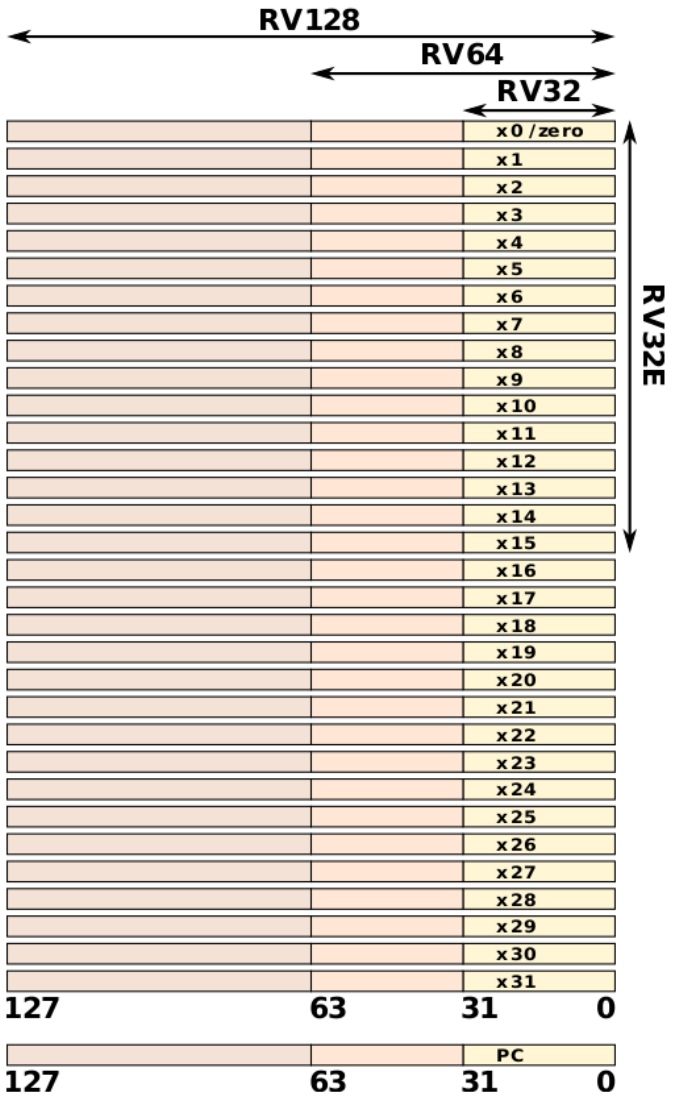
RISC-V 实现了一系列的伪指令，在此列出仅供参考，按英文字母顺序排列。

伪指令	基础指令	含义
beqz rs, offset	beq rs, x0, offset	寄存器为零分支跳转
bnez rs, offset	bne rs, x0, offset	寄存器不为零分支跳转
blez rs, offset	bge x0,rs,offset	寄存器小于等于零跳转
bgez rs, offset	bge rs, x0, offset	寄存器大于等于零跳转
bltz rs, offset	blt rs, x0, offset	寄存器小于零跳转
bgtz rs, offset	blt x0, xs, offset	寄存器大于零跳转
bgt rs, rt, offset	blt rt, rs, offset	比较大于分支跳转
ble rs, rt, offset	bge rt, rs, offset	比较小于等于分支跳转
bgtu rs, rt, offset	bltu rt, rs, offset	无符号比较大于分支跳转
bleu rs, rt, offset	bgeu rt, rs, offset	无符号比较小于等于分支跳转
call offset	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]	跳转 4KiB-4GiB 空间的函数
csrc csr, rs	csrrc x0, csr, rs	清除控制寄存器中对应比特
csrci csr, imm	csrrci x0, csr, imm	清除控制寄存器低 5 位中对应比特
csrs csr, rs	csrrs x0, csr, rs	置位控制寄存器中对应比特
csrsi csr, imm	csrrsi x0, csr, imm	置位控制寄存器低 5 位中对应比特
csrw csr, rs	csrrw x0, csr, rs	写控制寄存器中对应比特
csrwi csr, imm	csrrwi x0, csr, imm	写控制寄存器低 5 位中对应比特
fence	fence iorw, iorw	存储和外设同步指令
j offset	jal x0, offset	直接跳转指令
jal offset	jal x1, offset	子程序跳转和链接指令
jalr rs	jalr x1, rs, 0	子程序跳转寄存器和链接寄存器指令
jr rs	jalr x0, rs, 0	跳转寄存器指令
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	指令地址加载指令
li rd, immediate	根据立即数大小拆分为多条指令	立即数加载指令
l{b|h|w} rd, symbol, rt	auipc rt, symbol[31:12] l{b|h|w} rd, symbol[11:0](rt)	4GB 地址空间加载指令
mv rd, rs	addi rd, rs, 0	数据传送指令
neg rd, rs	sub rd, x0, rs	寄存器取负指令
nop	addi x0,x0,0	空指令
not rd, rs	xori rd, rs, -1	寄存器取反指令
ret	jalr x0, x1,0	子程序返回指令
