RV32 Reference Card

RV32IMFD Registers and Assembler Directives

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

* Within a given function `f`, all registers marked as `caller` are considered temporary and do not need to be saved. All registers marked as `callee` must be saved at function entrance, if used.

Register	ABI Name	Description	Saver*
f0-f7	ft0-ft7	FP temporaries	Caller
f8-f9	fs0-fs1	FP saved registers	Callee
f10-f11	fa0-fa1	FP Funct. arguments/return values	Caller
f12-f17	fa2-fa7	FP Funct. arguments	Caller
f18-f27	fs2-fs11	FP saved registers	Callee
f28-f31	ft8-ft11	FP temporaries	Caller

Assembler Directives:

- DECLARATION OF SEGMENTS:
  `.data` (for RW data) and `.text` (for program)
- DECLARATION OF DATA (WITHIN `.data` SEGMENT):
  `.byte`,`.half`,`.word`,`.string` – initialized list of values/characters
  `.zero` – N bytes initialized with value 0

Instruction Formats

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R-Type	funct7		rb	ra	funct3	rd	opcode
I-Type	$imm_{[11:0]}$			ra	funct3	rd	opcode
S-Type	$imm_{[11:5]}$		rb	ra	funct3	$imm_{[4:0]}$	opcode
B-Type	$imm_{[12\mid10:5]}$		rb	ra	funct3	$imm_{[4:1\mid11]}$	opcode
U-Type	$imm_{[31:12]}$					rd	opcode
J-Type	$imm_{[20\mid10:1\mid11\mid19:12]}$					rd	opcode
R3-Type	funct5	fmt	rb	ra	funct3	rd	opcode
R4-Type	rc	fmt	rb	ra	funct3	rd	opcode

Note: on B-Type and J-Type instructions the encoding omits bit 0 of the immediate field.

