# 5.B Maschinensprache und Assembler

- Die vom Prozessor *ausführbaren Befehle* liegen im Binärformat vor. Nur solche Befehle sind direkt ausführbar. So steht das Wort

  $$(02538820)_{16}$$

  für den *symbolischen Befehl*

  add $17, $18, $19 bzw. add $s1, $s2, $s3

- Während der Befehl $(02538820)_{16}$ für den Prozessor leicht "verständlich" ist, ist er für Menschen schwer lesbar. Für Menschen ist dagegen die Textzeile "add $s1, $s2, $s3" verständlicher.
  - Das "add" im Befehl wird *Mnemonic* genannt.
  - Die Registerbenennung kann in der gewünschten Form erfolgen.
  - Zusätzlich gibt es syntaktische Kennzeichnungen für verschiedene Adressierungsarten (siehe unten).

Zur Lösung dieses Problems verwendet man zur Programmierung auf Maschinenebene einen so genannten *Assembler*:

- Assembler lesen Texte zeilenweise und wandeln in der Regel jede Textzeile in einen Maschinenbefehl um.
- Ein typischer Befehl besteht aus einem Mnemonic (z.B. addi) und Operanden (z.B. $s1), letztere durch Kommata oder Blanks getrennt.
- Die Übersetzung von Assembler-Texten in Maschinenbefehle ähnelt der Compilierung bei höheren Programmiersprachen – der Prozess wird *Assemblierung* genannt. Nicht korrekt geformte Texte können Syntaxfehler enthalten, so dass kein Maschinenprogramm generiert werden kann.
- Es gibt eine 1-1-Übersetzung von Assembler-Befehlen zu Maschinen-Befehlen – man verfügt also bei weitem nicht über die Möglichkeiten einer höheren Programmiersprache.

Weitere Funktionen eines Assemblers:

- Hantieren mit *symbolischen Adressen*: Der Entwickler kann bestimmten Speicheradressen Namen (*Label*) zuordnen. Der Assembler erkennt diese Namen und fügt bei der Übersetzung die Originaladressen ein.
- Hantieren mit *symbolischen Sprungzielen*: Sprungbefehle führen den Programmablauf an einer anderen Stelle des Programms fort. Diese Stellen können symbolisch benannt werden. Bei relativen Sprüngen (z.B. "springe 100 Bytes nach vorne") berechnet der Assembler die Sprungdifferenz anhand des symbolischen Namens automatisch.
- Komfortables *Einfügen statischer Daten*: Neben den Maschinenbefehlen enthält ein Programm auch statische Bereiche, z.B. feste Zeichenketten oder numerische Konstanten. Über Assembler-Befehle können solche Datenbereiche komfortabel erstellt werden.
- Angabe von numerischen *Konstanten in verschiedenen Zahlensystemen*, z.B. "100" (dezimal) "0xFF" (hexadezimal).

- *Pseudobefehle*: häufig benutzte Befehle können durch eigene Befehlskürzel abgekürzt werden. So existiert in MIPS der Pseudobefehl li (load immediate), z.B. in

```
li $s1, 100              # Lade die Zahl 100 in das Register $s1
```

  der allerdings bei der Assemblierung in den Befehl

```
ori $s1, $zero, 100      # $s1 := 0 OR 100
```

  übersetzt wird.

  Der zweite Befehl ist inhaltsgleich aber für Menschen schwerer lesbar.

- *Makros* sind mehrzeilige Befehlssequenzen, die bei der Assemblierung textuell eingefügt werden. Das kann die Lesbarkeit erhöhen. Markos können auch parametrisiert werden, dürfen aber nicht mit Unterprogrammen verwechselt werden, da sie zur Assemblierzeit textuell eingefügt werden und keine Verwaltung zur Laufzeit erfordern.

## Ein MIPS-Beispielprogramm:

```
        .text                   # Schlüsselwort für den Anfang des Programmtexts
main:                           # Einstiegspunkt des Programms
    addi $s1, $zero, 20 # Lade 20 in das Register $s1 ($s1 := 0 + 20)
    sll $s1, $s1, 3         # Schiebe $s1 um 3 Bit nach links ($s1 := $s1 * 8)
    addi $s2, $s1, 10   # $s2 := $s1+10
```

Dieses Programm berechnet 20*8+10 und legt das Ergebnis in $*s*2 ab.

**Bemerkung:** Dieses Programm soll lediglich die Wirkungsweise von Maschinenbefehlen illustrieren. In der Realität würde man das Ergebnis der Rechnung (170) direkt dem Register $*s*2 zuweisen.

