2018

RISC-V 手册

一本开源指令集的指南

DAVID PATTERSON, ANDREW WATERMAN

翻译：勾凌睿、黄成、刘志刚
校阅：包云岗

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	`SLL rd,rs1,rs2`	`SLLW rd,rs1,rs2`
	Shift Left Log. Imm.	I	`SLLI rd,rs1,shamt`	`SLLIW rd,rs1,shamt`
	Shift Right Logical	R	`SRL rd,rs1,rs2`	`SRLW rd,rs1,rs2`
	Shift Right Log. Imm.	I	`SRLI rd,rs1,shamt`	`SRLIW rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRAW rd,rs1,rs2`
	Shift Right Arith. Imm.	I	`SRAI rd,rs1,shamt`	`SRAIW rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADDW rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDIW rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUBW rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	B	`BEQ rs1,rs2,imm`	
	Branch ≠	B	`BNE rs1,rs2,imm`	
	Branch <	B	`BLT rs1,rs2,imm`	
	Branch ≥	B	`BGE rs1,rs2,imm`	
	Branch < Unsigned	B	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	B	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	J	`JAL rd,imm`	
	Jump & Link Register	I	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
Environment	CALL	I	`ECALL`	
	BREAK	I	`EBREAK`	

Control Status Register (CSR)

Read/Write	I	`CSRRW`	`rd,csr,rs1`
Read & Set Bit	I	`CSRRS`	`rd,csr,rs1`
Read & Clear Bit	I	`CSRRC`	`rd,csr,rs1`
Read/Write Imm	I	`CSRRWI`	`rd,csr,imm`
Read & Set Bit Imm	I	`CSRRSI`	`rd,csr,imm`
Read & Clear Bit Imm	I	`CSRRCI`	`rd,csr,imm`

Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`LWU rd,rs1,imm`
				`LD rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`SD rs1,rs2,imm`

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	`MRET`
	Supervisor-mode trap return	R	`SRET`
Interrupt	Wait for Interrupt	R	`WFI`
MMU	Virtual Memory FENCE	R	`SFENCE.VMA rs1,rs2`

Examples of the 60 RV Pseudoinstructions

Branch = 0 (`BEQ rs,x0,imm`)	J	`BEQZ rs,imm`
Jump (uses `JAL x0,imm`)	J	`J imm`
MoVe (uses `ADDI rd,rs,0`)	R	`MV rd,rs`
RETurn (uses `JALR x0,0,ra`)	I	`RET`

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Float Load Word SP	CL	`C.FLW rd',rs1',imm`	`FLW rd',rs1',imm*8`
	Float Load Word	CI	`C.FLWSP rd,imm`	`FLW rd,sp,imm*8`
	Float Load Double	CL	`C.FLD rd',rs1',imm`	`FLD rd',rs1',imm*16`
	Float Load Double SP	CI	`C.FLDSP rd,imm`	`FLD rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Float Store Word	CS	`C.FSW rs1',rs2',imm`	`FSW rs1',rs2',imm*8`
	Float Store Word SP	CSS	`C.FSWSP rs2,imm`	`FSW rs2,sp,imm*8`
	Float Store Double	CS	`C.FSD rs1',rs2',imm`	`FSD rs1',rs2',imm*16`
	Float Store Double SP	CSS	`C.FSDSP rs2,imm`	`FSD rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
	AND	CR	`C.AND rd,rs1`	`AND rd,rd,rs1`
	AND Immediate	CI	`C.ANDI rd,imm`	`ANDI rd,rd,imm`
	OR	CR	`C.OR rd,rs1`	`OR rd,rd,rs1`
	eXclusive OR	CR	`C.XOR rd,rs1`	`AND rd,rd,rs1`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
	Shift Right Ari. Imm.	CI	`C.SRAI rd,imm`	`SRAI rd,rd,imm`
	Shift Right Log. Imm.	CI	`C.SRLI rd,imm`	`SRLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

Optional Compressed Extention: RV64C

All RV32C (except `C.JAL`, 4 word loads, 4 word strores) plus:

ADD Word (`C.ADDW`)	Load Doubleword (`C.LD`)
ADD Imm. Word (`C.ADDIW`)	Load Doubleword SP (`C.LDSP`)
SUBtract Word (`C.SUBW`)	Store Doubleword (`C.SD`)
	Store Doubleword SP (`C.SDSP`)

32-bit 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
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

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10 9	8 7	6 5	4 3	2 1 0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3		imm		rs2		op
CIW	funct3		imm			rd′	op
CL	funct3	imm	rs1′	imm		rd′	op
CS	funct3	imm	rs1′	imm		rs2′	op
CB	funct3	offset	rs1′	offset			op
CJ	funct3	jump target					op

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers `x1`-`x31` and the `PC` are 32 bits wide in RV32I and 64 in RV64I (`x0`=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-0,ft0-7	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	SETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,rs3
Multiply-SUB	R	VFMSUB	rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2
MERGE	R	VMERGE	rd,rs1,rs2
SELECT	R	VSELECT	rd,rs1,rs2

RISC-V calling convention and five optional extensions: 8 RV32M; 11 RV32A; 34 floating-point instructions each for 32- and 64-bit data (RV32F, RV32D); and 53 RV32V. Using regex notation, {} means set, so `FADD.{F|D}` is both `FADD.F` and `FADD.D`. RV32{F|D} adds registers `f0-f31`, whose width matches the widest precision, and a floating-point control and status register `fcsr`. RV32V adds vector registers `v0-v31`, vector predicate registers `vp0-vp7`, and vector length register `vl`. RV64 adds a few instructions: RVM gets 4, RVA 11, RVF 6, RVD 6, and RVV 0.

目录

参考卡 **1**
致谢 **7**
关于作者 **9**
前言 **10**
译者序 **12**
翻译团队 12
第一章 为什么要有 RISC-V? **13**
1.1 导言 13
1.2 模块化与增量型 ISA 14
1.3 ISA 设计 101 15
1.4 全书的总览 19
1.5 结束语 20
1.6 扩展阅读 21
第二章 RV32I：RISC-V 基础整数指令集 **23**
2.1 导言 23
2.2 RV32I 指令格式 23
2.3 RV32I 寄存器 26
2.4 RV32I 整数计算 27
2.5 RV32I 的 Load 和 Store 29
2.6 RV32I 条件分支 30
2.7 RV32I 无条件跳转 31
2.8 RV32I 杂项 31
2.9 使用插入排序比较 RV32I，ARM-32，MIPS-32 和 x86-32 指令集 32
2.10 结束语 32
2.11 扩展阅读 33
第三章 RISC-V 汇编语言 **41**
3.1 导言 41
3.2 函数调用规范（Calling convention） 41
3.3 汇编器 43
3.4 链接器 46
3.5 静态链接和动态链接 49
3.6 加载器 49
3.7 结束语 49

3.8 扩展阅读 50
第四章 乘法和除法指令 51
4.1 导言 51
4.2 结束语 53
4.3 扩展阅读 53
第五章 RV32F 和 RV32D：单精度和双精度浮点数 55
5.1 导言 55
5.2 浮点寄存器 55
5.3 浮点加载，存储和算术指令 56
5.4 浮点转换和搬运 60
5.5 其他浮点指令 60
5.6 使用 DAXPY 程序比较 RV32FD，ARM-32，MIPS-32 和 x86-32 62
5.7 结束语 62
5.8 扩展阅读 62
第六章 原子指令 65
6.1 导言 65
6.2 结束语 67
6.3 扩展阅读 67
第七章 压缩指令 69
7.1 导言 69
7.2 RV32GC，Thumb-2，microMIPS 和 x86-32 的比较 71
7.3 结束语 71
7.4 扩展阅读 71
第八章 向量 75
8.1 导言 75
8.2 向量计算指令 76
8.3 向量寄存器和动态类型 76
8.4 向量的 Load 和 Store 操作 78
8.5 向量操作期间的并行性 78
8.6 向量运算的条件执行 79
8.7 其他向量指令 79
8.8 例子：用 RV32V 写成的 DAXPY 程序 80
8.9 RV32V，MIPS-32 MSA SIMD 和 x86-32 AVX SIMD 的比较 81

8.10 结束语 83
8.11 扩展阅读 84
第九章 RV64：64 位地址指令 87
9.1 导言 87
9.2 使用插入排序来比较 RV64 与其他 64 位 ISA 91
9.3 程序大小 93
9.4 结束语 93
9.5 了解更多 94
第十章 RV32/64 特权架构 99
10.1 导言 99
10.2 简单嵌入式系统的机器模式 100
10.3 机器模式下的异常处理 101
10.4 嵌入式系统中的用户模式和进程隔离 104
10.5 现代操作系统的监管者模式 105
10.6 基于页面的虚拟内存 106
10.7 结束语 110
10.8 扩展阅读 111
第十一章 RISC-V 未来的可选扩展 113
11.1 “B”标准扩展：位操作 113
11.2 “E”标准扩展：嵌入式 113
11.3 “H”特权态架构扩展：支持管理程序（Hypervisor） 113
11.4 “J”标准扩展：动态翻译语言 113
11.5 “L”标准扩展：十进制浮点 113
11.6 “N”标准扩展：用户态中断 113
11.7 “P”标准扩展：封装的单指令多数据（Packed-SIMD）指令 114
11.8 “Q”标准扩展：四精度浮点 114
11.9 结束语 114
附录 A RISC-V 指令列表 115

对这本 RISC-V 手册的称赞

我喜欢 RISC-V 和这本书，因为它们优雅——简洁、扼要且完整。书中的评论无偿提供了一些历史，设计的动机，以及一些对于各种架构的批评。

——C. Gordan Bell，微软公司成员，Digital PDP-11 和 VAX-11 指令集架构的设计者

本书讲述了 RISC-V 可以做到什么，以及为什么它的设计师选择赋予它这些能力。更有趣的是，作者说出了为什么 RISC-V 省略了早期计算机中存在的一些东西。这些原因至少和 RISC-V 本身能做到什么与忽略了什么一样有意思。

——Ivan Sutherland，图灵奖获得者，被称作计算机图形学之父

RISC-V 会改变世界，这本书会助你成为改变的一部分。

——Michael B. Taylor，华盛顿大学教授

RISC-V 是学生学习指令集架构和汇编级编程的理想选择，而它们是以后用高级语言工作的基础。这本写得很清楚的书提供了对 RISC-V 的很好的介绍，再加上一些对其演化历史的深刻见解及与其它常见架构的比较。以过去的指令集架构为鉴，RISC-V 的设计者能够避免一些不必要的、不合理的特征，这让教学过程变得容易。即使它很简洁，它也足够强大，能在实际应用中广泛使用。很久以前我教过汇编编程的入门课，如果我现在去教这门课的话，我会很乐意用这本书作为教材。

——John Mashey，MIPS 指令集架构的设计者之一

这本书对于任何使用 RISC-V ISA 的人来说都是十分宝贵的参考。为了便于快速查阅，操作码按几种有用的格式呈现，这让编写和解释汇编代码变得简单。此外，对于如何使用这个 ISA 的解释和示例也让程序员的工作更容易。和其他 ISA 比较的部分很有意思，它们解释了 RISC-V 设计者们做出他们的设计决策的原因。

——Megan Wachs，博士，SiFive 工程师

致谢

David Patterson 把这本书献给他的父母：

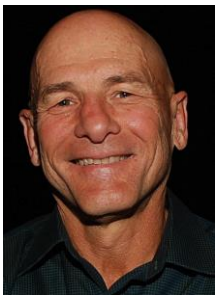
——给我的父亲 David，我从他那儿继承了创造力、运动天赋和为正义奋斗的勇气；以及

——给我的母亲 Lucy，我从她那儿继承了智慧、乐观和良好的性格。

感谢你们成为我如此伟大的榜样，你们让我明白了成为好的配偶、父亲和祖父的意义。

Andrew Waterman 把这本书献给他的父母 John 和 Elizabeth，他们即便在千里之外也支持着他。

关于作者



David Patterson 在加州大学伯克利分校担任计算机科学教授 40 年后于 2016 年退休，随后加入 Google 担任杰出工程师（distinguished engineer，Google 的职位）。他还担任了 RISC-V 基金会董事会副主席。过去，他曾被任命为伯克利计算机科学部（Computer Science Division）主席，并当选为计算机研究协会（Computing Research Association）主席和计算机协会（ACM，Association for Computing Machinery）主席。在 20 世纪 80 年代，他领导了四代精简指令集计算机（RISC，Reduced Instruction Set Computer）项目，伯克利最新的 RISC 因此得名“RISC Five”。他和 Andrew Waterman 都是 RISC-V 四位架构师中的一员。除了 RISC 以外，他最著名的研究项目有廉价磁盘冗余阵列（RAID，Redundant Arrays of Inexpensive Disks）和工作站网络（NOW，Networks of Workstations）。这项研究让他发表了许多论文，出版了 7 本书，获得了超过 35 个荣誉，包括当选国家工程院和国家科学院院士，名列硅谷工程师名人堂，还成为了计算机历史博物馆、ACM、IEEE 和两个 AAAS 组织的研究员。他的教学奖项包括杰出教学奖（加州大学伯克利分校），Karstrom 杰出教育家奖（ACM），Mulligan 教育奖章（IEEE）和本科教学奖（IEEE）。他还因为一本计算机体系结构方面的书和一本关于软件工程的书被文本和学术作家协会（Text and Academic Authors Association）授予计算机教科书卓越奖（“Texty”）。他在加州大学洛杉矶分校获得了他的所有学位，也被授予了杰出工程学院校友奖。他在南加州长大，乐趣是踢足球、和他的儿子一起骑自行车，以及和妻子在沙滩上散步。他们在高中时期就是情侣。在本书的测试版出版几天后，他们庆祝了 50 周年结婚纪念日。

Andrew Waterman 是 SiFive 的总工程师和联合创始人。SiFive 由 RISC-V 架构的创建者们建立，旨在提供基于 RISC-V 的低成本定制芯片。他在加州大学伯克利分校获得了计算机科学博士学位。在那里，他厌倦了现有的指令集架构的变幻莫测，于是共同设计了 RISC-V ISA 和第一台 RISC-V 微处理器。Andrew 是基于开源 RISC-V 的 Rocket 芯片生成器、Chisel 硬件构造语言以及 Linux 操作系统内核和 GNU C 编译器和 C 库的 RISC-V 端口的主要贡献者之一。他还有加州大学伯克利分校的硕士学位，这是 RISC-V 的 RVC 扩展的基础，他还有杜克大学的工学学士学位。

前言

欢迎！

RISC-V 自 2011 年推出以来迅速地普及。我们认为一个精简的程序员指南将进一步促进它的发展，并促使新人理解为什么它是一个有吸引力的指令集，以及它与传统指令集架构(ISA)的不同。

我们的灵感部分来源于其它指令集架构书籍，但我们希望 RISC-V 自身的简洁性能让我们写得比 See MIPS Run 一类 500 多页的详尽书籍少很多。我们把全书的长度控制到了前述的三分之一，至少在这个意义上我们成功了。实际上，介绍模块化 RISC-V 指令集的每个组成部分的十章只用了 100 页——即便为了有助于快速阅读，平均每页用到了一张图片（一共 75 张）。

