Grundlagen der Rechnerarchitektur

Relevante RISC-V Assemblerbefehle

(keine Garantie auf Vollständigkeit)

Load/Store

mv t1, t2	Move: Kopiert Wert aus Register t2 in Register t1
li t1, 100	Load Immediate: Setzt Register t1 auf 100
lw t1, 100(t2)	Load word (Register indirekt + displacement): Lädt Wert von Speicheradresse t2+100 und speichert Wert in t1
sw t1, 100(t2)	Store word: Speichert Wert aus t1 in den Hauptspeicher an Adresse t2+100
lb t1, 100(t2)	Load byte: Lädt 8 bit von Adresse t2+100 in Register t1
sb t1, 100(t2)	Store byte: Speichert letzte 8 bit aus t1 in den Hauptspeicher an Adresse t2+100
la t0, var	Lädt die (Anfangs-)Adresse der globalen Variable var (Datensegment) in das Register t0.

Jumps/Branches

j Label	Jump unconditionally: Springe zu Label (goto)
jr t1	Jump register unconditionally: Springe an Adresse, welche in t1 steht (z.B. *jr ra* für return in Funktionen)
jal Label	Jump and link: Setzt Befehlszähler (ra-Register) und springt zum Label
jalr t1	Jump and link register: Setzt Befehlszähler (ra-Register) und springt an Adresse, welche in t1 steht (z.B. für Funktionspointer)
beq t1, t2, Label	Branch if equal: Springe zu Label, falls t1 == t2
beqz t1, Label	Branch if equal zero: Springe zu Label, falls t1 == 0
bnez t1, Label	Branch not equal zero: Springe zu Label, falls t1 != 0
blt t1, t2, Label	Branch if less than: Springe zu Label, falls t1 < t2
bltz t1, Label	Branch if less than zero: Springe zu Label, falls t1 < 0
bgt t1, t2, Label	Branch if greater than: Springe zu Label, falls t1 > t2
bgtz t1, Label	Branch if greater than zero: Springe zu Label, falls t1 > 0
ecall	syscall (wird bestimmt durch Argumentregister, z.B. exit(EXIT_SUCCESS) ist a7 = 10; printf: a7 = 1)

Math. Operations

addi t1, t2, 100	t1 = t2 + 100	
add t1, t2, t3	t1 = t2 + t3	
subi t1, t2, 100	t1 = t2 - 100	
sub t1, t2, t3	t1 = t2 - t3	
mul t1, t2, t3	t1 = t2 * t3	
slli t1, t2, 10	t1 = t2 << 10	Denke daran, dass Multiplikationen mit Zweierpotenzen schneller über bitshifts nach links zu lösen sind!
srli t1, t2, 10	t1 = t2 >> 10	Denke daran, dass Divisionen mit Zweierpotenzen schneller über bitshifts nach rechts zu lösen sind!