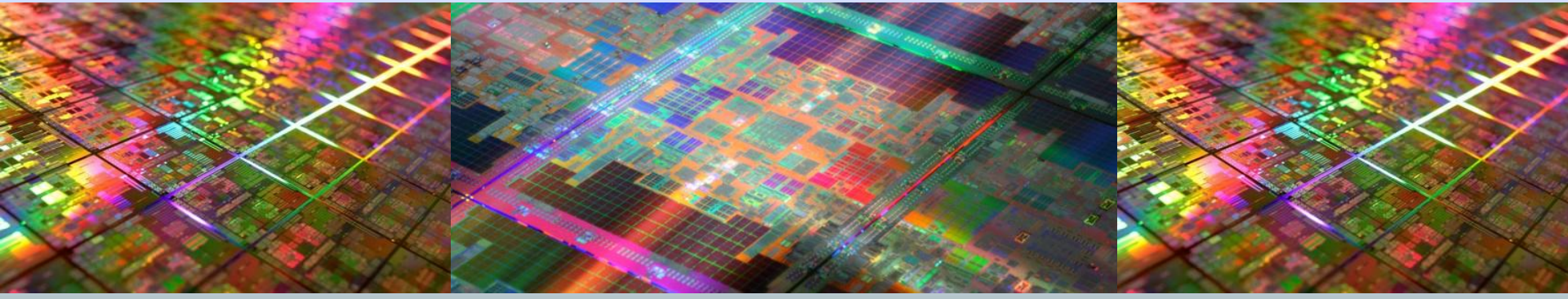
MIPS32 Befehlssatz

Technische Grundlagen der Informatik für Wirtschaftsinformatik

Stefan Podlipnig

TU Wien



Lernziele

- Kennenlernen einer Instruction Set Architecture (ISA) am Beispiel MIPS
  - Befehlssatz
  - Adressierungsarten
  - Instruktionsformate
  - Kompromisse beim Entwurf einer ISA
- Erstellen kleiner Programme in der MIPS-Assemblersprache
- Verständnis der Arbeitsweise von Assembler, Linker und Loader

EINLEITUNG

Maschinensprache

- Die elektronischen Schaltungen eines Rechners „kennen“ nur eine begrenzte Menge einfacher Maschinenbefehle (*instructions*)
- Maschinenbefehle bilden die Maschinensprache (*machine language*)
  - Unterschiedliche Rechner (CPUs) können unterschiedliche Maschinensprachen haben
- **Alle Programme** müssen vor der Ausführung in Abfolgen von Maschinenbefehlen umgewandelt werden
  - Befehle stehen als Bitmuster im Speicher
- Beispiel (MIPS32 Architektur, ein Befehl als Bitmuster)
  - `00000001000010010101000000100000`
    - Addiere binäre Werte in Register 8 und 9 und speichere Ergebnis in Register 10

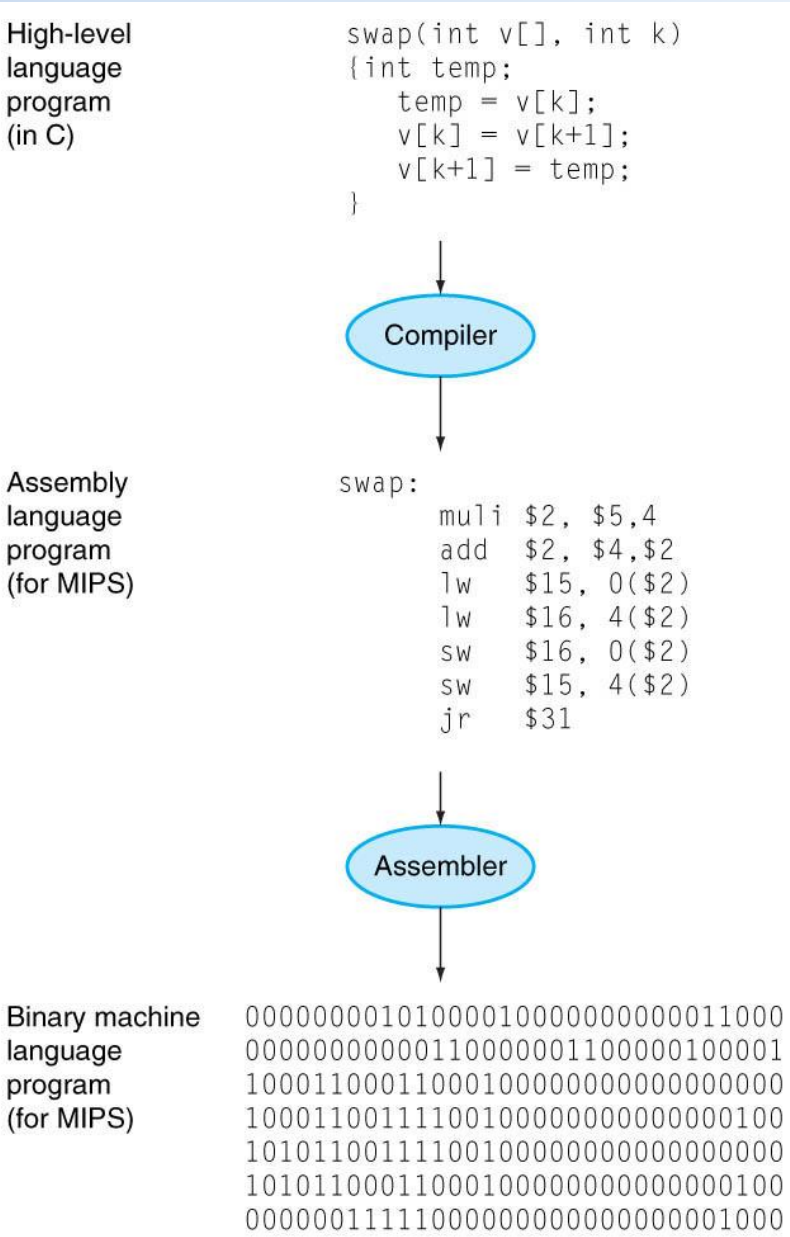
Assembler

- Maschinensprache ist für längere Programme ungeeignet
- Assemblersprache (*assembly language*)
  - Symbolische Darstellung von Maschinenbefehlen
  - Beispiel (MIPS32 Architektur)
    - **`add $t2, $t0, $t1`** entspricht `00000001000010010101000000100000`
- Assembler
  - Übersetzt die symbolische Version der Befehle in eine binäre Form

Höhere Programmiersprachen

- Programmieren in Assemblersprache ist sehr zeitaufwändig und fehleranfällig
- Daher gibt es höhere Programmiersprachen (*high level languages*)
  - Beispiele dafür sind C, C++ oder Java
- Rechner „versteht“ nicht die Befehle der höheren Programmiersprache
  - Programm muss übersetzt werden

Übersetzen eines C-Programms



MIPS

Befehlssatzarchitektur (ISA)

- Befehlssatz (*instruction set*)
  - Befehle, die eine bestimmte Architektur versteht
  - Unterschiedliche Rechner haben unterschiedliche Befehlssätze
    - Aber viele Aspekte werden ähnlich behandelt
- Befehlssatzarchitektur (*instruction set architecture*, ISA)
  - Schnittstelle zum Programmierer bzw. Compiler (Abstraktion!)
  - Prozessoren mit der selben ISA sind binärkompatibel
- ISA aus Sicht des Programmierers
  - Datenformate und Datentypen
  - Adressierung (Spezifikation der Operanden einer Operation)
  - Befehle (Operationen auf Daten, Ablaufsteuerung) und Befehlsformate
  - …

MIPS Befehlssatz

- In dieser Veranstaltung wird die MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) ISA eingesetzt
  - 1981 entwickelt
  - Grundlage für ersten RISC(Reduced Instruction Set Computer)-Prozessor
  - Heute überwiegend im Bereich eingebetteter Systeme eingesetzt
    - z.B. Router, Navigationsgeräte, DTV, DVD recorder, ...
- Es wird hier nur die 32-Bit MIPS ISA betrachtet (MIPS32)

MIPS32 Überblick

- Register/Register-basierte ISA
  - Operationen arbeiten mit Daten aus Registern
  - Register sind prozessorinterne sehr schnelle Speicherplätze
- MIPS32
  - 32 allgemeine Register (jeweils 32 Bit breit)
  - Zusätzliche Spezialregister, z.B. Befehlszähler (PC, *program counter*)
- Befehlskategorien
  - Arithmetisch
  - Datentransfer
  - Logisch
  - Bedingte Verzweigung
  - Unbedingter Sprung
  - Gleitkomma (Co-Prozessor)

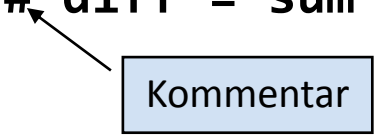
Arithmetische Operationen (allgemeines Konzept)

