32	33	34	35
A[9]	36	37	38	39
A[10]	40	41	42	43

Übersetzt in C wäre dies : A[5] = h + A[9] ;

Annahme : die Basisadresse des Arrays (also für A[0]) steht im Register $s3
der Wert h steht in Register $s2

Zeile 1 : In Registeradresse $s3 steht eine Memoryadresse, die Basisadresse von A.
Addiere zu dieser Adresse einen Offset von 4*9=36, das ergibt Adresse von A[9]
Lade das 32-Bit-Wort des berechneten Memoryadresse in das Register $t0

Zeile 2 : Addiere den Wert h, der im Register $s2 steht zu $t0
Speichere das Ergebnis in $t0

Zeile 3 : Nimm wieder die Registeradresse aus $s3, addiere Offset von 4*5=20
Speichere das 32-Bit-Wort aus $t0 am Speicherplatz „20“

Übung : Was macht dieses Programm ?

```
1.  lw   $t0, 0($s3)
2.  lw   $t1, 4($s3)
3.  lw   $t2, 8($s3)
4.  lw   $t3, 12($s3)
5.  add  $t5, $t0, $t1
6.  sll  $t5, $t5, 2
7.  add  $t6, $t2, $t3
8.  slr  $t6, $t6, 1
9.  sub  $t4, $t5, $t6
10. sw   $t4, 16($s3)
```

Es gibt ein Array im Speicher mit mind. 5 Werten

4 Werte daraus werden in den ersten 4 Zeilen in

die Register $t0, $t1, $t2, $t3 geladen

Wir definieren : $t0, $t1, $t2, $t3 = a,b,c,d

Zeile 5 : Berechne $t5=a+b

Zeile 6 : Shift Logical Left mit 2 == Multiplikation mit 4
Zeile 7 : Berechne $t6 = c+d
Zeile 8 : Shift Logical Right mit 1 == Division durch 2
Zeile 9 : Berechne $t4 = $t5 - $t6
Zeile 10 : Speichere den Registerswert aus $t4 ins Memory zurück
an die 5. Stelle im Array (0,4,8,12,**16**….)

C-Code für dieses Programm : `e = 4*(a+b) – (c+d)/2 ;`

Arithmetrische I-Type Instruktionen

Arithmetrische I-Type Instruktionen

⇒ I -Type Instruktionen heißen so, weil „immediately“ (= sofort, unmittelbar) eine Konstante direkt im Befehl angewendet wird.

⇒ R-Type Instruktionen hingegen bearbeiten Registerwerte

⇒ direkte Verarbeitung von Konstanten über die Befehlszeile macht Sinn, da man so „wertvolle“ Register spart und zeitraubende Speicherzugriffe vermeidet

⇒ Da Variablen sehr oft mit Konstanten verknüpft werden, gibt es eine Reihe an I-Type Befehlen, bei denen die Konstante direkt in der Befehlszeile eingegeben wird

⇒ **Design Prinzip: Make the common case fast**

⇒ da I-Type-Befehle dieselbe Struktur haben => 16 Bit (wie Offset bei lw/sw)

Arithmetrische I-Type Instruktionen

Instruktionsformat					
Instruktion	op 6 Bit	rs 5 Bit	rt 5 Bit	immediate 16 Bit	Bedeutung
addi	001000	rs	rt	Konstante	rt = rs + Konstante
andi	001100	rs	rt	Konstante	rt = rs & Konstante
ori	001101	rs	rt	Konstante	rt = rs or Konstante
xori	001110	rs	rt	Konstante	rt = rs xor Konstante

Beispiel:

Laden einer Konstanten (hier die Zahl 122) in das Register $s1:

```
addi    $s1, $zero, 122
```

Zero Register

- ⇒ Sehr oft benötigt wird der Wert Null als Konstante
- ⇒ MIPS stellt hierfür ein eigenes, unveränderbares Register zur Verfügung
- ⇒ Register Nummer 0 heißt Zero-Regíster $zero und enthält den Wert 0

Beispiel	add	$v0, $s0, $zero
Bedeutung :	$v0 wird der Inhalt von $s0 zugewiesen !	