在解释指令集设计的原理之后，我们将阐述 RISC-V 架构师在设计指令集的时候，如何在过去 40 年的指令集的基础上取其精华，去其糟粕。要评判一个指令集架构，不仅要看它包括了什么，而且要看它省略了什么。

随后我们会按顺序介绍这个模块化架构的每个组成部分。每一章都会包含一个用 RISC-V 汇编语言写成的程序，这是为了展示那一章所述的指令的用法，这样有助于汇编语言程序员学习 RISC-V 汇编。有时，我们还会列出用 ARM，MIPS 和 x86 写成的同样的程序，从而突出 RISC-V 在简洁性，以及成本、功耗、性能方面的优势。

为了增加本书的趣味性，我们在页边加入了将近 50 个侧边栏，这里面放了一些有关书中内容的评论，希望它们能带来一些乐趣。我们还在页边放了大约 75 个图片，用于展示设计良好 ISA 的例子。（我们充分利用了侧边的空间！）最后，对于那些愿意钻研的读者，我们在全书中加入了大概 25 段补充说明。如果你对某个主题感兴趣，可以深入研究这些可选部分。略过这些部分不会影响对书中的其他内容的理解，所以如果你对他们不感兴趣的话，尽管跳过它们。对于计算机体系结构爱好者，我们援引的 25 篇论文和书籍能够开阔你的视野。在写这本书的过程中，我们从它们当中学到了很多东西！

为什么引用了这么多名言？

我们认为引用这些名言也能增加本书的趣味性，因此我们把这 25 个引用分散在整本书里。 它们同样是一种将智慧从前辈传递给初学者的有效机制，且有助于为良好的 ISA 设计设定文化标准。我们希望读者也能了解一点该领域的历史，这就是为什么我们在全书中引用了众多著名计算机科学家和工程师的名言。

导言和参考

我们打算将这本薄薄的书作为RISC-V的介绍和参考资料，供有兴趣编写RISC-V代码的学生和嵌入式系统程序员使用。本书假设读者事先已经了解过至少一个指令集。如果没有，您可能希望浏览基于RISC-V的相关入门架构手册：*Computer Organization and Design RISC-V Edition: The Hardware Software Interface*。

这本书中的参考资料包括：

- **参考卡**——这个一页（两面）的RISC-V的精简描述囊括了RV32GCV和RV64GCV，同时包含了基本内容和所有已定义的指令扩展：RVI, RVM, RVA, RVF, RVD，甚至包括了尚处在开发阶段的RVV。
- **指令图**——每个指令扩展的半页图形描述（它们是每章的第一个图）以同样的格式列出了所有RISC-V指令的全称，让大家可以轻松查看每条指令的不同变种。见

图2.1、4.1、5.1、6.1、7.1、8.1、9.1、9.2、9.3和9.4。

- **操作码映射——**这些表格在一页中显示了指令布局，操作码，格式类型和每页指令扩展的指令助记符。见图2.3、3.3、3.4、4.2、5.2、5.3、6.2、7.6、7.5、7.7、9.5和10.1。（这些指令图和操作码映射启发了我们在书的副标题中使用单词图集。）
- **指令术语表——**附录A是对每个RISC-V指令和伪指令的详尽描述[1]。它包括所有内容：操作名称和操作数、英文描述、寄存器传输语言定义、它所在的RISC-V扩展、指令的全称、指令格式、显示操作码的指令图，以及紧凑版本指令的参照。令人惊讶的是，所有这些加起来不到50页。
- **索引——**它可以帮你通过指令全称或助记符找到描述指令说明、定义或图表的页面。它是按照字典的形式组织的。

勘误和补充内容

我们打算把勘误集中起来，每年发布几次更新。这本书的网站上会有本书的最新版本，还会简单介绍一下当前版本相对上一版本的改变。可在本书的网站（www.riscvbook.com）上查看勘误表的历史版本或报告新的错误。我们预先为您在这一版中发现的问题表示歉意。我们期待您的反馈意见，来帮助我们改进这本书。

本书的诞生过程

在2017年5月8日至11日在上海举行的第六届RISC-V研讨会上，我们认识到了对这么一本书的需求，几个星期后我们开始了编写。考虑到Patterson在写书方面的丰富经验，我们计划让他写大部分的章节。我们两人在组织方面进行了合作，并且是彼此章节的第一个评论者。Patterson撰写了第1、2、3、4、5、6、7、8、9、11章，参考卡和本前言，而Waterman写了第10章和附录A（本书的最大部分），并编写了书中的全部程序。Waterman还维护了Armando Fox提供的LaTeX工具，使我们能做出这本书。

我们在2017年秋季学期为800名加州大学伯克利分校的学生提供了这本教科书的测试版本。在融入了他们的反馈后，当2017年学期结束以后，第一个正式版将于2017年11月28日至30日在硅谷举办的第七届RISC研讨会上及时发布。

RISC-V是一个伯克利研究项目的副产品。该项目正在针对更容易地同时构建硬件和软件的目标进行开发。

致谢

我们要感谢Armando Fox，因为我们使用了他的LaTeX工具，以及采纳了他关于个人出版的建议。

我们最深切的感谢要送给那些读了本书早期的草稿并提出了有用建议的人，比如：Krste Asanovi´c, Nikhil Athreya, C. Gordon Bell, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor, Megan Wachs,….

最后，我们要感谢数百名加州大学伯克利分校学生在调试方面的付出以及他们的对这些素材的持续热忱！

David Patterson 和 Andrew Waterman

2017年9月1日于加州伯克利

[1] 定义 RV32V 的委员会没有赶在本书的测试版本之前完成他们的工作，所以我们在附录 A 中省略了这些指令。尽管到时候 RV32V 有可能会有一些微小的改变，第八章是我们对于它的最为接近的猜想。

译者序

大约四个月前的一天，我收到加州大学伯克利分校毕业的谭章熹博士的消息，告知图灵奖得主David Patterson教授（谭博士的导师）希望将他和Andrew Waterman一起完成的《The RISC-V Reader》翻译成中文。这让我想起了大约四年前的一天，我收到了正在加州大学伯克利分校做博士后的钱学海博士的邮件，告知David Patterson教授希望将他和Krste Asanovic教授一起撰写的文章在中文杂志上发表。当我收到了中文翻译稿《指令系统应该免费：RISC-V的案例》后，动用了专栏编委的一点小权利，立刻向《中国计算机学会通讯》强烈推荐了这篇文章。文章很快于2015年2月发表，然而略显遗憾的是彼时它并未得到广泛关注，不过却让我们团队“近水楼台先得月”，成为国内最早将体系结构前沿研究全面转到了RISC-V平台的团队。这一次，怀着对RISC-V的感激之情、对Patterson教授的敬仰之心，更是为了便于更多中国爱好者了解RISC-V的愿望，我欣然接受了谭博士的请求。

过去几年深入接触RISC-V后，我心中时常呈现出一种愿景——RISC-V很可能像Linux那样开启开源芯片设计的黄金时代。事实上，伯克利的科研侠客们发明RISC-V就是希望“Instruction Sets Want to be Free”——全世界任何公司、大学、研究机构与个人都可以开发兼容RISC-V指令集的处理器，都可以融入到基于RISC-V构建的软硬件生态系统，而不需要为指令集付一分钱。这是伟大的理想！

在开源软件生态中，Linux是整个生态的基石。基于Linux，人们开发Python、LLVM、GCC等完整的工具链，创造MySQL、Apache、Hadoop等大量开源软件，实验各种创新思想与技术，逐渐形成一个价值超过150亿美元的开源软件生态。这对中国的互联网产业的意义尤为重大，不仅提升了BAT等互联网企业的技术研发能力，也大大降低了互联网产业创新的门槛，如今3-5位开发人员在几个月时间里就能快速开发出一个互联网应用。在芯片设计领域，RISC-V有望像Linux那样成为计算机芯片与系统创新的基石。但是只有RISC-V又是远远不够的，更重要的是要形成一个基于RISC-V的开源芯片设计生态，包括开源工具链、开源IP、开源SoC等等。

RISC-V还只是星星之火，却已展露出燎原之潜力。作为全世界最大的芯片用户，中国一直希望能把芯片产业做大做强，各方也都在努力。借鉴开源软件对于中国互联网产业发展的作用，也许开源芯片设计是一条值得尝试的道路。这本中文版《RISC-V手册》，希望能成为这条道路上一个小路标。

包云岗
2018 年 11 月 3 日

翻译团队

- ✧ **包云岗** 中科院计算所研究员，先进计算机系统研究中心主任，长期从事计算机体系结构研究
- ✧ **勾凌睿** 中国科学院大学计算机科学与技术专业本科生，有志于计算机体系结构研究
- ✧ **黄成** 中科院计算所研究生，研究方向为计算机系统结构
- ✧ **刘志刚** 中科院计算所硕士研究生，开展基于 RISC-V 的前沿研究，曾获第二届全国大学生计算机系统能力培养大赛一等奖

第一章 为什么要有 RISC-V？

简约是复杂的最终形式。 ——列奥纳多·达·芬奇(Leonardo da Vinci)

1.1 导言

列奥纳多·达·芬奇（1452-1519）是一位文艺复兴时期的建筑师，工程师，雕塑家，同时也是一名画家，创作了著名的《蒙娜丽莎的微笑》

RISC-V（“RISC five”）的目标是成为一个通用的指令集架构（ISA）：

- 它要能适应包括从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器。
- 它应该能兼容各种流行的软件栈和编程语言。
- 它应该适应所有实现技术，包括现场可编程门阵列（FPGA）、专用集成电路（ASIC）、全定制芯片，甚至未来的设备技术。
- 它应该对所有微体系结构样式都有效：例如微编码或硬连线控制;顺序或乱序执行流水线; 单发射或超标量等等。
- 它应该支持广泛的专业化，成为定制加速器的基础，因为随着摩尔定律的消退，加速器的重要性日益提高。
- 它应该是稳定的，基础的指令集架构不应该改变。更重要的是，它不能像以前的专有指令集架构一样被弃用，例如AMD Am29000、Digital Alpha、Digital VAX、Hewlett Packard PA-RISC、Intel i860、Intel i960、Motorola 88000、以及Zilog Z8000。

RISC-V的不同寻常不仅在于它是一个最近诞生的指令集架构（它诞生于最近十年，而大多数其他指令集都诞生于20世纪70到80年代），而且在于它是一个开源的指令集架构。与几乎所有的旧架构不同，它的未来不受任何单一公司的浮沉或一时兴起的决定的影响（这一点让许多过去的指令集架构都遭了殃）。它属于一个开放的，非营利性质的基金会。RISC-V基金会的目标是保持RISC-V的稳定性，仅仅出于技术原因缓慢而谨慎地发展它，并力图让它之于硬件如同Linux之于操作系统一样受欢迎。图1.1列出了RISC-V基金会最大的企业成员，作为其活力的证明。

我们在页边加入了侧边栏，是希望能在里面放入一些有意思的评论。比如，RISC-V 最初是为加州大学伯克利分校的内部研究和课程开发的。外部人员的使用使它变得开放。RISC-V 架构师在开始收到有关网上 ISA 课程变化的投诉时就了解到了来自外部的兴趣。只有在架构师理解了需求之后，他们才会尝试把它变为一个开放的 ISA 标准。

>$50B		>$5B, <$50B		>$0.5B, <$5B	
Google	USA	BAE Systems	UK	AMD	USA
Huawei	China	MediaTek	Taiwan	Andes Technology	China
IBM	USA	Micron Tech.	USA	C-SKY Microsystems	China
Microsoft	USA	Nvidia	USA	Integrated Device Tech.	USA
Samsung	Korea	NXP Semi.	Netherlands	Mellanox Technology	Israel
		Qualcomm	USA	Microsemi Corp.	USA
		Western Digital	USA		

图1.1：2017年5月第六届RISC-V研讨会上RISC-V基金会的企业成员按年销售额排名。左栏公司的年销售额均超过500亿美元，中间栏目公司的销售额低于500亿美元但超过50亿美元，右栏的销售额低于50亿美元但超过5亿美元。RISC-V基金会包括另外25家小公司，5家初创公司（Antmicro Ltd，Blockstream，Esperanto Technologies，Greenwaves Technologies和SiFive），4家非营利组织（CSEM，Draper Laboratory，ICT和lowRISC）和6所大学（ETH Zurich，IIT Madras，National University of Defense Technology，Princeton和UC Berkeley）。 60个组织中的大多数总部都在美国以外。要了解更多信息，请访问www.riscv.org。

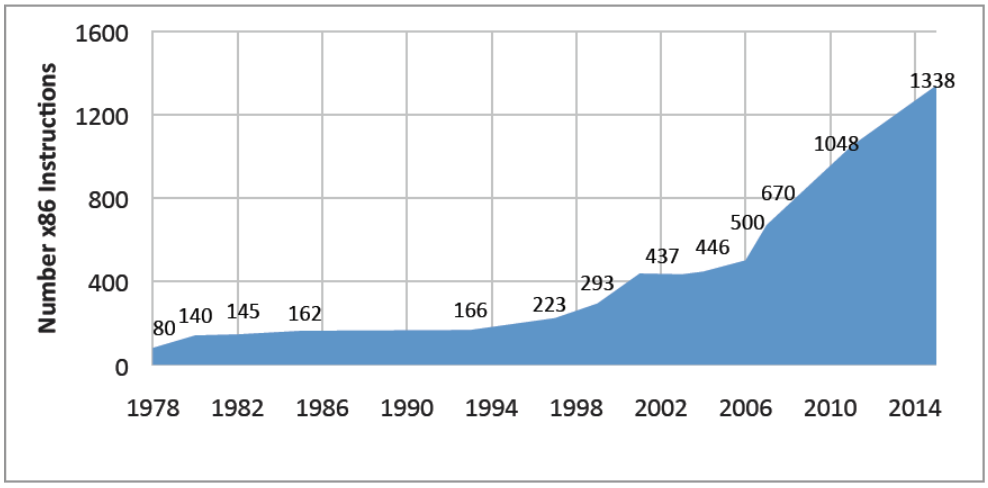


图1.2：x86指令集自诞生以来指令数量的增长。x86在1978年诞生时有80条指令，到2015年增长了16倍，到了1338条指令，并且仍在增长。令人惊讶的是这张图的数据仍显保守。2015年在英特尔的博客上有着3600条指令的统计结果[Rodgers and Uhlig 2017]，这意味着x86指令的增长速率提高到了（在1978年到2015年之内）每四天增长一条。我们是用汇编语言指令计算的，他们想必算入了机器语言指令。正如第八章所解释的那样，这个增长的很大一部分是因为x86 ISA依赖于SIMD指令来实现数据级并行。

```
The AL register is the default source and destination.
If the low 4-bits of AL register are > 9,
   or the auxiliary carry flag AF = 1,
 Then
    Add 6 to low 4-bits of AL and discard overflow
    Increment the high byte of AL
    Carry flag CF = 1
    Auxiliary carry flag AF = 1
 Else
    CF = AF = 0
Upper 4-bits of AL = 0
```

