RISC-V Privileged Architecture (Version 1.12) Summary

(Excluding Hypervisor and Debug Features)

Table I.9 Control and Status Registers (CSRs)

Name	Address	h Address (RV32)	Size	Type	Description
Traps					
mstatus/sstatus	300/100	310	64/XLEN	MRW/SRW	Status
medeleg	302		XLEN	MRW	Exception delegation
mideleg	303		XLEN (12)	MRW	Interrupt delegation
mie/sie	304/104		XLEN (12)	MRW/SRW	Interrupt enable
mtvec/stvec	305/105		XLEN	MRW/SRW	Trap vector
mscratch/sscratch	340/140		XLEN	MRW/SRW	Scratch (for trap stack pointer)
mepc/sepc	341/141		XLEN	MRW/SRW	Exception program counter
mcause/scause	342/142		XLEN	MRW/SRW	Trap cause
mtval/stval	343/143		XLEN	MRW/SRW	Bad address or instruction
mip/sip	344/144		XLEN (12)	MRW/SRW	Interrupt pending
Virtual Memory					
satp	180		XLEN	SRW	Address translation & protection
Physical Memory Protection					
pmpcfg0/2/.../14	3A0/3A2/.../3AE	3A1/3A3/.../3AF	64 (8 x 8)	MRW	PMP configuration
pmpaddr0/1/.../63	380/381/.../3EF		XLEN	MRW	PMP address
Floating-Point					
fcsr/fflags/frm	003/001/002			URW	Floating-point CSR/flags/rounding
Counters / Performance Monitoring					
mcounteren/scounteren	306/106		32	MRW/SRW	Counter enable
mcountinhibit	320		32	MRW	Counter inhibit
mhpmevent3/4/.../31	323/324/.../33F		XLEN	MRW	Counter event selectors
mcycle/cycle	B00/C00	B80/C80	64	MRW/URO	Cycle counter
time	C01	C81	64	URO	Time (from CLINT)
minstret/instret	B02/C02	B82/C82	64	MRW/URO	Instructions retired
mhpmcounter3/4/.../31 hpmcounter3/4/.../31	B03/B04/.../B1F C03/C04/.../C1F	B83/B84/... C83/C84/...	64	MRW/ URO	Performance-monitoring counters
Machine Information Registers & Configuration					
misa	301		XLEN	MRW	Instruction set
menvcfg/senvcfg	30A/10A	31A/-	64/XLEN	MRW/SRW	Environment configuration
mseccfg	747	757	64	MRW	Security configuration
mvendorid	F11		32	MRO	JEDEC vendor ID
marchid	F12		XLEN	MRO	Vendor-specific architecture ID
mimpid	F13		XLEN	MRO	Vendor-specific implementation ID
mhartid	F14		XLEN	MRO	Hart ID
mconfigptr	F15		XLEN	MRO	Configuration pointer
Sstc Extension					
stimecmp	14D	15D	64	SRW	Supervisor timer compare
Zkr Extension					
seed	015		32	MRW	Seed to provide entropy for random number generator

{M/S/U}{RW/RO} = {Machine/Supervisor/User}{Read & Write/Read Only}. Gray registers are read only. For 64-bit registers in RV32, the bottom 32 are in the base register and the upper 32 are in a corresponding h register (e.g., mcycleh).

Table I.10 CSR bitfields

Register	XLEN-1	XLEN-2	XLEN-3	...	37	36	35	34	33	32	31:23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mstatus	SD				MBE	SBE	**SXL**		**UXL**			TSR	TW	TVM	MXR	SUM	MPRV	XS		FS		MPP		VS		SPP	MPIE	UBE	SPIE		MIE		SIE	
sstatus	SD								**UXL**						MXR	SUM		XS		FS				VS		SPP		UBE	SPIE				SIE	
misa	MXL												V	U		S		Q				M				I	H		F	E	D	C		A
menvcfg	STCE	PBMTE	ADUE																								CBZE	CBCFE	CBIE					FIOM
senvcfg																											CBZE	CBCFE	CBIE					FIOM
fcsr																											frm			NV	DZ	OF	UF	NX
{m/s}cause	INT																																	cause
medeleg																			See Table I.13															
mideleg																						MEI		SEI		MTI		STI		MSI		SSI		
mie																						MEIE		SEIE		MTIE		STIE		MSIE		SSIE		
mip																						MEIP		SEIP		MTIP		STIP		MSIP		SSIP		
sie																								SEIE				STIE				SSIE		
sip																								SEIP				STIP				SSIP		

Gray register bits are read only. For 64-bit registers in RV32, the bottom 32 are in the base register and the upper 32 are in a corresponding h register (e.g., mcycleh). RV32 mstatus and sstatus place SD in bit 31. The bolded fields are in RV64 only: mstatush drops the SXL and UXL fields, leaving only MBE and SBE as bits 5:4; sstatus has no UXL field, so no sstatush is needed.

