RV64i Base Integer Instructions

Opcode	Instruction	Fmt	Example		Description	Notes
`li`	load immediate	*	`li`	`a0, 2`	`addi a0, zero, 2`	*pseudo*
`la`	load address	*	`la`	`a0, symbol`	`a0 = symbol`	*pseudo*, 2 instr
`add`	add	R	`add`	`a0, a1, a2`	`a0 = a1 + a2`	
`sub`	subtract	R	`sub`	`a0, a1, a2`	`a0 = a1 - a2`	
`xor`	bitwise exclusive or	R	`xor`	`a0, a1, a2`	`a0 = a1 ^ a2`	
`or`	bitwise or	R	`or`	`a0, a1, a2`	`a0 = a1 | a2`	
`and`	bitwise and	R	`and`	`a0, a1, a2`	`a0 = a1 & a2`	
`sll`	shift left logical	R	`sll`	`a0, a1, a2`	`a0 = a1 << a2`	
`srl`	shift right logical	R	`srl`	`a0, a1, a2`	`a0 = a1 >> a2`	
`sra`	shift right arith*	R	`sra`	`a0, a1, a2`	`a0 = a1 >> a2`	sign-extends
`slt`	set less than	R	`slt`	`a0, a1, a2`	`a0 = (a1 < a2) ?  1 :  0`	
`sltu`	set less than (u)	R	`sltu`	`a0, a1, a2`	`a0 = (a1 < a2) ?  1 :  0`	unsigned
`addi`	add immediate	I	`addi`	`a0, a1, 2`	`a0 = a1 + 2`	
`xori`	xor immediate	I	`xori`	`a0, a1, 2`	`a0 = a1 ^ 2`	
`ori`	or immediate	I	`ori`	`a0, a1, 2`	`a0 = a1 | 2`	
`andi`	and immediate	I	`andi`	`a0, a1, 2`	`a0 = a1 & 2`	
`slli`	shift left logical imm	I	`slli`	`a0, a1, 2`	`a0 = a1 << 2`	
`srli`	shift right logical imm	I	`srli`	`a0, a1, 2`	`a0 = a1 >> 2`	
`srai`	shift right arith imm	I	`srai`	`a0, a1, 2`	`a0 = a1 >> 2`	sign-extends
`slti`	set less than imm	I	`slti`	`a0, a1, 2`	`a0 = (a1 < 2) ?  1 :  0`	
`sltiu`	set less than imm (u)	I	`sltiu`	`a0, a1, 2`	`a0 = (a1 < 2) ?  1 :  0`	unsigned
`mv`	move (copy)	*	`mv`	`a0, a1`	`addi a0, a1, 0`	*pseudo*
`neg`	2s-complement negation	*	`neg`	`a0, a1`	`sub a0, zero, a1`	*pseudo*
`not`	bitwise not	*	`not`	`a0, a1`	`xori a0, a1, -1`	*pseudo*
`lb`	load byte	I	`lb`	`a0, 1(a1)`	`a0 = M[a1+1] (8 bits)`	
`lh`	load half	I	`lh`	`a0, 2(a1)`	`a0 = M[a1+2] (16 bits)`	
`lw`	load word	I	`lw`	`a0, 4(a1)`	`a0 = M[a1+4] (32 bits)`	
`ld`	load double word	I	`ld`	`a0, 8(a1)`	`a0 = M[a1+8] (64 bits)`	
`lbu`	load byte (u)	I	`lbu`	`a0, 1(a1)`	`a0 = M[a1+1] (8 bits)`	zero-extends
`lhu`	load half (u)	I	`lhu`	`a0, 2(a1)`	`a0 = M[a1+2] (16 bits)`	zero-extends
`lwu`	load word (u)	I	`lwu`	`a0, 4(a1)`	`a0 = M[a1+4] (32 bits)`	zero-extends
`l{b|h|w|d}`	load global	*	`ld`	`a0, symbol`	`a0 = M[symbol]`	*pseudo*, 2 instr
`sb`	store byte	S	`sb`	`a0, 1(a1)`	`M[a1+1] = a0 (8 bits)`	
`sh`	store half	S	`sh`	`a0, 2(a1)`	`M[a1+2] = a0 (16 bits)`	
`sw`	store word	S	`sw`	`a0, 4(a1)`	`M[a1+4] = a0 (32 bits)`	
`sd`	store double word	S	`sd`	`a0, 8(a1)`	`M[a1+8] = a0 (64 bits)`	
`s{b|h|w|d}`	store global	*	`sd`	`a0, symbol, t0`	`M[symbol] = a0 (uses t0)`	*pseudo*, 2 instr
`beq`	branch if $=$	B	`beq`	`a0, a1, 2b`	`if (a0 == a1) goto 2b`	