- Arithmetische oder logische Operation **op** hat die Form

```
op dest,src1,src2
```

  - **op** ist eine symbolische Bezeichnung der Operation, z.B. add oder sub
  - **src1** und **src2** sind Platzhalter für die Operanden der Operation
  - **dest** ist ein Platzhalter für das Ergebnis der Operation
- In einem Assemblerprogramm steht in jeder Zeile ein derartiger Befehl, z.B. (schematisch)

```
add sum,a,b          # sum = a + b
sub diff,sum,c       # diff = sum – c
```



Entwurfsprinzip 1

Einfachheit begünstigt Regelmäßigkeit

- Die Hardware für eine feste Anzahl an Operanden ist weniger komplex als eine Hardware für eine variable Anzahl an Operanden
- Einfachheit ermöglicht eine höhere Leistung bei geringeren Kosten

- Aber
  - Komplexere Befehle in einer Hochsprache erzeugen längere Befehlsfolgen in der Maschinensprache

Operanden arithmetischer Befehle

- Die Operanden `src1` und `src2` einer Operation sind Register des Registersatzes
- Ergebnis `dest` wird wieder in ein Register geschrieben
- Insgesamt gibt es 32 Register
- Für allgemeine Zwecke (Assemblerschreibweise)
  - `$s0, $s1,..., $s7` zur Speicherung von Variablen
  - `$t0, $t1,..., $t9` zur Speicherung temporärer Werte

Entwurfsprinzip 2

Kleiner ist schneller

- Eine zu große Anzahl an Registern kann zu einer längeren Taktzykluszeit führen
- Wahl der Anzahl der Register ist ein Kompromiss
- Anzahl wird auch durch erforderliche Anzahl von Bits in den Befehlsformaten beschränkt

Beispiel (arithmetische Operationen in MIPS)

- Beispiel (in Hochsprache, Variablen entsprechend initialisiert)

```
f = (g + h) – (i + j);
```

- Umsetzung in MIPS
  - Annahme: Werte für f, g, h, i, j werden (bzw. sind) in $s0, $s1, $s2, $s3, $s4 abgelegt

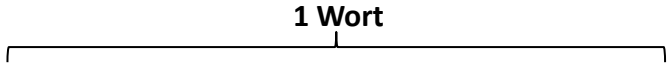
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Wort

- Ein Wort (*word*) ist die natürliche Zugriffseinheit in einem Rechner
- Entspricht in MIPS der Größe eines Registers (also 32 Bit)
- Typische Wortbreiten
  - n = 8: **Byte**, in Mikroprozessoren der ersten Generation (z.B. Intel 8080, Z80)
  - n = 16: **Halbwort** (bei Intel: Wort) in Minicomputern und Mikroprozessoren der zweiten Generation, wie z.B. PDP-11, Intel 8086, Motorola 68000
  - n = 32: **Wort** (bei Intel: Doppelwort) in Mikroprozessoren der dritten Generation, z.B. Intel Pentium, Motorola 68030
  - n = 64: **Doppelwort** (bei Intel: Quadwort) in aktuellen Prozessoren, z.B. Intel Core-i7

Speicheroperanden

- Der Hauptspeicher wird für komplexere Strukturen verwendet
  - z.B. Arrays
- Für Operationen darauf benötigt man Datentransfer-Befehle
  - Laden der Werte aus dem Hauptspeicher in die Register
  - Speichern der Ergebnisse aus den Registern in den Hauptspeicher
- Für den Zugriff benötigt man Speicheradressen
  - Speicher wird vereinfacht als ein großes eindimensionales Feld aus Bytes betrachtet, das mit 0 beginnend adressiert wird



Adresse	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**
Daten	00101100	10101100	00001000	01100000	00111100	10000011	11000010	01001000	…

  - Adressen aufeinanderfolgender Wörter unterscheiden sich um **4**
  - In MIPS müssen Wörter bei Adressen beginnen, die ein Vielfaches von 4 sind
    - **Ausrichtung an Wortgrenzen** (*alignment restriction*)

Daten laden und speichern

- Ladebefehl (`lw` = *load word*)

```
lw reg1, disp(reg2)
```

- Hierbei bezeichnet
  - `reg1` eines der 32 MIPS-Register als **Zielregister** für den Datentransfer
  - `reg2` eines der 32 MIPS-Register, das eine 32-Bit **Speicheradresse** enthält
  - `disp` (*displacement*) ein Offset (in Byte), der vor dem Zugriff zum Inhalt von `reg2` hinzuaddiert wird (Basis- oder Displacement-Adressierung)
- Speicherbefehl (`sw` = *store word*)

```
sw reg1, disp(reg2)
```

  - Hier ist `reg1` das **Quellregister**

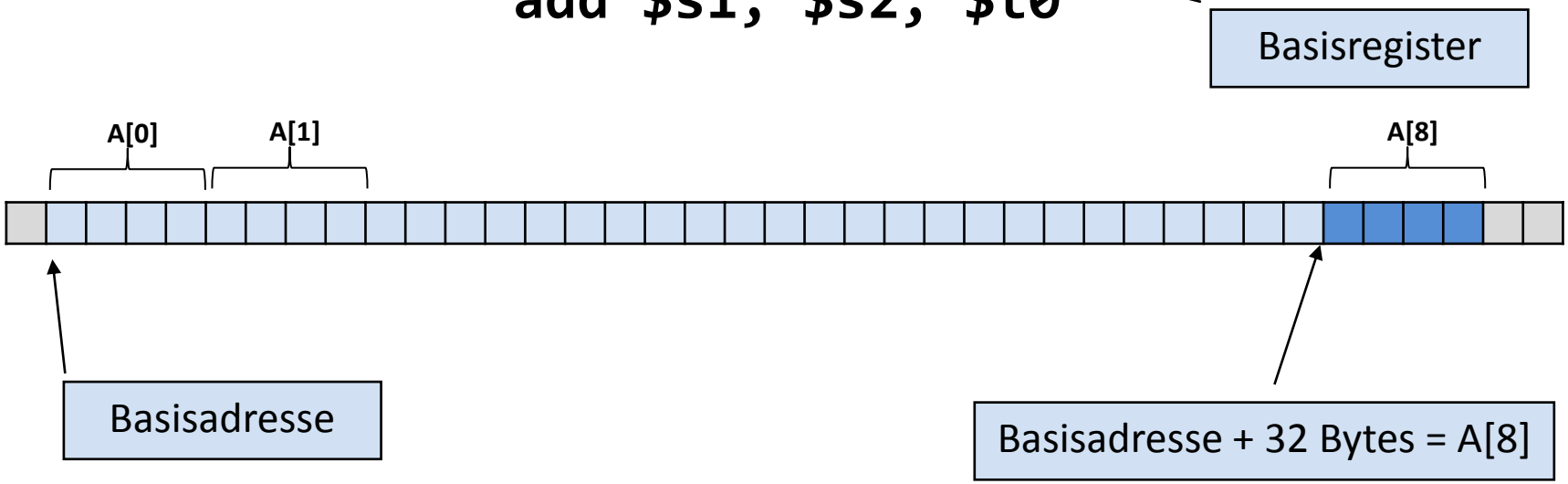
Beispiel 1 (Speicheroperanden)

- Beispiel (mit Arrayzugriff)

```
g = h + A[8];
```

  - g in $s1, h in $s2
  - Basisadresse (Anfang) von A in $s3
- MIPS-Code
  - Index 8 bedeutet Offset 32 (4 Bytes pro Wort)

```
lw  $t0, 32($s3)
add $s1, $s2, $t0
```



Beispiel 2 (Speicheroperanden)

- Beispiel (mit Arrayzugriff)

```
A[12] = h + A[8];
```

  - h in `$s2`, Basisadresse von A in `$s3`
- MIPS-Code

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

Register oder Hauptspeicher?

- Der Zugriff auf Register ist schneller
- Speicherzugriff erfordert immer Lade- bzw. Speicherbefehle
  - Ein Datentransferbefehl kann nur für einen Operanden verwendet werden
  - Code wird länger und mehr Befehle müssen ausgeführt werden
- Compiler versucht die am häufigsten verwendeten Variablen in Registern zu halten
  - Der Rest wird im Hauptspeicher abgelegt (Registerauslagerung, *register spilling*) und nur bei Bedarf geladen
    - Was wiederum die Auslagerung bestimmter Registerwerte bewirkt
  - Effiziente Ausnutzung der Register ist eine wichtige Compiler-Aufgabe

Konstante oder Direktoperanden?