图1.3：x86-32 ASCII Adjust after Addition（aaa）指令的描述。它以二进制编码十进制数（BCD）形式进行计算机运算，这种方式已经被扔进信息技术历史的垃圾堆里。x86还有三个相似的指令，分别执行减法操作（aas），乘法操作（aam），和除法操作（aad）。由于他们都是单字节指令，它们加起来占用了宝贵的操作码空间的1.6%（4/256）。

1.2 模块化与增量型 ISA

英特尔曾将其未来押在高端微处理器之上，但那时还需要很多年时间。为了对抗Zilog，英特尔开发了一款过渡产品，并给它起名为8086。它本应该是短命的，没有任何继任者，但事情并非如此。高端处理器姗姗来迟，等它最终出现时，它的性能并不如人意。因此，8086架构延续了下去——它变成32位处理器，最终演变为了64位处理器。它的名称不断变化（80186,80286，i386，i486，Pentium），但基础指令集保持不变。

——Stephen P. Morse, 8086的架构师[Morse 2017]

计算机体系结构的传统方法是增量ISA，新处理器不仅必须实现新的ISA扩展，还必须实现过去的所有扩展。目的是为了保持向后的二进制兼容性，这样几十年前程序的二进制版本仍然可以在最新的处理器上正确运行。这一要求与来自于同时发布新指令和新处理器的营销上的诱惑共同导致了ISA的体量随时间大幅增长。例如，图1.2显示了当今主导ISA 80x86的指令数量增长过程。这个指令集架构的历史可以追溯到1978年，在它的漫长生涯中，它平均每个月增加了大约三条指令。

这个传统意味着x86-32（我们用它表示32位地址版本的x86）的每个实现必须实现过去的扩展中的错误设计，即便它们不再有意义。例如，图1.3描述了x86的ASCII Adjust after Addition（aaa）指令，该指令早已失效。

作为一个类比，假设一家餐馆只提供固定价格的餐点，最初只是一顿包含汉堡和奶昔的小餐。随着时间的推移，它会加入薯条，然后是冰淇淋圣代，然后是沙拉，馅饼，葡萄酒，素食意大利面，牛排，啤酒，无穷无尽，直到它成为一顿大餐。食客可以在那家餐厅找到他们过去吃过的东西，尽管总的来说这样做可能没什么意义。这样做的坏处是，用餐者为每次晚餐支付的宴会费用不断增加。

如果软件使用来自可选扩展的省略的 RISC-V 指令，则硬件会在软件中捕获并执行所需的功能，作为标准库的一部分。

RISC-V的不同寻常之处，除了在于它是最近诞生的和开源的以外，还在于：和几乎所有以往的ISA不同，它是模块化的。它的核心是一个名为*RV32I*的基础ISA，运行一个完整的软件栈。RV32I是固定的，永远不会改变。这为编译器编写者，操作系统开发人员和汇编语言程序员提供了稳定的目标。模块化来源于可选的标准扩展，根据应用程序的需要，硬件可以包含或不包含这些扩展。这种模块化特性使得RISC-V具有了袖珍化、低能耗的特点，而这对于嵌入式应用可能至关重要。RISC-V编译器得知当前硬件包含哪些扩展后，便可以生成当前硬件条件下的最佳代码。惯例是把代表扩展的字母附加到指令集名称之后作为指示。例如，RV32IMFD将乘法（RV32M），单精度浮点（RV32F）和双精度浮点（RV32D）的扩展添加到了基础指令集（RV32I）中。

继续用我们刚才的类比来说，RISC-V提供的是菜单，而不是一顿应有尽有的自助餐。主厨只需要烹饪顾客需要的东西（而不是每次都做出一顿盛宴），顾客只需要按他们的订单付费。RISC-V无需仅仅为了市场吸引力而添加指令。RISC-V基金会会决定什么时候在菜单里添加新的选项，而他们只会出于技术原因这样做，而且要在由软硬件专家组成的委员会进行专门的公开讨论以后才会添加。即使那些新选择出现在了菜单上，它们仍是可选的，不会像在增量ISA中那样成为未来所有实现的必要组成部分。

1.3 ISA 设计 101

在介绍 RISC-V 这个 ISA 之前，了解计算机架构师在设计 ISA 时的基本原则和必须做出的权衡是有用的。如下的列表列出了七种衡量标准。页边放置了对应的七个图标，以突出显示 RISC-V 在随后章节中应对它们的实例。（印刷版的封底有所有图标的图例。）

- 成本（美元硬币）
- 简洁性（轮子）
- 性能（速度计）
- 架构和具体实现的分离（分开的两个半圆）
- 提升空间（手风琴）
- 程序大小（相对的压迫着一条线的两个箭头）
- 易于编程/编译/链接（儿童积木“像 ABC 一样简单”）

为了解释我们的意思，在这一节中我们会展示一些以往 ISA 所作出的选择。它们现在

看起来是不明智的，而 RISC-V 通常会做出更好的决定。

成本 处理器通过集成电路实现，通常称为芯片或晶粒。它们叫做晶粒是因为，它们由一些单个的圆形晶片被切割成许多单独的片得到。图 1.4 显示了 RISC-V 处理器的晶圆。成本对晶粒面积十分敏感：

$$cost \approx f(die\ area^2)$$

显然，晶粒越小，每个晶圆上能切割出来的晶粒越多。晶粒的大部分成本来自于处理过的晶圆本身。不太直观的是，晶粒越小，产率（生产出的可用晶粒所占的比例）越高。原因在于目前的硅生产工艺会在晶圆上留下一些散布的小瑕疵。因此晶粒越小，有缺陷部分所占比重会越低。

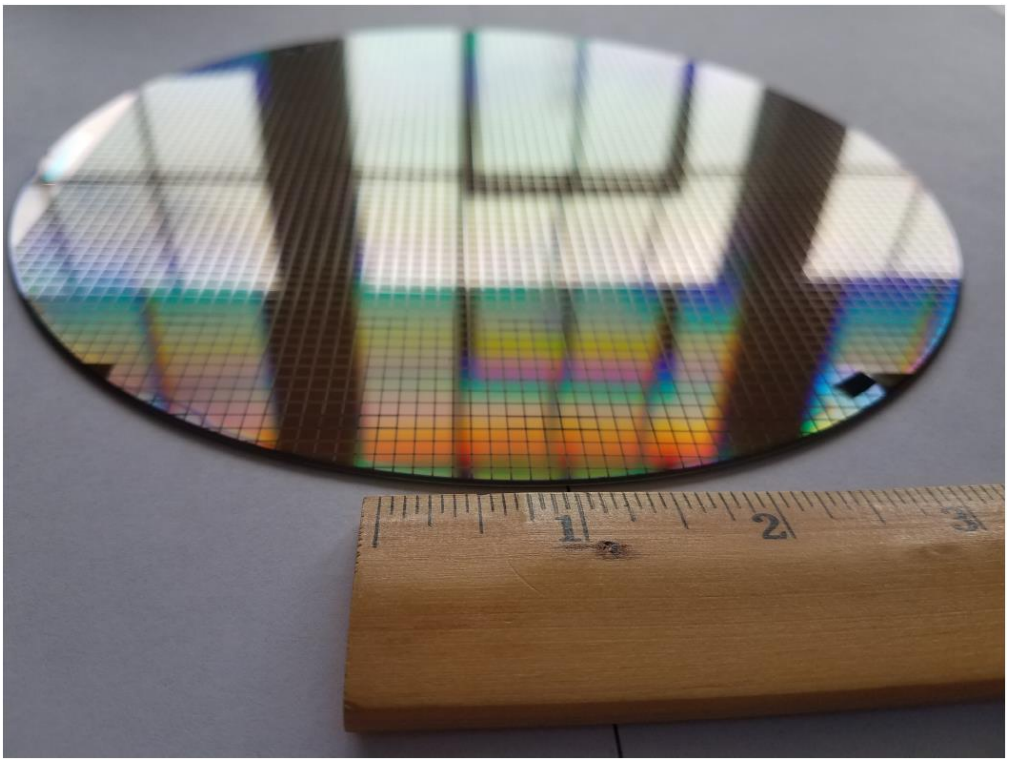


图1.4：由SiFive设计的直径为8英寸的RISC-V晶圆。 它有两种类型的RISC-V芯片，使用较旧的较大加工线。 FE310芯片为2.65mm×2.72mm，SiFive测试芯片为2.89mm×2.72mm。 一片晶圆上有1846片FE310和1866片SiFive测试芯片，总共3712个芯片。

架构师希望保持 ISA 的简洁性，从而缩小实现 ISA 的处理器的尺寸。我们将在随后的章节看到，RISC-V ISA 比 ARM-32 ISA 简洁得多。就简洁性造成的影响举例，我们把使用相同大小缓存（16KiB）的 RISC-V Rocket 处理器和采用相同技术（TSMC40GPLUS）的 ARM-32 Cortex A5 处理器进行比较。RISC-V 晶粒的大小是 0.27mm^2，而 ARM-32 晶粒的大小是 0.53mm^2。由于面积大一倍，ARM-32 Cortex A5 的晶粒成本是 RISC-V Rocket 的约 4（2^2）倍。即使晶粒的大小只缩小 10%，成本也将以 1.2（1.1^2）倍的比例缩小。

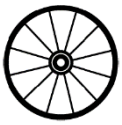
高端处理器可以通过将简单的指令组合在一起来提升性能，而不会因更大，更复杂的 ISA 给所有低端实现带来负担。这种技术称为宏观融合，因为它将“宏”指令融合在一起。

简洁性 鉴于成本对于复杂度的敏感性，架构师需要一个简单的 ISA 来缩小芯片面积。ISA 的简洁性还能缩短芯片的设计和验证时间，而它们可能构成了芯片开发的大部分成本。这些成本必须算到芯片的成本当中。这个开销取决于发货芯片的数量。简洁性还能降低文档成本，

让客户更容易了解如何使用这个 ISA。

以下是 ARM-32 的 ISA 复杂性的一个明显示例：

```
ldmiaeq SP!, {R4-R7, PC}
```



简单的处理器对嵌入式应用程序有益，因为它更容易预测执行时间。微控制器的汇编语言程序员通常希望保持精确的时序，因此他们会保持代码执行所需的时钟周期数可预测并可以手动数出来。

该指令代表 LoaD Multiple, Increment-Address, on EQual。它执行 5 次数据加载并写入 6 个寄存器，但仅当 EQ 条件码置位时才执行。此外，它将结果写入 PC，因此它也执行条件分支。真不少！

具有讽刺意味的是，即便需要实现的功能相同，简单指令也通常比复杂指令更容易被用到。例如，x86-32 有一个 enter 指令，该指令本应该是在进入一个创建一个栈帧的过程中执行的第一条指令（见第三章）。大多数编译器用两条简单的 x86-32 指令来代替它：

```
push ebp          # 将帧指针压入栈
mov  ebp, esp     # 把栈指针复制到帧指针
```

性能 除非是那些用于嵌入式应用的微型芯片，处理器的性能和成本通常都能成为架构师的关注对象。性能可以分解为如下三个因素：

$$\frac{instructions}{program} \times \frac{average\ clock\ cycles}{instruction} \times \frac{time}{clock\ cycle} = \frac{time}{program}$$

最后一个因素是时钟频率的倒数，因此 1 GHz 时钟频率意味着每个时钟周期的时间为 1 ns（$1 / 10^9$）。

即使一个简单的 ISA 可能在每个程序执行的指令数方面多于复杂的 ISA，但它可以通过更快的时钟频率或更低的平均单条指令周期数（CPI）来弥补。

例如，运行 CoreMark 测试程序[Gal-On, Levy 2012]（100000 次迭代）后，得到 ARM-32 Cortex-A9 的性能为：

$$\frac{32.27\ B\ instructions}{program} \times \frac{0.79\ clock\ cycles}{instruction} \times \frac{0.71\ ns}{clock\ cycle} = \frac{18.15\ secs}{program}$$

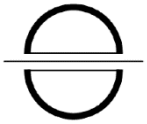
平均时钟周期数可以小于 1，因为 A9 和 BOOM [Celio et al. 2015]是所谓的超标量处理器，每个时钟周期执行多个指令。

对应地，RISC-V 的 BOOM 实现的性能为：

$$\frac{29.51\ B\ instructions}{program} \times \frac{0.72\ clock\ cycles}{instruction} \times \frac{0.67\ ns}{clock\ cycle} = \frac{14.26\ secs}{program}$$

在这个例子中，ARM 处理器执行的指令并不比 RISC-V 处理器少。正如我们将要看到的，简单的指令也是最常用到的指令，因此 ISA 的简洁性是最为重要的指标。对于这个程序，RISC-V 处理器在这三个因素中的每一个都获得了近 10%的优势，它们加起来导致了近 30%的性能优势。如果更简洁的 ISA 也能催生出更小的芯片，那么其性价比将非常出色。

架构和具体实现的分离 *架构和实现*之间最初的分离可以追溯到 20 世纪 60 年代，具体表现为：机器语言程序员了解架构后能写出正确的程序，却不一定能保证性能。对于架构师来说，为了在性能和成本上对某一特定时间的某种实现进行优化，而在 ISA 中包含某些指令，有时候是一件有诱惑性的事情。但这样做会给其他实现或者今后的实现带来负担。



今天的流水线处理器使用硬件预测器预测分支结果，这种方法的准确度可以超过 90%，并且适用于任何大小的流水线。他们只需要一种机制来刷新和重启流水线。

*延迟分支*是 MIPS-32 ISA 的一个令人遗憾的例子。条件分支导致流水线执行出现问题，因为处理器希望下一条要执行的指令总是已经在流水线上，但它不能确定它要的到底是顺序执行的下一条（如果分支未执行），还是分支目标地址的那一条（如果执行了分支）。对于它们的第一个五级流水的微处理器，这种优柔寡断可能导致流水线一个时钟周期的阻塞。MIPS-32 通过把分支操作重新定义在分支指令的下一条指令*执行完之后*发生，因此分支指令的下一条指令*永远*会被执行。程序员或编译器编写者要做的是把一些有用的指令放入*延迟槽*。

唉，这个“解决方案”对接下来有着更多流水级（于是在计算出分支结果之前取了更多的指令）的 MIPS-32 处理器并无益处，反而让 MIPS-32 程序员，编译器编写者，以及处理器设计者（因为增量 ISA 需要向后兼容，见 1.2 节）的生活变得更加艰难。此外，它让 MIPS-

32 的代码变得更加难懂（参见第 29 页图 2.10）。

虽然架构师不该为了有助于某个时间点的某一个特定实现而特意加入某些功能，但他们也不应该放入阻碍某些实现的功能。例如，如上一页所述，ARM-32 和其他一些 ISA 具有 Load Multiple 指令。这些指令可以提高单发射流水线设计的性能，但会降低多发射流水线的效率。原因在于这种直截了当的实现排除了与其他指令并行地调度 Load Multiple 的各个负载的可能，从而降低了这些处理器的指令吞吐量。

提升空间 随着摩尔定律（Moore's law）的终结，对性价比进行重大改进的唯一途径是为特定领域（例如深度学习，增强现实，组合优化，图形等）添加自定义指令。这意味着如今的 ISA 必须保留操作码空间以供未来的提升。