Table I.11 CSR bitfield legend

Bit Field	Meaning	Description
SD	**S**tate **D**irty	1 = any of XS, FS, VS dirty
{M/S/U}BE	{**M**achine/**S**upervisor/**U**ser} **B**ig **E**ndian	1 = Big-endian data
{M/S/U}XL	{**M**achine/**S**upervisor/**U**ser} Base ISA **XL**EN	01= 32; 10 = 64; 11 = 128
TSR	**T**rap **SR**ET	For virtualization
TW	**T**imeout **W**ait	1 = Trap on WFI after timeout
TVM	**T**rap **V**irtual **M**emory	For virtualization
MXR	**M**ake e**X**ecutable **R**eadable	1 = Allow loads from executable pages
SUM	**S**upervisor **U**ser **M**emory	1 = Supervisor can access user memory
MPRV	**M**odify **PR**i**V**ilege	1 = Apply memory privileges from MPP
{X/F/V}S	{Custom(**X**)/**F**loating-point/**V**ector} **S**tate dirty	00 = off; 01 = initial; 10 = clean; 11 = dirty
{M/S}PP	{**M**achine/**S**upervisor} **P**revious **P**rivilege	00 = User; 01 = Supervisor; 11 = Machine
{M/S}PIE	{**M**achine/**S**upervisor} **P**revious **I**nterrupt **E**nable	Restore IE = PIE after MRET/SRET
{M/S}IE	{**M**achine/**S**upervisor} global **I**nterrupt **E**nable	0 = disable all interrupts
STCE	**S**upervisor **T**imer **C**ompare **E**nable	1 = enable Sstc extension (see Table I.9)
PBMTE	**P**age-**B**ased **M**emory **T**ypes **E**nable	1 = enable Svpbmt extension (see Section 11.2.2.1.1XREF)
ADUE	**A**ccess/**D**irty **U**pdate **E**nable	1 = enable Svadu extension (see Section 8.2.3.5XREF)
CB{Z/CF}E	**C**ache **B**lock {**Z**ero/**C**lean&**F**lush} **E**nable	1 = enable CBO instructions (see Table I.21)
CBIE	**C**ache **B**lock **I**nvalidate **E**nable	00 = disable; 01 = flush; 11 = invalidate
FIOM	**F**ence of **I/O** implies **M**emory	For virtualization
frm	**F**loating-point **R**ounding **M**ode	0 = RNE; 1 = RTZ; 2 = RDN; 3 = RUP; 4 = RMM; 7 = DYN
fflags {NV, DZ, OF, UF, NX}	{i**NV**alid, **D**ivide by **Z**ero, **O**ver**F**low, **U**nder**F**low, i**N**e**X**act}	Floating-point flags
INT	Trap is an **INT**errupt	1 = trap is an interrupt
cause	**Cause** of trap	See Table I.13
{M/S}{E/T/S}I{E/P}	{**M**achine/**S**upervisor}{**E**xternal/**T**imer/**S**oftware} **I**nterrupt{**E**nable/**P**ending}	MEIP set by PLIC MTIP set by CLINT when mtime > mtimecmp MSIP set by CLINT SEIP set by PLIC or mip.SEIP STIP set by mip.STIP or when mtime > stimecmp SSIP set by mip.SSIP

Table I.12 misa bitfields

Bit	Character	Description
0	A	Atomic
1	B	Bit manipulation (Zba+Zbb+Zbs)
2	C	Compressed
3	D	Double-precision floating-point
4	E	RV32E Embedded ISA
5	F	Single-precision floating-point
6	G	At least IMAFD supported
7	H	Hypervisor
8	I	RV32/64/128I integer ISA
12	M	Integer multiply/divide
16	Q	Quad-precision floating-point
18	S	Supervisor mode
20	U	User mode
21	V	Vector
XLEN-1:XLEN-2	MXL	XLEN: 01 = 32, 10 = 64, 11 = 128

Table I.13 cause interrupt and exception codes

Cause	Interrupt	Exception
0		Instruction address misaligned
1	Supervisor Software	Instruction access fault
2		Illegal instruction
3	Machine Software	Breakpoint
4		Load address misaligned
5	Supervisor Timer	Load access fault
6		Store/AMO address misaligned
7	Machine Timer	Store/AMO access fault
8		Environment call from U-mode
9	Supervisor External	Environment call from S-mode
10		
11	Machine External	Environment call from M-mode
12		Instruction page fault
13		Load page fault
14		
15		Store/AMO page fault