- Konstanten direkt angeben
  - Einfacher und schneller, als die Konstanten aus dem Speicher zu laden
- Alle Befehle deren symbolischer Name auf „i“ endet
  - i steht für *immediate operand*
  - Beispiel für die Addition

```
addi reg2, reg1, const
```

- Dabei bezeichnet
  - **`reg2`** das Zielregister
  - **`reg1`** das Register mit dem ersten Operanden
  - **`const`** eine ganzzahlige positive oder negative 16-Bit Konstante
- Beispiel

```
addi $t1, $t0, 100
```

Entwurfsprinzip 3

Optimiere den häufig vorkommenden Fall

- Konstanten werden häufig als Operanden verwendet
  - Meist sind die Werte auch nicht sehr groß
- Direkte Verwendung in Befehlen ermöglicht schnellere Ausführung
  - Keine Ladebefehle

Konstante 0

- Register 0 (`$zero`) enthält fest den Wert 0
  - Kann nicht überschrieben werden
- Beispiel für Verwendung
  - Daten zwischen Registern verschieben (move-Befehl)

```
add $t2, $s1, $zero
```

  - Weitere alternative Form in MIPS

```
move $t2, $s1
```

    - Solch ein Befehl wird Pseudobefehl (*pseudo instruction*) genannt
    - Assembler übersetzt Pseudobefehle zuerst in richtige MIPS-Befehle und diese dann in Maschinensprache

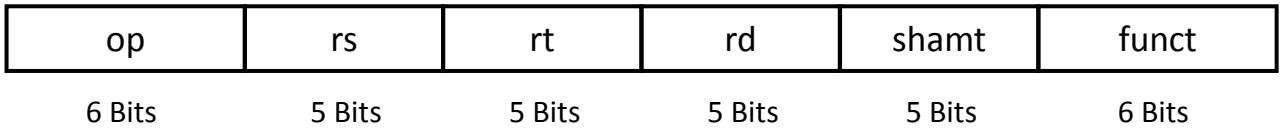
Vorzeichenerweiterung

- Wie konvertiert man eine Binärzahl, die n Bits hat, in eine Binärzahl mit mehr als n Bits?
  - Der Wert sollte gleich bleiben
- Das höchstwertige Bit (Vorzeichenbit) wird nach links repliziert
- Beispiel (Zweierkomplement)
  - 8-Bit Darstellung auf 16-Bit Darstellung
  - +3: 0000 0011 => 0000 0000 0000 0011
  - –3: 1111 1101 => 1111 1111 1111 1101
- Anwendung im MIPS-Befehlssatz
  - Beispiel: Befehl `addi`
    - Die Konstante hat im Befehl 16 Bits
    - Der Wert im Register hat 32 Bits
    - Bei der Addition müssen beide Werte 32 Bits haben
  - Weitere Beispiele folgen noch

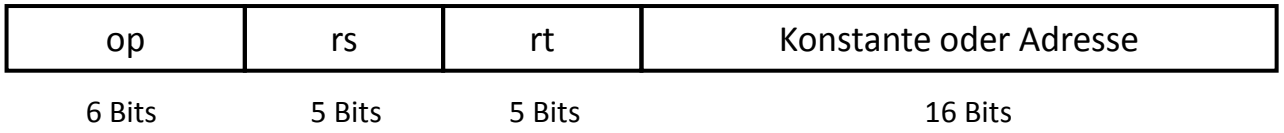
INSTRUKTIONSFORMATE

Darstellung der Befehle im Rechner

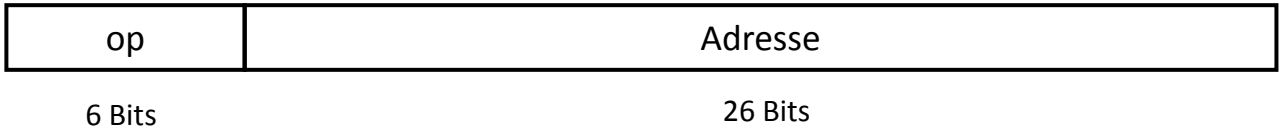
- MIPS-Befehle
  - Werden als 32-Bit Befehlsworte kodiert
- 3 verschiedene Formate für Befehle
  - R-Format



  - I-Format



  - J-Format

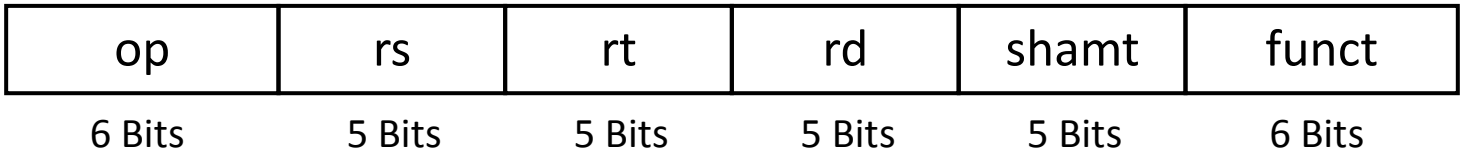


MIPS32 Register

Name	Nummer	Nutzung
`$zero`	0	Der konstante Wert 0
`$at`	*1*	*Für Assembler*
`$v0 - $v1`	2-3	Rückgabewerte (Unterprogramme) und zur Auswertung von Ausdrücken
`$a0 - $a3`	4-7	Argumente (Unterprogramme)
`$t0 - $t7`	8-15	Temporäre Variablen
`$s0 - $s7`	16-23	Gespeicherte Variablen
`$t8 - $t9`	24-25	Weitere temporäre Variablen
`$k0 - $k1`	*26-27*	*Für Betriebssystem reserviert*
`$gp`	28	Globaler Zeiger
`$sp`	29	Stackzeiger
`$fp`	30	Rahmenzeiger
`$ra`	31	Rücksprungadresse

R-Format

- Aufbau



- Das R-Format enthält 6 verschiedene Felder
  - Im Feld **op** (*operation code*, *opcode*) wird die Operation und das Format des Befehls kodiert
  - Das Feld **rs** (*first source register*) kodiert den ersten Quelloperanden (s = source)
  - Das Feld **rt** (*second source register*) kodiert den zweiten Quelloperanden
  - Das Feld **rd** (*destination register*) kodiert das Zielregister
  - Im Feld **shamt** (*shift amount*) wird für Schiebebefehle die Schiebedistanz angegeben
  - Das Feld **funct** (*function*) dient zur Auswahl einer bestimmten Variante einer Operation

Beispiel (R-Format)

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

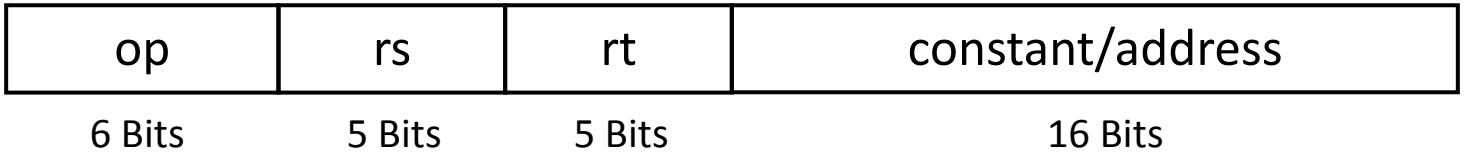
- **`add $t0, $s1, $s2`**

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

- 00000010001100100100000000100000 = 0x02324020

I-Format

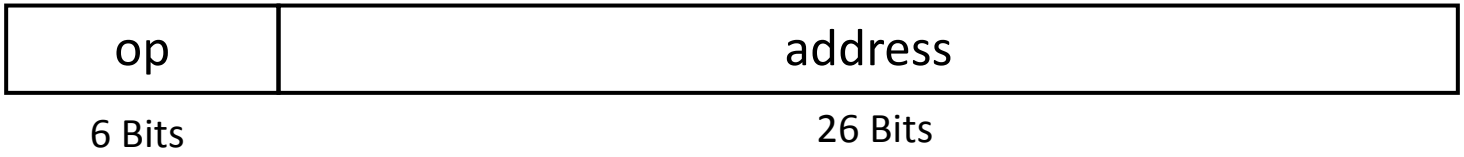
- Für Lade-/Speicherbefehle oder Befehle mit Kostanten *(immediate operand*) wird das I-Format verwendet



- Dabei bezeichnet
  - Das Feld **op** den Befehl
  - Das Feld **rs** das erste Register
  - Das Feld **rt** das zweite Register
  - Das Feld **constant/address** entweder einen konstanten (*immediate*) Operanden oder einen Offset
  - Für die Konstante stehen nur 16 Bit zur Verfügung
    - Konstanten sind nur im Bereich von –32768 bis 32767 möglich

J-Format

- Für Sprungbefehle existiert das J-Format



- Dabei bezeichnet
  - Das Feld **op** den Befehl
  - Das Feld **address** eine 26-Bit Adresse
- Das Feld address enthält keine vollständige 32-Bit Adresse
  - Wird nach einem bestimmten Schema auf 32 Bit gebracht

Entwurfsprinzip 4

Ein guter Entwurf erfordert gute Kompromisse

- Kompromisse bei MIPS
  - Alle Befehle haben dieselbe Länge (32 Bits)
  - Verschiedene Befehlsarten haben unterschiedliche Befehlsformate
    - Aber nur 3 Formate
  - 32 Register
    - 5 Bits pro Register im Befehlswort

