THE LINUX FOUNDATION | Training & Certification

RISC-V

RISC-V Assembly Programming

Links / References

- Ripes
- Qemu
- Debian wiki for RISC-V
- RISC-V GNU compiler toolchain
- DQIB
- Debian unstable packages
- Visual Studio Code
- PlatformIO
- Code of the example programs
- RISC-V Assembly Programmer's Manual
- RISC-V ELF Specification
- RISC-V application Binary interface (ABI)
- RISC-V Calling Conventions
- Hash Functions
- Sieve of Eratosthenes

Tables

Common Extensions	
Abbreviation	**Standard extension (subset)**
M	integer multiplication and division
A	atomic instructions
F	single-precision floating point

D	double-precision floating point
Q	quad-precision floating point
C	compressed instructions
V	vector operations

RISC-V Registers		
Register	**ABI name**	**Description**
x0	zero	Zero constant
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0 / fp	Saved / Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function args. / return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries
pc	-	Program counter

Encoding						
Type/Bit	**31:25**	**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
B	imm[12,10:5]	rs2	rs1	funct3	imm[4:1, 11]	opcode
U	imm[31:12]				rd	opcode
J	imm[20, 10:1, 11, 19:12]				rd	opcode

Arithmetic and logical instructions (immediates)					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
addi	ADD Immediate	I	0010011	0x0	rd = rs1 + imm
xori	XOR Immediate	I	0010011	0x4	rd = rs1 ^ imm
ori	OR Immediate	I	0010011	0x6	rd = rs1 | imm
andi	AND Immediate	I	0010011	0x7	rd = rs1 & imm
slli	Shift Left Logical Imm.	I	0010011	0x1	imm[11:5]=0x00, rd = rs1 << imm[4:0]
srli	Shift Right Logical Imm.	I	0010011	0x5	imm[11:5]=0x00, rd = rs1 << imm[4:0]
srai	Shift Right Arith. Imm.	I	0010011	0x5	imm[11:5]=0x20, rd = rs1 >> imm[4:0]
slti	Set Less Than Imm.	I	0010011	0x2	rd = (rs1 < imm)? 0:1
sltiu	Set Less Than Imm. Un.	I	0010011	0x3	rd = (rs1 < imm)? 0:1

Arithmetic and logical instructions (registers)						
instruction	**name**	**format**	**opcode**	**funct3**	**funct7**	**description**
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
sra	Shift Right Arith.	R	0110011	0x5	0x20	rd = rs1 >> rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)? 0:1
sltu	Set Less Than Un.	R	0110011	0x3	0x00	rd = (rs1 < rs2)? 0:1

Load and save instructions					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
lb	Load Byte	I	0000011	0x0	rd = M[rs1+imm][7:0]
lh	Load Half	I	0000011	0x1	rd = M[rs1+imm][15:0]
lw	Load Word	I	0000011	0x2	rd = M[rs1+imm][31:0]
lbu	Load Byte Un.	I	0000011	0x0	rd = M[rs1+imm][7:0]

lhu	Load Half Un.	I	0000011	0x0	rd = M[rs1+imm][15:0]
sb	Store Byte	S	0100011	0x0	M[rs1+imm][7:0] = rs2[7:0]
sh	Store Half	S	0100011	0x1	M[rs1+imm][15:0] = rs2[15:0]
sw	Store Word	S	0100011	0x2	M[rs1+imm][31:0] = rs2[31:0]

Upper immediate instructions					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
lui	Load Upper Imm.	U	0110111	-	rd = imm << 12
auipc	Add Upper Imm. to PC	U	0010111	-	rd = PC + (imm << 12)

Conditional branching instructions					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
beq	Branch ==	B	1100011	0x0	if (rs1 == rs2) pc+=imm
bne	Branch !=	B	1100011	0x1	if (rs1 != rs2) pc+=imm
blt	Branch <	B	1100011	0x4	if (rs1 < rs2) pc+=imm
bge	Branch >=	B	1100011	0x5	if (rs1 >= rs2) pc+=imm
bltu	Branch < Un.	B	1100011	0x6	if (rs1 < rs2) pc+=imm
bgeu	Branch >= Un.	B	1100011	0x7	if (rs1 >= rs2) pc+=imm

Unconditional jump instructions					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
jal	Jump and Link	J	1101111	-	if (rs1 == rs2) pc+=imm
jalr	Jump and Link Register	I	1100111	0x0	if (rs1 != rs2) pc+=imm

System interface					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
ecall	Environment Call	I	1110011	0x0	imm = 0, rd = rs1 = 0, transfer control to system
ebreak	Environment Break	I	1110011	0x0	imm = 1, rd = rs1 = 0, transfer control to debugger

Memory ordering					
instruction	**name**	**format**	**opcode**	**funct3**	**description**
`fence`	Fence	I	0001111	0x0	rd, rs1 reserved. Normal fence for all memory access types has imm = 0b000011111111.