## 5.C Die MIPS-Befehle

Die MIPS-Befehle orientieren sich an folgenden *vier Design-Prinzipien*:

- *Simplicity favors regularity* ("Einfachheit bevorzugt Regelmäßigkeit"):
  - Nur eine Befehlslänge von 32 Bit
  - Nur drei Befehlsformate
  - Innerhalb eines Befehlsformats: feste Anzahl von Operanden
- *Smaller is faster* ("Kleiner ist schneller"):
  - Eine kleine Anzahl von Registern begünstigt eine schnelle Befehlsabarbeitung.

- *Good design demands compromise* ("Gutes Design erfordert einen Kompromiß"):
  - Einiges ist unbequem, z.B. immer exakt drei Operanden für bestimmte Befehle zu haben.
  - Einige wünschenswerte Befehle fehlen, z.B. Register-Kopien.
  - Ein Teil der Unbequemlichkeit wird durch Pseudo-Befehle behoben.
- *Make the common case fast* ("Mache den üblichen Fall schnell"):
  - Beispiel der Angabe von Konstanten: Für 16-Bit Konstanten ("der übliche Fall") kann MIPS dies in einem 32-Bit-Befehl ausdrücken. Für größere Konstanten ("der unübliche Fall") erfordert dies 2 Befehle.

**Bezeichnungen:**

- *Rd, Rs, Rt*: Register $0...$31 bzw. $*v*0 – $*v*1, $*a*0 – $*a*3, $*t*0 – $*t*7 etc.
- *I*: Direktoperand (immediate): Operand wird durch 16 Bit gegeben, die auf 32 Bit erweitert werden:
  Beispiel :1000 stellt den direkten Wert der Zahl 1000 dar.
- *Hi, Lo*: Multiplikations-, Divisionsergebnis
- *Label*: Sprungziel
- *shamt*: Feste Zahl von Schiebeschritten (vergleichbar mit I)
- *Address(Rs)*: Zugriff auf die Speicherzelle Mem[Address+Rs],
  Beispiel 1000($*s*1) für Zugriff auf Mem(1000+$*s*1)

Die Bedeutung der Spalte F (Format) wird weiter unten erklärt.

## Arithmetische Operationen:

| Befehl | Beschreibung | Vorzeichen | F |
|---|---|---|---|
| add Rd, Rs, Rt | Rd := Rs + Rt | mit | R |
| addu Rd, Rs, Rt | Rd := Rs + Rt | ohne | R |
| addi Rt, Rs, I | Rt := Rs + I | mit | I |
| addiu Rt, Rs, I | Rt := Rs + I | ohne | I |
| sub Rd, Rs, Rt | Rd := Rs – Rt | mit | R |
| subu Rd, Rs, Rt | Rd := Rs – Rt | ohne | R |
| div Rs, Rt | Lo := Rs/Rt, Hi := Rs mod Rt | mit | R |
| divu Rs, Rt | Lo := Rs/Rt, Hi := Rs mod Rt | ohne | R |
| mult Rs, Rt | (Hi, Lo) := Rs · Rt<br>Hi = oberen 32 Bits, Lo = unteren 32 Bits | mit | R |
| multu Rs, Rt | (Hi, Lo) := Rs · Rt<br>Hi = oberen 32 Bits, Lo = unteren 32 Bits | ohne | R |

# Bitweise logische Verknüpfungen:

| Befehl | Beschreibung | F |
|---|---|---|
| and Rd, Rs, Rt | $Rd := Rs \wedge Rt$ | R |
| andi Rt, Rs, I | $Rt := Rs \wedge I$ | I |
| nor Rd, Rs, Rt | $Rd := \overline{Rs \vee Rt}$ | R |
| or Rd, Rs, Rt | $Rd := Rs \vee Rt$ | R |
| ori Rt, Rs, I | $Rt := Rs \vee I$ | I |
| xor Rd, Rs, Rt | $Rd := Rs \oplus Rt$ | R |
| xori Rt, Rs, I | $Rt := Rs \oplus I$ | I |

## Schiebeoperationen:

| Befehl | Beschreibung | F |
|---|---|---|
| sll Rd, Rt, shamt | Rd := Rt links-geschoben um shamt Bits | R |
| sllv Rd, Rt, Rs | Rd := Rt links-geschoben um Rs Bits | R |
| srl Rd, Rt, shamt | Rd := Rt rechts-geschoben um shamt Bits | R |
| srlv Rd, Rt, Rs | Rd := Rt rechts-geschoben um Rs Bits | R |
| sra Rd, Rt, shamt | Rd := Rt rechts-geschoben um shamt Bits (arithmetisch, d.h. Erhaltung des Vorzeichens) | R |
| srav Rd, Rt, Rs | Rd := Rt rechts-geschoben um Rs Bits (arithmetisch, d.h. Erhaltung des Vorzeichens) | R |