在 20 世纪 70 年代和 80 年代，当摩尔定律如日中天的时候，很少有人考虑为未来的提升节省操作码空间。相反，架构师们重视长地址和立即数字段以减少每个程序执行的指令数（这是前一页上有关性能的方程式中的第一个因素）。

一个能说明缺少操作码空间的弊端的例子是，ARM-32 的架构师后来试图通过向以前统一的 32 位 ISA 中添加 16 位指令来缩减代码长度，但根本就没有空间了。因此，唯一的解决方案是先用 16 位指令来创建一个新的 ISA（Thumb），然后同时用 16 位指令和 32 位指令来组成另外一个 ISA（Thumb-2），并用一个模式位在两种长度的指令间切换。为了切换模式，程序员或编译器会跳转到一个最低有效位为 1 的字节地址。这种方法有效的前提是，在正常的 16 位和 32 位指令中，该位应该是 0。

上面提到的 ARM-32 指令 ldmiaeq 甚至更复杂，因为当它分支时它也可以将 ARM-32 从 Thumb/Thumb-2 两种模式中切换。

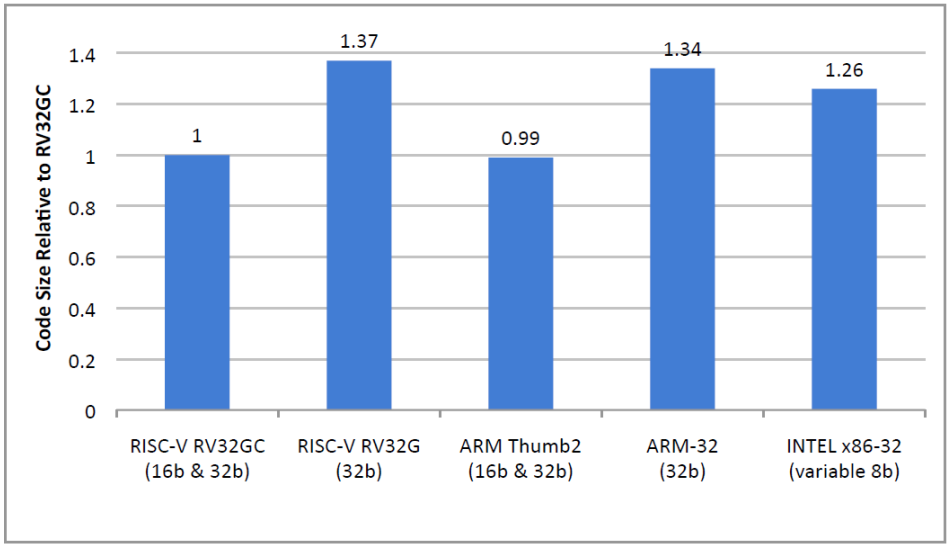


图1.5：RV32G，ARM-32，x86-32，RV32C和Thumb-2程序的相对大小。最后两个ISA是以短代码长度为目标的。 这些程序是使用GCC编译器的SPEC CPU2006基准测试。与RV32C相比，Thumb-2的代码短小的优势是由于在进入程序时Load and Store Multiple的节省。RV32C没有包含它们，以保持与RV32G指令的一对一映射，RV32G省略了Load and Store Multiple以降低高端处理器的实现复杂性（见下文）。第七章介绍了RV32C。RV32G表示RISC-V扩展（RV32M，RV32F，RV32D和RV32A）的流行组合，正确称为RV32IMAFD。[Waterman 2016]

程序大小 程序越小，存储它所需的芯片面积就越小(这对于嵌入式设备来说可能是一个巨大的成本)。实际上，这个问题促使 ARM 架构师在 Thumb 和 Thumb-2 ISA 中追加了一些更短的指令。更小的程序还能减少指令缓存的未命中问题，从而节省了功耗（因为片外 DRAM 访问比片上 SRAM 访问耗能更多），也提高了性能。短的代码长度是 ISA 架构师的目标之一。

x86-32 ISA 的指令可以短至 1 字节，也可以长达 15 字节。你可能会觉得 x86 的这种可变字节长度的指令写成的程序一定会比用一些 ISA（比如 ARM-32，RISC-V）中 32 位定长指令写的要更短。逻辑上，可变字节长度指令的程序也应该小于仅由 16 位和 32 位定长指令组成的 ISA（比如 Thumb-2 和使用 RV32C 扩展的 RISC-V，参见第七章）。图 1.5 显示，当所有指令都是 32 位长时，ARM-32 和 RISC-V 代码比 x86-32 长 6%到 9%，而令人惊讶的是，x86-32 代码比同时提供 16 位和 32 位指令的压缩版本（Thumb-2 和 RV32C）大 26%。

例如一个 15 字节的 x86-32 指令是 lock add dword ptr ds:[esi+ecx*4+0x12345678],0xefcdab89。它汇编成（十六进制）：67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef。最后 8 个字节是 2 个地址，前 7 个字节指定原子的存储器操作，加操作，32 位数据，数据段寄存器，2 个地址寄存器和缩放索引寻址模式。1 字节指令的例子是汇编成 40 的指令 inc eax。

虽然使用新的可变字节长度指令的新 ISA 可能会导致比 RV32C 和 Thumb-2 更短的代码，但 20 世纪 70 年代设计第一个 x86 的架构师并不关心这个问题。此外，考虑到增量 ISA（第 1.2 节）对于向后二进制兼容性的要求，数百条新的 x86-32 指令比预期要长。它们有着一到两个字节长前缀的负担，这迫使它们使用原始 x86 的有限的空余操作码空间。

易于编程/编译/链接 由于寄存器中的数据访问起来要比存储器中的快得多，编译器在寄存器分配方面一定要做得很好。这件事在有许多寄存器的时候变得更加容易。鉴于这一点，ARM-32 有 16 个寄存器，而 x86-32 只有 8 个。大多数现代 ISA（包括 RISC-V）都有 32 个整型寄存器。毫无疑问，有了更多的寄存器，编译器和汇编程序员的工作会更加轻松。

编译器和汇编语言程序员的另一个问题是弄清楚一个代码序列的执行速度。我们可以看到，一般每条 RISC-V 指令最多用一个时钟周期执行（忽略缓存未命中）。但正如我们之前看到的，ARM-32 和 x86-32 都有需要很多个时钟周期执行（即使所有缓存都命中）的指令。此外，与 ARM-32 和 RISC-V 不同，x86-32 的算术指令操作数可以在存储器中，而不必都在寄存器里。复杂的指令和位于存储器中的操作数使得处理器的设计人员难以保证性能的可预测性。

ISA 支持*位置无关代码*（PIC）非常有用，因为这样它就支持动态链接（参见第 3.5 节），原因在于在不同程序中共享库代码可以驻留在不同地址。PC 相关的分支和数据寻址是 PIC 的福音。虽然几乎所有的 ISA 都提供与 PC 相关的分支，但 x86-32 和 MIPS-32 省略了与 PC 相关的数据寻址。

补充说明：ARM-32，MIPS-32 和 x86-32

这是一个可选部分，如果对某个主题感兴趣的话，读者可以深入研究它们，但它们对于理解本书的其余部分并不必要。例如，我们对于 ISA 的称呼不是官方名称。32 位地址 ARM ISA 有许多版本，第一个诞生于 1986 年，最新版本在 2005 年出现，称为 ARMv7。ARM-32 通常是指 ARMv7 ISA。MIPS 也有许多 32 位版本，但我们指的是原版，称为 MIPS I（“MIPS32”是一个更新的，不同于我们称之为 MIPS-32 的 ISA）。英特尔的第一个 16 位地址架构是 1978 年的 8086，其中 80386 ISA 在 1985 年扩展到 32 位地址。我们的 x86-32 表示法通常是指 IA-32，它的 x86 ISA 的 32 位地址版本。鉴于这些 ISA 的数不清的变体的存在，我们发现我们的非标准术语反而最不容易混淆。

1.4 全书的总览

本书假设您在 RISC-V 之前已经了解过其他指令集。如果没有，请查看我们基于 RISC-V 的相关入门架构书[Patterson 和 Hennessy 2017]。

第二章介绍了 RV32I，它是 RISC-V 固定不变的基础整数指令集，是 RISC-V 的核心内容。第三章解释了第二章中没有介绍的其余 RISC-V 汇编语言内容，包括调用约定和一些

用于链接的巧妙技巧。汇编语言包括所有符合规则的 RISC-V 指令和一些 RISC-V 指令集外的有用指令。这些伪指令是实际指令的巧妙变体，它们简化了编写汇编语言程序的过程，同时避免了使 ISA 复杂化。

接下来的三章阐述了 RISC-V 的标准扩展。当它们添加到 RIV32I 中的时候，我们统称 RV32G（G 代表一般）：

- 第四章：乘法和除法（RV32M）
- 第五章：浮点操作（RV32F 和 RV32D）
- 第六章：原子操作（RV32A）

参考卡也被称为绿色卡片，这来源于 20 世纪 60 年代的 ISA 的单页纸板摘要的背景颜色的阴影。为了易读性，我们将背景保持白色，而不是延续历史而使其为绿色。

第 3 页和第 4 页的 RISC-V“参考卡”是本书中所有 RISC-V 指令（RV32G，RV64G 和 RV32 / 64V）的摘要。

第七章介绍了可选的压缩扩展 RV32C，它是 RISC-V 优雅性的一个绝佳例子。通过把 16 位指令限制为现有 32 位 RV32G 指令的短版本，它们几乎是没有代价的。汇编程序可以选择指令大小，这使得汇编语言程序员和编译器忘记 RV32C。将 16 位 RV32C 指令转换成 32 位 RV32G 指令的硬件解码器只需要 400 个门，这即使在最简单的 RISC-V 实现中也只占百分之几。

第八章介绍了向量扩展 RV32V。当与众多强大的单指令多数据（*SIMD*）指令（ARM-32，MIPS-32，x86-32）相比时，向量指令成为了 ISA 优雅性的另一个例证。实际上，图 1.2 中添加到 x86-32 的数百条指令都是 SIMD，还有数百条指令即将问世。RV32V 甚至比大多数向量 ISA 更简单，因为它通过向量寄存器指定数据类型和长度，而不是将这两者嵌入到操作码中。RV32V 也许是大家从传统的基于 SIMD 的 ISA 转到 RISC-V 的最为可能的原因。

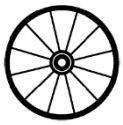
第九章展示了 RV64G，它是 RISC-V 的 64 位地址版本。正如该章节所说的那样，RISC-V 的架构师只需要拓宽寄存器，并加入一些字、双字或长版的 RV32G 指令，就可以把地址从 32 位扩展为 64 位。

第十章介绍了系统指令，说明了 RISC-V 如何处理分页以及机器、用户和监管者权限模式。

最后一章简要介绍了 RISC-V 基金会目前正在考虑增加的其它扩展。

接下来是本书最大的一个部分，附录 A。它是按字母表顺序排列的指令集摘要。它定义了完整的 RISC-V ISA 以及上面提到的所有扩展，还有大概 50 页的全部伪指令。这是 RISC-V 简洁性的证明。

这本书的最后一部分是索引。



1.5 结束语

冯·诺伊曼先前版本的精心编写的报告非常有影响力，以至于这种计算机通常被称为冯·诺伊曼架构，尽管这份报告是基于其他人的工作。它是在第一台存储程序计算机开始运行的三年前编写的！

> 用形式逻辑的方法可以很容易看出，存在某种[指令集]在理论上足以控制和执行任意顺序的操作……从当前的观点出发，选择一个[指令集]时考虑得更多更实际的问题是：[指令集]要求的设备简单性，在实际重要的问题中有明确应用和解决该类问题的速度。
>
> ——冯·诺伊曼（von Neumann）等，1947

RISC-V 是一个最新的，清晰的，简约的，开源的 ISA，它以过去 ISA 所犯过的错误为鉴。RISC-V 架构师的目标是让它在从最小的到最快的所有计算设备上都能有效工作。遵循冯诺依曼 70 年前的建议，这个 ISA 强调简洁性来保证它的低成本，同时有着大量的寄存器和透明的指令执行速度，从而帮助编译器和汇编语言程序员将实际的重要问题转换为适

当的高效代码。

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

图1.6：ISA手册的页数和字数来自[Waterman and Asanovi'c 2017a]，[Waterman and Asanovi'c 2017b]，[Intel Corporation 2016]，[ARM Ltd. 2014]。读完需要的时间按每分钟读200个单词，每周读40小时计算。 基于[Baumann 2017]的图1的一部分。

复杂度的一个标准是文档的大小。图 1.6 显示了以页数和单词数衡量的 RISC-V、ARM-32 和 x86-32 指令集手册的大小。如果你把读手册作为全职工作，每天 8 小时，每周 5 天，那么需要半个月读完 ARM-32 手册，需要整整一个月读完 x86-32 手册。有这样的复杂程度，大概没有一个人能完全理解 ARM-32 或 x86-32。用这种常识来度量，RISC-V 的复杂度只有 ARM-32 的$\frac{1}{12}$,x86-32 的$\frac{1}{10}$到$\frac{1}{30}$。实际上，包含所有扩展的 RISC-V ISA 摘要只有两页（参见参考卡）。

这个袖珍的，开源的 ISA 于 2011 年推出，现在由一个基金会提供支持。该基金会通过长期讨论后严格依据技术理由添加可选扩展的方式来改进它。开源性让 RISC-V 的免费的、共享的实现成为可能，从而降低了成本，也减少了将不为人知的邪恶秘密隐藏在处理器之中的可能性。

然而，只有硬件不能组成一个系统。软件开发成本可能使硬件开发成本相形见绌。因此虽然稳定的硬件很重要，但稳定的软件更甚于此。这些软件需要包括操作系统，引导加载程序，参考软件和大众化的软件工具。基金会保证整个 ISA 的稳定性，而固定不变的基础指令集意味着核心的 RV32I 作为软件栈的目标永远不会改变。通过它的普适性和开源性，RISC-V 可以挑战主流专有 ISA 的主导地位。

优雅是一个很少应用于 ISA 的词，但在阅读本书后，你可能会同意我们把它用于 RISC-V。我们将用页边的蒙娜丽莎图标来凸显我们认为体现出优雅性的特征。

1.6 扩展阅读

ARM Ltd. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition, 2014. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/.

A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

C. Celio, D. Patterson, and K. Asanovic. The Berkeley Out-of-Order Machine (BOOM): an industry-competitive, synthesizable, parameterized RISC-V processor. *Tech. Rep. UCB/EECS-2015–167, EECS Department, University of California, Berkeley*, 2015.