Extension ‘M’						
instruction	**name**	**format**	**opcode**	**funct3**	**description**	**instruction**
`mul`	Multiply	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
`mulh`	Multiply High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
`mulhsu`	Multiply High Sign/Uns.	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
`mulhu`	Multiply Unsigned	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
`div`	Divide	R	0110011	0x4	0x01	rd = rs1 / rs2
`divu`	Divide Unsigned	R	0110011	0x5	0x01	rd = rs1 / rs2
`rem`	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
`remu`	Remainder Unsigned	R	0110011	0x7	0x01	rd = rs1 % rs2

Assembler directives		
Directive	**Arguments**	**Description**
`.text`		change to .text section
`.data`		change to .data section
`.rodata`		change to .rodata section
`.bss`		change to .bss section
`.section`	.text, .data, .rodata, .bss	change to section given by arguments
`.equ`	name, value	define name for value
`.ascii`	“string”	begin string without null terminator
`.asciz`	“string”	begin string with null terminator

.string	“string”	same as .asciz
.byte	expression [,expression]*	8-bit comma separated words
.half	expression [,expression]*	16-bit comma separated words
.word	expression [,expression]*	32-bit comma separated words
.dword	expression [,expression]*	64-bit comma separated words
.zero	integer	zero bytes
.align	integer	align to the power of 2
.globl	symbol_name	make symbol_name apparent in symbol table

Pseudo instructions for load, store, and complement		
Pseudo instruction	**Base instruction(s)**	**Description**
la rd, symbol	**auipc rd, symbol[31:12]** **addi rd, symbol[11:0]**	Load address (non position independent code - non-PIC)
la rd, symbol	**auipc rd, symbol@GOT[31:12]** **l{w|d} rd, symbol[11:0](rd)**	Load address (position independent code PIC)
lla ra, symbol	**auipc rd, symbol[31:12]** **addi rd, rd, symbol[11:0]**	Load local address
lga rd, symbol	**auipc rd, symbol@GOT[31:12]** **l{w|d} rd,** **symbol@GOT[11:0](rd)**	Load global address
l{b|h|w|d} rd, symbol	**auipc rd, symbol[31:12]** **l{b|h|w|d} rd,** **symbol[11:0](rd)**	Load global
s{b|h|w|d} rs, symbol, rd	**auipc rd, symbol[31:12]** **s{b|h|w|d} rs,** **symbol[11:0](rd)**	Store global
nop	**addi x0, x0, 0**	No operation
li rd, imm	***Different instructions***	Load immediate
mv rd, rs	**addi rd, rs, 0**	Copy register
not rd, rs	**xori rd, rs, -1**	1’s complement
neg rd, rs	**sub rd, x0, rs**	2’s complement
negw rd, rs	**subw rd, x0, rs**	2’s complement word
la rd, symbol	**auipc rd, symbol[31:12]** **addi rd, symbol[11:0]**	Load address (non position independent code - non-PIC)

`la rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol[11:0](rd)`	Load address (position independent code PIC)
`lla ra, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address

`lga rd, symbol`	`auipc rd, symbol@GOT[31:12]`
	`l{w|d} rd,`
	`symbol@GOT[11:0](rd)`
		Load global address

Pseudo instructions for extending and conditional bit setting		
Pseudo instruction	**Base instruction(s)**	**Description**
`sext.{b|h|w} rd, rs`	`different instructions`	sign extend
`zext.{b|h|w} rd, rs`	`different instructions`	zero extend
`seqz rd, rs`	`sltiu rd, rs, 1`	rd = (rs == 0)? 1:0
`snez rd, rs`	`sltu rd, x0, rs`	rd = (rs != 0)? 1:0
`sltz rd, rs`	`slt rd, rs, x0`	rd = (rs < 0)? 1:0
`sgtz rd, rs`	`slt rd, x0, rs`	rd = (rs > 0)? 1:0

Pseudo instructions for conditional branching		
Pseudo instruction	**Base instruction(s)**	**Description**
`beqz rs, imm`	`beq rs, x0, imm`	if (rs == 0) PC+=imm
`bnez rs, imm`	`bne rs, x0, imm`	if (rs != 0) PC+=imm
`blez rs, imm`	`bge x0, rs, imm`	if (rs <= 0) PC+=imm
`bgez rs, imm`	`bge rs, x0, imm`	if (rs >= 0) PC+=imm
`bltz rs, imm`	`blt rs, x0, imm`	if (rs < 0) PC+=imm
`bgtz rs, imm`	`blt x0, rs, imm`	if (rs > 0) PC+=imm
`bgt rs, rt, imm`	`blt rt, rs, imm`	if (rs > rt) PC+=imm
`ble rs, rt, imm`	`bge rt, rs, imm`	if (rs <= rt) PC+=imm
`bgtu rs, rt, imm`	`bltu rt, rs, imm`	if (rs > rt) PC+=imm, unsign.
`bleu rs, rt, imm`	`bgeu rt, rs, imm`	if (rs <= rt) PC+=imm, unsign.

Pseudo instructions for unconditional jumping		
Pseudo instruction	**Base instruction(s)**	**Description**

`j imm`	`jal x0, imm`	PC += imm
`jal imm`	`jal x1, imm`	x1 = PC+4; PC += imm
`jr rs`	`jalr x0, rs, 0`	PC = rs
`jalr rs`	`jalr x1, rs, 0`	x1 = PC+4; PC = rs
`ret`	`jalr x0, x1, 0`	PC = x1
`call imm`	`auipc x6, imm[31:12]` `jalr x1, x6, imm[11:0]`	x1 = PC+4; PC = imm
`tail imm`	`auipc x6, imm[31:12]` `jalr x0, x6, imm[11:0]`	PC = imm

Application binary interface (ABI) and user-mode			
Register	**ABI alias**	**Description**	**Saver**
x0	zero	zero constant	-
x1	ra	return address	caller
x2	sp	stack pointer	callee / function
x3	gp	global pointer	- / should not be used from user
x4	tp	thread pointer	- / should not be used from user
x5-x7	t0-t2	temporaries	caller
x8	s0 / fp	saved / Frame pointer	callee / function
x9	s1	saved register	callee / function
x10-x11	a0-a1	function args. / return values	caller
x12-x17	a2-a7	function arguments	caller
x18-x27	s2-s11	saved registers	callee / function
x28-x31	t3-t6	temporaries	caller
pc	-	program counter	-