RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

①

RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`add,addw`	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
`addi,addiw`	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
`and`	R	AND	R[rd] = R[rs1] & R[rs2]	
`andi`	I	AND Immediate	R[rd] = R[rs1] & imm	
`auipc`	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
`beq`	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
`bge`	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
`bgeu`	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
`blt`	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
`bltu`	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
`bne`	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
`csrrc`	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
`csrrci`	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
`csrrs`	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
`csrrsi`	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
`csrrw`	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
`csrrwi`	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
`ebreak`	I	Environment BREAK	Transfer control to debugger	
`ecall`	I	Environment CALL	Transfer control to operating system	
`fence`	I	Synch thread	Synchronizes threads	
`fence.i`	I	Synch Instr & Data	Synchronizes writes to instruction stream	
`jal`	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
`jalr`	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
`lb`	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
`lbu`	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
`ld`	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
`lh`	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
`lhu`	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
`lui`	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
`lw`	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
`lwu`	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
`or`	R	OR	R[rd] = R[rs1] | R[rs2]	
`ori`	I	OR Immediate	R[rd] = R[rs1] | imm	
`sb`	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
`sd`	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
`sh`	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
`sll,sllw`	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
`slli,slliw`	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
`slt`	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
`slti`	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
`sltiu`	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
`sltu`	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
`sra,sraw`	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
`srai,sraiw`	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
`srl,srlw`	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
`srli,srliw`	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
`sub,subw`	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
`sw`	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
`xor`	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
`xori`	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit regsiters
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F regsiter
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET

②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`mul,mulw`	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
`mulh`	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
`mulhsu`	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
`mulhu`	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
`div,divw`	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
`divu`	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
`rem,remw`	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
`remu,remuw`	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`fld,flw`	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
`fsd,fsw`	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
`fadd.s,fadd.d`	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
`fsub.s,fsub.d`	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
`fmul.s,fmul.d`	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
`fdiv.s,fdiv.d`	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
`fsqrt.s,fsqrt.d`	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
`fmadd.s,fmadd.d`	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
`fmsub.s,fmsub.d`	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
`fmnsub.s,fmnsub.d`	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
`fmnadd.s,fmnadd.d`	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
`fsgnj.s,fsgnj.d`	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
`fsgnjn.s,fsgnjn.d`	R	Negative SiGN source	F[rd] = { (!F[rs2]<63>), F[rs1]<62:0>}	7)
`fsgnjx.s,fsgnjx.d`	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
`fmin.s,fmin.d`	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
`fmax.s,fmax.d`	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
`feq.s,feq.d`	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
`flt.s,flt.d`	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
`fle.s,fle.d`	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
`fclass.s,fclass.d`	R	Classify Type	R[rd] = class(F[rs1])	7,8)
`fmv.s.x,fmv.d.x`	R	Move from Integer	F[rd] = R[rs1]	7)
`fmv.x.s,fmv.x.d`	R	Move to Integer	R[rd] = F[rs1]	7)
`fcvt.s.d`	R	Convert from DP to SP	F[rd] = single(F[rs1])	
`fcvt.d.s`	R	Convert from SP to DP	F[rd] = double(F[rs1])	
`fcvt.s.w,fcvt.d.w`	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
`fcvt.s.l,fcvt.d.l`	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
`fcvt.s.wu,fcvt.d.wu`	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
`fcvt.s.lu,fcvt.d.lu`	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
`fcvt.w.s,fcvt.w.d`	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
`fcvt.l.s,fcvt.l.d`	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
`fcvt.wu.s,fcvt.wu.d`	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
`fcvt.lu.s,fcvt.lu.d`	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
`beqz`	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	`beq`
`bnez`	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	`bne`
`fabs.s,fabs.d`	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	`fsgnx`
`fmv.s,fmv.d`	FP Move	F[rd] = F[rs1]	`fsgnj`
`fneg.s,fneg.d`	FP negate	F[rd] = –F[rs1]	`fsgnjn`
`j`	Jump	PC = {imm,1b'0}	`jal`
`jr`	Jump register	PC = R[rs1]	`jalr`
`la`	Load address	R[rd] = address	`auipc`
`li`	Load imm	R[rd] = imm	`addi`
`mv`	Move	R[rd] = R[rs1]	`addi`
`neg`	Negate	R[rd] = –R[rs1]	`sub`
`nop`	No operation	R[0] = R[0]	`addi`
`not`	Not	R[rd] = ~R[rs1]	`xori`
`ret`	Return	PC = R[1]	`jalr`
`seqz`	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	`sltiu`
`snez`	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	`sltu`

