Musterlösung Blatt 7: Adressierung

Einleitung: Die Operanden eines Assembler-Befehls können in unterschiedlichen Formen angegeben werden. Operanden und Werte können beispielsweise **implizit** (auch **inhärent**) im Op-Code des Befehls enthalten sein. So arbeitet die Instruktion *jal* immer auf dem Register *$ra*, ohne dass dies expizit angegeben werden muss. Bei der **unmittelbaren** (**immediate**) Adressierung werden konstante Werte direkt in einem der Operandenfelder der Instruktion angegeben, z.B. *li $t0 42*. Im Weiteren besteht die Möglichkeit **Register**, sowie **Speicher** zu adressieren; dies kann **direkt** oder **indirekt** erfolgen (einstufige oder zweistufige Adressierung). Adressen können **absolut** oder **relativ** (Speicher-relativ oder Register-relativ) berechnet werden. Die **Indexregister-Adressierung** ist eine Form der Speicher-relativen Adressierung, da hier ein Registerinhalt als *Displacement* relativ zu einer in der Instruktion gegebenen Speicheradresse addiert wird. Zu den Register-relativen Adressierungsarten zählt die **Basisregisteradressierung** ggf. **mit Index**. Bei der relativen Adressierung wird die *effektive Adresse* durch eine **Summenbildung** der einzelnen Adressteile gebildet.
Nicht jede Architektur unterstützt alle oben genannten Adressierungsarten für jede Instruktion, insbesondere sind Speicherzugriffe bei Load/Store-Architekturen nur durch load- (lw) und store-Befehle (sw) möglich.

Aufgabe 1: Adressierung in der Praxis

In dieser Aufgabe sollen Sie die aus der Vorlesung bekannten Adressierungsarten für den Pentium und den PowerPC betrachten. Sie sollen dabei lernen, die Designentscheidungen bezüglich Adressierung auf den verschiedenen Architekturen nachzuvollziehen.

a) Aus welchen Adressteilen wird beim Pentium die effektive Adresse berechnet?

> Effektive Adresse =
> Segmentregister + Basisregister + (Indexregister * Skalierung) + Displacement
> z.B. `DS + EBX + (EAX * 4) + 5EH`

b) Welche Adressierungsarten kennt der PowerPC? Wie wird die effektive Adresse bestimmt?

> Direkt, Registerinhalt + Offset und Registerinhalt + Registerinhalt.
> Die effektive Adresse berechnet sich aus der Summe der einzelnen Adressteile.

c) Welche Befehle können beim PowerPC den Hauptspeicher adressieren?

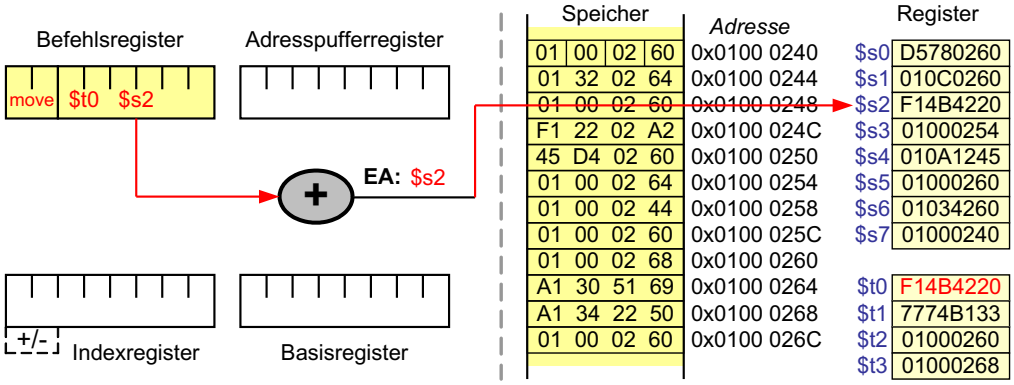
> Nur die load/store-Befehle haben genau eine Hauptspeicheradresse als Operand.

d) Vergleichen Sie die beiden Adressierungsarten. Welche Vor- und Nachteile beinhalten die beiden Adressierungsarten und deren Beschränkung auf bestimmte Befehle?

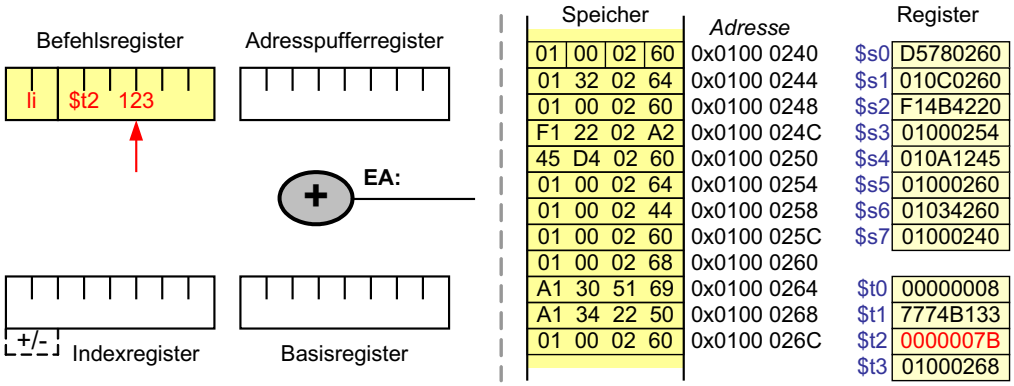
> Load/store Architekturen haben mehr Register, da alle Operationen außer load/store nur auf Registern definiert sind. Eine hohe Anzahl an Registern kostet viel Platz, beschleunigt dagegen die Ausführung. Da bei load/store Architekturen weniger Instruktionen zur Verfügung stehen ist das Steuerwerk kleiner, dafür sind im Allgemeinen mehr Assembler Instruktionen notwendig. Ziel einer Architektur ist es nun zwischen diesen Werten ein sinnvolles Trade-off zu finden.

