RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
`ADD rd, rs1, rs2`	Add	R	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	Subtract	R	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	Add immediate	I	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Set less than	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, imm12`	Set less than immediate	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Set less than unsigned	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Set less than immediate unsigned	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Load upper immediate	U	`rd ← imm20 << 12`
`AUIP rd, imm20`	Add upper immediate to PC	U	`rd ← PC + imm20 << 12`

Logical Operations

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

Load / Store Operations

Mnemonic	Instruction	Type	Description
`LD rd, imm12(rs1)`	Load doubleword	I	`rd ← mem[rs1 + imm12]`
`LW rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SD rs2, imm12(rs1)`	Store doubleword	S	`rs2 → mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → mem[rs1 + imm12]`

Branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 = rs2` `pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 ≠ rs2` `pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 ≥ rs2` `pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

32-bit instruction format

Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - 17 - Function arguments
a0 - a1 - Return value(s)

AVR and RISC-V Instructions

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	AVR	AVR Description
`ADD rd, rs1, rs2`	Add	`ADD rd, rs`	`rd ← rd + rs`
`SUB rd, rs1, rs2`	Subtract	`SUB rd, rs`	`rd ← rd - rs`
`ADDI rd, rs1, imm12`	Add immediate	`SUBI rd, -imm8`	`rd ← rd - (-rs)`
`ADDI rd, rs1, -imm12`	Subtract immediate	`SUBI rd, imm8`	`rd ← rd - imm8`
`LUI rd, imm20`	Load upper immediate	`LDI rd, imm8`	`rd ← imm8`
`AUIP rd, imm20`	Add upper immediate to PC	`LDI rd, imm8`	`rd ← imm8`

Logical Operations

Mnemonic	Instruction	AVR	AVR Description
`AND rd, rs1, rs2`	AND	`AND rd, rs`	`rd ← rd & rs`
`OR rd, rs1, rs2`	OR	`OR rd, rs`	`rd ← rd | rs`
`XOR rd, rs1, rs2`	XOR	`EOR rd, rs`	`rd ← rd ^ rs`
`ANDI rd, rs1, imm12`	AND immediate	`ANDI rd, imm8`	`rd ← rd & imm8`
`ORI rd, rs1, imm12`	OR immediate	`ORI rd, imm8`	`rd ← rd | imm8`
`XORI rd, rs1, imm12`	XOR immediate	`LDI rs, imm8` `EOR rd, rs`	
`SLL rd, rs1, rs2`	Shift left logical	`LSL rd`	`rd ← rd << 1`
`SRL rd, rs1, rs2`	Shift right logical	`LSR rd`	`rd ← rd >> 1`
`SRA rd, rs1, rs2`	Shift right arithmetic	`ASR rd`	`rd ← rd >> 1`
	Rotate Left through carry	`ROL rd`	`rd ← rd << 1` `rd[0] ← C, C ← rd[7]`
	Rotate Right through carry	`ROR rd`	`rd ← rd >> 1` `rd[7] ← C, C ← rd[0]`
	Swap nibbles	SWAP rd	`rd[3..0] ⟷ rd[7..4]`

AVR Register File

r0	r1	r2	r3	r4	r5	r6	r7
r8	r9	r10	r11	r12	r13	r14	r15
r16	r17	r18	r19	r20	r21	r22	r23
r24	r25	r26	r27	r28	r29	r30	r31

AVR Register File

r0	r1	r2	r3	r4	r5	r6	r7
r8	r9	r10	r11	r12	r13	r14	r15
r16	r17	r18	r19	r20	r21	r22	r23
r24	r25	XL	XH	YL	YH	ZL	ZH
		X		Y		Z	

Callee saved registers (subroutine must save)