RV32IM Instructions

	Instruction		Name	Encoding Type	Opcode	funct3	funct7	RTL Description
-	nop		No Operation	*Pseudo-inst.:* addi x0,x0,0				-
Move	li	xd,imm	Load (move) Immediate	*Pseudo-inst.:* Myriad sequence				xd ← imm
	la	xd,symbol	Load Address	*Pseudo-inst.:* auipc + addi				xd ← imm *(symbol)*
	lui	xd,imm	Load Upper Imm	U	0110111			xd ← imm << 12
	auipc	xd,imm	Add Upper Imm to PC	U	0010111			xd ← PC + (imm << 12)
	mv	xd,rs	Move Register Value	*Pseudo-inst.:* addi xd, xs, 0				xd ← xs
Arithmetic, Logic, Shift	neg	xd,xs	Negate (2's complement)	*Pseudo-inst.:* sub xd, x0, xs				xd ← - xs
	add	xd,xa,xb	ADD	R	0110011	0x0	0x00	xd ← xa + xb
	addi	xd,xa,imm	ADD Immediate	I	0010011	0x0		xd ← xa + imm
	sub	xd,xa,xb	SUB	R	0110011	0x0	0x20	xd ← xa - xb
	not	xd,xs	NOT	*Pseudo-inst.:* xori xd, xs, -1				xd ← ~ xs
	and	xd,xa,xb	AND	R	0110011	0x7	0x00	xd ← xa & xb
	andi	xd,xa,imm	AND Immediate	I	0010011	0x7		xd ← xa & imm
	or	xd,xa,xb	OR	R	0110011	0x6	0x00	xd ← xa | xb
	ori	xd,xa,imm	OR Immediate	I	0010011	0x6		xd ← xa | imm
	xor	xd,xa,xb	XOR	R	0110011	0x4	0x00	xd ← xa ^ xb
	xori	xd,xa,imm	XOR Immediate	I	0010011	0x4		xd ← xa ^ imm
	sll	xd,xa,xb	Shift Left Logical	R	0110011	0x1	0x00	xd ← xa << $xb_{[4:0]}$
	slli	xd,xa,imm	Shift Left Logical Imm	I	0010011	0x1		xd ← xa << $imm_{[4:0]}$
	srl	xd,xa,xb	Shift Right Logical	R	0110011	0x5	0x00	xd ← xa >> $xb_{[4:0]}$
	srli	xd,xa,imm	Shift Right Logical Imm	I	0010011	0x5		xd ← xa >> $imm_{[4:0]}$
	sra	xd,xa,xb	Shift Right Arithmetic	R	0110011	0x5	0x20	xd ← xa >> $xb_{[4:0]}$
	srai	xd,xa,imm	Shift Right Arith Imm	I	0010011	0x5		xd ← xa >> $imm_{[4:0]}$
Multiply, Divide	mul	xd,xa,xb	Multiply	R	0110011	0x0	0x01	xd ← $(xa \times xb)_{[31:0]}$
	mulh	xd,xa,xb	Multiply High (S×S)	R	0110011	0x1	0x01	xd ← $(xa \times xb)_{[63:32]}$
	mulsu	xd,xa,xb	Multiply High (S×U)	R	0110011	0x2	0x01	xd ← $(xa \times xb)_{[63:32]}$
	mulu	xd,xa,xb	Multiply High (U×U)	R	0110011	0x3	0x01	xd ← $(xa \times xb)_{[63:32]}$
	div	xd,xa,xb	Divide	R	0110011	0x4	0x01	xd ← xa / xb
	divu	xd,xa,xb	Divide (U)	R	0110011	0x5	0x01	xd ← xa / xb
	rem	xd,xa,xb	Remainder	R	0110011	0x6	0x01	xd ← xa % xb
	remu	xd,xa,xb	Remainder (U)	R	0110011	0x7	0x01	xd ← xa % xb
Load, Store	l{b|h|w}	xd,symbol	Load From Global Symbol	*Pseudo-inst.:* auipc + l{b|h|w}				xd ← M[symbol] (B/H/W)
	lb	xd,imm(xa)	Load Byte	I	0000011	0x0		xd ← M[xa+imm] (B)
	lh	xd,imm(xa)	Load Halfword	I	0000011	0x1		xd ← M[xa+imm] (H)
	lw	xd,imm(xa)	Load Word	I	0000011	0x2		xd ← M[xa+imm] (W)
	lbu	xd,imm(xa)	Load Byte (U)	I	0000011	0x2		xd ← M[xa+imm] (BU)
	lhu	xd,imm(xa)	Load Halfword (U)	I	0000011	0x2		xd ← M[xa+imm] (HU)
	s{b|h|w}	xs,symbol,xt	Store to Global Symbol	*Pseudo-inst.:* auipc + s{b|h|w}				M[symbol] ← xs *(modifies* xt*)*
	sb	xb,imm(xa)	Store Byte	S	0100011	0x0		M[xa+imm] ← $xb_{[7:0]}$
	sh	xb,imm(xa)	Store Half	S	0100011	0x1		M[xa+imm] ← $xb_{[15:0]}$
	sw	xb,imm(xa)	Store Word	S	0100011	0x2		M[xa+imm] ← $xb_{[31:0]}$

RV32IM Instructions (continued)