Kontrollstrukturen in MIPS

Stored Program Konzept (von Neumann Konzept)

Ein von Neumann-Rechner arbeitet Programmbefehle nach folgenden Regeln ab.

- ⇒ Befehle (zusammen mit Daten) sind in einem linearen Adressraum abgelegt.
- ⇒ Ein Befehls-Adressregister (Befehlszähler, Program Counter) zeigt auf den nächsten auszuführenden Befehl
- ⇒ Befehle werden aus einer Zelle des Speichers gelesen und dann ausgeführt
- ⇒ der Inhalt des Befehlszählers wird anschließend um Eins erhöht
- ⇒ neben sequentiellen Befehlen gibt es zusätzlich Sprung-(Jump) Befehle, die den Inhalt des Befehlszählers um einen anderen Wert als +1 verändern
- ⇒ und Verzweigungs-(Branch) Befehle, die in Abhängigkeit eines decision-Bits den Befehlszähler um Eins erhöhen oder einen Sprung ausführen.

- ⇒ Bei MIPS erhöht sich der Program Counter nicht um 1, sondern immer um 4
- ⇒ Bei MIPS ist der Program Counter für den Programmierer nicht direkt zugänglich !

MIPS: 4-bit-Befehlssatz

Unbedingter Sprung (jump): j offset

- Zweck: kodieren von Schleifen (loops) oder Verzweigungen
  - ➔Ändert linearen Kontrollfluss
  - ➔PC = PC + (offset * 4)

- Befehlsformat: J-TYPE-Instruktionen

- Beispiel:

Instruktionsformat			
Instruktion	op 6 Bit	address 26 Bit	Bedeutung
j	000010	0000 0000 0000 0000 0000	j 0 / Jump to 0

- PC: Aktualisierung
  - die niederwertigen 26+2 Bits des PC werden mir dieser Konstante überschrieben
    (shift left 2, da unterste 2 Bits immer 0 sind → 4-Byte-Alignment)
  - die höherwertigen 4 Bits bleiben gleich (abhängig vom PC)
  - Sprungweite maximal 2^{28}

Weitere Instruktionen für Kontrollstrukturen

⇒ Bedingter Sprung J (BEQ, BNE)

Instruktionsformat					
Instruktion	op 6 Bit	rs 5 Bit	rt 5 Bit	immediate 16 Bit	Bedeutung
beq	000100	Reg #1	Reg #2	Offset	if reg1 == reg2 goto Offset
bne	000101	Reg #1	Reg #2	Offset	if reg1 != reg2 goto Offset

- neuer Wert wird in den Program Counter geladen, wenn die Inhalte 2er Register
  - gleich (beq, branch if equal) oder
  - ungleich (bne, branch if not equal) sind
  -
- **I-Type Format** mit Opcode, 2 Register und 16-Bit Offset

- Im Gegensatz zum unbedingten Sprungbefehl wird keine direkte Sprungadresse
- angegeben, sondern ein Offset, der zum aktuellen Wert des PC addiert wird.

- bedingte Anweisungen und Schleifen können nun in MIPS programmiert werden

if-else Strukturen in MIPS

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

```
bne     $s3, $s4, Else
add     $s0, $s1, $s2
j       Exit
Else:   sub $s0, $s1, $s2
Exit:
```

f, g, h, i, j
seien abgelegt in
$s0 bis $s4

Exit: , Else : werden als **Label** bezeichnet, ist kein Teil der Programmiersprache
=> Label kann beliebigen Namen annehmen !

MIPS-Code-Erläuterung

Zeile 1: falls i und j ungleich sind, gehe zum Label „Else“ (Zeile 5)
Zeile 2: falls i und j gleich waren, landet man hier. Berechnung von f=g+h
Zeile 3 : Folgebefehl nach Zeile 2 : Gehe zum Label „Exit“ (Zeile 5)
Zeile 4 : Label „Else:“ . Falls i ≠ j war, kommt man von Zeile 1. Berechnung von f=g-h
Zeile 5 : Label „Exit:“. Ende des „Unterprogramms“

Adressberechnung/Zuweisungen für Label : wird vom MIPS-Assembler erledigt !