Used for indirect addressing

Load / Store Operations

Mnemonic	Instruction	AVR	AVR Description
`LB rd, imm12(rs1)`	Load byte	`LDD rd, rs+imm6`	`rd ← mem[rs + imm6]`
`LB rd, 0(rs1)`	Load Indirect	`LD rd, rs`	`rd ← mem[rs]`
`LB rd, 0(r1)` `ADDI r1, r1, 1`	Load Indirect and post-increment	`LD rd, rs+`	`rd ← mem[rs], rs ← rs+1`
`ADDI r1, r1, -1` `LB rd, 0(r1)`	Load Indirect and pre-decrement	`LD rd, -rs`	`rs ← rs-1, rd ← mem[rs]`
`LI rd, imm12`	Load Immediate	`LDI rd, imm8`	`rd ← imm8`
`LUI r1, r1, imm20` `LB rd, imm12(r1)`	Load Direct from data space (32-bit)	`LDS rd, imm16`	`rd ← mem[imm16]`
`SB rs2, imm12(rs1)`	Store byte	`STD rd+imm6, rs`	`mem[rd + imm6] ← rs`
`SB rs2, 0(rs1)`	Store Indirect	`ST rd, rs`	`mem[rd] ← rs`
`SB r2, 0(r1)` `ADDI r1, r1, 1`	Store Indirect and post-increment	`ST rd+, rs`	`mem[rd] ← rs, rd ← rd+1`
`ADDI r1, r1, -1` `SB r2, 0(r1)`	Store Indirect and pre-decrement	`ST -rd, rs`	`rd ← rd-1, mem[rd] ← rs`
`LUI r1, r1, imm20` `SB r2, imm12(r1)`	Store Direct to data space (32-bit)	`STS imm16, rs`	`mem[imm16] ← rs`

Branching

Mnemonic	Instruction	AVR	AVR Description
`BEQ rs1, rs2, imm12`	Branch equal	`BREQ imm7`	`if Z == 1` `pc ← pc + imm7`
`BNE rs1, rs2, imm12`	Branch not equal	`BRNE imm7`	`if Z == 0` `pc ← pc + imm7`
`BGE rs1, rs2, imm12`	Branch greater than or equal	`BRGE imm7`	`if N ^ V == 0` `pc ← pc + imm7`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	`BRSH imm7`	`if C == 0` `pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	`BRLT imm7`	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	`BRLO imm7`	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JALR zero, imm12(zero)`	Jump	`JMP imm16`	`pc ← imm16`
`JAL zero, imm20`	Relative jump	`RJMP imm12`	`pc ← pc + imm12`
`JALR zero, imm12(rs1)`	Indirect jump	`IJMP`	`pc ← r31:r30`
`JAL rd, imm20`	Jump and link	`RCALL imm12`	`stack ← pc + 2` `pc ← pc + imm12`
`JALR rd, imm12(zero)`	Long call	`CALL imm16`	`stack ← pc + 2` `pc ← imm16`
`JALR rd, imm12(rs1)`	Jump and link register	`RET`	`pc ← stack`
`JALR rd, imm12(rs1)`	Jump and link register	`ICALL`	`stack ← pc + 2` `pc ← r31:r30`

16-bit RISC-V Compared with 16-bit AVR

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Instruction	RISC-V	Description	AVR	Description
ADD Immediate	`ADDI Rd, K`	`Rd ← Rd + K`	`SUBI Rd, -K`	`Rd ← Rd - (-K)`
ADD Immediate, scaled by 16 to SP	`ADDI16SP K`	`SP ← SP + 16 × K`		
ADD Immediate, scaled by 4 to SP	`ADDI4SPN K`	`SP ← SP + 4 × K`		
ADD	`ADD Rd, Rs2`	`Rd ← Rd + Rs2`	`ADD Rd, Rr`	
SUBstract	`SUB Rd, Rs2`	`Rd ← Rd - Rs2`	`SUB Rd, Rr`	`Rd ← Rd - Rr`

Logical Operations

Instruction	RISC-V	Description	AVR	Description
AND	`AND Rd, Rs2`	`Rd ← Rd & Rs2`	`AND Rd, Rs2`	`Rd ← Rd & Rr`
AND Immediate	`ANDI Rd, K`	`Rd ← Rd & K`	`ANDI Rd, K`	`Rd ← Rd & K`
OR	`OR Rd, Rs2`	`Rd ← Rd | Rs2`	`OR Rd, Rs2`	`Rd ← Rd | Re`
eXclusive OR	`XOR Rd, Rs2`	`Rd ← Rd ^ Rs2`	`EOR Rd, Rs2`	`Rd ← Rd ^ Re`
Shift Left Logical Immediate	`SLLI Rd, K`	`Rd ← Rd << K`	`LSL Rd`	`Rd ← Rd << 1`
Shift Right Logical Immediate	`SRLI Rd, K`	`Rd ← Rd >> K`	`LSR Rd`	`Rd ← Rd >> 1`
Shift Right Arithmetic Immediate	`SRAI Rd, K`	`Rd ← Rd >> K`	`ASR Rd`	`Rd ← Rd >> 1`