	Instruction	Name	Encoding				RTL Description
			Type	Opcode	funct3	funct7	
Compare	slt xd,xa,xb	Set Less Than	R	0110011	0x2	0x00	xd ← (xa < xb)?1:0
	slti xd,xa,imm	Set Less Than Imm	I	0010011	0x2		xd ← (xa < imm)?1:0
	sltu xd,xa,xb	Set Less Than (U)	R	0110011	0x3	0x00	xd ← (xa < xb)?1:0
	sltiu xd,xa,imm	Set Less Than Imm (U)	I	0010011	0x3		xd ← (xa < imm)?1:0
	seqz xd,xs	Set Equal Zero	*Pseudo-inst.:* sltiu xd, xs, 1				xd ← (xs == 0)?1:0
	snez xd,xs	Set Not Equal Zero	*Pseudo-inst.:* sltu xd, x0, xs				xd ← (xs != 0)?1:0
	sltz xd,xs	Set Less Than Zero	*Pseudo-inst.:* slt xd, xs, x0				xd ← (xs < 0)?1:0
	sgtz xd,xs	Set Greater Than Zero	*Pseudo-inst.:* slt xd, x0, xs				xd ← (xs > 0)?1:0
Flow control (branch, jump, call, ret)	beq xa,xb,imm	Branch Equal	B	1100011	0x0		if(xa==xb) PC ← PC + imm
	bne xa,xb,imm	Branch Not Equal	B	1100011	0x1		if(xa!=xb) PC ← PC + imm
	bgt xs, xt, offset	Branch Greater Than	*Pseudo-inst.:* blt xt, xs, offset				if (xs> xt) PC ← symbol
	bge xa,xb,imm	Branch Greater or Equal	B	1100011	0x5		if(xa>=xb) PC ← PC + imm
	ble xs, xt, offset	Branch Less or Equal	*Pseudo-inst.:* bge xt, xs, offset				if (xs<=xt) PC ← symbol
	blt xa,xb,imm	Branch Less Than	B	1100011	0x4		if(xa< xb) PC ← PC + imm
	bgtu xs, xt, offset	Branch Less Than (U)	*Pseudo-inst.:* bltu xt, xs, offset				if (xs> xt) PC ← symbol
	bgeu xa,xb,imm	Branch Greater or Equal (U)	B	1100011	0x7		if(xa>=xb) PC ← PC + imm
	bleu xs, xt, offset	Branch Less or Equal (U)	*Pseudo-inst.:* bgeu xt, xs, offset				if (xs<=xt) PC ← symbol
	bltu xa,xb,imm	Branch Less Than (U)	B	1100011	0x6		if(xa< xb) PC ← PC + imm
	beqz xs, symbol	Branch Equal Zero	*Pseudo-inst.:* beq xs, x0, offset				if (xs==0) PC ← symbol
	bnez xs, symbol	Branch Not Equal Zero	*Pseudo-inst.:* bne xs, x0, offset				if (xs!=0) PC ← symbol
	blez xs, symbol	Branch Less or Equal Zero	*Pseudo-inst.:* bge x0, xs, offset				if (xs<=0) PC ← symbol
	bgez xs, symbol	Branch Greater or Equal Zero	*Pseudo-inst.:* bge xs, x0, offset				if (xs>=0) PC ← symbol
	bltz xs, symbol	Branch Less Than Zero	*Pseudo-inst.:* blt xs, x0, offset				if (xs< 0) PC ← symbol
	bgtz xs, symbol	Branch Greater Than Zero	*Pseudo-inst.:* blt x0, xs, offset				if (xs> 0) PC ← symbol
	j symbol	Jump	*Pseudo-inst.:* jal x0, offset				PC ← symbol
	jal xd,imm	Jump And Link	J	1101111			xd ← PC+4; PC ← PC + imm
	jal symbol	Jump And Link To Symbol	*Pseudo-inst.:* jal x1, offset				x1 ← PC + 4; PC ← symbol
	jr xs	Jump Register	*Pseudo-inst.:* jalr x0, xs, 0				PC ← xs
	jalr xs	Jump And Link Register	*Pseudo-inst.:* jalr x1, xs, 0				x1 ← PC + 4; PC ← xs
	jalr xd,xa,imm	Jump And Link Register	I	1100111	0x0		xd ← PC+4; PC ← xa + imm
	call symbol	Call subroutine	*Pseudo-inst.:* auipc + jalr				x1 ← PC + 4; PC ← symbol
	ret	Return from subroutine	*Pseudo-inst.:* jalr x0, x1, 0				PC ← x1
OS	ecall	Environment Call	I	1110011	imm=0x0, others=0		SEPC ← PC + 4; PC ← STVEC
	ebreak	Environment Break	I	1110011	imm=0x1, others=0		SEPC ← PC + 4; PC ← STVEC
	sret	Exception return	I	1110011	imm=0x102, others=0		PC ← SEPC

RV32FD Floating Point Instructions (not available on Ripes)