Von-Neumann-Konzept (Wiederholung)

- Rechner von heute basieren auf zwei Grundprinzipien
  - Befehle werden in Form von Zahlen kodiert
  - Programme werden wie Zahlen im Hauptspeicher gespeichert, um gelesen und geschrieben zu werden
- Diese Prinzipien führen zum Von-Neumann-Konzept
  - Befehle werden wie Daten behandelt
  - Vereinfacht die Speicherhardware (einheitliche Sicht)
  - Programme können auf Programmen operieren
    - Compiler, Linker
  - Binärkompatibilität
    - Rechner mit derselben ISA „verstehen“ die gleichen fertigen Programme in Binärdarstellung

LOGISCHE OPERATIONEN

Logische Operationen

- Befehle für die bitweise Manipulation von Daten

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitweises UND	&	&	and, andi
Bitweises ODER	|	|	or, ori
Bitweises NICHT	~	~	nor

- Bei NOR
  - Erst wenn ein Operand 0 ist, dann entspricht der Befehl einem NICHT (ansonsten NOR)
- Können für das Extrahieren und Einfügen von Bit-Gruppen in ein MIPS-Wort benutzt werden

Schiebebefehle

- Verschieben eines Operanden nach links oder rechts um eine konstante Anzahl von Bitpositionen (Feld `shamt`)

Name	Beispiel	Notation in Pseudocode
`Shift Left Logical` (logischer Linksshift)	`sll $s0, $s1, 4`	`$s0 = $s1 << 4`
`Shift Right Logical` (logischer Rechtsshift)	`srl $s0, $s1, 10`	`$s0 = $s1 >> 10`

- Auswirkung
  - `sll` um i Bits entspricht einer Multiplikation mit 2^i
    - Beispiele:
      - 00000110 << 2 ergibt 00011000
      - 00000110 << 6 ergibt 10000000 (ein 1er fällt weg, Ergebnis nicht mehr korrekt)
  - `srl` um i Bits entspricht einer Division mit 2^i
    - Beispiele:
      - 00001100 >> 2 ergibt 00000011
      - 00001110 >> 2 ergibt 00000011 (ein 1er fällt weg, nur ganzzahliges Ergebnis)

UND-Verknüpfung, ODER-Verknüpfung

- UND-Verknüpfung für das Maskieren von Bits
  - Einige Bits auswählen, andere auf 0 setzen
- Beispiel

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

- ODER-Verknüpfung für das Setzen bestimmter Bits auf 1
- Beispiel

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NICHT-Verknüpfung

- Zum Invertieren der Bits
- MIPS bietet eine NOR-Operation mit 3 Operanden an
- Ein Operator 0 -> NICHT
- Beispiel

nor $t0, $t1, $zero

$zero	0000 0000 0000 0000 0000 0000 0000 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

32-Bit Konstanten

- Die meisten Konstanten sind sehr klein
  - 16 Bit (immediate) reichen aus
- Für größere Konstanten

`lui rt, constant`

  - Kopiert 16-Bit Konstante in die linken 16 Bits von `rt`
  - Setzt die rechten 16 Bits von `rt` auf 0
- Beispiel für Darstellung von 4000000

`lui $s0, 61`	`0000 0000 0011 1101`	`0000 0000 0000 0000`
`ori $s0, $s0, 2304`	`0000 0000 0011 1101`	`0000 1001 0000 0000`

VERZWEIGUNGEN

Verzweigung

- Bedingte Verzweigung in MIPS (ähnlich zu if-Anweisung)

```
beq reg1, reg2, label
bne reg1, reg2, label
```

- Dabei gilt
  - Ist bei `beq` (*branch on equal*) der Inhalt von `reg1` **gleich** dem Inhalt von `reg2`, so wird zur Sprungmarke `label` verzweigt
  - Ist bei `bne` (*branch on not equal*) der Inhalt von `reg1` **ungleich** dem Inhalt von `reg2`, so wird zur Sprungmarke `label` verzweigt
  - Ist die Bedingung nicht erfüllt, so wird nicht verzweigt und die Abarbeitung des Programms mit dem nachfolgenden Befehl fortgesetzt
- Unbedingte Verzweigung zu einer Sprungmarke

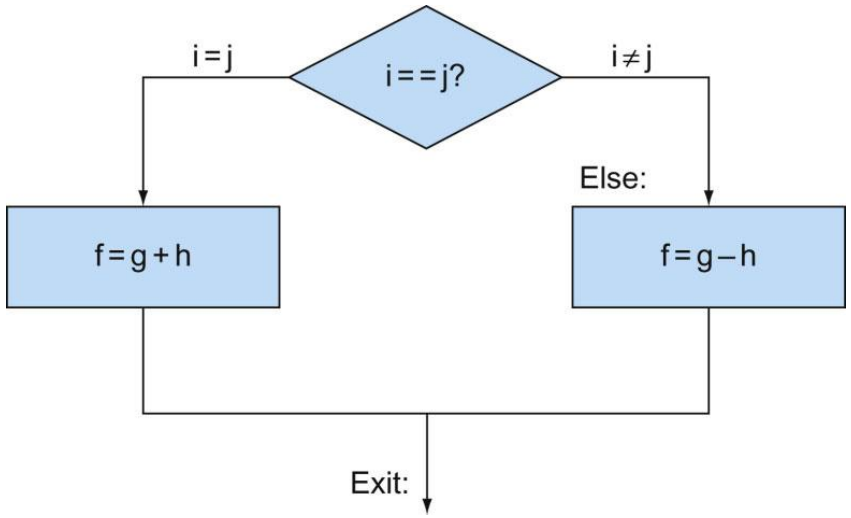
```
j label
```

Beispiel (if-Anweisung)

- Beispiel

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

  - f, g, h, i, j in $s0, $s1, $s2, $s3, $s4



- MIPS-Code (Fragment)

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

Label im Programm für Programmierer - Assembler berechnet dann die Adressen im Speicher

Beispiel (Schleifen)

- Beispiel

```
while (save[i] == k){
    i += 1;
}
```

  - i in $s3, k in $s5, Basisadresse von save in $s6

- MIPS-Code (Fragment)

```
        …
Loop:   sll  $t1, $s3, 2      # berechne 4 * i in $t1
        add  $t1, $t1, $s6    # Adresse von save[i] in $t1
        lw   $t0, 0($t1)      # lade save[i] in $t0
        bne  $t0, $s5, Exit   # Sprung bei save[i] ≠ k
        addi $s3, $s3, 1      # i = i + 1
        j    Loop             # unbedingter Sprung
Exit:   …
```

Weitere Befehle für Verzweigungen

- Befehle zum Überprüfen, ob ein Registerinhalt kleiner oder größer als der Inhalt eines anderen Registers ist

```
slt reg1, reg2, reg3
slti reg1, reg2, const
```

- Dabei gilt
  - Bei `slt` (*set on less than*) wird überprüft, ob der Inhalt von `reg2` kleiner als der Inhalt von `reg3` ist
  - Bei `slti` (*set on less than immediate*) wird überprüft, ob der Inhalt von `reg2` kleiner als die Konstante `const` ist
  - Es wird `reg1` auf 1 gesetzt, wenn Bedingung erfüllt ist, ansonsten auf 0
  - Es erfolgt ein Sprung erst durch einen folgenden `beq` oder `bne` Befehl (unter Benutzung von `reg1`)

Entwurfsprinzipien für Verzweigungen

- Warum enthält MIPS nicht gleich direkt Befehle für die Vergleiche auf der vorhergehenden Folie?
  - Z.B. blt, bge etc.?
- Hardware für <,≥, … ist viel komplexer als für =, ≠
  - Das würde bei allen Verzweigungen zu mehr Aufwand und einer geringeren Taktgeschwindigkeit führen
- beq und bne entsprechen dem häufig vorkommenden Fall
  - Siehe Entwurfsprinzip 3

Vorzeichenbehafteter und vorzeichenloser Vergleich

- Vorzeichenbehafteter Vergleich (*signed*): `slt`, `slti`
- Vorzeichenloser Vergleich (*unsigned*): `sltu`, `sltiu`
- Beispiel