While – Loop Strukturen in MIPS

```
while (Array [i] == k)
i += 1;
end
```

```
loop:   sll $t1, $s3, 2
add     $t1, $t1, $s6
lw      $t0, 0($t1)
bne     $t0, $s5, Exit
addi    $s3, $s3, 1
j       loop
Exit:
```

i,k abgelegt in $s3,$s5

Arraybasisadresse sei abgelegt in $s6

MIPS-Code-Erläuterung

Zeile 1: Offsetberechnung für Zugriff auf i-tes Element im Array
sll mit 2 => Faktor 4 => temporäre Variable $t1=4*$s3 (Byte-Adressierung)
Zeile 2 : Berechnung der Adresse = Arraybasisadresse + Offset (aus Zeile 1)
Zeile 3 : Einladen des Arrayelements aus dem Memory ins CPU-Register $t0
Zeile 4 : Falls das eingeladene Arrayelement Array [i] ≠ k, gehe zum Label Exit
ansonsten ist logischerweise Array [i] = k und man landet in Folgezeile 5
Zeile 5 : Addition ausführen
Zeile 6 : Springe zum Label „Loop:“
Zeile 7 : Label „Exit:“. Ende des „Unterprogramms“

Behelfskontrukte für bedingte Sprünge

⇒ MIPS-Architektur sieht keine Sprungbefehle für die Fälle a<b bzw. a>b vor

⇒ Es gibt nur die Sprungbefehle beq und bne, man kann also für a<b bzw. a>b nur verzweigen, indem man a<b bzw a>b gesondert abprüft und dann in einer Folgeprogrammzeile spingt !

Grund : Dies ermöglicht eine einfachere Hardware !

⇒ **Design Prinzip: Gute Designs brauchen gute Kompromisse**

Kompromiss : Zwei neue Befehle slt und slti, die eine Hilfsvariable auf 0 oder 1 setzen, was in der Folge durch beq oder bne abgefragt werden kann.

Behelfskontrukte für bedingte Sprünge

⇒ **Die Befehle slt (set on less than, R-Type) und slti (set on less than (I-Type)**

slt $t0, $s3, $s4 (*set on less than, r-type*)

$t0 wird auf 1 gesetzt, wenn Registerinhalt von $s3 kleiner als Registerinhalt von $s4.

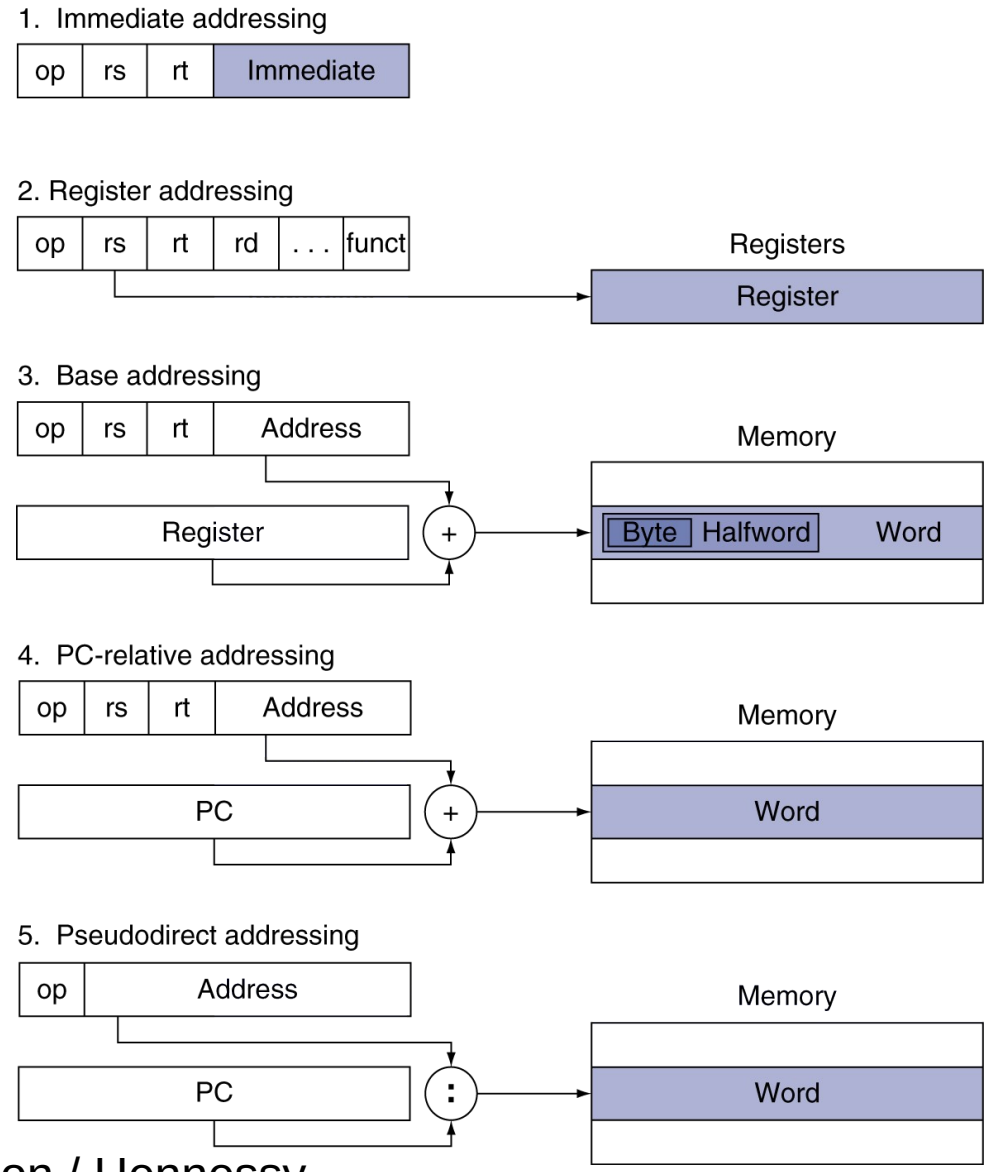
slti $t0, $s2, 10 (*set on less than, i-type*)