	{"F"|"D"} Instruction		Name	Encoding				RTL Description
				Type	Opcode	funct3	funct5	
LD, ST	fl{w|d}	fd,symbol	FP Load from Symbol	*Pseudo-inst.:* auipc + fl{w|d}				fd ← M[symbol]
	fl{w|d}	fd,imm(fa)	FP Load	I	0000111	{010|011}		fd ← M[fa+imm]
	fs{w|d}	fb,symbol,xt	FP Store to Symbol	*Pseudo-inst.:* auipc + fs{w|d}				M[symbol] ← fb *(modifies* xt*)*
	fs{w|d}	fb,imm(fa)	FP Store	S	0100111	{010|011}		M[fa+imm] ← fb
Move	fmv.{s|d}	fd,fs	FP Sign Injection	*Pseudo-inst.:* fsgnj.{s|d} fd,fs,fs				fd = fs
	fsgnj.{s|d}	fd,fa,fb	FP Sign Injection	R3	1010011	000	{0x10|0x11}	fd = abs(fa) * sgn(fb)
	fsgnjn.{s|d}	fd,fa,fb	FP Sign Neg Injection	R3	1010011	001	{0x10|0x11}	fd = abs(fa) * -sgn(fb)
	fsgnjx.{s|d}	fd,fa,fb	FP Sign Xor Injection	R3	1010011	010	{0x10|0x11}	fd = fa * sgn(fb)
	fabs.{s|d}	fd,fs	FP Absolute Value	*Pseudo-inst.:* fsgnjx.{s|d} fd,fs,fs				fd = |fs|
	fneg.{s|d}	fd,fs	FP Negative Value	*Pseudo-inst.:* fsgnjn.{s|d} fd,fs,fs				fd = -fs
	fmin.{s|d}	fd,fa,fb	FP Minimum	R3	1010011	000	{0x10|0x11}	fd = min(fa, fb)
	fmax.{s|d}	fd,fa,fb	FP Maximum	R3	1010011	001	{0x10|0x11}	fd = max(fa, fb)
Arithmetic	fadd.{s|d}	fd,fa,fb	FP Add	R3	1010011	rm	{0x00|0x01}	fd = fa + fb
	fsub.{s|d}	fd,fa,fb	FP Sub	R3	1010011	rm	{0x04|0x05}	fd = fa - fb
	fmul.{s|d}	fd,fa,fb	FP Mul	R3	1010011	rm	{0x08|0x09}	fd = fa * fb
	fdiv.{s|d}	fd,fa,fb	FP Div	R3	1010011	rm	{0x0c|0x0d}	fd = fa / fb
	fsqrt.{s|d}	fd,fa	FP Square Root	R3	1010011	rm	{0x2c|0x2d}	fd = sqrt(fa)
Fused	fmadd.{s|d}	fd,fa,fb,fc	FP Fused Mul-Add	R4	1000011	rm		fd = fa * fb + fc
	fmsub.{s|d}	fd,fa,fb,fc	FP Fused Mul-Sub	R4	1000111	rm		fd = fa * fb - fc
	fnmadd.{s|d}	fd,fa,fb,fc	FP Neg Fused Mul-Add	R4	1001111	rm		fd = -fa * fb + fc
	fnmsub.{s|d}	fd,fa,fb,fc	FP Neg Fused Mul-Sub	R4	1001011	rm		fd = -fa * fb - fc
Convert	fcvt.{s|d}.w	fd,xa *	FP Conv from Sign Int	R3	1010011	rm	{0x68|0x69}	fd = (float) xa
	fcvt.{s|d}.wu	fd,xa *	FP Conv from Uns Int	R3	1010011	rm	{0x68|0x69}	fd = (float) xa
	fcvt.w.{s|d}	xd,fa *	FP Convert to Int	R3	1010011	rm	{0x60|0x61}	xd = (int32_t) fa
	fcvt.wu.{s|d}	xd,fa *	FP Convert to Int	R3	1010011	rm	{0x60|0x61}	xd = (uint32_t) fa
	fmv.x.w	xd,fa	Move SP FP to Int	R3	1010011	000	0x70	xd = *((int*) &fa)
	fmv.w.x	fd,xa	Move Int to SP FP	R3	1010011	000	0x78	fd = *((float*) &xa)
Compare	feq.{s|d}	xd,fa,fb	FP Equal	R3	1010011	010	{0x50|0x51}	xd = (fa == fb) ? 1 : 0
	flt.{s|d}	xd,fa,fb	FP Less Than	R3	1010011	001	{0x50|0x51}	xd = (fa < fb) ? 1 : 0
	fle.{s|d}	xd,fa,fb	FP Less or Equal	R3	1010011	000	{0x50|0x51}	xd = (fa <= fb) ? 1 : 0
	fclass.{s|d}	xd,fa	FP Classify	R3	1010011	001	{0x70|0x71}	xd = (fa type?):0..9

∗ – To encode fcvt.{s|d}.w and fcvt.w.{s|d} set xb=0; for fcvt.{s|d}.w and fcvt.w.{s|d} set xb=1.
rm – Floating point rounding mode. Set to "000" to select round to nearest.