S. Gal-On and M. Levy. Exploring CoreMark - a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. September 2016.

S. P. Morse. The Intel 8086 chip and the future of microprocessor design. *Computer*, 50(4): 8–9, 2017.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

S. Rodgers and R. Uhlig. X86: Approaching 40 and still going strong, 2017.

J. L. von Neumann, A.W. Burks, and H. H. Goldstine. Preliminary discussion of the logical design of an electronic computing instrument. *Report to the U.S. Army Ordnance Department*, 1947.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017a. URL https://riscv.org/specifications/privileged-isa/.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017b. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

第二章 RV32I：RISC-V 基础整数指令集

…提升计算性能并且让用户能切实享受到性能提升的唯一方法是同时设计编译器和计算机。这样软件用不到的特性将不会被实现在硬件上…

——Frances Elizabeth “Fran” Allen, 1981

Frances Elizabeth “Fran” Allen
(1932-)被授予图灵奖主要是因为她在优化编译器方面的工作。图灵奖是计算机科学的最高奖项。

2.1 导言

图 2.1 是 RV32I 基础指令集的一页图形表示。对于每幅图，将有下划线的字母从左到右连接起来，即可组成完整的 RV32I 指令集。对于每一个图，集合标志{}内列举了指令的所有变体，变体用加下划线的字母或下划线字符_表示。特别的，下划线字符_表示对于此指令变体不需用字符表示。例如，下图表示了这四个 RV32I 指令：slt，slti，sltu，sltiu:

$$\underline{\text{s}}\text{et }\underline{\text{l}}\text{ess }\underline{\text{t}}\text{han}\left\{\begin{matrix}\underline{\ }\\ \underline{\text{i}}\text{mmediate}\end{matrix}\right\}\left\{\begin{matrix}\underline{\ }\\ \underline{\text{u}}\text{nsigned}\end{matrix}\right\}$$

我们使用这些图（下面几章的第一个图），旨在对本章的指令给出一个进行快速、深入的概述。

2.2 RV32I 指令格式

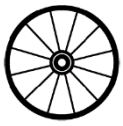


图 2.2 显示了六种基本指令格式，分别是：用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。图 2.3 使用图 2.2 的指令格式列出了图 2.1 中出现的所有 RV32I 指令的操作码。

即使是指令格式也能从一些方面说明 RISC-V 更简洁的 ISA 设计能提高提高性能功耗比。首先，指令只有六种格式，并且所有的指令都是 32 位长，这简化了指令解码。ARM-32，还有更典型的 x86-32 都有许多不同的指令格式，使得解码部件在低端实现中偏昂贵，在中高端处理器设计中容易带来性能挑战。第二，RISC-V 指令提供三个寄存器操作数，而不是像 x86-32 一样，让源操作数和目的操作数共享一个字段。当一个操作天然就需要有三个不同的操作数，但是 ISA 只提供了两个操作数时，编译器或者汇编程序程序员就需要多使用一条 move（搬运）指令，来保存目的寄存器的值。第三，在 RISC-V 中对于所有指令，要读写的寄存器的标识符总是在同一位置，意味着在解码指令之前，就可以先开始访问寄存器。在许多其他的 ISA 中，某些指令字段在部分指令中被重用作为源目的地，在其他指令中又被作为目的操作数（例如，ARM-32 和 MIPS-32）。因此，为了取出正确的指令字段，我们需要时序本就可能紧张的解码路径上添加额外的解码逻辑，使得解码路径的时序更为紧张。第四，这些格式的立即数字段总是符号扩展，符号位总是在指令中最高位。这意味着可能成为关键路径的立即数符号扩展，可以在指令解码之前进行。

立即数的符号扩展甚至有助于逻辑指令。例如，x＆0xfffffff0 在 RISC-V 中只需要一条 andi 指令，但在 MIPS-32 中需要两条指令（addiu 用于加载常量，然后是 and），因为 MIPS 零扩展逻辑立即数。ARM-32 需要添加一个额外的指令 bic，它执行 rx＆immediate 以补偿零扩展的立即数.

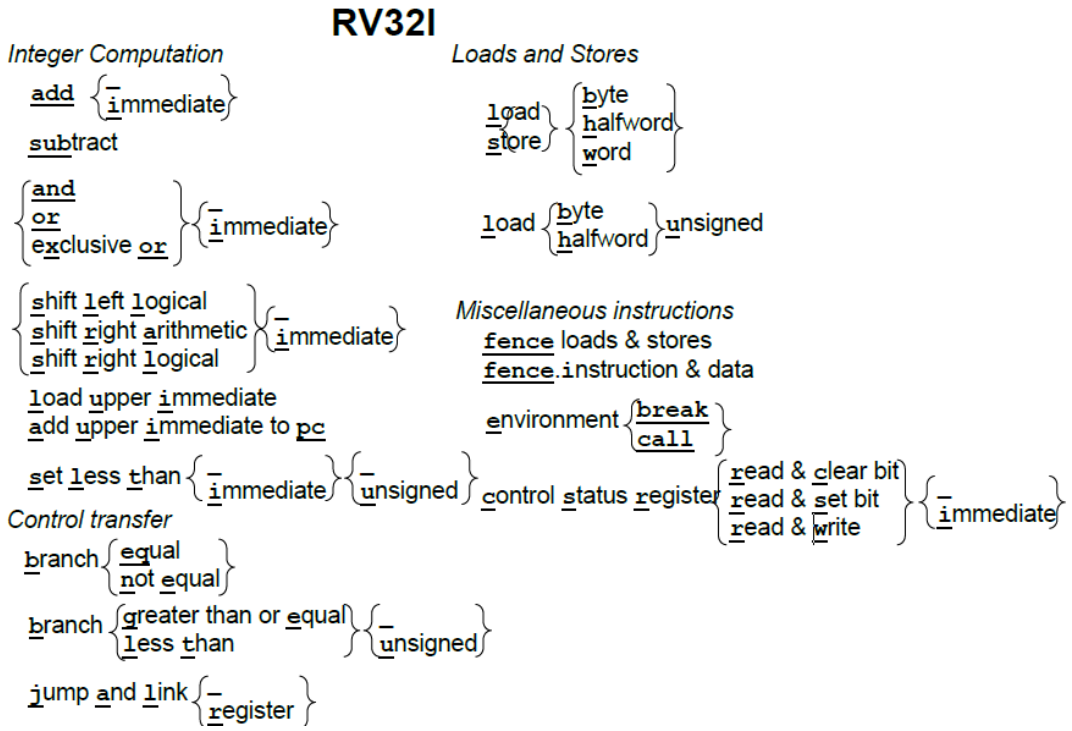


图 2.1：RV32I 指令图示。把带下划线的字母从左到右连接就组成了 RV32I 指令。花括号{}表示集合中垂直方向上的每个项目都是指令的不同变体。集合中的下划线_意味着不包含这个字母的也是一个指令名称。例如，左上角附近的符号表示以下六个指令：and，or，xor，andi，ori，xori。

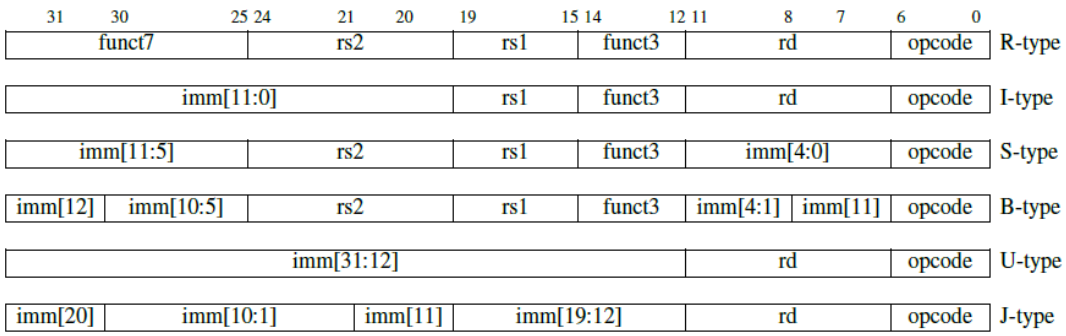


图 2.2：RISC-V 指令格式。我们用生成的立即数值中的位置（而不是通常的指令立即数域中的位置）(imm[*x*])标记每个立即数子域。第十章解释了控制状态寄存器指令使用 I 型格式的稍微不同的做法。（本图基于 Waterman 和 Asanovi'c 2017 的图 2.2）。

31 25 24 20 19 15 14 12 11 7 6 0						
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

图 2.3：RV32I 带有指令布局，操作码，格式类型和名称的操作码映射。（此图基于[Waterman and Asanovi'c 2017]的表 19.2。）

补充说明：B 类型和 J 类型指令

如下所述，分支指令（B 类型）的立即数字段在 S 类型的基础上旋转了 1 位。跳转指令（J 类型）的直接字段在 U 类型的基础上旋转了 12 位。因此,RISC-V 实际上只有四种基本格式，但我们可以保守地认为它有六种格式。

为了帮助程序员，所有位全部是 0 是非法的 RV32I 指令。因此，试图跳转到被清零的内存区域的错误跳转将会立即触发异常，这可以帮助调试。类似地，所有位全部是 1 的指令也是非法指令，它将捕获其他常见的错误，诸如未编程的非易失性内存设备、断开连接的内存总线或者坏掉的内存芯片。

为了给 ISA 扩展留出足够的空间，最基础的 RV32I 指令集只使用了 32 位指令字中的编码空间的不到八分之一。架构师们也仔细挑选了 RV32I 操作码，使拥有共同数据通路的指令的操作码位有尽可能多的位的值是一样的，这简化了控制逻辑。最后,当我们看到，B 和 J 格式的分支和跳转地址必须向左移动 1 位以将地址乘以 2，从而给予分支和跳转指令更大的跳转范围。RISC-V 将立即数中的位从自然排布进行了一些移位轮换，将指令信号的扇出和立即数多路复用的成本降低了近两倍，这也简化了低端实现中的数据通路逻辑。

RISC-V 实现对可选扩展使用相同的操作码，例如 RV32M，RV32F 等。针对特定处理器的非标准扩展只能使用 RISC-V 中的保留操作码空间。

有什么不同之处？在这一章和后面的章节的结束部分，我们将描述 RISC-V 与其他指令集的不同之处。这种对比通常是描述相比于其他指令集，RISC-V 少了什么。省略什么特性和包括什么特性一样，都能体现架构师的精心设计。

ARM-32 指令集 12 位的立即字段不仅仅是一个常量，而是一个函数的输入，此函数根据 12 位立即数的输入来产生一个常量：8 位被零扩展到全宽度，然后被循环右移。右移的位数是 12 位立即数中剩余 4 位的值乘 2。设计者希望在 12 位中编码更多有用的常数来减少执行指令的数量。在大多数指令格式中，ARM-32 也将十分宝贵的四位编码空间拿出来专门用于条件执行。这些条件执行指令不仅使用频率低而且增加了乱序处理器的复杂性。

补充说明：乱序执行处理器

这是一种高速的、流水化的处理器。它们一有机会就执行指令，而不是在按照程序顺序。这种处理器的一个关键特性是寄存器重命名，把程序中的寄存器名称映射到大量的内部物理寄存器。条件执行的问题是不管条件是否成立，都必须给这些指令中的寄存器分配相应的物理寄存器。但内部物理寄存器的可用性是影响乱序处理器的关键性能资源。

2.3 RV32I 寄存器

图 4 列出了 RV32I 寄存器以及由 RISC-V 应用程序二进制接口（ABI）所定义的寄存器名称。在我们的示例代码中，我们将使用 ABI 名称，使它们更容易阅读。为了满足汇编语言程序员和编译器编写者，RV32I 有 31 寄存器加上一个值恒为 0 的 x0 寄存器。与之相比，ARM-32 只有 16 个寄存器，x86-32 甚至只有 8 个寄存器。

有什么不同之处？为常量 0 单独分配一个寄存器是 RISC-V ISA 能如此简单的一个很大的因素。第 3 章的第 36 页的图 3 给出了许多 ARM-32 和 x86-32 的原生指令操作，这两个指令集中没有零寄存器。我们可以用 RV32I 指令完成功能相同的操作，只需使用零寄存器作为操作数。

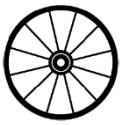
程序计数器（PC）是 ARM-32 的 16 个寄存器之一，这意味着任何改变寄存器的指令都有可能导致分支跳转。PC 作为一个寄存器使硬件分支预测变得复杂，因为在典型的 ISA 中，仅 10%-20%的指令为分支指令，而在 ARM-32 中，任何指令都有可能是分支指令。而分支预测的准确性对于良好的流水线性能至关重要。另外将 PC 作为一个寄存器也意味着可用的通用寄存器少了一个。

目前除最便宜的处理器以外的所有处理器都使用流水线来获得良好的性能。与工业装配线一样，它们通过一次执行多条指令来获得更高的吞吐量。为了实现这一目标，处理器可以预测分支结果，这个操作的准确度可以超过 90%。若进行了错误预测，指令会重新执行。早期的微处理器有一个 5 级流水线，这意味着 5 条指令并行执行。最近的处理器有 10 多个流水级。

2.4 RV32I 整数计算

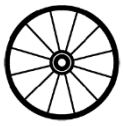
附录 A 给出了所有 RISC-V 指令的细节信息，包括格式和操作码。在本节以及接下来的章节的类似小节中，我们将给出 ISA 的一些概述。这能够让有基础的汇编语言程序员了解 RISC-V，同时也顺便说明 RISC-V 的特性如何满足第一章中阐述的七个 ISA 指标。

简单的算术指令（add, sub）、逻辑指令（and, or, xor），以及图 2.1 中的移位指令（sll, srl, sra）和其他 ISA 差不多。他们从寄存器读取两个 32 位的值，并将 32 位结果写入目标寄存器。RV32I 还提供了这些指令的立即数版本。和 ARM-32 不同，立即数总是进行符号扩展，这样子如果需要，我们可以用立即数表示负数，正因为如此，我们并不需要一个立即数版本的 sub。



程序可以根据比较结果生成布尔值。为应对这种使用场景下，RV32I 提供一个*当小于时置位*的指令。如果第一个操作数小于第二个操作数，它将目标寄存器设置为 1，否则为 0。不出所料，对这个指令，有一个有符号版本（slt）和无符号版本（sltu），分别用于处理有符号和无符号整数比较。相应的，上述两条指令也有立即数版本的（slti，sltiu）。正如我们将要看到的，虽然 RV32I 分支指令可以检查两个寄存器之间的所有关系，但一些条件表达式涉及多对寄存器之间的关系。对于这些表达式，编译器或汇编语言程序员可以将 slt 以及与或异或等逻辑指令组合使用来解决更复杂的条件表达式。

图 2.1 剩下的两条整数计算指令主要用于构造大的常量数值和链接。加载立即数到高位（lui）将 20 位常量加载到寄存器的高 20 位。接着便可以使用标准的立即指令来创建 32 位常量。这样子，仅使用 2 条 32 位 RV32I 指令，便可构造一个 32 位常量。向 PC 高位加上立即数（auipc）让我们仅用两条指令，便可以基于当前 PC 以任意偏移量转移控制流或者访问数据。将 auipc 中的 20 位立即数与 jalr（参见下面）中 12 位立即数的组合，我们可以将执行流转移到任何 32 位 PC 相对地址。而 auipc 加上普通加载或存储指令中的 12 位立即数偏移量，使我们可以访问任何 32 位 PC 相对地址的数据。