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

slt  $t0, $s0, $s1  # -1 < +1                ⇒    $t0 = 1
sltu $t0, $s0, $s1  # +4294967295 > +1       ⇒    $t0 = 0
```

UNTERPROGRAMME

Unterprogramme

- Unterprogramme (Prozedur oder Funktion) stellen ein Hilfsmittel zur Strukturierung von Programmen dar
  - Wiederverwendung von Code
  - Parametrisierung
- Folgende Schritte sind beim Aufruf eines Unterprogramms erforderlich
  - Die Parameter werden an einer vereinbarten Stelle (Speicher oder Register) abgelegt
  - Ablaufsteuerung wird an das Unterprogramm (*callee*) übergeben
  - Im Unterprogramm muss Speicher für lokale Variablen bereitgestellt werden
  - Unterprogramm wird bis zum Ende ausgeführt
  - Ergebnis wird an einer Stelle abgelegt, auf die das aufrufende Unterprogramm (*caller*) zugreifen kann
  - Ablaufsteuerung muss an das aufrufende Unterprogramm an die Stelle unmittelbar nach dem Aufruf zurückgegeben werden

Registerbenutzung

- `$a0` – `$a3`: Argumente
- `$v0`, `$v1`: Rückgabewerte
- `$t0` – `$t9`: Temporäre Werte
  - Können vom aufgerufenen Unterprogramm überschrieben werden
- `$s0` – `$s7`: Lokale Variablen
  - Müssen vom aufgerufenen Unterprogramm gespeichert bzw. wieder hergestellt werden
- `$gp`: Globaler Zeiger auf statische Daten
- `$sp`: Stackzeiger
- `$fp`: Framezeiger
- `$ra`: Rücksprungadresse

Aufruf eines Unterprogramms

- MIPS bietet den Befehl `jal` (*jump and link*) an

```
jal proc
```

  - Realisiert den Aufruf eines Unterprogramms `proc`
  - Es wird zur Startadresse des Unterprogramms gesprungen
  - Die Rücksprungadresse (Stand des aktuellen Befehlszählers + 4) wird im Register `$ra` gespeichert
- Der Rücksprung aus einem Unterprogramm erfolgt mit

```
jr $ra
```

  - `jr` (*jump register*) bewirkt einen unbedingten Sprung zu der im Register `$ra` gespeicherten Adresse

Beispiel (einfach)

- Beispiel: Berechnung von | x2 – x1 | (x2 in $s2, x1 in $s1)