RISC-V Virtual Memory

Table I.14 satp bitfields

Field	Meaning	RV64	RV32
Mode	RV64: 0 = bare; 8 = SV39; 9 = SV48; 10 = SV57 RV32: 0 = bare; 1 = RV32	63:60	31
ASID	Address space ID	59:44	30:22
PPN	Physical page number of root page table	43:0	21:0

Table I.15 Virtual address bitfields

Format	63:57	56:48	47:39	38:32	31:30	29:22	21	20:12	11:0
SV32	n/a				VPN[1]		VPN[0]		Page Offset
SV39	extend			VPN[2]		VPN[1]		VPN[0]	
SV48	extend		VPN[3]						
SV57	extend	VPN[4]							

Table I.16 XWR encodings

X	W	R	Meaning
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	Reserved
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	Reserved
1	1	1	Read-write-execute page

Table I.17 Physical address bitfields

Format	55:48	47:39	38:34	33:30	29:22	21	20:12	11:0
SV32	n/a			PPN[1]		PPN[0]		Page Offset
SV39	PPN[2]				PPN[1]		PPN[0]	
SV48	PPN[3]		PPN[2]					
SV57	PPN[4]	PPN[3]						

Virtual addresses are 32 bits for RV32 and 64 bits for RV64.
Physical addresses are 34 bits for RV32 and 56 bits for RV64.

Table I.18 Page table entry bitfields

ISA	63	62:61	60:54	53:32	31:10	9:8	7	6	5	4	3	2	1	0
RV32	n/a				PPN	**RSW**: **R**eserved for **S**oft**w**are	**D**irty	**A**ccessed	**G**lobal	**U**ser	e**X**ecutable	**W**ritable	**R**eadable	**V**alid
RV64	**NAPOT**: 0 = normal 1 = 64 KiB huge page	**PBMT**: 0 = Use PMA 1 = Uncacheable Memory 2 = Uncacheable I/O	reserved	PPN										

RISC-V Physical Memory Protection

Table I.19 pmpaddr0-63 bitfields

ISA	63:54	53:32	31:0
RV32	n/a	n/a	33:2
RV64	0	55:34	33:2

Table I.20 pmpcfg0-15 bitfields

7	6:5	4:3	2	1	0
Locked	00	**A**lignment: 0 = off 1 = **TOR**: **T**op **O**f **R**ange 2 = **NA4**: **N**aturally **A**ligned **4**-byte 3 = **NAPOT**: **N**aturally **A**ligned **P**ower **O**f **T**wo	e**X**ecutable	**W**ritable	**R**eadable

Each pmpcfg register contains 8 (RV64) or 4 (RV32) copies of the 8-bit configuration fields shown in Table I.20, so each pmpcfg register supports up to 8 (RV64) or 4 (RV32) pmpaddr registers. To support all 64 pmpaddr registers, RV64 uses 64-bit pmpcfg0, 2, 8,... 14 and RV32 uses 32-bit pmpcfg0, 1, 2,...15.

RISC-V Cache Management

Table I.21 Zicbo*: Cache management operation instructions

op	funct3	$imm_{11:0}$	Registers	Type	Instruction	Description	Operation
Zicbom: Cache Block Management							
0001111	010	000000000000	rd = 0	I	`cbo.inval (rs1)`	invalidate block	invalidate block holding [rs1]
0001111	010	000000000001	rd = 0	I	`cbo.clean (rs1)`	clean (write back) block	write back block holding [rs1]
0001111	010	000000000010	rd = 0	I	`cbo.flush (rs1)`	flush block	flush block holding [rs1]
Zicboz: Cache Block Zero							
0001111	010	000000000100	rd = 0	I	`cbo.zero  (rs1)`	zero block	zero block holding [rs1]
Zicbop: Cache Block Prefetch							
0010011	110	imm[11:5], 00000	rd = 0	I	`prefetch.i imm(rs1)`	instruction prefetch	I$ Prefetch [Address]
0010011	110	imm[11:5], 00001	rd = 0	I	`prefetch.r imm(rs1)`	data read prefetch	D$ Prefetch [Address]

RISC-V Additional Compressed Instructions

Table I.22 RVC/Zcb/Zcf/Zcd: Additional compressed (16-bit) instructions (continued from Table I.6)