$t0 wird auf 1 gesetzt, wenn der Registerinhalt von $s2 kleiner als 10 ist.

Mit slt, slti, beq, bne und dem festen Wert 0 (Zero-Register) sind alle relativen Bedingungen , d.h. = ≠ ≤ ≥ < > abfragbar.

```
slt $t0, $s3, $s4          # if $s3 < $s4
bne $t0, $zero, Label      #   goto Label
```

Adressierungsmöglichkeiten im Überblick



Quelle: Patterson / Hennessy

MIPS Assembler Pseudoinstruktionen / Makros

- Neben Anweisungen die direkt in Maschineninstruktionen umgesetzt werden, gibt es auch Makros
- Hierfür wird manchmal als Hilfsregister `$at` benutzt

- Register → Register-Kopie

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

- Branch if less than

```
blt $t0, $t1, Lbl      →   slt $at, $t0, $t1
                           bne $at, $zero, Lbl
```

- `$at` steht für („assembler temporary“)

Funktionsaufrufe in MIPS

Unterstützung von Funktionen & Prozeduren (1)

⇒ Bisher wurden Programmblöcke (Prozeduren) vorgestellt
und deren Funktion erläutert

⇒ zu klären ist noch, wie Prozeduren in ein Programm eingebunden werden d.h.
- ⇒ Handling von Übergabeparametern
- ⇒ Vorgehensweise, wenn man mehr Parameter als Register zur Verfügung hat

⇒ Aufrufendes Programm muss Übergabeparameter in einem Register bereitstellen
Prozedur muss auf dieses Register zugreifen können

⇒ Prozedur muss Programmabarbeitung übernehmen
und das Unterprogramm abarbeiten

⇒ Prozedur muss Ergebnisse in einem Register bereitstellen
Das ursprünglich aufrufende Programm muss auf dieses Register zugreifen können

⇒ Ursprünglich aufrufendes Programm muss Programmabarbeitung übernehmen

Unterstützung von Funktionen & Prozeduren (2)

MIPS Konvention

Übergabeparameter an Prozedur (Callee)

⇒ Register $a0,…,$a3

Rückgabeparameter an aufrufendes Programm (Caller)

=> Register $v0, $v1

Rückkehradresse

=> *Return Address Register $ra*

Befehle für Prozeduren : jal (Jump and Link) und jr (Jump Register)