s{b|h|w} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w} rd, symbol[11:0](rt)	4GiB 地址空间存储指令
seqz rd, rs	sltiu rd, rs, 1	寄存器为 0 置 1 指令
sgtz rd, rs	slt rd, rs, x0, rs	寄存器大于 0 置 1 指令
sltz rd, rs	slt rd, rs, rs, x0	寄存器小于 0 置 1 指令
snez rd, rs	sltu rd, rs, x0, rs	寄存器不为 0 置 1 指令
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	寄存器不链接跳转子程序指令

RISC-V Registers 寄存器

ABI（Application Binary Interface）描述了程序如何使用寄存器。
- 以字母 t（临时）开头的寄存器可用于任何目的。
- 以字母 a（参数）开头的寄存器用于传递给函数的参数。
- 以字母 s（已保存）开头的寄存器是跨函数调用保留的寄存器（sp 除外）。

Register name	ABI name (ABI（应用程序二进制接口）名称)	Description (描述)	Saved by
32 integer registers 32 个整数寄存器			
x0	zero	Always zero 始终为零	
x1	ra	Return address 返回地址	Caller
x2	sp	Stack pointer 堆栈指针	Callee
x3	gp	Global pointer 全局指针	
x4	tp	Thread pointer 线程指针	
x5	t0	Temporary / alternate return address 临时 / 备用返回地址	Caller
x6–7	t1–2	Temporary 临时寄存器	Caller
x8	s0/fp	Saved register / frame pointer 保存的寄存器 / 框架指针	Callee
x9	s1	Saved register 保存的寄存器	Callee
x10–11	a0–1	Function argument / return value 函数参数 / 返回值	Caller
x12–17	a2–7	Function argument 函数参数	Caller
x18–27	s2–11	Saved register 保存的寄存器	Callee
x28–31	t3–6	Temporary 临时寄存器	Caller
32 floating-point extension registers 32 个浮点扩展寄存器			
f0–7	ft0–7	Floating-point temporaries 浮点临时寄存器	Caller
f8–9	fs0–1	Floating-point saved registers 浮点保存寄存器	Callee
f10–11	fa0–1	Floating-point arguments/return values 浮点参数 / 返回值	Caller
f12–17	fa2–7	Floating-point arguments 浮点参数	Caller
f18–27	fs2–11	Floating-point saved registers 浮点保存寄存器	Callee
f28–31	ft8–11	Floating-point temporaries 浮点临时寄存器	Caller