op	$instr_{15:10}$	funct	Type	Compressed Instruction	32-Bit Equivalent
Zcb: Simple code-saving compressed instructions (some require Zba, Zbb, M, or RV64)					
00	100000	–	CLB	`c.lbu     rd',  imm(rs1')`	`lbu rd',  (ZeroExt(imm))  (rs1')`
00	100001	1	CLH	`c.lh      rd',  imm(rs1')`	`lh  rd',  (ZeroExt(imm)*2)(rs1')`
00	100001	0	CLH	`c.lhu     rd',  imm(rs1')`	`lhu rd',  (ZeroExt(imm)*2)(rs1')`
00	100010	–	CSB	`c.sb      rs2', imm(rs1')`	`sb  rs2', (ZeroExt(imm))  (rs1')`
00	100011	0	CSH	`c.sh      rs2', imm(rs1')`	`sh  rs2', (ZeroExt(imm)*2)(rs1')`
00	100111	11000	CU	`c.zext.b rd'`	`andi   rd', rd', 0xFF`
01	100111	11001	CU	`c.sext.b rd'`	`sext.b rd', rd'`
01	100111	11010	CU	`c.zext.h rd'`	`zext.h rd', rd'`
01	100111	11011	CU	`c.sext.h rd'`	`sext.h rd', rd'`
01	100111	11101	CU	`c.not     rd'`	`xori   rd', rd', -1`
01	100111	10	CA	`c.mul     rd',  rs2'`	`mul    rd', rd', rs2'`
01	100111	11100	CU	`c.zext.w rd'`	`add.uw rd', rd', zero`
{C/Zcf}F (RV32 only): Compressed single-precision floating-point loads and stores					
00	011–––	–	CL	`c.flw     fd',  imm(rs1')`	`flw fd',  (ZeroExt(imm)*4)(rs1')`
00	111–––	–	CS	`c.fsw     fs2', imm(rs1')`	`fsw fs2', (ZeroExt(imm)*4)(rs1')`
10	011–––	–	CI	`c.flwsp   fd,   imm`	`flw fd,   (ZeroExt(imm)*4)(sp)`
10	111–––	–	CSS	`c.fswsp   fs2,  imm`	`fsw fs2,  (ZeroExt(imm)*4)(sp)`
{C/Zcd}D: Compressed double-precision floating-point loads and stores					
00	001–––	–	CL	`c.fld     fd',  imm(rs1')`	`fld fd',   (ZeroExt(imm)*8)(rs1')`
00	101–––	–	CS	`c.fsd     fs2', imm(rs1')`	`fsd fs2',  (ZeroExt(imm)*8)(rs1')`
10	001–––	–	CI	`c.fldsp   fd,   imm`	`fld fd,    (ZeroExt(imm)*8)(sp)`
10	101–––	–	CSS	`c.fsdsp   fs2,  imm`	`fsd fs2,   (ZeroExt(imm)*8)(sp)`

