RISC-V Instruction Set Reference

Author: **ddonle.com**

RV32I/RV64I Base ISA with M, A, F/D, and Zicsr Extensions

RISC-V Operands

Name	Example	Comments
32 registers	`x0–x31`	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register `x0` is hardwired to 0.
2^{30} **memory words**	`Memory[0]`, `Memory[4]`, ..., `Memory[4,294,967,292]`	Accessed only by data transfer instructions. RISC-V uses byte addresses; sequential word accesses differ by 4. Holds data structures, arrays, and spilled registers.

RISC-V Assembly Language

Category	Instruction	Example	Meaning	Comments	ISA
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add	**RV32I/64I**
Arithmetic	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract	**RV32I/64I**
Arithmetic	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants	**RV32I/64I**
Set <	Set less than	`slt x5, x6, x7`	`x5=(x6<x7)?1:0`	Signed comparison; result 0 or 1	**RV32I/64I**
Set <	Set less than imm.	`slti x5, x6, 20`	`x5=(x6<20)?1:0`	Signed compare with constant	**RV32I/64I**
Set <	Set less than (u)	`sltu x5, x6, x7`	`x5=(x6<x7)?1:0`	Unsigned comparison	**RV32I/64I**
Set <	Set less than imm. (u)	`sltiu x5, x6, 20`	`x5=(x6<20)?1:0`	Unsigned compare with constant	**RV32I/64I**
Data Transfer	Load word	`lw x5, 40(x6)`	`x5=Memory[x6+40]`	Word from memory to register	**RV32I/64I**
Data Transfer	Load word, unsigned	`lwu x5, 40(x6)`	`x5=Memory[x6+40]`	Unsigned word from memory to register	**RV32I/64I**
Data Transfer	Store word	`sw x5, 40(x6)`	`Memory[x6+40]=x5`	Word from register to memory	**RV32I/64I**
Data Transfer	Load halfword	`lh x5, 40(x6)`	`x5=Memory[x6+40]`	Halfword from memory to register	**RV32I/64I**
Data Transfer	Load halfword (u)	`lhu x5, 40(x6)`	`x5=Memory[x6+40]`	Unsigned halfword to register	**RV32I/64I**
Data Transfer	Store halfword	`sh x5, 40(x6)`	`Memory[x6+40]=x5`	Halfword from register to memory	**RV32I/64I**
Data Transfer	Load byte	`lb x5, 40(x6)`	`x5=Memory[x6+40]`	Byte from memory to register	**RV32I/64I**
Data Transfer	Load byte, unsigned	`lbu x5, 40(x6)`	`x5=Memory[x6+40]`	Byte unsigned from memory to register	**RV32I/64I**
Data Transfer	Store byte	`sb x5, 40(x6)`	`Memory[x6+40]=x5`	Byte from register to memory	**RV32I/64I**
Data Transfer	Load reserved	`lr.d x5, (x6)`	`x5=Memory[x6]`	Load; 1st half of atomic swap	*A ext.*
Data Transfer	Store conditional	`sc.d x7,x5,(x6)`	`Mem[x6]=x5; x7=0/1`	Store; 2nd half of atomic swap	*A ext.*
Data Transfer	Load upper imm.	`lui x5, 0x12345`	`x5=0x12345000`	20-bit const shifted left 12 bits	**RV32I/64I**
Data Transfer	Add upper imm. to PC	`auipc x5, 0x12345`	`x5=PC+0x12345000`	PC-relative address construction	**RV32I/64I**
Logical	And	`and x5, x6, x7`	`x5=x6 & x7`	Three reg. operands; bit-by-bit AND	**RV32I/64I**