`bne`	branch if $\neq$	B	`bne`	`a0, a1, 2f`	`if (a0 != a1) goto 2f`	
`blt`	branch if $<$	B	`blt`	`a0, a1, 2b`	`if (a0 <  a1) goto 2b`	
`ble`	branch if $\leq$	*	`ble`	`a0, a1, 2f`	`bge a1, a0, 2f`	*pseudo*
`bgt`	branch if $>$	*	`bgt`	`a0, a1, 2b`	`blt a1, a0, 2b`	*pseudo*
`bge`	branch if $\geq$	B	`bge`	`a0, a1, 2f`	`if (a0 >= a1) goto 2f`	
`bltu`	branch if $<$ (u)	B	`bltu`	`a0, a1, 2b`	`if (a0 <  a1) goto 2b`	unsigned
`bleu`	branch if $\leq$ (u)	*	`bleu`	`a0, a1, 2f`	`bgeu a1, a0, 2f`	unsigned, *pseudo*
`bgtu`	branch if $>$ (u)	*	`bgtu`	`a0, a1, 2b`	`bltu a1, a0, 2b`	unsigned, *pseudo*
`bgeu`	branch if $\geq$ (u)	B	`bgeu`	`a0, a1, 2f`	`if (a0 >= a1) goto 2f`	unsigned
`beqz`	branch if $= 0$	*	`beqz`	`a0, 2b`	`if (a0 == 0) goto 2b`	*pseudo*
`bnez`	branch if $\neq 0$	*	`bnez`	`a0, 2f`	`if (a0 != 0) goto 2f`	*pseudo*
`bltz`	branch if < 0	*	`bltz`	`a0, 2b`	`if (a0 < 0) goto 2b`	*pseudo*
`blez`	branch if ≤ 0	*	`blez`	`a0, 2f`	`if (a0 <= 0) goto 2f`	*pseudo*
`bgtz`	branch if > 0	*	`bgtz`	`a0, 2b`	`if (a0 > 0) goto 2b`	*pseudo*
`bgez`	branch if ≥ 0	*	`bgez`	`a0, 2f`	`if (a0 >= 0) goto 2f`	*pseudo*
`jal`	jump and link	J	`jal`	`ra, label`	`ra = pc+4; jump to label`	
`jalr`	jump and link reg	I	`jalr`	`ra, a1`	`ra = pc+4; jump to a1`	
`call`	call subroutine	*	`call`	`label`	`ra = pc+4; jump to label`	
`j`	jump	*	`j`	`label`	`jump to label`	
`lui`	load upper imm	U	`lui`	`a0, 1234`	`a0 = 1234 << 12`	
`auipc`	add upper imm to pc	U	`auipc`	`a0, 1234`	`a0 = pc + (1234 << 12)`	
`ecall`	environment call	I	`ecall`		`system call (calls the OS)`	
`ebreak`	environment break	I	`ebreak`		`break to debugger`	

RV64m Multiply Extension

Opcode	Instruction	Fmt	Example	Description	Notes
`mul`	multiply	R	`mul a0, a1, a2`	`a0 = a1 * a2`	
`mulh`	multiply high	R	`mulh a0, a1, a2`	`a0 = a1 * a2 (high bits)`	
`mulsu`	multiply high (s*u)	R	`mulsu a0, a1, a2`	`a0 = a1 * a2 (high bits)`	a1 signed, a2 unsigned
`mulu`	multiply high (u)	R	`mulu a0, a1, a2`	`a0 = a1 * a2 (high bits)`	unsigned
`div`	divide	R	`div a0, a1, a2`	`a0 = a1 / a2`	
`divu`	divide (u)	R	`divu a0, a1, a2`	`a0 = a1 / a2`	unsigned
`rem`	remainder	R	`rem a0, a1, a2`	`a0 = a1 % a2`	
`remu`	remainder (u)	R	`remu a0, a1, a2`	`a0 = a1 % a2`	unsigned

Registers

Register	ABI Name	Description	Saver
`x0`	`zero`	Zero constant	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0 / fp`	Saved / frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Fn args/return values	Caller
`x12-x17`	`a2-a7`	Fn args	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller

Core Instruction Formats