RISC-V Bit Manipulation

Table I.23 RVB/Zb*: Bit manipulation instructions

op	funct3	funct7/ imm$_{11:5}$	rs2/ imm$_{4:0}$	Type	Instruction	Description	Operation
Zba: Address Generation							
0110011	010	0010000	rs2	R	sh1add rd,rs1,rs2	shift left by 1 and add	rd = rs2 + (rs1 << 1)
0110011	100	0010000	rs2	R	sh2add rd,rs1,rs2	shift left by 2 and add	rd = rs2 + (rs1 << 2)
0110011	110	0010000	rs2	R	sh3add rd,rs1,rs2	shift left by 3 and add	rd = rs2 + (rs1 << 3)
Zba (RV64 only): Shift and Add 32-bit Unsigned Word (U.W.)							
0111011	010	0010000	rs2	R	sh1add.uw rd,rs1,rs2	shift u.w. left 1, add	rd = rs2 + ZeroExt(rs1$_{31:0}$) << 1
0111011	100	0010000	rs2	R	sh2add.uw rd,rs1,rs2	shift u.w. left 2, add	rd = rs2 + ZeroExt(rs1$_{31:0}$) << 2
0111011	110	0010000	rs2	R	sh3add.uw rd,rs1,rs2	shift u.w. left 3, add	rd = rs2 + ZeroExt(rs1$_{31:0}$) << 3
0111011	000	0000100	rs2	R	add.uw rd,rs1,rs2	add u.w.	rd = rs2 + ZeroExt(rs1$_{31:0}$)
0011011	001	000010*	uimm$_{4:0}$	I	slli.uw rd,rs1,uimm	shift left logical imm. u.w.	rd = ZeroExt(rs1$_{31:0}$) << uimm
Zbb: Basic Bit Manipulation							
0110011	100	0000101	rs2	R	min rd,rs1,rs2	minimum	rd = min(rs1, rs2)
0110011	101	0000101	rs2	R	minu rd,rs1,rs2	unsigned minimum	rd = minu(rs1, rs2)
0110011	110	0000101	rs2	R	max rd,rs1,rs2	maximum	rd = max(rs1, rs2)
0110011	111	0000101	rs2	R	maxu rd,rs1,rs2	unsigned maximum	rd = maxu(rs1, rs2)
0110011	100	0100000	rs2	R	orn rd,rs1,rs2	or inverted operand	rd = rs1 | ~rs2
0110011	111	0100000	rs2	R	andn rd,rs1,rs2	and inverted operand	rd = rs1 & ~rs2
0110011	110	0100000	rs2	R	xnor rd,rs1,rs2	xor inverted operand	rd = rs1 ^ ~rs2
0110011	001	0110000	rs2	R	rol rd,rs1,rs2	rotate left	rd = rol(rs1, rs2$_{4/5:0}$)
0110011	101	0110000	rs2	R	ror rd,rs1,rs2	rotate right	rd = ror(rs1, rs2$_{4/5:0}$)
0010011	101	0010100	00111	I	orc.b rd,rs1	bitwise or, combine on bytes	rd = orc.b(rs1)
011•011	100	0000100	00000	I	zext.h rd,rs1	zero-extend halfword	rd = ZeroExt(rs1$_{15:0}$)
0010011	001	0110000	00000	I	clz rd,rs1	count leading zeros	rd = clz(rs1)
0010011	001	0110000	00010	I	cpop rd,rs1	population count	rd = popcnt(rs1)
0010011	001	0110000	00001	I	ctz rd,rs1	count trailing zeros	rd = clz(rev(rs1))
0010011	001	0110000	00100	I	sext.b rd,rs1	sign-extend byte	rd = SignExt(rs1$_{7:0}$)
0010011	001	0110000	00101	I	sext.h rd,rs1	sign-extend halfword	rd = SignExt(rs1$_{15:0}$)
0010011	101	011010•	11000	I	rev8 rd,rs1	byte-wise reverse	rd = revbyte(rs1)
0010011	101	011000*	uimm$_{4:0}$	I	rori rd,rs1,uimm	rotate right immediate	rd = ror(rs1, uimm)
Zbb (RV64 only): Basic Bit Manipulation on 32-bit Words							
0111011	001	0110000	rs2	R	rolw rd,rs1,rs2	rotate left word	rd = SignExt(rol32(rs1$_{31:0}$, rs2$_{4:0}$))
0111011	101	0110000	rs2	R	rorw rd,rs1,rs2	rotate right word	rd = SignExt(ror32(rs1$_{31:0}$, rs2$_{4:0}$))
0011011	101	0110000	uimm$_{4:0}$	I	roriw rd,rs1,uimm	rotate right immediate word	rd = SignExt(ror32(rs1$_{31:0}$, uimm))
0011011	001	0110000	00000	I	clzw rd,rs1	count leading zeros word	rd = clz(rs1$_{31:0}$)
0011011	001	0110000	00010	I	cpopw rd,rs1	population count word	rd = popcnt(rs1$_{31:0}$)
0011011	001	0110000	00001	I	ctzw rd,rs1	count trailing zeros word	rd = clz(rev(rs1$_{31:0}$))
Zbc: Carry-Less (CL) Multiplication							
0110011	001	0000101	rs2	R	clmul rd,rs1,rs2	cl multiply (lower half)	rd = (rs1 @ rs2)$_{XLEN-1:0}$
0110011	011	0000101	rs2	R	clmulh rd,rs1,rs2	cl multiply (upper half)	rd = (rs1 @ rs2)$_{2*XLEN-1:XLEN}$
0110011	010	0000101	rs2	R	clmulr rd,rs1,rs2	cl multiply reversed	rd = rev((rev(rs1) @ rev(rs2))$_{XLEN-1:0}$)
Zbs: Single-Bit Operations							
0110011	001	0100100	rs2	R	bclr rd,rs1,rs2	bit clear	rd = rs1 & ~(1 << rs2$_{4/5:0}$)
0110011	001	0110100	rs2	R	binv rd,rs1,rs2	bit invert	rd = rs1 ^ (1 << rs2$_{4/5:0}$)
0110011	001	0010100	rs2	R	bset rd,rs1,rs2	bit set	rd = rs1 | (1 << rs2$_{4/5:0}$)
0110011	101	0100100	rs2	R	bext rd,rs1,rs2	bit extract	rd = |(rs1 & (1 << rs2$_{4/5:0}$))
0010011	001	010010*	uimm$_{4:0}$	I	bclri rd,rs1,uimm	bit clear immediate	rd = (rs1 & ~(1 << uimm)
0010011	001	011010*	uimm$_{4:0}$	I	binvi rd,rs1,uimm	bit invert immediate	rd = (rs1 ^ (1 << uimm
0010011	001	001010*	uimm$_{4:0}$	I	bseti rd,rs1,uimm	bit set immediate	rd = (rs1 | (1 << uimm)
0010011	101	010010*	uimm$_{4:0}$	I	bexti rd,rs1,uimm	bit extract immediate	rd = |(rs1 & (1 << uimm))

* = 0 for RV32, uimm$_5$ for RV64.