Category	Instruction	Example	Meaning	Comments	ISA
Logical	Inclusive or	`or x5, x6, x8`	`x5=x6 | x8`	Three reg. operands; bit-by-bit OR	**RV32I/64I**
Logical	Exclusive or	`xor x5, x6, x9`	`x5=x6 ^ x9`	Three reg. operands; bit-by-bit XOR	**RV32I/64I**
Logical	And immediate	`andi x5, x6, 20`	`x5=x6 & 20`	Bit-by-bit AND reg. with constant	**RV32I/64I**
Logical	Inclusive or imm.	`ori x5, x6, 20`	`x5=x6 | 20`	Bit-by-bit OR reg. with constant	**RV32I/64I**
Logical	Exclusive or imm.	`xori x5, x6, 20`	`x5=x6 ^ 20`	Bit-by-bit XOR reg. with constant	**RV32I/64I**
Shift	Shift left logical	`sll x5, x6, x7`	`x5=x6 << x7`	Shift left by register	**RV32I/64I**
Shift	Shift right logical	`srl x5, x6, x7`	`x5=x6 >> x7`	Logical shift right by register	**RV32I/64I**
Shift	Shift right arith.	`sra x5, x6, x7`	`x5=x6 >> x7`	Arithmetic shift right by register	**RV32I/64I**
Shift	Shift left logical imm.	`slli x5, x6, 3`	`x5=x6 << 3`	Shift left by immediate	**RV32I/64I**
Shift	Shift right logical imm.	`srli x5, x6, 3`	`x5=x6 >> 3`	Logical shift right by immediate	**RV32I/64I**
Shift	Shift right arith. imm.	`srai x5, x6, 3`	`x5=x6 >> 3`	Arithmetic shift right by immediate	**RV32I/64I**
Cond. Branch	Branch if equal	`beq x5,x6,100`	`if(x5==x6) PC+100`	PC-relative; branch if equal	**RV32I/64I**
Cond. Branch	Branch if not equal	`bne x5,x6,100`	`if(x5!=x6) PC+100`	PC-relative; branch if not equal	**RV32I/64I**
Cond. Branch	Branch if less than	`blt x5,x6,100`	`if(x5<x6) PC+100`	PC-relative; branch if less (signed)	**RV32I/64I**
Cond. Branch	Branch if ≥	`bge x5,x6,100`	`if(x5>=x6) PC+100`	PC-relative; branch if ≥ (signed)	**RV32I/64I**
Cond. Branch	Branch if less (u)	`bltu x5,x6,100`	`if(x5<x6) PC+100`	PC-relative; branch if less (unsigned)	**RV32I/64I**
Cond. Branch	Branch if ≥ (u)	`bgeu x5,x6,100`	`if(x5>=x6) PC+100`	PC-relative; branch if ≥ (unsigned)	**RV32I/64I**
Uncond. Branch	Jump and link	`jal x1, 100`	`x1=PC+4; PC+100`	PC-relative procedure call	**RV32I/64I**
Uncond. Branch	Jump and link reg.	`jalr x1, 100(x5)`	`x1=PC+4; x5+100`	Indirect call; procedure return	**RV32I/64I**
Memory Order	Fence	`fence`	—	Enforce memory access ordering	**RV32I/64I**
Memory Order	Fence instruction	`fence.i`	—	Synchronize instruction & data streams	**RV32I/64I**
Environment	Environment call	`ecall`	—	System call / supervisor request	**RV32I/64I**
Environment	Environment break	`ebreak`	—	Transfer control to debugger	**RV32I/64I**
Mult/Div.	Multiply	`mul x5,x6,x7`	`x5=x6*x7` (low 32b)	Lower 32 bits of product	*M ext.*
Mult/Div.	Multiply high	`mulh x5,x6,x7`	`x5=x6*x7` (high 32b)	Upper 32 bits, signed×signed	*M ext.*
Mult/Div.	Multiply high (s×u)	`mulhsu x5,x6,x7`	`x5=x6*x7` (high)	Upper 32 bits, signed×unsigned	*M ext.*
Mult/Div.	Multiply high (u)	`mulhu x5,x6,x7`	`x5=x6*x7` (high)	Upper bits, unsigned×unsigned	*M ext.*
Mult/Div.	Divide	`div x5,x6,x7`	`x5=x6/x7`	Signed integer division	*M ext.*
Mult/Div.	Divide unsigned	`divu x5,x6,x7`	`x5=x6/x7`	Unsigned integer division	*M ext.*
Mult/Div.	Remainder	`rem x5,x6,x7`	`x5=x6%x7`	Remainder of signed division	*M ext.*