⇒ Format : `jal` Prozeduradresse bzw. `jr $ra`

⇒ `jal` springt zur Prozeduradresse und
sichert Adresse der nächsten Instruktion (Programcounter + 4) in $ra

⇒ Rückkehr mit `jr $ra`

Unterstützung von Funktionen & Prozeduren (3)

Vorgangsweise

⇒ Aufrufendes Programm „Caller“ schreibt Übergabeparameter in $a0,…,$a3

⇒ Caller springt mit jal Prozeduradresse ins Unterprogramm „Callee“

⇒ Callee führt Prozedur aus

⇒ Callee schreibt Prozedurergebnisse in $v0, $v1

⇒ Callee überstellt an Caller mit dem Befehl jr $ra

Unterstützung von Funktionen & Prozeduren (4)

Problem bei Prozeduren :

=> oft mehr Übergabeparameter und/oder Prozedurergebnisse als o.g. Register
=> ebenso fehlen Register bei verschachtelten Prozeduren

Beispiel :

Callee benutzt aus Mangel an zu Verfügung stehender Register auch Register, die vom Caller verwendet werden.

⇒ Diese Registerinhalte müssen ausgelagert werden ins Memory.
⇒ Nach Ausführung des Unterprogramms müssen die ursprünglichen Register wieder mit den Inhalten gefüllt sein, die vor dem Aufruf des Unterprogramms drin standen

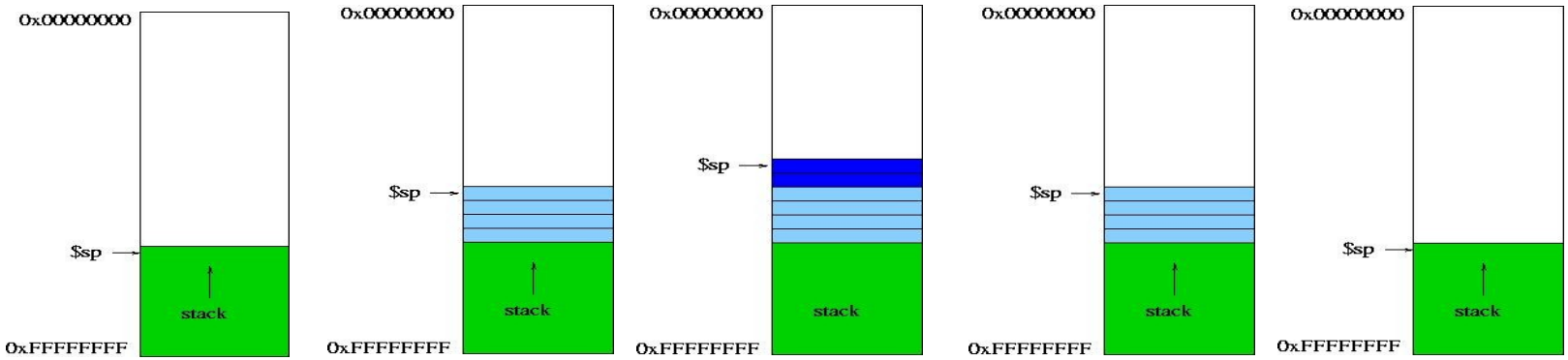
⇒ Die ideale Struktur, um so eine Aufgabe zu lösen ist ein **Stack** (sog. Kellerspeicher mit Last-In-First-Out-Funktion)

=> ein Register $sp, der Stackpointer zeigt auf den Beginn eines Speicherbereichs in diesem Speicherbereich werden lokale Variablen zwischengespeichert

MIPS: 4-bit-Befehlssatz

Stackfüllung vor, während und nach Prozeduraufrufen

Grün() ruft Hellblau() auf , Übergabe der Argumente über den Stack
Hellblau() ruft Dunkelblau() auf , Übergabe der Argumente über den Stack
Dunkelblau() ist fertig, stellt den ursprünglichen Stack wieder her
Hellblau() ist fertig, stellt den ursprünglichen Stack wieder her



=> Stackpointer $sp zeigt zum “obersten” Element im Stack
=> Frame Pointer $FP zeigt zum ersten Wort des Stackframes

=> Nach Abschluss jedes Unterprogramms ist der jeweils unterliegende Teil des Stacks wieder so wie vorher und alle ursprünglichen Register wurden wiederhergestellt.