• = 0 for rv32, 1 for rv64. rev8 and zext.h are versions of more general GREV, PACK, and PACKW instructions that may be provided in a future enhancement of the bit manipulation specification and that use the op and funct7 fields in more general ways.

RISC-V Atomic Memory Operations

Table I.24 RVA: RISC-V atomic memory operation (AMO) instructions

op	funct3	funct7/ imm$_{11:5}$	rs2	Type	Instruction	Description	Operation
0101111	010	00010, aq, rl	00000	I	`lr.w rd, (rs1)`	load reserved	`rd=SignExt([rs1]), reserve [rs1]`

0101111	010	00011, aq, rl	rs2	R	`sc.w rd, rs2, (rs1)`	store conditional	`if ([rs1] still reserved)`
							`[rs1]`$_{31:0}$ `= rs2`$_{31:0}$`, rd = 0 #success`
							`else rd = 1 #fail`

0101111	010	00001, aq, rl	rs2	R	`amoswap.w rd, rs2, (rs1)`	amo swap	`rd=[rs1], [rs1]`$_{31:0}$ `= rs2`
0101111	010	00000, aq, rl	rs2	R	`amoadd.w rd, rs2, (rs1)`	amo add	`rd=[rs1], [rs1]`$_{31:0}$ `= [rs1] + rs2`
0101111	010	00100, aq, rl	rs2	R	`amoxor.w rd, rs2, (rs1)`	amo xor	`rd=[rs1], [rs1]`$_{31:0}$ `= [rs1] ^ rs2`
0101111	010	01100, aq, rl	rs2	R	`amoand.w rd, rs2, (rs1)`	amo and	`rd=[rs1], [rs1]`$_{31:0}$ `= [rs1] & rs2`
0101111	010	01000, aq, rl	rs2	R	`amoor.w rd, rs2, (rs1)`	amo or	`rd=[rs1], [rs1]`$_{31:0}$ `= [rs1] | rs2`
0101111	010	10000, aq, rl	rs2	R	`amomin.w rd, rs2, (rs1)`	amo min	`rd=[rs1], [rs1]`$_{31:0}$ `= min ([rs1], rs2)`
0101111	010	10100, aq, rl	rs2	R	`amomax.w rd, rs2, (rs1)`	amo max	`rd=[rs1], [rs1]`$_{31:0}$ `= max ([rs1], rs2)`
0101111	010	11000, aq, rl	rs2	R	`amominu.w rd, rs2, (rs1)`	amo min unsign.	`rd=[rs1], [rs1]`$_{31:0}$ `= minu([rs1], rs2)`
0101111	010	11100, aq, rl	rs2	R	`amomaxu.w rd, rs2, (rs1)`	amo max unsign.	`rd=[rs1], [rs1]`$_{31:0}$ `= maxu([rs1], rs2)`

In this table, [rs1] = SignExt([rs1] $_{31:0}$), the sign-extended contents of memory address rs1. RV64A adds .d versions of all these instructions (i.e., lr.d, amoswap.d, etc.), with funct3 = 011; these operate on a 64-bit operand (double word). The aq (acquire) and rl (release) bits indicate additional ordering. When aq = 1, later memory operations in this hart are only seen after AMO. When rl = 1, other RISC-V harts will see all previous memory operations in this hart before AMO. If aq = rl = 1, all later/previous memory operations in this hart must occur after/before AMO.

RISC-V Integer Conditional Instructions

Table I.25 Zicond: Integer conditional instructions

op	funct3	funct7	rs2	Type	Instruction	Description	Operation
0110011	101	0000111	rs2	R	`czero.eqz rd, rs1, rs2`	conditional zero on equal zero	`rd=(rs2 == 0) ? 0 : rs1`
0110011	111	0000111	rs2	R	`czero.nez rd, rs1, rs2`	conditional zero on not equal zero	`rd=(rs2 != 0) ? 0 : rs1`