示例：整数和浮点寄存器的使用

`lb t0, 8(sp)`	Load a byte from the memory address (sp + 8) into register t0. 从内存地址（sp + 8）加载一个字节到寄存器 t0。
`sb t0, 8(sp)`	Store a byte from register t0 into memory address (sp + 8). 将寄存器 t0 中的一个字节存储到内存地址（sp + 8）。
`add a0, t0, t1`	Add the value in t0 to the value in t1 and store the sum in a0. 将 t0 中的值与 t1 中的值相加，并将和存储在 a0 中。
`addi a0, t0, -10`	Add the value in t0 to the value -10 and store the sum in a0. 将 t0 中的值与 -10 相加，并将和存储在 a0 中。
`sub a0, t0, t1`	Subtract the value in t1 from value in t0 and store the difference in a0. 从 t0 中的值减去 t1 中的值，并将差值存储在 a0 中。
`mul a0, t0, t1`	Multiply the value in t0 with the value in t1 and store the product in a0. 将 t0 中的值与 t1 中的值相乘，并将积存储在 a0 中。
`div a1, s3, t3`	Divide the value in s3 (numerator) by the value in t3 (denominator) and store the quotient in a1. 将 s3（分子）中的值除以 t3（分母）中的值，并将商存储在 a1 中。
`rem a1, s3, t3`	Divide the value in s3 (numerator) by the value in t3 (denominator) and store the remainder in a1. 将 s3（分子）中的值除以 t3（分母）中的值，并将余数存储在 a1 中。
`and a3, t3, s3`	Perform the logical AND of t3 and s3 and store the result in a3. 对 t3 和 s3 进行逻辑与运算，并将结果存储在 a3 中。
`or a3, t3, s3`	Perform the logical OR of t3 and s3 and store the result in a3. 对 t3 和 s3 进行逻辑或运算，并将结果存储在 a3 中。
`xor a3, t3, s3`	Perform the logical XOR of t3 and s3 and store the result in a3. 对 t3 和 s3 进行逻辑异或运算，并将结果存储在 a3 中。

`flw ft0, 0(sp)`	Load in register ft0 the value at memory + 0 将内存 + 0 处的值加载到寄存器 ft0 中
`flw ft1, 4(sp)`	Load in register ft1 the value at memory + 4 将内存 + 4 处的值加载到寄存器 ft1 中
`fadd.s ft2, ft0, ft1`	Add the value in ft0 to the value ft1 and store the sum in ft2. Note: the fadd.s instruction to tell the RISC-V processor to add two single-precision values (ft0 and ft1) and store it as a single precision value into ft2.

将 ft0 中的值添加到值 ft1 中，并将总和存储在 ft2 中。

注意：fadd.s 指令告诉 RISC-V 处理器将两个单精度值（ft0 和 ft1）相加，并将其作为单精度值存储到 ft2 中。

32位 RISC-V 指令格式

32-bit RISC-V instruction formats

Format 格式	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register 寄存器	funct7							rs2					rs1					funct3			rd					opcode						
Immediate 立即数	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate 上位立即数	imm[31:12]																				rd					opcode						
Store 存储	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch 分支	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump 跳转	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

RISC-V

Examples of arithmetic instructions

Instruction	Type	Example	Description
Add	R	add rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
Subtract	R	sub rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
Add immediate	I	addi rd, rs1, imm12	R[rd] = R[rs1] + SignExt(imm12)
Set less than	R	slt rd, rs1, rs2	R[rd] = (R[rs1] < R[rs2])? 1 : 0
Set less than immediate	I	slti rd, rs1, imm12	R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0
Set less than unsigned	R	sltu rd, rs1, rs2	R[rd] = (R[rs1] <u R[rs2])? 1 : 0
Set less than immediate unsigned	I	sltiu rd, rs1, imm12	R[rd] = (R[rs1] <u SignExt(imm12))? 1 : 0
Load upper immediate	U	lui rd, imm20	R[rd] = SignExt(imm20 << 12)
Add upper immediate to PC	U	auipc rd, imm20	R[rd] = PC + SignExt(imm20 << 12)

RISC-V

Examples: logical instructions

Instruction	Type	Example	Description
AND	R	and rd, rs1, rs2	R[rd] = R[rs1] & R[rs2]
OR	R	or rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
XOR	R	xor rd, rs1, rs2	R[rd] = R[rs1] ^ R[rs2]
AND immediate	I	andi rd, rs1, imm12	R[rd] = R[rs1] & SignExt(imm12)
OR immediate	I	ori rd, rs1, imm12	R[rd] = R[rs1] | SignExt(imm12)
XOR immediate	I	xori rd, rs1, imm12	R[rd] = R[rs1] ^ SignExt(imm12)
Shift left logical	R	sll rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
Shift right logical	R	srl rd, rs1, rs2	R[rd] = R[rs1] >> R[rs2] (logical)
Shift right arithmetic	R	sra rd, rs1, rs2	R[rd] = R[rs1] >> R[rs2] (arithmetic)
Shift left logical immediate	I	slli rd, rs1, shamt	R[rd] = R[rs1] << shamt
Shift right logical imm.	I	srli rd, rs1, shamt	R[rd] = R[rs1] >> shamt (logical)
Shift right arithmetic immediate	I	srai rd, rs1, shamt	R[rd] = R[rs1] >> shamt (arithmetic)