有什么不同之处？首先，RISC-V 中没有字节或半字宽度的整数计算操作。操作始终是以完整的寄存器宽度。内存访问需要的能量比算术运算高几个数量级。因此低宽度的数据访问可以节省大量的能量，但低宽度的运算不会。ARM-32 具有一个不寻常的功能，对于大多数算术逻辑运算中的一个操作数，你可以选择对它进行移位。尽管这些指令的使用频率很低，但它使数据路径和数据通路更加复杂。与此相对的是，RV32I 提供了单独的移位指令。

RV32I 也不包含乘法和除法，它们包含在可选的 RV32M 扩展中（参见第 4 章）。与 ARM-32 和 x86-32 不同，即使处理器没有添加乘除法扩展，完整的 RISC-V 软件栈也可以运行，这可以缩小嵌入式芯片的面积。MIPS-32 汇编程序可能用一系列移位以及加法指令来替换乘法，以提高性能，这可能会使程序员看到处理器执行了汇编程序中没有的指令，进而造成混淆。RV32I 可以忽略了这些特性：循环移位指令和整数算术溢出检测，这两个特性都可以用若干条 RV32I 指令来实现（参见第 2.6 节）。

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

31 … 0
pc

32

图 2.4：RV32I 的寄存器。第 3 章解释了 RISC-V 调用约定，各种指针（sp，gp，tp，fp），保存寄存器（s0-s11）和临时寄存器（t0-t6）背后的基本原理（基于[Waterman and Asanovi'c 2017]的图 2.1 和表 20.1）。

补充说明：“位操作”指令

RISC-V 基金会正在考虑把 rotate 之类的位操作指令作为可选指令扩展 RV32B 的一部分（见第 11 章）。

补充说明：利用 *xor* 指令进行的花式操作

您可以在不使用中间寄存器的情况下交换两个值！此代码交换 x1 和 x2 的值。我们将证明留给读者。提示：异或操作是交换的 $(a \oplus b = b \oplus a)$，结合的 $((a \oplus b) \oplus c = a \oplus (b \oplus c))$，是它自己的逆操作 $(a \oplus a = 0)$，并且有一个单位元$(a \oplus 0 = a)$。

```
xor x1,x1,x2 # x1' == x1^x2, x2' == x2
xor x2,x1,x2 # x1' == x1^x2, x2' == x1'^x2 == x1^x2^x2 == x1
xor x1,x1,x2 # x1" == x1'^x2' == x1^x2^x1 == x1^x1^x2 == x2, x2' == x1
```

不论这个操作多么奇妙，RISC-V 充足的寄存器使得编译器通常可以找到一个临时寄存器，因此很少使用这个异或的交换操作。

2.5 RV32I 的 Load 和 Store

除了提供 32 位字（lw，sw）的加载和存储外，图 2.1 中说明，RV32I 支持加载有符号和无符号字节和半字（lb，lbu，lh，lhu）和存储字节和半字（sb，sh）。有符号字节和半字符号扩展为 32 位再写入目的寄存器。即使是自然数据类型更窄，低位宽数据也是被扩展后再处理，这使得后续的整数计算指令能正确处理所有的 32 位。在文本和无符号整数中常用的无符号字节和半字，在写入目标寄存器之前都被无符号扩展到 32 位。

加载和存储的支持的唯一寻址模式是符号扩展 12 位立即数到基地址寄存器，这在 x86-32 中被称为位偏移寻址模式[Irvine 2014]。

有什么不同之处？RV32I 省略了 ARM-32 和 x86-32 的复杂寻址模式。另外，ARM-32 提供的寻址模式并非适用于所有数据类型，但 RV32I 寻址不会歧视任何数据类型。RISC-V 可以模仿某些 x86 寻址模式。例如，将立即数字段设置为 0 即与 x86 中的寄存器间接寻址效果相同。与 x86-32 不同，RISC-V 没有特殊的堆栈指令。将 31 个寄存器中的某一个作为堆栈指针（见图 2.4），标准寻址模式使用起来和压栈（push）和出栈（pop）类似，并且不增加 ISA 的复杂性。与 MIPS-32 不同，RISC-V 不支持延迟加载（*delayed load*）。与延迟分支的设计相似，为了更好的适应五级流水线，MIPS-32 重新定义了 load 指令的语义，load 上来的数据在 load 指令两个指令后才可用。但是对于后来出现的更长的流水线，延迟加载带来的收益逐渐消失，因此 RISC-V 不支持延迟加载。

虽然 ARM-32 和 MIPS-32 要求存储在内存中的数据，要按照数据的自然大小进行边界对齐，但是 RISC-V 没有这个要求。移植旧的代码有时需要未对齐的访问。对于不对齐访问，一种选择是在基础 ISA 中禁止不对齐访问，然后提供一些单独的指令用于不对齐访问，例如 MIPS-32 中的 Load Word Left 和 Load Word Right。然而，这会使寄存器访问变得复杂，因为 lwl 并且 lwr 需要对寄存器进行部分写，而不是简单地对寄存器进行完整的

写。支持不对齐访问的，另一种方法就是让普通的加载和存储指令支持不对齐访问，这简化了整体设计。

补充说明：字节序问题
RISC-V 选择了小尾端字节序，因为它在商业上占主导地位：所有 x86-32 系统，Apple iOS，谷歌 Android 操作系统和微软 Windows for ARM 都是低字节优先序。由于字节顺序仅在同时以按字访问和按字节访问同一份数据时才会有影响，字节序只会影响很少一部分的程序员。

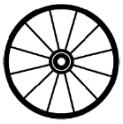
2.6 RV32I 条件分支

RV32I 可以比较两个寄存器并根据比较结果上进行分支跳转。比较可以是：相等（beq），不相等 （bne），大于等于（bge），或小于（blt）。最后两种比较有符号比较，RV32I 也提供相应的无符号版本比较的：bgeu 和 bltu。剩下的两个比较关系（大于和小于等于）可以通过简单地交换两个操作数，即可完成比较。因为 $x < y$ 表示 $y > x$ 且 $x \geq y$ 表示 $y \leq x$。

bltu 允许使用单个指令检查有符号数组的边界，因为任何负索引都将比任何非负边界更大！

由于 RISC-V 指令长度必须是两个字节的倍数——关于可选的双字节指令 ，请参考第七章——分支指令的寻址方式是 12 位的立即数乘以 2，符号扩展它，然后将得到值加到 PC 上作为分支的跳转地址。PC 相对寻址可用于位置无关的代码，简化了链接器和加载器的工作（第 3 章）。

有什么不同之处？如上所述，RISC-V 去掉了 MIPS-32，Oracle SPARC 等指令集中被广为诟病的延迟分支特性等。对于条件分支，它还没有像 ARM-32 和 x86-32 那样使用条件码。条件码的存在使得大多数指令都需要隐式设置一些额外状态，这使乱序执行的依赖计算复杂化。最后，它省略了 x86-32 中的循环指令：loop，loope，loopz，loopne，loopnz。



补充说明：不使用条件码实现大位宽数据的加法
在 RV32I 中是通过 sltu 计算进位来实现的：

```
add a0,a2,a4 #  加低 32  位: a0 = a2 + a4
sltu a2,a0,a2 #  若  (a2+a4) < a2 那么 a2' = 1,  否则 a2' = 0
add a5,a3,a5 #  加高 32 位: a5 = a3 + a5
add a1,a2,a5 #  加上低 32 位的进位
```

补充说明：获取 PC
当前的 PC 可以通过将 auipc 的 U 立即数字段设置为 0 来获得。对于 x86-32，要想读取 PC，你需要先进行函数调用，（这样子可以将 PC 推入堆栈）;然后被调用的函数可以从堆栈中读取刚被压栈的 PC，最后将 PC 值返回给调用者（需要再弹出堆栈）。因此，或许当前的 PC 至少需要 1 个 store，2 个 load 和 2 个跳转！

补充说明：软件检查溢出

大部分（但不是所有）程序都忽略整数算术溢出，因此 RISC-V 依赖于软件溢出检查。检查无符号加法的溢出只需要在指令后添加一个额外的分支指令：addu t0，t1，t2; bltu t0，t1，overflow。

对于带符号的加法，如果已知一个操作数的符号，则溢出检查只需要在加法后添加一条分支指令：addi t0，t1，+ imm; blt t0，t1，overflow。

这覆盖了常见的加立即数的情况。对于一般的带符号加法，我们需要在加法指令后添加三个附加指令，当且仅当一个操作数为负数时，结果才能小于另一个操作数，否则就是溢出。

```
add t0, t1, t2
slti t3, t2, 0          # t3 = (t2<0)
slt t4, t0, t1          # t4 = (t1+t2<t1)
bne t3, t4, overflow # 若 (t2<0) && (t1+t2>=t1)
                      # || (t2>=0) && (t1+t2<t1)则为溢出
```

2.7 RV32I 无条件跳转

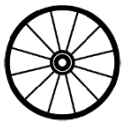


图 2.1 中的跳转并链接指令（jal）具有双重功能。若将下一条指令 PC + 4 的地址保存到目标寄存器中，通常是返回地址寄存器 ra（见图 2.4），便可以用它来实现过程调用。如果使用零寄存器（x0）替换 ra 作为目标寄存器，则可以实现无条件跳转，因为 x0 不能更改。像分支一样，jal 将其 20 位分支地址乘以 2，进行符号扩展后再添加到 PC 上，便得到了跳转地址。

跳转和链接指令的寄存器版本（jalr）同样是多用途的。它可以调用地址是动态计算出来的函数，或者也可以实现调用返回（只需 ra 作为源寄存器，零寄存器（x0）作为目的寄存器）。Switch 和 case 语句的地址跳转，也可以使用 jalr 指令，目的寄存器设为 x0。

寄存器窗口通过使用多于 32 个寄存器来加速函数调用。新函数将在调用时获得 32 个寄存器的新集合或窗口。为了传递参数，窗口重叠，这意味着一些寄存器位于两个相邻的窗口中。

有什么不同之处？RV32I 避开了错综复杂的程序调用指令，例如 x86-32 的进入和离开指令，或 Intel Itanium，Oracle SPARC 和 Cadence Tensilica 中的寄存器窗口。

2.8 RV32I 杂项

图 2.1 中的控制状态寄存器指令 (csrrc、csrrs、csrrw、csrrci、csrrsi、csrrwi)，使我们可以轻松地访问一些程序性能计数器。对于这些 64 位计数器，我们一次可以读取 32 位。这些计数器包括了系统时间，时钟周期以及执行的指令数目。

在 RISC-V 指令集中，ecall 指令用于向运行时环境发出请求，例如系统调用。调试器使用 ebreak 指令将控制转移到调试环境。

fence 指令对外部可见的访存请求，如设备 I / O 和内存访问等进行串行化。外部可见指对处理器的其他核心、线程，外部设备或协处理器可见。fence.i 指令同步指令和数据流。在执行 fence.i 指令之前，对于同一个硬件线程，RISC-V 不保证用存储指令写到内存指令区的数据可以被取指令取到。

第 10 章介绍 RISC-V 系统指令。

```
void insertion_sort(long a[], size_t n)
{
  for (size_t i = 1, j; i < n; i++) {
    long x = a[i];
    for (j = i; j > 0 && a[j-1] > x; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

图 2.5: C 语言版的插入排序。虽然看起来简单，插入排序比复杂的排序算法有许多优势：对于小数据集来说，它是内存使用效率高、速度快，同时还有适应性强、稳定、能在线处理的特点。gcc 编译器生成了以下四个数字的代码。我们设置优化标志以减少代码大小，因为这产生了最容易理解的代码。

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC
Instructions	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

图 2.6: 插入排序在不同指令集下生成的指令数目以及指令大小。第 7 章会介绍 ARM Thumb-2, microMIPS 以及 RV32C 指令集。

有什么不同之处？RISC-V 使用内存映射 I / O 而不是像 x86-32 一样，使用 in，ins，insb，insw 和 out，out，outsb 等指令来进行 I/O。为支持字符串处理，RISC-V 实现了字节存取，而不是像 x86-32 那样实现了 rep，movs，coms，scas，lods 等 16 条特殊的字符串处理指令。

2.9 使用插入排序比较 RV32I，ARM-32，MIPS-32 和 x86-32 指令集

我们已经介绍了 RISC-V 基本指令集，并说明了与 ARM-32，MIPS-32 和 x86-32 相比，它做了哪些取舍。我们现在通过真实程序来进行一场直接的较量。图 2.5 显示了我们的基准测试——用 C 实现的插入排序。图 2.6 是一个表，它总结了在编译到不同 ISA 后，插入排序的指令数和字节数。

图 2.8 至 2.11 显示了插入排序编译生成的 RV32I，ARM-32，MIPS-32 和 x86-32 的汇编代码。尽管强调简单性，RISC-V 版本使用相同数目或更少的指令， 并且不同架构的代码大小非常接近。在此示例中，RISC-V 的比较、执行分支指令和 ARM-32 和 x86-32 中花式繁多的寻址模式以及入栈出栈指令一样，能够节省大量指令。

在本章和后续章节中，我们将代码示例移到章节文本结尾之后，以便保持内容的流畅。

2.10 结束语

那些不记得过去的人，注定要重复过去。——George Santayana, 1905

图 2.7 使用第 1 章中的七个 ISA 设计指标来组织前面提到的一些过去的指令集中学习到的经验教训，并说明了这些经验教训对 RV32I 设计的积极影响。我们并不是说 RISC-V 是第一个拥有这些积极结果的 ISA。事实上，RV32I 从 RISC-I，它的曾祖父母[Patterson 2017]那里，继承了如下这些特性：

所有 RISC-V 指令的谱系记录在[Chen & Patterson 2016]中。

- 32 位字节可寻址的地址空间
- 所有指令均为 32 位长
- 31 个寄存器，全部 32 位宽，寄存器 0 硬连线为零
- 所有操作都在寄存器之间（没有寄存器到内存的操作）
- 加载/存储字加上有符号和无符号加载/存储字节和半字
- 所有算术，逻辑和移位指令都有立即数版本的指令
- 立即数总是符号扩展
- 仅提供一种数据寻址模式（寄存器+立即数）和 PC 相对分支
- 无乘法或除法指令
- 一个指令，用于将大立即数加载到寄存器的高位，这样加载 32 位常量到寄存器只需要两条指令

RISC-V 的出现比过去的 ISA 晚了四分之一到三分之一个世纪后开始，这使它的设计者得以实践 Santayana 的建议，即借用之前指令集中好的设计，但不重复它们不好的瑕疵 - 包括 RISC-I 指令集中的瑕疵。另外 RISC-V 基金会将通过可选的指令集扩展的方式缓慢扩展着指令集，以避免出现困扰过去的成功指令集的疯狂的增量发展。

Lindy 效应 [Lin 2017]观察到技术或想法的未来预期寿命与其年龄成正比。它经受住了时间的考验，所以它过去存活的时间越长，它在未来的生存时间就越长。如果这个假设成立，RISC 架构可能在很长一段时间都会是一个好的设计。

补充说明：RV32I 是否与众不同？

早期的微处理器有单独的浮点运算芯片，所以那些浮点运算指令是可选的。摩尔定律使得我们很快就将所有功能（包括浮点运算）都实现了在同一块芯片上，而且模块化在指令集中逐渐消失。在更简单的处理器中只实现完整的指令集的子集，并利用软件异常来模拟未实现的指令，如同数十年前的在 IBM 360 的 44 型号和 Digital Equipment microVAX。RV32I 的不同之处在于完整的软件堆栈只需要 RV32I 中的基本指令，因此，对于 RV32G 中未实现的指令，RV32I 处理器无需通过软件异常来进行模拟。在这方面，最接近 RISC-V 的 ISA 可能是 Tensilica Xtensa，它是专为嵌入式应用设计的。它的指令集包含有 80 条基础指令。并且它的指令集旨在被用户根据自己的需求扩展一些加速指令，以加速其应用程序。与 Tensilica Xtensa 相比，RV32I 具有更简单的基础 ISA，具有 64 位地址版本，并且对超级计算机和微控制器都提供了针对性的指令集扩展。

2.11 扩展阅读

Lindy effect, 2017. URL https://en.wikipedia.org/wiki/Lindy_effect.

T. Chen and D. A. Patterson. RISC-V genealogy. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html.

W. Hohl and C. Hinds. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, 2016.

K. R. Irvine. *Assembly language for x86 processors*. Prentice Hall, 2014.

D. Patterson. How close is RISC-V to RISC-I?, 2017.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

	过去的错误			RV32I (2011)吸取的经验教训
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	
成本	必须支持整数乘除法	必须支持整数乘除法	8 位以及 16 位操作、必须支持整数乘除法	无 8 位、16 位操作、可选的整数乘除法支持(RV32M)
简洁性	无零寄存器、条件指令执行、复杂的寻址模式、栈操作指令（push/pop）、算术/逻辑指令中存在移位	立即数支持零扩展及符号扩展、一些算术指令会造成溢出异常	无零寄存器、复杂的过程调用指令(enter/leave)、栈指令(push/pop)、复杂寻址模式、循环指令	寄存器 x0 专门用于存放常数 0、立即数只进行符号扩展 、一种数据寻址模式、没有条件执行、没有复杂的函数调用指令以及栈指令、算术指令不抛异常、使用单独的移位指令来处理移位操作
性能	分支指令使用条件码、在不同格式的指令中，源和目的寄存器的位置不同、加载多个计算得到的立即数、PC 是一个通用寄存器	在不同格式的指令中，源和目的寄存器的位置不同	分支指令使用条件码、每个指令中最多只能使用两个寄存器	使用同一条指令实现比较及跳转（不使用条件码）、每条指令三个寄存器 、不能一次 load 多个数据、不同指令格式中，源及目的寄存器字段位置固定、立即数是常数（不是由计算得出的）、PC 不是通用寄存器
架构和具体实现的分离	将 PC 像普通寄存器一样读写，这样暴露了流水线长度	分支指令延迟槽、Load 指令延迟槽、乘除法使用单独的 HI，LO 寄存器	寄存器不是通用的(AX,CX,DX, DI,SI 有特殊用途)	分支指令没有延迟槽、Load 指令无延迟槽、通用寄存器
增长空间	有限的指令码空间	有限的指令码空间		大量可用的指令码空间
程序大小	仅有 32bit 指令(Thumb-2 是作为一个独立的 ISA)	仅 32bit 指令(microMIPS 是作为一个独立的ISA)	指令长度可用是不同字节，但这是一个很不好的选择	32 位指令+16 位 RV32C 扩展
易于编程/编译/链接	仅 15 个寄存器内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	仅 15 个寄存器内存数据必须对齐、不规则的数据寻址模式、不一致的性能计数器	31 个寄存器、数据可用不对齐、PC 相对的数据寻址模式 、对称的数据寻址模式、定义在架构中的性能计数器

图2.7：RISC-V架构师从过去指令集设计的错误中吸取的教训。通常的教训是避免过去的ISA“优化”。经验和教训按照第一章中提出的七个ISA指标进行分类。在成本，简单性和性能下列出的许多指令集特性可以互换，因为这只是设计的偏好问题，但不管它们出现在哪里，它们都很重要。

