1 I-Type

In der folgenden Tabelle gilt: $m = m_d(ea(c))$ wobei $ea(c) = rs(c) +_{32} sxtimm(c)$

opc	Mnemonic	Assembler-Syntax	d	Effect
Data Transfer				
100 000	lb	lb *rt rs imm*	1	rt = sxt(m)
100 001	lh	lh *rt rs imm*	2	rt = sxt(m)
100 011	lw	lw *rt rs imm*	4	rt = m
100 100	lbu	lbu *rt rs imm*	1	rt = 0^{24}m
100 101	lhu	lhu *rt rs imm*	2	rt = 0^{16}m
101 000	sb	sb *rt rs imm*	1	m = rt[7:0]
101 001	sh	sh *rt rs imm*	2	m = rt[15:0]
101 011	sw	sw *rt rs imm*	4	m = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000	addi	addi *rt rs imm*		rt = rs + sxt(imm)
001 001	addiu	addiu *rt rs imm*		rt = rs + sxt(imm)
001 010	slti	slti *rt rs imm*		rt = (rs < sxt(imm) ? 1 : 0)
001 011	sltui	sltui *rt rs imm*		rt = (rs < sxt(imm) ? 1 : 0)
001 100	andi	andi *rt rs imm*		rt = rs $\wedge$ zxt(imm)
001 101	ori	ori *rt rs imm*		rt = rs $\vee$ zxt(imm)
001 110	xori	xori *rt rs imm*		rt = rs $\oplus$ zxt(imm)
001 111	lui	lui *rt imm*		rt = imm0^{16}

opc	rt	Mnemonic	Assembler-Syntax	Effect
Branch				
000 001	00000	bltz	bltz *rs imm*	pc = pc + (rs < 0 ? imm00 : 4)
000 001	00001	bgez	bgez *rs imm*	pc = pc + (rs $\geq$ 0 ? imm00 : 4)
000 100		beq	beq *rs rt imm*	pc = pc + (rs = rt ? imm00 : 4)
000 101		bne	bne *rs rt imm*	pc = pc + (rs $\neq$ rt ? imm00 : 4)
000 110	00000	blez	blez *rs imm*	pc = pc + (rs $\leq$ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz *rs imm*	pc = pc + (rs > 0 ? imm00 : 4)

Bei $X \in \{lb, lh, lw, lbu, lhu, sb, sh, sw\}$ wurde in der Vorlesung Programmierung 2 folgende Assembler-Syntax eingeführt:

X *rt* $imm(rs)$

Dies ist als gleichwertig anzusehen, im Rahmen der Systemarchitektur-Vorlesung betrachten wir beide Schreibweisen als korrekt.

2 R-type

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 000	sll	sll *rd rt sa*	rd = sll(rt,sa)
000000	000 010	srl	srl *rd rt sa*	rd = srl(rt,sa)
000000	000 011	sra	sra *rd rt sa*	rd = sra(rt,sa)
000000	000 100	sllv	sllv *rd rt rs*	rd = sll(rt,rs)
000000	000 110	srlv	srlv *rd rt rs*	rd = srl(rt,rs)
000000	000 111	srav	srav *rd rt rs*	rd = sra(rt,rs)
Arithmetic, Logical Operation				
000000	100 000	add	add *rd rs rt*	rd = rs + rt
000000	100 001	addu	addu *rd rs rt*	rd = rs + rt
000000	100 010	sub	sub *rd rs rt*	rd = rs − rt
000000	100 011	subu	subu *rd rs rt*	rd = rs − rt
000000	100 100	and	and *rd rs rt*	rd = rs $\wedge$ rt
000000	100 101	or	or *rd rs rt*	rd = rs $\vee$ rt
000000	100 110	xor	xor *rd rs rt*	rd = rs $\oplus$ rt
000000	100 111	nor	nor *rd rs rt*	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt *rd rs rt*	rd = (rs < rt ? 1 : 0)
000000	101 011	sltu	sltu *rd rs rt*	rd = (rs < rt ? 1 : 0)
Jumps, System Call				
000000	001 000	jr	jr *rs*	pc = rs
000000	001 001	jalr	jalr *rd rs*	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call

Coprocessor Instructions					
opcode	rs	fun	Mnemonic	Assembler-Syntax	Effect
010000	10000	011 000	eret	eret	Exception Return
010000	00100		movg2s	movg2s *rd rt*	spr[rd] := gpr[rt]
010000	00000		movs2g	movs2g *rd rt*	gpr[rt] := spr[rd]

3 J-type

opc	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j *iindex*	pc = bin_{32}(pc+4)[31:28]iindex00
000 011	jal	jal *iindex*	R31 = pc + 4 pc = bin_{32}(pc+4)[31:28]iindex00

Assembler-Syntax

Ein Assembler übersetzt Programme geschrieben in Assembler-Syntax in Maschinencode. Bei der MIPS besteht der Maschinencode aus 4 Byte langen Instruktionswords in denen wie in der Vorlesung definiert Quell- und Zielregister, sowie Konstanten kodiert sind.

Wir halten im Folgenden ein paar Konventionen zu der in der Vorlesung verwendeten Assembler-Syntax fest:

Konstanten: Die Immediate-Konstante imm, sowie $iindex$ werden im Allgemeinen als Dezimalzahlen angegeben. Diese werden bei der Übersetzung Modulo 2^{16} in Binärzahlen konvertiert. Es ist jedoch auch möglich, durch Voranstellen des Präfixes $0b$, Konstanten in binärer Repräsentation anzugeben. Diese werden dann auf 16-Bit zero-extended.

Beispiel:

- 25 ergibt im Instruktionsword als Immediate-Konstante: $imm = bin_{16}(25 \textbf{ mod } 2^{16})$
- $0b0110$ ergibt als Immediate-Konstante $imm = 0^{12}0110 = zext(0110)$

Register: Wir beschreiben Quell- und Zielregister im Allgemeinen durch ihre *gpr*-Adresse in dezimaler Repräsentation. Zwecks besserer Lesbarkeit und zur leichteren Unterscheidung von Konstanten und Registern schreiben wir bisweilen auch Ri für Register i.

Beispiel:

- add 12 2 1 // addiert den Wert von Register 2 und 1, und speichert das Ergebnis in Register 12.
- addi $R12$ $R2$ 1 // addiert den Wert von Register 2 zur Immediate-Konstante 1 und speichert das Ergebnis in Register 12.