RISC-V Additional Floating-Point Instructions

Table I.26 Zfa: Additional RISC-V Floating-Point Instructions

op	funct3	funct7/ imm$_{11:5}$	rs2/ imm$_{4:0}$	Type	Flags NV DZ OF UF NX	Instruction	Description	Operation
1010011	000	11110, fmt	00001	I	– – – – –	`fli.* fd, rs1`	load immediate	`fd = imm[rs1]`
1010011	010	00101, fmt	fs2	R	x – – – –	`fminm.* fd, fs1, fs2`	min	`fd = min(fs1, fs2)`
1010011	011	00101, fmt	fs2	R	x – – – –	`fmaxm.* fd1, fs1, fs2`	max	`fd = max(fs1, fs2)`
1010011	rm	01000, fmt	00100	I	x – – – –	`fround.* fd, fs1`	round to integer	`fd = round(fs)`
1010011	rm	01000, fmt	00101	I	x – – – x	`froundnx.* fd, fs1`	round to integer	`fd = round(fs)`
1010011	101	10100, fmt	fs2	R	x – – – –	`fltq.* rd, fs1, fs2`	compare <	`rd = (fs1 < fs2)`
1010011	100	10100, fmt	fs2	R	x – – – –	`fleq.* rd, fs1, fs2`	compare ≤	`rd = (fs1 ≤ fs2)`
						Only RV32D		
1010011	000	1110001	00001	I	– – – – –	`fmvh.x.d rd, fs1`	move high	`rd = fs1[63:31]`
1010011	000	1011001	rs2	R	– – – – –	`fmvp.d.x fd, rs1, rs2`	move pair	`fd = {rs2, rs1}`
						Only RV64Q		
1010011	000	1110011	00001	I	– – – – –	`fmvh.x.q rd, fs1`	move high	`rd = fs1[127:64]`
1010011	000	1011011	rs2	R	– – – – –	`fmvp.q.x fd, rs1, rs2`	move pair	`fd = {rs2, rs1}`
						Only D Extension		
1010011	001	1100001	01000	I	x – – – x	`fcvtmod.w.d rd, fs1, rtz`	modular convert	`rd = SignExt(trunc(fs1) mod `2^{32}`)`

imm[rs1] is a floating-point immediate specified by rs1, as shown in Table 16.52. In fcvtmod.w.d, rtz is the rounding mode.