```
        or $a0,$zero,$s2    # x2 nach a0 (Parameter 1)
        or $a1,$zero,$s1    # x1 nach a1 (Parameter 2)
        jal absdi
        add …               # Nach Rücksprung hier weiter
        …
absdi:  sub $v0,$a0,$a1     # berechnet diff = x2–x1
        slti $t0,$v0,0      # ist diff < 0?
        beq $t0,$zero,ret   # Sprung falls positiv
        sub $v0,$zero,$v0   # berechnet –diff
ret:    jr $ra
        …
```

Unterprogramm

Stack

- Wenn das Unterprogramm weitere Register benötigt
  - Die Inhalte von Registern, die im aufrufenden Programm genutzt werden, müssen gerettet werden
- Die Inhalte einiger Register werden zu Beginn der Prozedur auf dem **Stack** (auch als **Keller** bezeichnet) gesichert
- Der Stack ist ein **LIFO Puffer** (*Last In First Out*) im Arbeitsspeicher eines Programms
  - Ein Stackzeiger (*SP*, *Stack Pointer*) zeigt auf den zuletzt reservierten Eintrag
  - Der Stack wächst von oben nach unten, d.h. von höherwertigen zu niedrigwertigen Adressen
- In MIPS ist **`$sp`** für den **Stackzeiger** vorgesehen

Beispiel (einfacher Aufruf, C-Code)

- C-Code für eine einfache Funktion, die z.B. aus dem Hauptprogramm aufgerufen werden kann
- Code

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

  - Argumente g, h, i, j in `$a0`, `$a1`, `$a2`, `$a3`
  - f (lokale Variable) in `$s0`
    - `$s0` muss am Stack gesichert werden, da es ja im aufrufenden Unterprogramm verwendet werden könnte (Konvention)
  - Resultat in `$v0`

Beispiel (einfacher Aufruf, MIPS-Code)

- MIPS-Code (Fragment) für das Unterprogramm

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

Speichere $s0 am Stack

Unterprogramm

Ergebnis

$s0 wiederherstellen

Zurückspringen

Geschachtelte Unterprogramme (Ausblick)

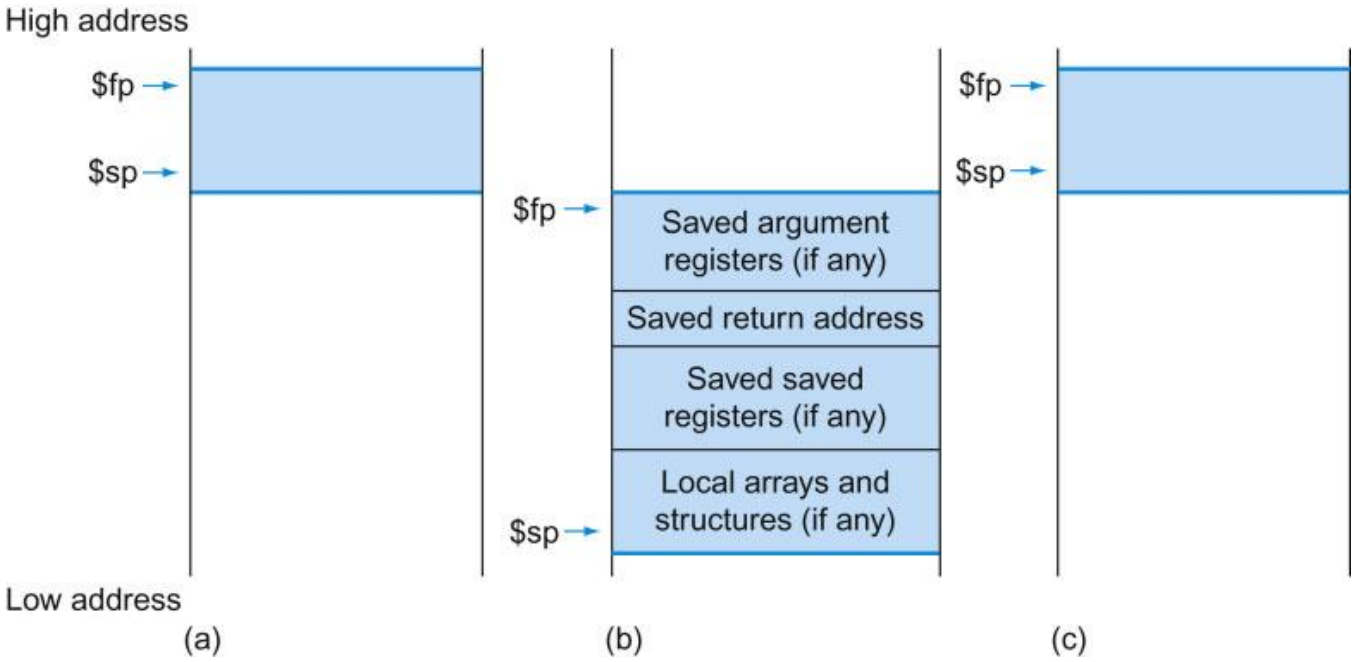
- Unterprogramme können auch wieder andere Unterprogramme aufrufen
  - Geschachtelte Aufrufe: A ruft B, B ruft C usw.
  - Rekursive Aufrufe
- In diesem Fall müssen gerettet werden
  - Die Rücksprungadresse
  - Argumente
  - Lokale Variablen
- Am Ende jedes Unterprogramms müssen die Werte vom Stack wieder hergestellt (geladen) werden

Lokale Daten am Stack (1)

- Stack beinhaltet bei einer Ausführung
  - Gerettete Werte
  - Lokale Variablen eines Unterprogramms, die nicht in Register passen
- Prozeduraufrufrahmen (*procedure call frame*)
  - Segment im Stack, das die geretteten Register und lokalen Variablen einer Prozedur (Unterprogramm) enthält
- Rahmenzeiger (*frame pointer*)
  - Zeigt auf das erste Wort im Prozedurrahmen
  - Ändert sich nicht während der Ausführung eines Unterprogramms
    - Stackzeiger kann sich ändern
  - Ist eine feste Basis für die Ermittlung lokaler Speicherreferenzen

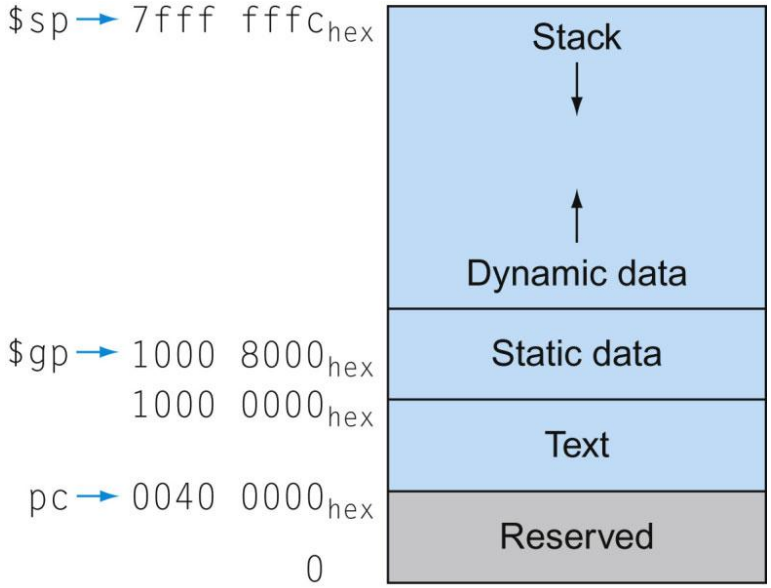
Lokale Daten am Stack (2)

- Stackzuordnung vor (a), während (b) und nach (c) einem Unterprogrammaufruf



Speicheraufteilung in MIPS

- Stack
  - Prozedurrahmen
- Dynamische Daten (*Dynamic data*)
  - Heap
  - Für dynamische Daten (new in Java)
- Statische Daten (*Static data*)
  - Globale Variablen
  - Statische Variablen
  - Konstanten
  - $gp wird so initialisiert, dass man mit ± einem Offset auf die Daten zugreifen kann
- Textsegment (*Text*)
  - Bereich für den MIPS-Maschinencode



ADRESSIERUNGSARTEN

Adressbildung bei Verzweigungen

- Bei Verzweigungen werden angegeben (I-Format)
  - Opcode
  - zwei Register
  - Zieladresse (16 Bit)
- Zieladressen sind nicht weit von der Adresse der Verzweigung entfernt
  - Vorwärts oder rückwärts (vom Befehlszähler aus gesehen)
- **Befehlszählerrelative Adressierung (*PC-relative addressing*)**
  - Addition eines Offsets zum Stand des aktualisierten Befehlszählers (PC + 4)
  - Die 16 Bits werden von der Hardware um 2 Bits nach links verschoben
    - Multiplikation mit 4 (Wortgröße)
    - Offset liegt daher im Bereich -2^{15} bis $+2^{15} - 1$ Wörter

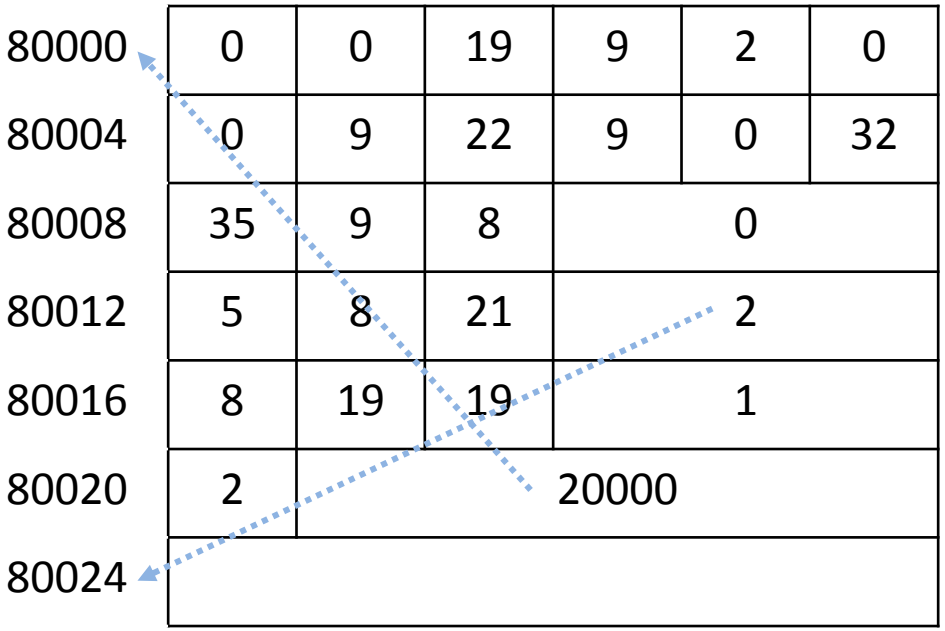
Adressbildung bei Sprüngen

- Bei Sprüngen (j und jal) wird das J-Format verwendet
  - Opcode (6 Bits)
  - Adresse (26 Bits)
- **Pseudodirekte Adressierung (*pseudodirect addressing*)**
  - Adresse bei Sprungbefehlen ergibt sich durch Konkatenation aus
    - Obere 4 Bits des Befehlszählers
    - 26 Bit Adresse und 2 Null-Bits (Multiplikation mit 4, d.h. 28 Bits)

Beispiel (Sprungadresse)

- Beispiel mit Schleife (Label an der Adresse 80000, dezimale Werte)

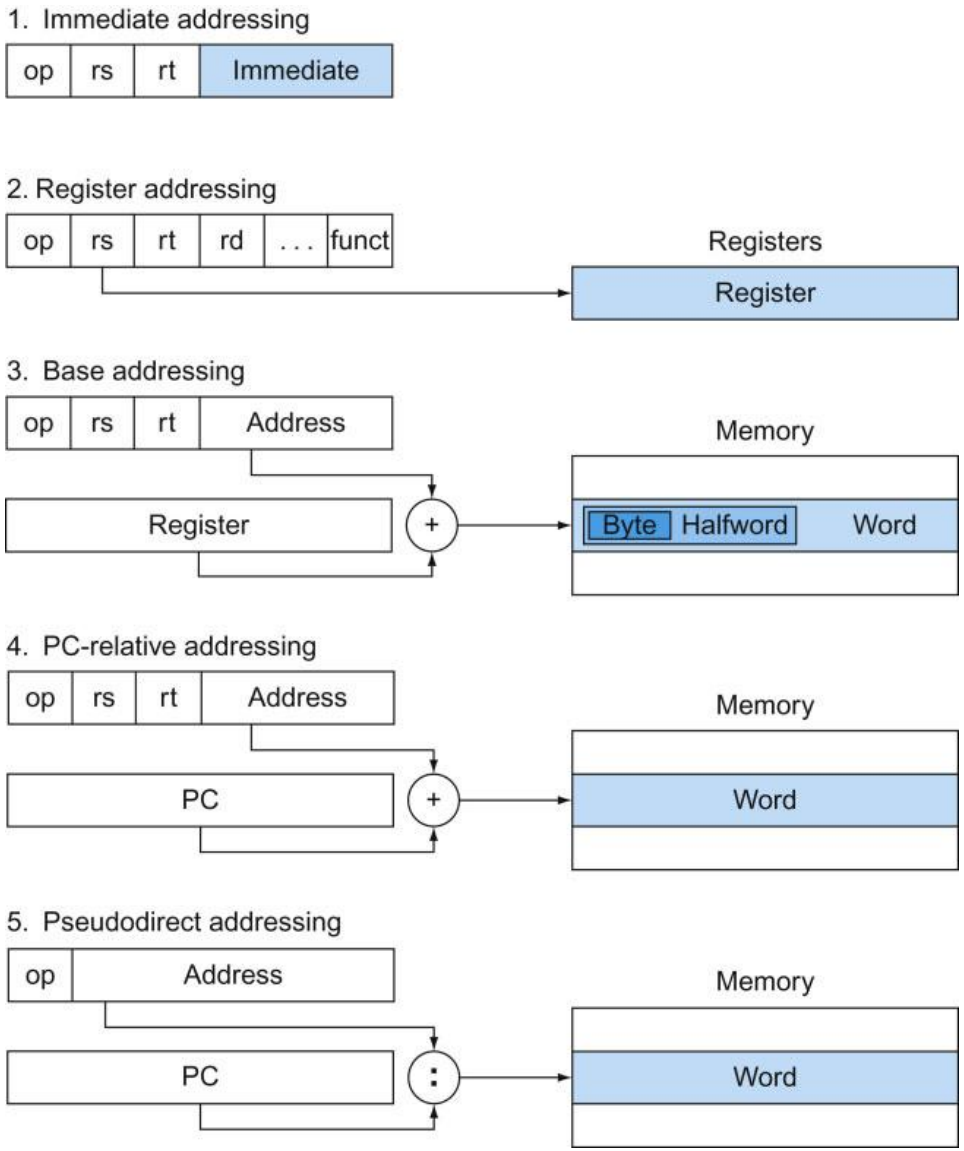
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



Arten der Adressierung in MIPS (Überblick)

1. **Direkte Adressierung (*Immediate addressing*)**
   - Operand ist eine Konstante, die direkt im Befehlswort kodiert ist
2. **Registeradressierung (*Register addressing*)**
   - Operand steht in einem Register
3. **Basis- oder Displacement-Adressierung (*Base addressing*)**
   - Operand steht im Speicher auf der Adresse, die sich aus der Addition eines Registerinhalts und des Displacements ergibt
4. **Befehlszählerrelative Adressierung (*PC-relative addressing*)**
   - Adresse ergibt sich durch Addition eines Offsets zum Stand des aktuellen Befehlszählers
5. **Pseudodirekte Adressierung (*Pseudodirect addressing*)**
   - Adresse bei Sprungbefehlen ergibt sich durch Konkatenation der oberen 4 Bit des Befehlszählers, des 26-Bit Felds des Befehls und 2 Null-Bits

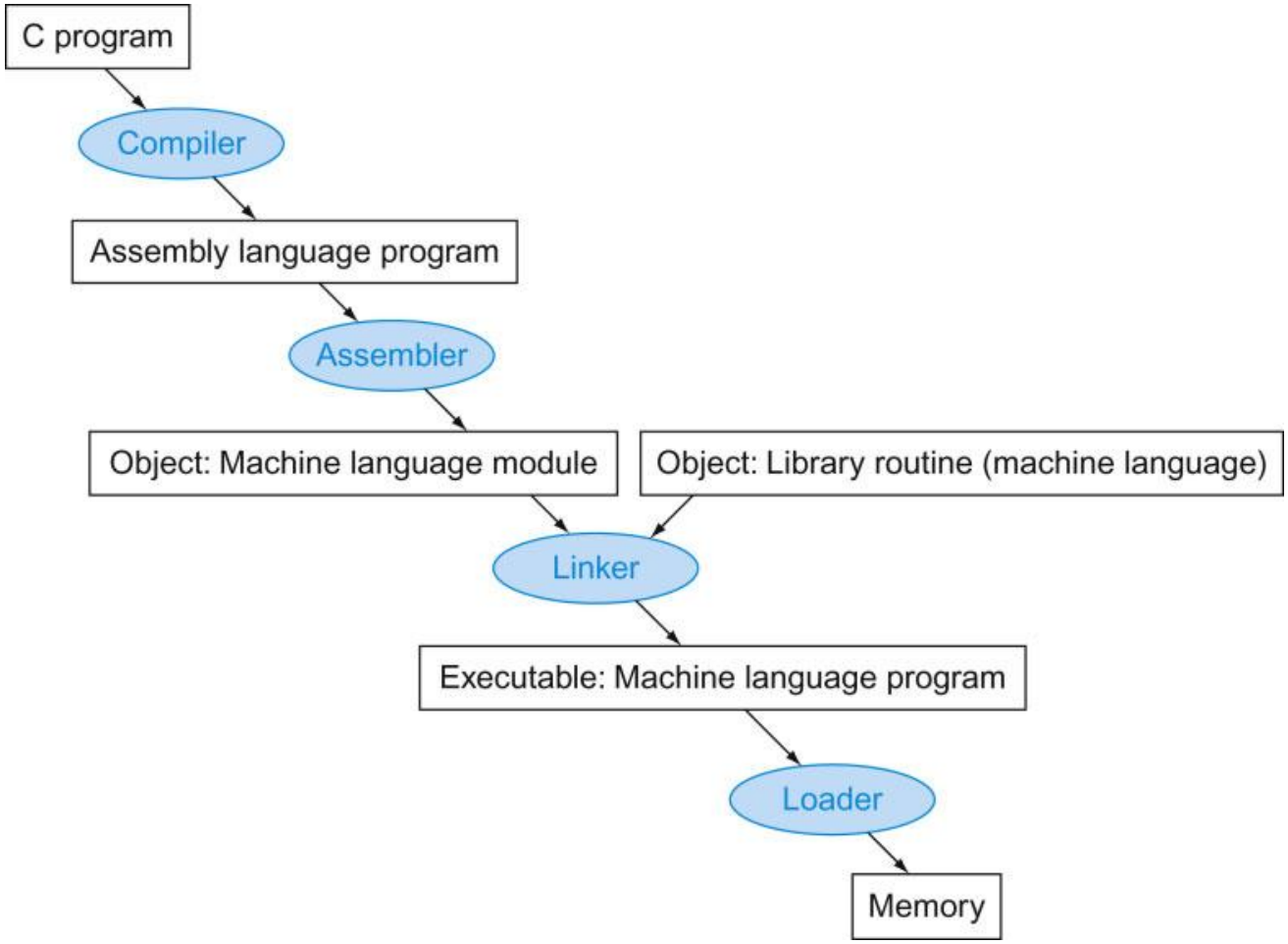
Arten der Adressierung in MIPS (2)



ÜBERSETZEN EINES PROGRAMMS

Übersetzen und Starten eines Programms

- Übersetzungshierarchie für ein C-Programm



Assembler (1)

- Erlaubt Verwendung von **symbolischen Namen**
- Realisiert eine Vielzahl von **Pseudobefehlen**
- Erlaubt unterschiedliche Zahlenformate
  - z.B. dezimal, binär, oktal, hexadezimal
- Unterstützt **Betriebssystemaufrufe** (*system calls*)
- Erzeugt eine Objektdatei mit unterschiedlichen Inhalten, wie z.B.
  - **Header** mit Informationen über Größe und Positionen der einzelnen Teile der Objektdatei
  - **Programmsegment** (**Textsegment**) mit binärem Maschinenprogramm
  - **Datensegment** mit statischen Daten
- Detailliertes Format der Objektdatei ist vom Betriebssystem abhängig

Assembler (2)

- Auswahl einiger **Direktiven** des MIPS Assemblers zur Steuerung der Assemblierung
  - **`.text`** markiert Beginn des Programmsegments
  - **`.data`** markiert Beginn des Datensegments
  - **`.word w1,w2,...,wn`** füllt Speicher mit den n 32-Bit Worten **`w1,w2,...,wn`**
  - **`.byte b1,b2,...,bn`** füllt Speicher byteweise mit den angegebenen Inhalten **`b1,b2,...,bn`**
  - **`.space num`** reserviert Speicher für **`num`** Bytes (nicht initialisiert)
  - **`.asciiz str`** stellt den String **`str`** in den Speicher (mit Terminierung durch Byte Null)

Assembler (3)

- Mögliche **Systemaufrufe** in MIPS Assembler bei Verwendung eines Simulators (Auswahl)

Code	Bezeichnung	Argumente und Wirkung
1	**print_int**	Gibt Inhalt von $a0 aus (Integer)
4	**print_string**	Gibt einen String (mit Terminierung durch Byte Null) ab Adresse $a0 aus
5	**read_int**	Liest in $v0 einen Wert ein
8	**read_string**	Liest ab Adresse $a0 einen String aus $a1 Zeichen ein
10	**exit**	Gibt Kontrolle an das Betriebssystem zurück

- Schema