Aufgabe 2: Berechnung der effektiven Adresse

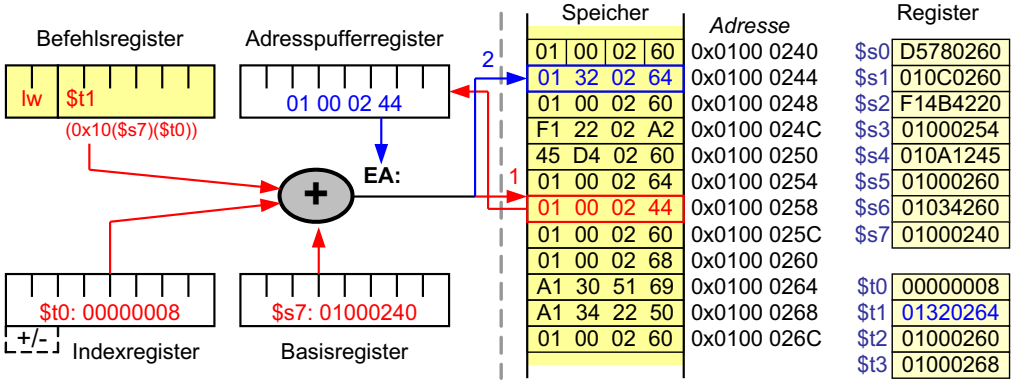
a) move $t0, $s2 Adressierungsart: Direkte Registeradressierung



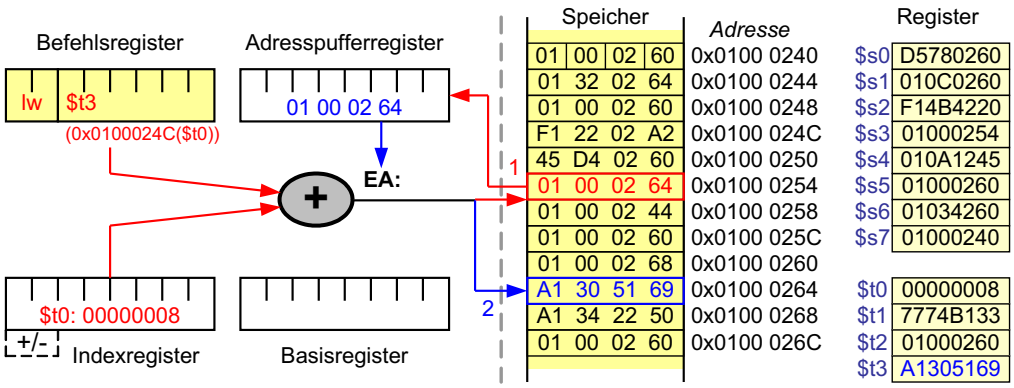
b) li $t2, 123 Adressierungsart: Immediate Adressierung



c) lw $t1 (0x10($s7)($t0)) Adressierungsart: Indirekte Basisregisteradressierung mit Index



d) lw $t3 (0x0100024C($t0)) Adressierungsart: Indirekte Indexregisteradressierung



Aufgabe 3: Adressierung am Beispiel von Datentypen

a) Assembler Code: Wie in Aufgabe 1) bereits angesprochen, kann man hier im Vergleich leicht erkennt, dass der Code der Load/Store Architektur umfangreicher ist als bei der nicht Load/Store Architektur (hier als ein CISC angenommen). Dafür ist die Adressbildung in den einzelnen Befehlen des CISC komplexer.

load / store (MIPS)

```
        .data
        .align 2
txt:    .asciiz „Hallo Welt“
arr:    .space 40
length: .word 0x0A
arrP:   .word arr

        .text
        .globl main

main:   la $t0, txt
        lw $t2, arrP
        li $t3, 0x00
loop:   lbu $s1, ($t0)
        bez $t0, exit
        add $t4, $t2, $t3
        sw $s1, 00($t4)
        addi $t3, 4
        addi $t0, 1
        j loop
exit:
```

CISC

```
        .data
        .align 2
txt:    .asciiz „Hallo Welt“
arr:    .space 40
length: .word 0x0A
arrP:   .word arr

        .text
        .globl main

main:   li $t0, 0x00
loop:   lbu $s0, 0x0(txt)($t0)
        bez $s0, exit
        sw $s0, (0x0(arrP))($t0:4)
        inc $t0
        j loop
exit:
```