REGISTER NAME, USE, CALLING CONVENTION

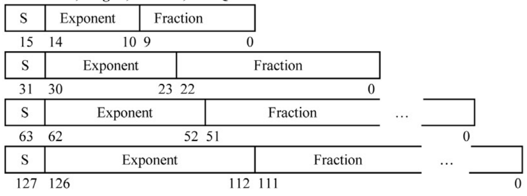
REGISTER	NAME	USE	SAVER
`x0`	`zero`	The constant value 0	N.A.
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	--
`x4`	`tp`	Thread pointer	--
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/Frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Function arguments/Return values	Caller
`x12-x17`	`a2-a7`	Function arguments	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller
`f0-f7`	`ft0-ft7`	FP Temporaries	Caller
`f8-f9`	`fs0-fs1`	FP Saved registers	Callee
`f10-f11`	`fa0-fa1`	FP Function arguments/Return values	Caller
`f12-f17`	`fa2-fa7`	FP Function arguments	Caller
`f18-f27`	`fs2-fs11`	FP Saved registers	Callee
`f28-f31`	`ft8-ft11`	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD ④

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

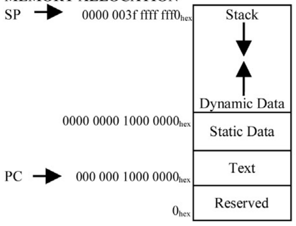
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



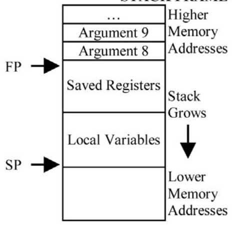
OPCODES IN NUMERICAL ORDER BY OPCODE ③

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
`lb`	I	0000011	000		03/0
`lh`	I	0000011	001		03/1
`lw`	I	0000011	010		03/2
`ld`	I	0000011	011		03/3
`lbu`	I	0000011	100		03/4
`lhu`	I	0000011	101		03/5
`lwu`	I	0000011	110		03/6
`fence`	I	0001111	000		0F/0
`fence.i`	I	0001111	001		0F/1
`addi`	I	0010011	000		13/0
`slli`	I	0010011	001	0000000	13/1/00
`slti`	I	0010011	010		13/2
`sltiu`	I	0010011	011		13/3
`xori`	I	0010011	100		13/4
`srli`	I	0010011	101	0000000	13/5/00
`srai`	I	0010011	101	0100000	13/5/20
`ori`	I	0010011	110		13/6
`andi`	I	0010011	111		13/7
`auipc`	U	0010111			17
`addiw`	I	0011011	000		1B/0
`slliw`	I	0011011	001	0000000	1B/1/00
`srliw`	I	0011011	101	0000000	1B/5/00
`sraiw`	I	0011011	101	0100000	1B/5/20
`sb`	S	0100011	000		23/0
`sh`	S	0100011	001		23/1
`sw`	S	0100011	010		23/2
`sd`	I	0100011	011		23/3
`add`	R	0110011	000	0000000	33/0/00
`sub`	R	0110011	000	0100000	33/0/20
`sll`	R	0110011	001	0000000	33/1/00
`slt`	R	0110011	010	0000000	33/2/00
`sltu`	R	0110011	011	0000000	33/3/00
`xor`	R	0110011	100	0000000	33/4/00
`srl`	R	0110011	101	0000000	33/5/00
`sra`	R	0110011	101	0100000	33/5/20
`or`	R	0110011	110	0000000	33/6/00
`and`	R	0110011	111	0000000	33/7/00
`lui`	U	0110111			37
`addw`	R	0111011	000	0000000	3B/0/00
`subw`	R	0111011	000	0100000	3B/0/20
`sllw`	R	0111011	001	0000000	3B/1/00
`srlw`	R	0111011	101	0000000	3B/5/00
`sraw`	R	0111011	101	0100000	3B/5/20
`beq`	SB	1100011	000		63/0
`bne`	SB	1100011	001		63/1
`blt`	SB	1100011	100		63/4
`bge`	SB	1100011	101		63/5
`bltu`	SB	1100011	110		63/6
`bgeu`	SB	1100011	111		63/7
`jalr`	I	1100111	000		67/0
`jal`	UJ	1101111			6F
`ecall`	I	1110011	000	000000000000	73/0/000
`ebreak`	I	1110011	000	000000000001	73/0/001
`CSRRW`	I	1110011	001		73/1
`CSRRS`	I	1110011	010		73/2
`CSRRC`	I	1110011	011		73/3
`CSRRWI`	I	1110011	101		73/5
`CSRRSI`	I	1110011	110		73/6
`CSRRCI`	I	1110011	111		73/7

MEMORY ALLOCATION



STACK FRAME



RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together