Observation: Designations fl{w|d} and fs{w|d} represent the corresponding instructions for SP FP load/store (flw/fsw) and for DP FP load/store (fld/fsd). Similarly, designations f__.{s|d} are used to represent the corresponding SP (f__.s) and DP (f__.d) instructions.

Special purpose registers and instructions (not available on Ripes)

Relevant Control and Status Registers (CSRs):

Type	Register	Full Name	CSR ID
Exception handling	sepc	Supervisor Exception PC	0x141
	scause	Supervisor Exception Cause	0x142
	stvec	Supervisor Trap Vector Base Register	0x105
	sip	Supervisor Interrupt-Pending Register	0x144
	sie	Supervisor Interrupt-Enable Register	0x104
	sstatus	Supervisor Status Register	0x100
Virtual Memory	satp	Supervisor Address Translation and Protection	0x180

	31	···	5	4	3	2	1	0	
scause	<number>								states the number of the first pending interruption
stvec	<address>								states the address of the Interrupt Service Routine
sie	INT31	···	INT5	INT4	INT3	INT2	INT1	INT0	0 - Interruptions masked; 1 - Interruptions enabled
sip	INT31	···	INT5	INT4	INT3	INT2	INT1	INT0	0 - No pending interruption; 1 - Pending interruption
sstatus	-	···	SPIE	-	-	-	GIE	-	

GIE - Global Interrupt Enable (set to zero to disable ALL interruptions/exceptions). When entering an interrupt service routine (ISR), the value of GIE is first copied to SPIE and then set to zero. When leaving the ISR, its value is restored using SPIE.

SPIE - GIE backup copy. The value of GIE is automatically copied to SPIE when entering the interrupt service routine.

	Instruction	Name	Encoding Type	Opcode	funct3	funct5	RTL Description
Handling of CSR registers	csrrc xd,csr,xa *	CSR read and clear	I	1110011	011		xd ← CSR[csr]; CSR[csr] ← CSR[csr] & ∼xa
	csrrci xd,csr,imm2 *	CSR read and clear	I	1110011	111		xd ← CSR[csr]; CSR[csr] ← CSR[csr] & ∼imm2
	csrrs xd,csr,xa *	CSR read and set	I	1110011	010		xd ← CSR[csr]; CSR[csr] ← CSR[csr] | xa
	csrrsi xd,csr,imm2 *	CSR read and set	I	1110011	110		xd ← CSR[csr]; CSR[csr] ← CSR[csr] | imm2
	csrrw xd,csr,xa *	CSR read and write	I	1110011	001		xd ← CSR[csr]; CSR[csr] ← xa
	csrrwi xd,csr,imm2 *	CSR read and write	I	1110011	101		xd ← CSR[csr]; CSR[csr] ← imm2
	csrr xd,csr	CSR read	*Pseudo-inst.:* csrrs rd,csr,x0				xd ← CSR[csr]
	csrw csr,xa	CSR write	*Pseudo-inst.:* csrrw x0,csr,xa				CSR[csr] ← xa
	csrc csr,xa	CSR clear bits	*Pseudo-inst.:* csrrw x0,csr,xa				CSR[csr] ← CSR[csr] & ∼xa
	csrs csr,xa	CSR set bits	*Pseudo-inst.:* csrrs x0,csr,xa				CSR[csr] ← CSR[csr] | xa
	csrwi csr,imm2	CSR write	*Pseudo-inst.:* csrrwi x0,csr,imm2				CSR[csr] ← imm2
	csrci csr,imm2	CSR clear bits	*Pseudo-inst.:* csrrwi x0,csr,imm2				CSR[csr] ← CSR[csr] & ∼imm2
	csrsi csr,imm2	CSR set bits	*Pseudo-inst.:* csrrsi x0,csr,imm2				CSR[csr] ← CSR[csr] | imm2

∗ – To encode the instructions use field imm to specify csr and ra to specify imm2.