# Vergleichen von Inhalten:

| Befehl | Beschreibung | Vorzeichen | F |
|---|---|---|---|
| slt Rd, Rs, Rt | Rd := 1 wenn Rs < Rt,<br>Rd := 0 wenn Rs ≥ Rt | mit | R |
| sltu Rd, Rs, Rt | Rd := 1 wenn Rs < Rt,<br>Rd := 0 wenn Rs ≥ Rt | ohne | R |
| slti Rt, Rs, I | Rt := 1 wenn Rs < I,<br>Rt := 0 wenn Rs ≥ I | mit | I |
| sltiu Rt, Rs, I | Rt := 1 wenn Rs < I,<br>Rt := 0 wenn Rs ≥ I | ohne | I |

## Laden und Speichern:

| Befehl | Beschreibung | F |
|---|---|---|
| mfhi Rd | Rd := Hi | R |
| mflo Rd | Rd := Lo | R |
| lui Rt, I | Rt := $I \cdot 2^{16}$<br>(oberste 16 Bit = I , unterste 16 Bit = 0) | I |
| lb Rt, Address(Rs) | Rt := Byte in Mem[Address + Rs]<br>(Vorzeichen auf 32 Bit erweitert) | I |
| lbu Rt, Address(Rs) | Rt := Byte in Mem[Address + Rs]<br>(keine Vorzeichenerweiterung) | I |
| sb Rt, Address(Rs) | Byte in Mem[Address + Rs] := Rt | I |
| lw Rt, Address(Rs) | Rt := Word in Mem[Address + Rs] | I |
| sw Rt, Address(Rs) | Word in Mem[Address + Rs] := Rt | I |

## Relative Sprünge (Branch):

| Befehl | Beschreibung | Vorzeichen | F |
|---|---|---|---|
| beq Rs, Rt, Label | Springe wenn Rs =Rt | egal | I |
| bne Rs, Rt, Label | Springe wenn Rs ≠ Rt | egal | I |
| bgez Rs, Label | Springe wenn Rs ≥ 0 | mit | I |
| bgtz Rs, Label | Springe wenn Rs > 0 | mit | I |
| blez Rs, Label | Springe wenn Rs ≤ 0 | mit | I |
| bltz Rs, Label | Springe wenn Rs < 0 | mit | I |

## Absolute Sprünge (Jump):

| Befehl | Beschreibung | F |
|---|---|---|
| j Label | Springe | J |
| jal Label | Unterprogrammaufruf | J |
| jr Rs | Springe zur Adresse in Rs | R |
| jalr Rs | Unterprogrammaufruf zur Adresse in Rs | R |

## Sonstige Befehle:

| Befehl | Beschreibung | F |
|---|---|---|
| syscall | Betriebssystemaufruf | R |
| break n | Exception n aufrufen | R |

## Die wichtigsten Pseudobefehle:

| Befehl | Beschreibung |
|---|---|
| li Rd, I | Rd := I |
| la Rd, Label | Rd := Adresse des Labels<br>(Achtung: nicht den Inhalt der Speicherzelle kopieren, sondern die Adresse der Speicherzelle) |
| move Rd, Rs | Rd := Rs |

## Adressierungsarten:

- Durch den Befehl wird die jeweilige *Adressierungsart* festgelegt, also die Art, wie auf einen Operanden zugegriffen wird, z.B.
  - add: Registeradressierung
  - addi: Direktoperand
- Nicht bei jeder Assemblersprache ist das so. Bei einigen Assemblersprachen gibt es *ein einziges Mnemonic* für *verschiedene Adressierungsarten*. Der Assembler erkennt dann syntaktisch die jeweilige Variante und fügt automatisch die verschiedenen Binärbefehle ein.

  Beispiel Z80-Assembler

```
ADD A, B       # Addiere A := A + B (A, B sind CPU-Register)
ADD A, 100     # Addiere A := A + 100 (addiere Direktoperand 100)
ADD A, (HL)    # Addiere A := A + Mem(HL) (HL ist ein Zeigerregister)
```

## Adressierungsarten des MIPS:

- *Direktwert* (immediate): eine Zahl *i*, die direkt im Befehl steht
- *Register*: ein Register, darstellt durch eine Registernummer r, z.B. $r = 17$ für $s1
- *Speicher*: Speicherinhalt einer Adresse *a*: Mem[*a*]
- *Register-indirekt*: eine Registernummer *r* definiert ein Register, das als Zeiger auf eine Speicheradresse benutzt wird: Mem[$*r*]
- *Register-indirekt mit Versatz*: eine Registernummer *r* definiert eine Zeigerregister, zu dem eine Adresse *a* hinzuaddiert wird: Mem[*a* + $*r*]

**Bemerkung:** MIPS stellt die Adressierungsarten *Speicher* und *Register-indirekt* immer als Register-indirekt mit Versatz dar:

- Speicher: Mem[*a* + $*zero*]
- Register-indirekt: Mem[0 + $*r*]