RISC-V Cryptography Instructions

Table I.27 : Zbk*/Zkn*: RISC-V scalar cryptography instructions

op	funct3	funct7/ instr$_{31:25}$	rs2/ instr$_{24:20}$	Type	Instruction	Description	Operation
Zbkb: Bit Manipulation for Cryptography (also includes rol, ror, rori, andn, orn, xnor, rev8 from Zbb, Table I.23)							
0110011	100	0000100	rs2	R	pack rd,rs1,rs2	pack XLEN/2 in XLEN	rd = {rs2$_{XLEN/2-1:0}$,rs1$_{XLEN/2-1:0}$}
0110011	111	0000100	rs2	R	packh rd,rs1,rs2	pack 8 in 16	rd = ZeroExt{rs2$_{7:0}$,rs1$_{7:0}$}
0010011	101	0110100	00111	I	brev8 rd,rs1	bit reverse	rd = reversebitsinbytes(rs1)
Zbkb (RV32 only): Bit Manipulation for Cryptography							
0010011	001	0000100	01111	I	zip rd,rs1	hi/low -> odd/even	rd$_{even}$ = rs1$_{15:0}$; rd$_{odd}$ = rs1$_{31:16}$
0010011	101	0000100	01111	I	unzip rd,rs1	odd/even -> hi/low	rd$_{15:0}$ = rs1$_{odd}$; rd$_{31:16}$ = rs1$_{even}$
Zbkb (RV64 only): Bit Manipulation for Cryptography on 32-bit Words (also includes rolw, rorw, rorrwi, Table I.23)							
0111011	100	0000100	rs2	R	packw rd,rs1,rs2	pack 16 in 32	rd = ZeroExt({rs2$_{15:0}$,rs1$_{15:0}$})
Zbkc: Carry-less Multiply (see clmul and clmulh from Zbc, Table I.23)							
Zbkx: Bit Manipulation for Cryptography							
0110011	010	0010100	rs2	R	xperm4 rd,rs1,rs2	permute nibbles	rd=lookup4(rs1,rs2)
0110011	100	0010100	rs2	R	xperm8 rd,rs1,rs2	permute bytes	rd=lookup8(rs1,rs2)
Zknd (RV32 only): AES Decryption							
0110011	000	{bs, 10101}	rs2	R	aes32dsi rd,rs1,rs2,bs	final round	rd =rs1^(isbox(rs2[bs])<<bs*8)
0110011	000	{bs, 10111}	rs2	R	aes32dsmi rd,rs1,rs2,bs	middle round	rd =rs1^(imixcols(isbox(rs2[bs]))<<bs*8)
Zknd (RV64 only): AES Decryption & Inverse Key Schedule							
0110011	000	0011101	rs2	R	aes64ds rd,rs1,rs2	final round	rd =isbox(ishiftrows(rs2,rs1))
0110011	000	0011111	rs2	R	aes64dsm rd,rs1,rs2	middle round	rd =imixcols(isbox(shiftrows(rs2,rs1)))
0010011	001	0011000	00000	I	aes64im rd,rs1	InvMixCols	rd =imixcols(rs1)
Zkne (RV32 only): AES Encryption							
0110011	000	{bs, 10001}	rs2	R	aes32esi rd,rs1,rs2,bs	final round	rd =rs1^(sbox(rs2[bs])<<bs*8)
0110011	000	{bs, 10011}	rs2	R	aes32esmi rd,rs1,rs2,bs	middle round	rd =rs1^(mixcols(sbox(rs2[bs]))<<bs*8)
Zkne (RV64 only): AES Encryption							
0110011	000	0011001	rs2	R	aes64es rd,rs1,rs2	final round	rd =sbox(shiftrows(rs2,rs1))
0110011	000	0011011	rs2	R	aes64esm rd,rs1,rs2	middle round	rd =mixcols(sbox(shiftrows(rs2,rs1))
Zknd and Zkne (RV64 only): AES Key Schedule							
0010011	001	0011000	{1, rnum}	I	aes64ks1i rd,rs1,rnum	key schedule	rd ={2{rcon[rnum]^sbox(rot(rs1))}}
0110011	000	0111111	rs2	R	aes64ks2 rd,rs1,rs2	key schedule	w0 =rs1$_{63:32}$^rs2$_{31:0}$; rd={w0^rs2$_{63:32}$,w0}
Zknh: SHA2-256 Hash*							
0010011	001	0001000	00010	I	sha256sig0 rd,rs1	$\sigma_0^{(256)}$ function	rd =(rs1 ror 7) ^(rs1 ror 18)^(rs1 >> 3)
0010011	001	0001000	00011	I	sha256sig1 rd,rs1	$\sigma_1^{(256)}$ function	rd =(rs1 ror 17)^(rs1 ror 19)^(rs1 >> 10)
0010011	001	0001000	00000	I	sha256sum0 rd,rs1	$\Sigma_0^{(256)}$ function	rd =(rs1 ror 2) ^(rs1 ror 13)^(rs1 ror 22)
0010011	001	0001000	00001	I	sha256sum1 rd,rs1	$\Sigma_1^{(256)}$ function	rd =(rs1 ror 6) ^(rs1 ror 11)^(rs1 ror 25)
Zknh (RV32 only): SHA2-512 Hash							
0110011	000	0101110	rs2	R	sha512sig0h rd,rs1,rs2	$\sigma_0^{(512)}$ high word	rd=({rs2,rs1}>>1)$_{31:0}$ ^({rs2,rs1}>>8)$_{31:0}$ ^(rs1 >>7)
0110011	000	0101010	rs2	R	sha512sig0l rd,rs1,rs2	$\sigma_0^{(512)}$ low word	rd=({rs2,rs1}>>1)$_{31:0}$ ^({rs2,rs1}>>8)$_{31:0}$ ^({rs2,rs1}>>7)$_{31:0}$
0110011	000	0101111	rs2	R	sha512sig1h rd,rs1,rs2	$\sigma_1^{(512)}$ high word	rd=({rs2,rs1}>>29)$_{31:0}$^({rs2,rs1}>>19)$_{31:0}$ ^(rs1 >>6)
0110011	000	0101011	rs2	R	sha512sig1l rd,rs1,rs2	$\sigma_1^{(512)}$ low word	rd=({rs2,rs1}>>29)$_{31:0}$^({rs2,rs1}>>19)$_{31:0}$ ^({rs2,rs1}>>6)$_{31:0}$
0110011	000	0101000	rs2	R	sha512sum0r rd,rs1,rs2	$\Sigma_0^{(512)}$ half	rd=({rs2,rs1}>>7)$_{31:0}$ ^({rs2,rs1} >>2)$_{31:0}$ ^({rs2,rs1}>>28)$_{31:0}$
0110011	000	0101001	rs2	R	sha512sum1r rd,rs1,rs2	$\Sigma_1^{(512)}$ half	rd=({rs2,rs1}>>9)$_{31:0}$ ^({rs2,rs1}>>14)$_{31:0}$ ^({rs2,rs1}>>18)$_{31:0}$
Zknh (RV64 only): SHA2-512 Hash							
0010011	001	0001000	00110	I	sha512sig0 rd,rs1	$\sigma_0^{(512)}$ function	rd =(rs1 ror 1) ^(rs1 ror 8) ^(rs1 >> 7)
0010011	001	0001000	00111	I	sha512sig1 rd,rs1	$\sigma_1^{(512)}$ function	rd =(rs1 ror 19)^(rs1 ror 61)^(rs1 >> 6)
0010011	001	0001000	00100	I	sha512sum0 rd,rs1	$\Sigma_0^{(512)}$ function	rd =(rs1 ror 28)^(rs1 ror 34)^(rs1 ror 39)
0010011	001	0001000	00101	I	sha512sum1 rd,rs1	$\Sigma_1^{(512)}$ function	rd =(rs1 ror 14)^(rs1 ror 18)^(rs1 ror 41)

bs=2-bit byte select. rnum=4-bit round number. *RV64's SHA2-256 instructions operate on rs1's low 32 bits and sign-extend the final result to 64 bits.