RISC-V

Examples: data transfer instructions

Instruction	Type	Example	Description
Load doubleword	I	ld rd, imm12(rs1)	R[rd] = Mem8[R[rs1] + SignExt(imm12)]
Load word	I	lw rd, imm12(rs1)	R[rd] = SignExt(Mem4[R[rs1] + SignExt(imm12)])
Load halfword	I	lh rd, imm12(rs1)	R[rd] = SignExt(Mem2[R[rs1] + SignExt(imm12)])
Load byte	I	lb rd, imm12(rs1)	R[rd] = SignExt(Mem1[R[rs1] + SignExt(imm12)])
Load word unsigned	I	lwu rd, imm12(rs1)	R[rd] = ZeroExt(Mem4[R[rs1] + SignExt(imm12)])
Load halfword unsigned	I	lhu rd, imm12(rs1)	R[rd] = ZeroExt(Mem2[R[rs1] + SignExt(imm12)])
Load byte unsigned	I	lbu rd, imm12(rs1)	R[rd] = ZeroExt(Mem1[R[rs1] + SignExt(imm12)])
Store doubleword	S	sd rs2, imm12(rs1)	Mem8[R[rs1] + SignExt(imm12)] = R[rs2]
Store word	S	sw rs2, imm12(rs1)	Mem4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)
Store halfword	S	sh rs2, imm12(rs1)	Mem2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)
Store byte	S	sb rs2, imm12(rs1)	Mem1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)

RISC-V

Examples: control flow instructions

Instruction	Type	Example	Description
Branch equal	SB	beq rs1, rs2, imm12	if (R[rs1] == R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch not equal	SB	bne rs1, rs2, imm12	if (R[rs1] != R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch greater than or equal	SB	bge rs1, rs2, imm12	if (R[rs1] >= R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch greater than or equal unsigned	SB	bgeu rs1, rs2, imm12	if (R[rs1] >=u R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch less than	SB	blt rs1, rs2, imm12	if (R[rs1] < R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch less than unsigned	SB	bltu rs1, rs2, imm12	if (R[rs1] <u R[rs2]) pc = pc + SignExt(imm12 << 1)
Jump and link	UJ	jal rd, imm20	R[rd] = PC + 4 PC = PC + SignExt(imm20 << 1)
Jump and link register	I	jalr rd, imm12(rs1)	R[rd] = PC + 4 PC = (R[rs1] + SignExt(imm12)) & (~1)

RISC-V

Assembler pseudo-instructions

Pseudo-instruction	Base instruction(s)	Description
li rd, imm	addi rd, x0, imm	Load immediate
la rd, symbol	auipc rd, D[31:12]+D[11] addi rd, rd, D[11:0]	Load absolute address where D = symbol – pc
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
bgt{u} rs, rt, offset	blt{u} rt, rs, offset	Branch if > (u: unsigned)
ble{u} rs, rt, offset	bge{u} rt, rs, offset	Branch if ≥ (u: unsigned)
b{eq|ne}z rs, offset	b{eq|ne} rs, x0, offset	Branch if { = |≠ }
b{ge|lt}z rs, offset	b{ge|lt} rs, x0, offset	Branch if { ≥ | < }
b{le|gt}z rs, offset	b{ge|lt} x0, rs, offset	Branch if { ≤ | > }
j offset	jal x0, offset	Unconditional jump
call offset	jal ra, offset	Call subroutine (near)
ret	jalr x0, 0(ra)	Return from subroutine
nop	addi x0, x0, 0	No operation