```
# RV32I (19 instructions, 76 bytes, or 52 bytes with RVC)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00450693  addi  a3,a0,4    # a3 is pointer to a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  jalr  x0,x1,0    # return from function
Continue Outer Loop:
  10: 0006a803  lw    a6,0(a3)   # x = a[i]
  14: 00068613  addi  a2,a3,0    # a2 is pointer to a[j]
  18: 00070793  addi  a5,a4,0    # j = i
Inner Loop:
  1c: ffc62883  lw    a7,-4(a2)  # a7 = a[j-1]
  20: 01185a63  bge   a6,a7,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01162023  sw    a7,0(a2)   # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ffc60613  addi  a2,a2,-4   # decrement a2 to point to a[j]
  30: fe0796e3  bne   a5,x0,1c   # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00279793  slli  a5,a5,0x2  # multiply a5 by 4
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107a023  sw    a6,0(a5)   # a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00468693  addi  a3,a3,4    # increment a3 to point to a[i]
  48: fc1ff06f  jal   x0,8       # jump to Outer Loop
```

图2.8：插入排序的RV32I代码如图2.5所示。十六进制的地址在左边， 接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是评论以及注释。RV32I分配两个寄存器用以指向a[j]和a[j-1]。RV32I有很多寄存器，其中一些被ABI预留用于函数调用。与其他ISA不同，它会跳过保存和恢复寄存器值到内存的过程。虽然代码大小大于x86-32，但使用可选的RV32C指令（请参阅第七章）缩小了指令大小的差距。注意RV32I中的一条比较和分支指令顶得上ARM-32和x86-32比较所需的三条指令。

```
# ARM-32 (19 instructions, 76 bytes; or 18 insns/46 bytes with Thumb-2)
# r0 points to a[0], r1 is n, r2 is j, r3 is i, r4 is x
  0: e3a03001 mov  r3, #1              # i = 1
  4: e1530001 cmp  r3, r1              # i vs. n (unnecessary?)
  8: e1a0c000 mov  ip, r0              # ip = a[0]
  c: 212fff1e bxcs lr                  # don't let return address change ISAs
 10: e92d4030 push {r4, r5, lr}        # save r4, r5, return address
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!       # x = a[i] ; increment ip
 18: e1a02003 mov  r2, r3              # j = i
 1c: e1a0e00c mov  lr, ip              # lr = a[0] (using lr as scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]       # r5 = a[j-1]
 24: e1550004 cmp  r5, r4              # compare a[j-1] vs. x
 28: da000002 ble  38                  # if a[j-1]<=a[i], jump to Exit Inner Loop
 2c: e2522001 subs r2, r2, #1          # j--
 30: e40e5004 str  r5, [lr], #-4       # a[j] = a[j-1]
 34: 1afffff9 bne  20                  # if j != 0, jump to Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1          # i++
 3c: e1530001 cmp  r3, r1              # i vs. n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14                  # if i < n, jump to Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}        # restore r4, r5, and return address
```

图 2.9：图 2.5 中插入排序的 ARM-32 代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释、评论。由于寄存器不足，为了腾出两个空寄存器，以便之后重用，ARM-32 将两个寄存器的值保存到堆栈中（和返回地址放在一起）。它使用了一种将 i 和 j 缩放为字节地址的寻址方式。鉴于分支跳转需要同时适用于 ARM-32 和 Thumb-2，bxcs 首先设置返回的最低有效位保存前地址为 0。条件码使得我们在递减 j 后在检查它时可以少用一条比较指令，但在其他地方比较仍然需要三条指令。

```
# MIPS-32 (24 instructions, 96 bytes, or 56 bytes with microMIPS)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
   0: 24860004 addiu a2,a0,4   # a2 is pointer to a[i]
   4: 24030001 li    v1,1      # i = 1
Outer Loop:
   8: 0065102b sltu  v0,v1,a1  # set on i < n
   c: 14400003 bnez  v0,1c     # if i<n, jump to Continue Outer Loop
  10: 00c03825 move  a3,a2     # a3 is pointer to a[j] (slot filled)
  14: 03e00008 jr    ra        # return from function
  18: 00000000 nop             # branch delay slot unfilled
Continue Outer Loop:
  1c: 8cc80000 lw    t0,0(a2)  # x = a[i]
  20: 00601025 move  v0,v1     # j = i
Inner Loop:
  24: 8ce9fffc lw    t1,-4(a3) # t1 = a[j-1]
  28: 00000000 nop             # load delay slot unfilled
  2c: 0109502a slt   t2,t0,t1  # set a[i] < a[j-1]
  30: 11400005 beqz  t2,48     # if a[j-1]<=a[i], jump to Exit Inner Loop
  34: 00000000 nop             # branch delay slot unfilled
  38: 2442ffff addiu v0,v0,-1  # j--
  3c: ace90000 sw    t1,0(a3)  # a[j] = a[j-1]
  40: 1440fff8 bnez  v0,24     # if j != 0, jump to Inner Loop
  44: 24e7fffc addiu a3,a3,-4  # decr. a2 to point to a[j] (slot filled)
Exit Inner Loop:
  48: 00021080 sll   v0,v0,0x2 #
  4c: 00821021 addu  v0,a0,v0  # v0 now byte address oi a[j]
  50: ac480000 sw    t0,0(v0)  # a[j] = x
  54: 24630001 addiu v1,v1,1   # i++
  58: 1000ffeb b     8         # jump to Outer Loop
  5c: 24c60004 addiu a2,a2,4   # incr. a2 to point to a[i] (slot filled)
```

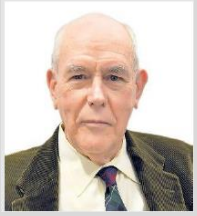
图2.10：图2.5中插入排序的MIPS-32代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释。MIPS-32代码中有三条nop指令，这增加了它的长度。两个是由于延迟分支，另一个是由于延迟加载。编译器无法找到有用的指令来填充延迟槽。延迟的分支也使代码更难理解，因为不管分支会不会跳转，延迟槽中的指令都会被执行。例如，地址5c处的最后一条指令（addiu）是循环的一部分，尽管它是在分支指令之后。

```
# x86-32 (20 instructions, 45 bytes)
# eax is j, ecx is x, edx is i
# pointer to a[0] is in memory at address esp+0xc, n is in memory at esp+0x10
  0: 56             push esi                # save esi on stack (esi needed below)
  1: 53             push ebx                # save ebx on stack (ebx needed below)
  2: ba 01 00 00 00 mov  edx,0x1            # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]      # ecx is pointer to a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]     # compare i vs. n
  f: 73 19          jae  2a <Exit Loop>     # if i >= n, jump to Exit Outer Loop
 11: 8b 1c 91       mov  ebx,[ecx+edx*4]    # x = a[i]
 14: 89 d0          mov  eax,edx            # j = i
Inner Loop:
 16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
 1a: 39 de          cmp  esi,ebx            # compare a[j-1] vs. x
 1c: 7e 06          jle  24 <Exit Loop>     # if a[j-1]<=a[i],jump Exit Inner Loop
 1e: 89 34 81       mov  [ecx+eax*4],esi    # a[j] = a[j-1]
 21: 48             dec  eax                # j--
 22: 75 f2          jne  16 <Inner Loop>    # if j != 0, jump to Inner Loop
Exit Inner Loop:
 24: 89 1c 81       mov  [ecx+eax*4],ebx    # a[j] = x
 27: 42             inc  edx                # i++
 28: eb e1          jmp  b <Outer Loop>     # jump to Outer Loop
Exit Outer Loop:
 2a: 5b             pop  ebx                # restore old value of ebx from stack
 2b: 5e             pop  esi                # restore old value of esi from stack
 2c: c3             ret                     # return from function
```

图 2.11：图 2.5 中插入排序的 x86-32 代码。十六进制的地址在左边，接下来是十六进制的机器语言代码，然后是汇编语言指令，最后是注释。由于缺少寄存器，x86-32 将将两个寄存器保存在堆栈中，以便腾出这两个寄存器供后续使用。而且，本来在 RV32I 中可以分配到寄存器的两个变量（n 和指向 a[0]的指针），现在是保存在内存中的。它使用扩展下标索引寻址模式，这对于访问[i]和[j]具有良好效果。 这里的 20 条 x32-86 指令中有 7 个是只有一个字节那么长，这使得对于这个简单的程序，x86-32 的代码规模很小。x86 有两个流行的汇编语言版本：Intel / Microsoft 和 AT＆T / Linux。我们使用英特尔语法，部分原因是它将目的地放在左边，而源操作数放在右边，与 RISC-V，ARM-32 和 MIPS-32 的操作数顺序一致。而 AT＆T 的操作数顺序则与之相反（并且对于寄存器操作数，需要在名字前加上%）。对于一些程序员来说，这看似微不足道的事情几乎是一个宗教问题。我们这里做出这样的选择，纯粹是因为教学方便，而非因为所谓“正统的信仰”。

第三章 RISC-V 汇编语言

Ivan Sutherland（1938-），因为在1962年发明出Sketchpad而获得图灵奖，被誉为计算机图形学之父。Sketchpad是现代计算机的图形用户界面的先驱。



给看似困难的问题找到简单的解法往往让人心满意足，而且最好的解法常常是都简单的。

——Ivan Sutherland

3.1 导言

图 3.1 表明了从 C 程序翻译成为可以在计算机上执行的机器语言程序的四个经典步骤。这一章的内容包括了后三个步骤，不过我们要从汇编语言在 RISC-V 函数调用规范中的作用开始说起。

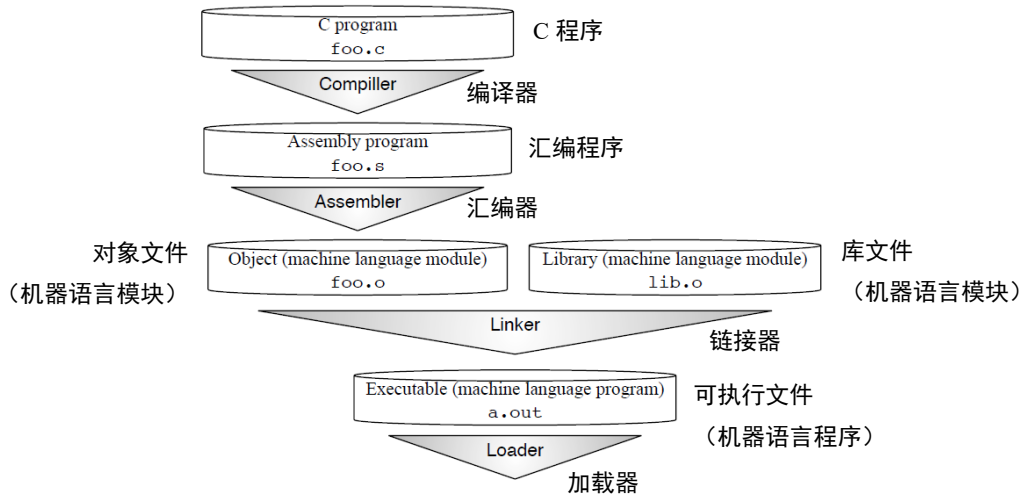


图 3.1 从 C 源代码翻译为可运行程序的步骤。这是从逻辑上进行的划分，实际中一些步骤会被结合起来，加速翻译过程。我们在这里使用了 Unix 的文件后缀命名习惯，分别对应 MS-DOS 中的.C, .ASM, .OBJ, .LIB 和.EXE。

3.2 函数调用规范（Calling convention）

函数调用过程通常分为 6 个阶段[Patterson and Hennessy 2017]。

1. 将参数存储到函数能够访问到的位置；
2. 跳转到函数开始位置（使用 RV32I 的 jal 指令）；
3. 获取函数需要的局部存储资源，按需保存寄存器；
4. 执行函数中的指令；
5. 将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源；

6. 返回调用函数的位置（使用 ret 指令）。

为了获得良好的性能，变量应该尽量存放在寄存器而不是内存中，但同时也要注意避免频繁地保存和恢复寄存器，因为它们同样会访问内存。

RISC-V 有足够多的寄存器来达到两全其美的结果：既能将操作数存放在寄存器中，同时也能减少保存和恢复寄存器的次数。其中的关键在于，在函数调用的过程中不保留部分寄存器存储的值，称它们为*临时寄存器*；另一些寄存器则对应地称为*保存寄存器*。不再调用其它函数的函数称为*叶函数*。当一个叶函数只有少量的参数和局部变量时，它们可以都被存储在寄存器中，而不会“溢出（spilling）”到内存中。但如果函数参数和局部变量很多，程序还是需要把寄存器的值保存在内存中，不过这种情况并不多见。

函数调用中其它的寄存器，要么被当做保存寄存器来使用，在函数调用前后值不变；要么被当做临时寄存器使用，在函数调用中不保留。函数会更改用来保存返回值的寄存器，因此它们和临时寄存器类似；用来给函数传递参数的寄存器也不需要保留，因此它们也类似于临时寄存器。对于其它一些寄存器，调用者需要保证它们在函数调用前后保持不变：比如用于存储返回地址的寄存器和存储栈指针的寄存器。图 3.2 列出了寄存器的 RISC-V 应用程序二进制接口（ABI）名称和它们在函数调用中是否保留的规定。

寄存器 Register	接口名称 ABI Name	描述 Description	在调用中是否保留？ Preserved across call?
x0	zero	Hard-wired zero 硬编码 0	—
x1	ra	Return address 返回地址	No
x2	sp	Stack pointer 栈指针	Yes
x3	gp	Global pointer 全局指针	—
x4	tp	Thread pointer 线程指针	—
x5	t0	Temporary/alternate link register 临时寄存器	No /备用链接寄存器
x6–7	t1–2	Temporaries 临时寄存器	No
x8	s0/fp	Saved register/frame pointer 保存寄存器	Yes /帧指针
x9	s1	Saved register 保存寄存器	Yes
x10–11	a0–1	Function arguments/return values 函数参数	No /返回值
x12–17	a2–7	Function arguments 函数参数	No
x18–27	s2–11	Saved registers 保存寄存器	Yes
x28–31	t3–6	Temporaries 临时寄存器	No
f0–7	ft0–7	FP temporaries 浮点临时寄存器	No
f8–9	fs0–1	FP saved registers 浮点保存寄存器	Yes
f10–11	fa0–1	FP arguments/return values 浮点参数/返回值	No
f12–17	fa2–7	FP arguments 浮点参数	No
f18–27	fs2–11	FP saved registers 浮点保存寄存器	Yes
f28–31	ft8–11	FP temporaries 浮点临时寄存器	No

图 3.2 RISC-V 整数和浮点寄存器的汇编助记符。RISC-V 有足够的寄存器，如果过程或方法不产生其它调用，就可以自由使用由 ABI 分配的寄存器，不需要保存和恢复。调用前后不变的寄存器也称为“由调用者保存的寄存器”，反之则称为“由被调用者保存的寄存器”。浮点寄存器将第 5 章进行解释。（这张图源于[Waterman and Asanovi´c 2017]的表 20.1。）

根据 ABI 规范，我们来看看标准的 RV32I 函数入口和出口。下面是函数的开头：