Branching

Instruction	RISC-V	Description	AVR	Description
Branch if EQual to Zero	`BEQZ Rs1, K`	`if Rs1 = 0` `  PC ← PC + K`	`TST Rr` `BEQZ K`	`if Rr = 0` `  PC ← PC + K`
Branch if Not Equal to Zero	`BNEZ Rs1, K`	`if Rs1 ≠ 0` `  PC ← PC + K`	`TST Rr` `BRNE K`	`if Rr ≠ 0` `  PC ← PC + K`
Jump	`J K`	`PC ← PC + K`	`RJMP K`	`PC ← PC + K + 1`
Jump Register	`JR Rs1`	`PC ← Rs1`	`IJMP`	`PC ← Z`
Jump And Link	`JAL K`	`Ra ← PC + 2` `PC ← PC + K`	`RCALL K`	`stack ← PC + 2` `PC ← PC + K`
Jump And Link Register	`JALR Rs1`	`Ra ← PC + 2` `PC ← Rs1`	`ICALL`	`stack ← PC + 2` `PC ← Z`

Load / Store Operations

Instruction	RISC-V	Description	AVR	Description
Load Word indirect with displacement	`LW Rd, K(Rs1)`	`Rd ← M[Rs1 + K]`	`LDD Rd, Y + K`	`Rd ← M[Y + K]`
Load indirect	`LW Rd, 0(Rs1)`	`Rd ← M[Rs1]`	`LD Rd, Y`	`Rd ← M[Y]`
Load Indirect and post-increment	`LW Rd, 0(Rs1)` `ADDI Rs1, 1`	`Rd ← M[Rs1]` `Rs1 ← Rs1 + 1`	`LD Rd, Y+`	`Rd ← M[Y]` `Y ← Y + 1`
Load Indirect and pre-decrement	`ADDI Rs1, -1` `LW Rd, 0(Rs1)`	`Rs1 ← Rs1 - 1` `Rd ← M[Rs1]`	`LD Rd, -Y`	`Y ← Y - 1` `Rd ← M[Y]`
Load Word, Stack Pointer relative	`LWSP Rd, K`	`Rd ← M[SP + K]`		
Load Immediate	`LI Rd, K`	`Rd ← K`	`LDI Rd, K`	`Rd ← K`
Load Upper Immediate	`LUI Rd, K`	`Rd ← K << 12`		
Move (copy register)	`MV Rd, Rs2`	`Rd ← Rs2`	`MOV Rd, Rr`	`Rd ← Rr`
Copy register pair			`MOVW Rd, Rr`	`Rd+1:Rd ← Rr+1:Rr`
Store Word indirect with displacement	`SW Rs2, K(Rs1)`	`Rs2 → M[Rs1 + K]`	`STD Y + K, Rr`	`M[Y + K] ← Rr`
Store indirect	`SW Rs2, 0(Rs1)`	`Rs2 → M[Rs1]`	`ST Y, Rr`	`M[Y] ← Rr`
Store Indirect and post-increment	`SW Rs2, 0(Rs1)` `ADDI Rs1, 1`	`Rs2 → M[Rs1]` `Rs1 ← Rs1 + 1`	`ST Y+, Rr`	`M[Y] ← Rr` `X ← X + 1`
Store Indirect and pre-decrement	`ADDI Rs1, -1` `SW Rs2, 0(Rs1)`	`Rs1 ← Rs1 - 1` `Rs2 → M[Rs1]`	`ST -Y, Rr`	`Y ← Y - 1` `M[Y] ← Rr`
Store Word, Stack Pointer relative	`SWSP Rs2, K`	`Rs2 → M[K]`		

Legend

Rd - Destination register

Rr, Rs1, Rs2 - Source registers

K - Constant

M - Memory

Y - One of the X, Y, Z registers on AVR

PC - Program Counter

SP - Stack Pointer

AVR Register File

r0	r1	r2	r3	r4	r5	r6	r7
r8	r9	r10	r11	r12	r13	r14	r15
r16	r17	r18	r19	r20	r21	r22	r23
r24	r25	XL	XH	YL	YH	ZL	ZH
		X		Y		Z	

Callee saved registers (subroutine must save)

Used for indirect addressing

AVR Register File

Callee saved registers
(subroutine must save)

r0	r1	r2	r3	r4	r5	r6	r7
r8	r9	r10	r11	r12	r13	r14	r15
r16	r17	r18	r19	r20	r21	r22	r23
r24	r25	XL	XH	YL	YH	ZL	ZH

X Y Z

Used for indirect addressing

RISC-V Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

RISC-V Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - 17 - Function arguments
a0 - a1 - Return value(s)