```
li $v0, code        # Pseudobefehl "load immediate"
lw $a0, addr        # Argument laden, wenn benötigt
syscall
```

Beispiel (einfaches Assemblerprogramm)

```
        .data

x:      .word 12
y:      .word 14
z:      .word 18
res:    .space 4
str:    .asciiz " Fertig."

        .text

main:   lw $t0, x
        lw $t1, y
        lw $t2, z
        add $t0, $t0, $t1
        add $t0, $t0, $t2
        sw $t0, res
        li $v0, 1
        move $a0, $t0
        syscall
        li $v0, 4
        la $a0, str
        syscall
        li $v0, 10
        syscall
```

Daten (Datensegment)

Code (Textsegment)

Verwendete Pseudobefehle:
`lw` mit Marke (`lw $t0, x`)
`sw` mit Marke (`sw $t0, res`)
`li` (Laden einer Konstante)
`move` (Verschieben von Werten)
`la` (Laden von Adressen)

Gleitkommazahlen in MIPS (1)

- Einfache und doppelte Genauigkeit
  - Arithmetische Operationen (z.B. `add, sub, mul, div`) erweitert
    - mit Suffix **`.s`** (*single precision*)
    - mit Suffix **`.d`** (*double precision*)
- 32 zusätzliche Register für Gleitkommazahlen
  - **`$f0`** bis **`$f31`**
  - Bei einigen MIPS CPUs haben diese nur eine Breite von 32 Bit
    - Für doppelte Genauigkeit werden zwei Register verbunden
    - Daher sind dort nur `$f0, $f2, $f4`, ... zulässig
  - Beispiele:

```
add.s $f2,$f3,$f4
mul.s $f5,$f2,$f3
sub.d $f6,$f8,$f12
div.d $f6,$f6,$f14
```

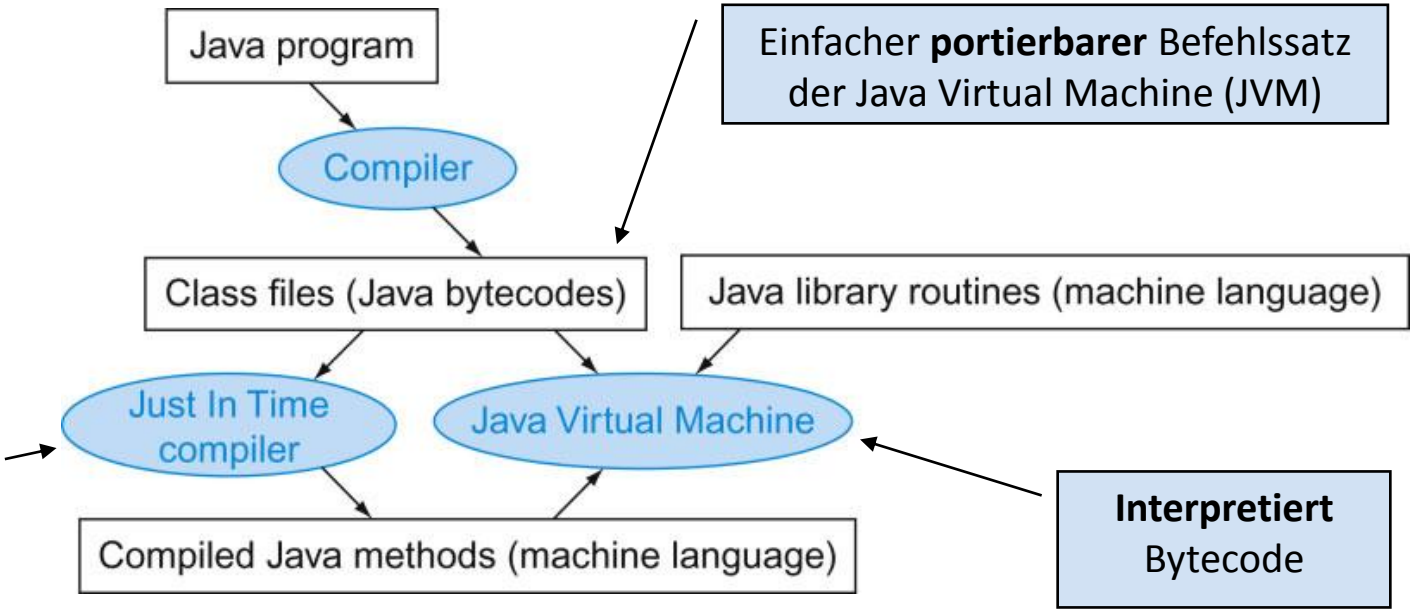
Gleitkommazahlen in MIPS (2)

- Zugriff auf Gleitkommazahlen
  - Assembler-Direktiven **`.float`** und **`.double`**
  - Laden und Speichern von 32-Bit Gleitkommazahlen mit **`lwc1`** bzw. **`swc1`** (Koprozessor c1)
  - Zusätzliche Pseudo-Instruktionen **`l.s`**, **`l.d`**, **`s.s`** und **`s.d`**

Linker und Loader

- Eine Objektdatei ist nicht ausführbar, da
  - sie noch keine absoluten Adressen enthält
  - sie häufig externe Referenzen enthält (zu Daten und Unterprogrammen in anderen Objektdateien oder Bibliotheken)
- Der **Linker** fügt mehrere Objektdateien zusammen und generiert ein **ausführbares Binärprogramm**
- Ausführung eines Programms durch den **Loader**
  - Einlesen des Headers, um die Größe des Daten- und Textsegmentes zu ermitteln
  - Bereitstellen von Arbeitsspeicher für Daten und Text
  - Laden von Daten und Text in den Arbeitsspeicher
  - Initialisieren einiger Register und des Stackzeigers
  - Aufruf einer Startprozedur, die das Hauptprogramm aufruft

Java



Einfacher **portierbarer** Befehlssatz der Java Virtual Machine (JVM)

Übersetzt Bytecode von oft benutzten Methoden in die Maschinensprache des Rechners

Interpretiert Bytecode

ENTWURF EINER ISA

RISC (*Reduced Instruction Set Computer*)

- RISC-Prozessoren
  - verfügen über einen vergleichsweise einfachen Befehlssatz
  - unterstützen meist nur elementare Adressierungsarten
- Konsequenzen
  - \+ RISC-Befehle können sehr effizient ausgeführt werden
  - \+ Einfacherer Aufbau der Prozessoren
    - \+ Kostenvorteile
    - \+ Geringer Verbrauch
  - – Code wird länger
- Beispiele
  - MIPS, ARM

CISC (*Complex Instruction Set Computer*)

- CISC-Prozessoren
  - besitzen einen umfangreichen Befehlssatz und Befehle sind mächtiger als bei RISC-Prozessoren
  - bieten viele Adressierungsarten an
  - besitzen nur wenige Register
- Konsequenzen
  - \+ Komplexe Aufgaben können mir einer geringeren Anzahl von Befehlen erledigt werden
  - \+ Code kann sehr kompakt geschrieben werden
  - – Einzelner Befehl benötigt viele Taktzyklen
  - – Umfangreiche Befehlssätze werden meist nur zu einem Bruchteil ausgenutzt
  - – Komplexe Prozessoren
- Beispiel
  - x86-Architektur von Intel

ARM und MIPS im Vergleich

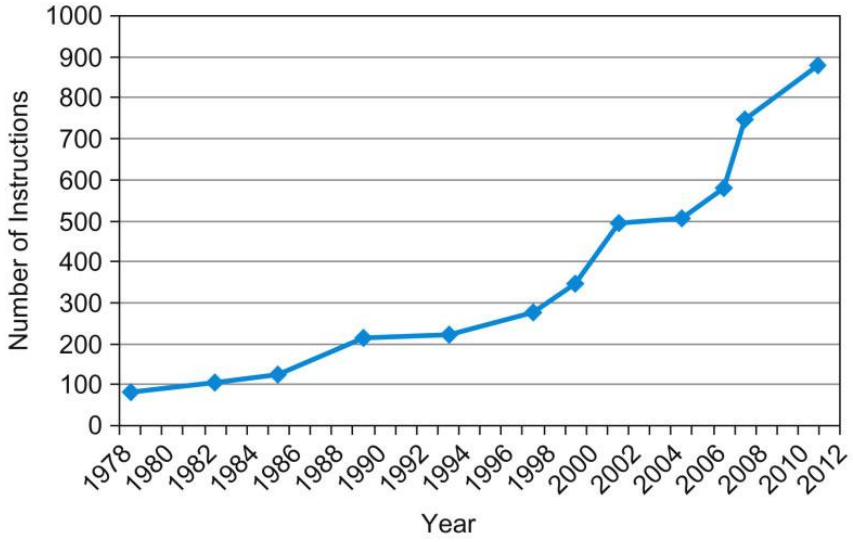
- ARM: Gebräuchlichste Befehlssatzarchitektur für Tablets und Smartphones
- Sehr ähnlich zu MIPS

	ARM	MIPS
Einführungsdatum	1985	1985
Befehlsgröße	32 Bit	32 Bit
Adressraum (Größe, Modell)	32-Bit flach	32-Bit flach
Datenausrichtung	Ausgerichtet	Ausgerichtet
Datenadressierungsmodi	9	3
Ganzzahlenregister (Anzahl, Modell, Größe)	15 GPR × 32-bit	32 GPR × 32-bit

  - GPR = General Purpose Register

Intel x86 ISA (1)

- Nach den Prozessoren der 8086/8088-Reihe benannt
  - 8086 ist ein 1978 eingeführter 16-Bit Prozessor
  - Mittlerweile etliche Generationen mit vielen Erweiterungen
- Generelle Eigenschaften
  - CISC-Befehlssatz mit variabler Instruktionslänge
  - Kompliziertere Speicheradressierung
  - Little-Endian-Architektur
  - Umfangreicher Befehlssatz (über die Jahre erweitert)



Intel x86 ISA (2)

- Heutige x86-Prozessoren sind hybride CISC/RISC-Prozessoren
- Hardware übersetzt Befehle in einfachere Mikrooperationen konstanter Länge
  - Einfach: 1 → 1
  - Komplex: 1 → viele Mikrooperationen
- Verarbeitung der Mikrobefehle erfolgt ähnlich dem RISC-Prinzip

ZUSAMMENFASSUNG

Zusammenfassung

- Entwurfsprinzipien und MIPS-Befehlssatz
  - Einfachheit begünstigt Regelmäßigkeit
  - Kleiner ist schneller
  - Optimiere den häufig vorkommenden Fall
  - Ein guter Entwurf erfordert gute Kompromisse
- Software/Hardware-Schichten
  - Compiler, Assembler, Hardware
- RISC und CISC
  - MIPS, ARM
  - x86

Fehlschluss 1

Leistungsfähigere Befehle bedeuten höhere Leistung

- Er werden weniger Befehle benötigt
- Komplexe Befehle sind aber komplizierter zu verwenden
  - Können möglicherweise zu langsameren Code führen
- Compiler sind sehr gut beim Erzeugen schneller Maschinenprogramme aus einfachen Befehlen

Fehlschluss 2

Programmieren in Assemblersprache erzielt die beste Leistung

- Setzt sehr gute Kenntnisse in Assembler voraus
- Moderne Compiler erzeugen meist besseren Code für moderne Prozessoren
- Assemblerprogramme haben mehr Codezeile
  - Mehr Möglichkeiten für Fehler
  - Geringere Produktivität

Literatur

- Grundliteratur
  - D. A. Patterson, J. L. Hennessy: **Computer Organization and Design**, 5. Auflage, Morgan Kaufmann, 2013 – Kapitel 2