Unterstützung von Funktionen & Prozeduren (5)

⇒ Um ein “wildes Durcheinander” und unnötiges Register Spilling zu vermeiden, gibt es bei MIPS folgende Konvention

$t0-$t9:
⇒ Dies sind temporäre Register und können von Callees beliebig verändert werden

$s0-s7:
⇒ sog. “saved registers”, Inhalte dieser Register **müssen** vom Callee (mittels Operationen auf dem Stack) wiederhergestellt werden

⇒ Es ist Aufgabe des Compilers (bzw. Ihre Aufgabe, wenn Sie in Assembler programmieren), bei Unterprogrammen ggf. Speicher auf dem Stack zu allokieren,

Variablen dort zwischenzuspeichern und nach Ablauf des Unterprogramms die Ausgangsregister wieder “sauber”, d.h. im Ausgangszustand wieder herstellen.

Beispiel : C-Subprogramm in MIPS-Assembler

```
int leaf_example (int g, int h,int i, int j, int k)
{
 int f:
    f = (g+h) - ((i+j)+k)
 return f;
}
```

g, h, i, j **seien abgelegt in $a0 … $a3 und $t3**
f soll $s0 entsprechen

```
leaf_example:
addi    $sp, $sp, -16
sw      $t3, 12($sp)
sw      $t1, 8($sp)
sw      $t0, 4($sp)
sw      $s0, 0($sp)
add     $t0, $a0, $a1
add     $t1, $a2, $a3
add     $t1, $t3, $t1
sub     $s0, $t0, $t1
add     $v0, $s0, $zero
lw      $s0, 0($sp)
lw      $t0, 4($sp)
lw      $t1, 8($sp)
lw      $t3, 12($sp)
addi    $sp, $sp, 16
jr      $ra
```

Num Register auf Stack retten!

Stackpointer um 4*4=16 erhöhen

pop $s0

pop $t1

pop $t0

pop $t3

4 Werte vom Register auf Stack gerettet!

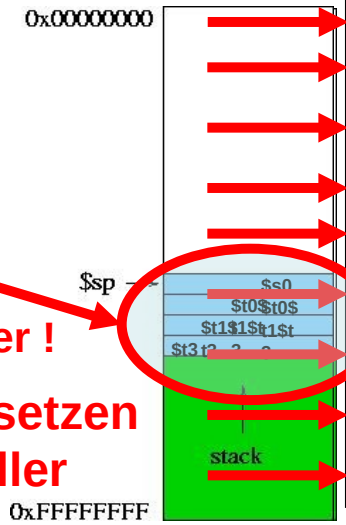
$s0 = Berechnung

$v0 = Rückgaberegister !

Subroutine

Stackpointer zurücksetzen

Rücksprung zum Caller



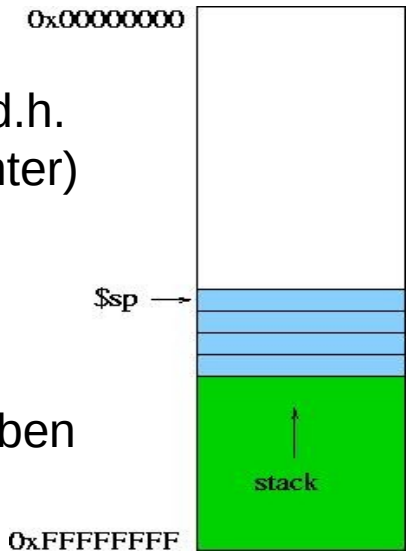
MIPS: 4-bit-Befehlssatz

Anmerkung zum Stack bei MIPS

⇒ Stackorganisation ist physikalisch nicht vorhanden
  ⇒ kein Hardware-Stack !

⇒ Stack wird „softwaremäßig emuliert“, d.h.
  mit Stackpointer (und ggf. Framepointer)
  wird ein Speicherbereich reserviert
  und in diesem gearbeitet

⇒ Rechner ohne viel internen Speicher
  (z.B. die Atmel AVR ohne SRAM) haben
  i.d.R. einen Hardwarestack !