Category	Instruction	Example	Meaning	Comments	ISA
Mult/Div.	Remainder unsigned	`remu x5,x6,x7`	`x5=x6%x7`	Remainder of unsigned division	*M ext.*
Atomic Ops	Atomic swap	`amoswap.d x5,x6,(x7)`	`x5=Mem[x7]; Mem[x7]=x6`	Atomic swap in memory	*A ext.*
Atomic Ops	Atomic add	`amoadd.d x5,x6,(x7)`	`x5=Mem[x7]; Mem[x7]+=x6`	Atomic add to memory	*A ext.*
Atomic Ops	Atomic and	`amoand.d x5,x6,(x7)`	`x5=Mem[x7]; Mem[x7]&=x6`	Atomic AND to memory	*A ext.*
Atomic Ops	Atomic or	`amoor.d x5,x6,(x7)`	`x5=Mem[x7]; Mem[x7]|=x6`	Atomic OR to memory	*A ext.*
Atomic Ops	Atomic xor	`amoxor.d x5,x6,(x7)`	`x5=Mem[x7]; Mem[x7]^=x6`	Atomic XOR to memory	*A ext.*
Atomic Ops	Atomic min/max	`amomin/max.d x5,(x7)`	—	Atomic signed/unsigned min or max	*A ext.*
Floating Point	FP load word	`flw f5, 40(x6)`	`f5=Memory[x6+40]`	Load 32-bit float from memory	*F/D ext.*
Floating Point	FP store word	`fsw f5, 40(x6)`	`Memory[x6+40]=f5`	Store 32-bit float to memory	*F/D ext.*
Floating Point	FP load double	`fld f5, 40(x6)`	`f5=Memory[x6+40]`	Load 64-bit double from memory	*F/D ext.*
Floating Point	FP store double	`fsd f5, 40(x6)`	`Memory[x6+40]=f5`	Store 64-bit double to memory	*F/D ext.*
Floating Point	FP add	`fadd.s f5,f6,f7`	`f5=f6+f7`	Single-precision FP add	*F/D ext.*
Floating Point	FP subtract	`fsub.s f5,f6,f7`	`f5=f6-f7`	Single-precision FP subtract	*F/D ext.*
Floating Point	FP multiply	`fmul.s f5,f6,f7`	`f5=f6*f7`	Single-precision FP multiply	*F/D ext.*
Floating Point	FP divide	`fdiv.s f5,f6,f7`	`f5=f6/f7`	Single-precision FP divide	*F/D ext.*
Floating Point	FP square root	`fsqrt.s f5,f6`	`f5=sqrt(f6)`	Single-precision FP square root	*F/D ext.*
Floating Point	FP compare equal	`feq.s x5,f6,f7`	`x5=(f6==f7)?1:0`	FP compare; result to integer reg	*F/D ext.*
Floating Point	FP compare less than	`flt.s x5,f6,f7`	`x5=(f6<f7)?1:0`	FP compare; result to integer reg	*F/D ext.*
Floating Point	FP int→float	`fcvt.s.w f5,x6`	`f5=(float)x6`	Convert integer to single-precision	*F/D ext.*
Floating Point	FP float→int	`fcvt.w.s x5,f6`	`x5=(int)f6`	Convert single-precision to integer	*F/D ext.*
CSR (Zicsr)	CSR read/write	`csrrw x5,csr,x6`	`x5=CSR; CSR=x6`	Atomic read/write CSR	*Zicsr*
CSR (Zicsr)	CSR read and set	`csrrs x5,csr,x6`	`x5=CSR; CSR|=x6`	Atomic read and set bits in CSR	*Zicsr*
CSR (Zicsr)	CSR read and clear	`csrrc x5,csr,x6`	`x5=CSR; CSR&=~x6`	Atomic read and clear bits	*Zicsr*
CSR (Zicsr)	CSR read/write imm.	`csrrwi x5,csr,5`	`x5=CSR; CSR=5`	Read/write CSR with immediate	*Zicsr*
CSR (Zicsr)	CSR read/set imm.	`csrrsi x5,csr,5`	`x5=CSR; CSR|=5`	Read and set CSR bits with immediate	*Zicsr*
CSR (Zicsr)	CSR read/clear imm.	`csrrci x5,csr,5`	`x5=CSR; CSR&=~5`	Read and clear CSR bits w/ imm.	*Zicsr*

ISA Key: RV32I/64I = Base integer ISA | *M ext.* = Integer Mult/Divide | *A ext.* = Atomic memory operations | *F/D ext.* = Single/double-precision floating point | *Zicsr* = Control and status register instructions

Note: `>>` is logical (`srl`/`srli`) or arithmetic (`sra`/`srai`) depending on variant. Double-precision FP uses `.d` suffix. Atomics support `.w` and `.d` variants.