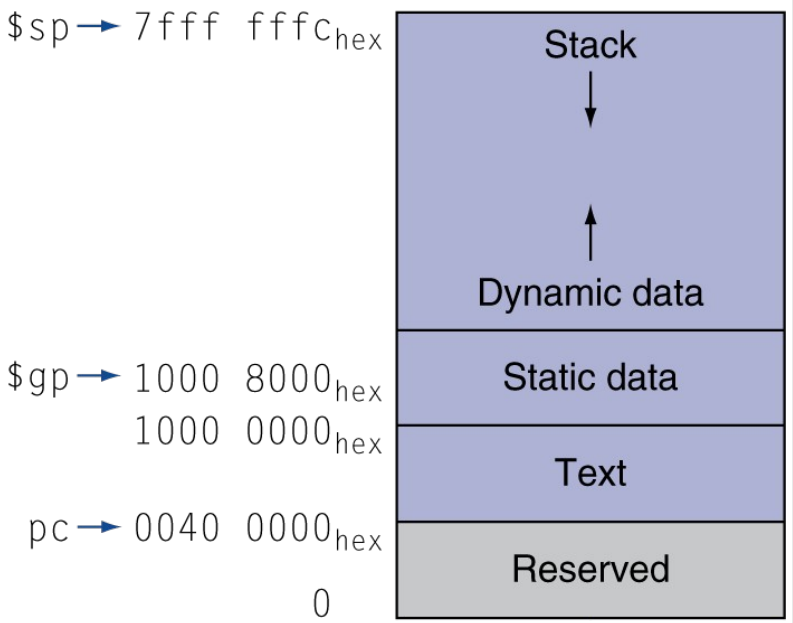

```
…
addi    $sp, $sp, -16
sw      $t3, 12($sp)
sw      $t1, 8($sp)
sw      $t0, 4($sp)
sw      $s0, 0($sp)
….
lw      $s0, 0($sp)
lw      $t0, 4($sp)
lw      $t1, 8($sp)
lw      $t3, 12($sp)
addi    $sp, $sp, 16
…..
```

⇒ Die Begriffe push/pull stammen vom Hardwarestack.
  ⇒ keine MIPS-Befehle, Beispiel x86

⇒ Es gibt reine Stackrechner (0-Adress-Befehle, Zugriff immer auf Top-of-Stack)
  ⇒ HP48, RPN
  ⇒ JVM

MIPS Memory Layout

- Stack: Stapelspeicher
  - `$sp` zeigt auf Ende des Stacks
- Dynamische Daten
  - Heap, Speicherverwaltung durch Programm (malloc, calloc in C, new in Java)
- Static Data
  - globale Variablen, Konstanten (in C als static deklarierte Variablen)
  - `$gp` zeigt auf Mitte des Static Data Segments (für vorzeichenbehaftete Offset-Addressierung)
- Text: Maschinencode, eigentliches Programm



Patterson/Hennessy

MIPS Assembler

- Kommentar:
  - beginnt mit einem „#“ - Symbol
- Assembler Direktiven
  - Direktiven sind interne Anweisungen wie z.B. für die Speichernutzung
  - Direktiven beginnen mit „.“ und sind reservierte Wörter
- Beispiele für Assembler Direktiven

```
.align 2          # Speicher-Elementgrenze = word
.ascii str        # str ist nicht null-terminiert
.asciiz str       # str ist null-terminiert
.data             # folgender Teil wird ins Datensegment
.globl sym        # globales Symbol
.text             # folgender Teil ins Text Segment
```

MARS / qtSPIM Syscalls

- System Calls sind Betriebssystem-Aufrufe
- System Calls werden durch die Instruktion `syscall` aktiviert
- Die eigentliche Syscall-Funktion wird durch Register $v0 festgelegt
- Beispiele:
  - `print string:  $v0=4  $a0 = string`
  - `print int:     $v0=1  $a0 = integer`
  - `read int:      $v0=5  $v0 = integer`
  - `read char:     $v0=12 $v0 = char`
  - `exit:          $v0=10`