```
entry_label:
  addi sp,sp,-framesize     # Allocate space for stack frame      调整栈指针（sp 寄存器）分配栈帧
                            # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)   # Save return address (ra register) 保存返回地址（ra 寄存器）
  # save other registers to stack if needed 按需保存其它寄存器
  ... # body of the function  函数体
```

如果参数和局部变量太多，在寄存器中存不下，函数的开头会在栈中为函数帧分配空间，来存放。当一个函数的功能完成后，它的结尾部分释放栈帧并返回调用点：

```
# restore registers from stack if needed   按需恢复其它寄存器
lw   ra,framesize-4(sp)  # Restore return address register 恢复返回地址
addi sp,sp, framesize    # De-allocate space for stack frame 释放栈帧空间
ret                      # Return to calling point  返回调用点
```

我们很快将会看到使用这套 ABI 的一个例子，但首先我们需要对汇编的其它部分进行一些解释。

补充说明：保存寄存器和临时寄存器为什么不是连续编号的？
为了支持 RV32E——一个只有 16 个寄存器的嵌入式版本的 RISC-V（参见第 11 章），只使用寄存器 x0 到 x15——一部分保存寄存器和一部分临时寄存器都在这个范围内。其它的保存寄存器和临时寄存器在剩余 16 个寄存器内。RV32E 较小，但由于和 RV32I 不匹配，目前还没有编译器支持。

3.3 汇编器

在 Unix 系统中，这一步的输入是以.s 为后缀的文件，比如 foo.s；在 MS-DOS 中则是.ASM。

图 3.1 中的汇编器的作用不仅仅是从处理器能够理解的指令产生目标代码，还能翻译一些扩展指令，这些指令对汇编程序员或者编译器的编写者来说通常很有用。这类指令在巧妙配置常规指令的基础上实现，称为伪指令。图 3.3 和 3.4 列出了 RISC-V 伪指令，前者中要求 x0 寄存器始终为 0，后者中则没有这种要求。例如，之前提到的 ret 实际上是一个伪指令，汇编器会用 jalr x0, x1, 0 来替换它（见图 3.3）。大多数的 RISC-V 伪指令依赖于 x0。因此，把一个寄存器硬编码为 0 便于将许多常用指令——如跳转（jump）、返回（return）、等于 0 时转移（branch on equal to zero）——作为伪指令，进而简化 RISC-V 指令集。

图 3.5 为经典的 C 程序 Hello World，编译器产生的汇编指令如图 3.6，其中使用了图 3.2 的调用规范和图 3.3、3.4 的伪指令。

Hello World 程序通常是一个新设计处理器上运行的第一个程序。设计者通常把能运行操作系统并成功打印出“Hello World”作为新的芯片能工作的标志。他们会马上发邮件给领导和同事，告诉他们这个结果，然后出去搓一顿。

汇编程序的开头是一些汇编指示符（assemble directives）。它们是汇编器的命令，具有告诉汇编器代码和数据的位置、指定程序中使用的特定代码和数据常量等作用。图 3.9 是 RISC-V 的汇编指示符。其中图 3.6 中用到的指示符有：

- .text：进入代码段。
- .align 2：后续代码按 22 字节对齐。
- .globl main：声明全局符号“main”。
- .section .rodata：进入只读数据段
- .balign 4：数据段按 4 字节对齐。
- .string “Hello, %s!\n”：创建空字符结尾的字符串。
- .string “world”：创建空字符结尾的字符串。

汇编器产生如图 3.7 的目标文件，格式为标准的可执行可链接文件（ELF）格式[TIS Committee 1995]。

伪指令 Pseudoinstruction	基础指令 Base Instruction(s)	含义 Meaning	
nop	addi x0, x0, 0	No operation	无操作
neg rd, rs	sub rd, x0, rs	Two's complement	补码
negw rd, rs	subw rd, x0, rs	Two's complement word	字的补码
snez rd, rs	sltu rd, x0, rs	Set if $\neq$ zero	非 0 则置位
sltz rd, rs	slt rd, rs, x0	Set if $<$ zero	小于 0 则置位
sgtz rd, rs	slt rd, x0, rs	Set if $>$ zero	大于 0 则置位
beqz rs, offset	beq rs, x0, offset	Branch if $=$ zero	为 0 则转移
bnez rs, offset	bne rs, x0, offset	Branch if $\neq$ zero	非 0 则转移
blez rs, offset	bge x0, rs, offset	Branch if $\leq$ zero	小于等于 0 则转移
bgez rs, offset	bge rs, x0, offset	Branch if $\geq$ zero	大于等于 0 则转移
bltz rs, offset	blt rs, x0, offset	Branch if $<$ zero	小于 0 则转移
bgtz rs, offset	blt x0, rs, offset	Branch if $>$ zero	大于 0 则转移
j offset	jal x0, offset	Jump	跳转
jr rs	jalr x0, rs, 0	Jump register	寄存器跳转
ret	jalr x0, x1, 0	Return from subroutine	从子过程返回
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine	尾调用远程子过程
rdinstret[h] rd	csrrs rd, instret[h], x0	Read instructions-retired counter	读取过时指令计数器
rdcycle[h] rd	csrrs rd, cycle[h], x0	Read cycle counter	读取周期计数器
rdtime[h] rd	csrrs rd, time[h], x0	Read real-time clock	读取实时时钟
csrr rd, csr	csrrs rd, csr, x0	Read CSR	读 CSR 寄存器
csrw csr, rs	csrrw x0, csr, rs	Write CSR	写 CSR 寄存器
csrs csr, rs	csrrs x0, csr, rs	Set bits in CSR	CSR 寄存器置零位
csrc csr, rs	csrrc x0, csr, rs	Clear bits in CSR	CSR 寄存器清
csrwi csr, imm	csrrwi x0, csr, imm	Write CSR, immediate	立即数写入 CSR
csrsi csr, imm	csrrsi x0, csr, imm	Set bits in CSR, immediate	立即数置位 CSR
csrci csr, imm	csrrci x0, csr, imm	Clear bits in CSR, immediate	立即数清除 CSR
frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register	读取 FP 控制/状态寄存器
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register	写入 FP 控制/状态寄存器
frrm rd	csrrs rd, frm, x0	Read FP rounding mode	读取 FP 舍入模式
fsrm rs	csrrw x0, frm, rs	Write FP rounding mode	写入 FP 舍入模式
frflags rd	csrrs rd, fflags, x0	Read FP exception flags	读取 FP 例外标志
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags	写入 FP 例外标志

图 3.3 依赖于 x0 的 RISC-V 伪指令。附录 A 包含了这些 RISC-V 的伪指令和真实指令。在 RV32I 中，那些读取 64 位计数器的指令默认读取低 32 位，增加“h”时读取高 32 位。（这张图源于[Waterman and Asanovi´c 2017]的表 20.2 和表 20.3。）

伪指令 Pseudoinstruction	基础指令 Base Instruction(s)	含义 Meaning	
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address	取局部地址
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address	取地址
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global	读取全局量
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global	存储全局量
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global	读取浮点全局量
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global	存储浮点全局量
`li rd, immediate`	*Myriad sequences*	Load immediate	读取立即数
`mv rd, rs`	`addi rd, rs, 0`	Copy register	复制寄存器
`not rd, rs`	`xori rd, rs, -1`	One's complement	反码
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word	有符号扩展字
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero	为 0 时置位
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register	复制单精度寄存器
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value	单精度绝对值
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate	单精度相反数
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register	复制双精度寄存器
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value	双精度绝对值
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate	双精度相反数
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >	大于时转移
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤	小于等于时转移
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned	无符号大于时转移
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned	无符号小于等于时转移
`jal offset`	`jal x1, offset`	Jump and link	跳转并链接
`jalr rs`	`jalr x1, rs, 0`	Jump and link register	跳转并链接寄存器
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine	远程调用子过程
`fence`	`fence iorw, iorw`	Fence on all memory and I/O	内存和 I/O 屏障
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register	交换 FP 控制/状态寄存器
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode	交换 FP 舍入模式
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags	交换 FP 例外标志

图 3.4 不依赖于 x0 寄存器的 RISC-V 伪指令。在 la 指令一栏，GOT 代表全局偏移表（Global Offset Table），记录动态链接库中的符号的运行时地址。附录 A 包含了这些 RISC-V 的伪指令和真实指令。（这张图源于[Waterman and Asanovi´c 2017]的表 20.2 和表 20.3。）

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

图 3.5 C 语言的 Hello World 程序（hello.c）。

```
  .text                     # Directive: enter text section          指示符：进入代码段
  .align 2                  # Directive: align code to 2^2 bytes     指示符：按 2^2 字节对齐代码
  .globl main               # Directive: declare global symbol main  指示符：声明全局符号 main
main:                       # label for start of main   main 开始标记
  addi sp,sp,-16            # allocate stack frame      分配栈帧
  sw   ra,12(sp)            # save return address       存储返回地址
  lui  a0,%hi(string1)      # compute address of        计算 string1 的地址
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # compute address of        计算 string2 的地址
  addi a1,a1,%lo(string2)   #   string2
  call printf               # call function printf       调用 printf 函数
  lw   ra,12(sp)            # restore return address    恢复返回地址
  addi sp,sp,16             # deallocate stack frame    释放栈帧
  li   a0,0                 # load return value 0       读取返回值
  ret                       # return                    返回
  .section .rodata          # Directive: enter read-only data section  指示符：进入只读数据段
  .balign 4                 # Directive: align data section to 4 bytes 指示符：按 4 字节对齐数据
string1:                    # label for first string                   第一个字符串标记
  .string "Hello, %s!\n"    # Directive: null-terminated string        指示符：空字符结尾的字符串
string2:                    # label for second string                  第二个字符串标记
  .string "world"           # Directive: null-terminated string        指示符：空字符结尾的字符串
```

图 3.6 RISC-V 汇编语言的 Hello World 程序（hello.s）。

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

图 3.7 RISC-V 机器语言的 Hello World 程序（hello.o）。**位置 8 到 1c 这六条指令的地址字段为 0，将在后面由链接器填充。目标文件的符号表记录了链接器所需的标签和地址。**

3.4 链接器

链接器允许各个文件独立地进行编译和汇编，这样在改动部分文件时，不需要重新编译全部源代码。链接器把新的目标代码和已经存在的机器语言模块（如函数库）等“拼接”起来。链接器这个名字源于它的功能之一，即编辑所有对象文件的跳转并链接指令（jump and link）中的链接部分。它其实是链接编辑器（link editor）的简称，图 3.1 中的这一步骤过去就被称为链接编辑。在 Unix 系统中，链接器的输入文件有.o 后缀，输出 a.out 文件；在 MS-DOS 中输入文件后缀为.OBJ 或.LIB，输出.EXE 文件。

图 3.10 展示了一个典型的 RISC-V 程序分配给代码和数据的内存区域，链接器需要调整对象文件的指令中程序和数据的地址，使之与图中地址相符。如果输入文件中的是*与位置无关的代码*（PIC），链接器的工作量会有所降低。PIC 中所有的指令转移和文件内的数据访

问都不受代码位置的影响。如第 2 章所言，RV32I 的相对转移（PC-relative branch）特性使得程序更易于实现 PIC。

除了指令，每个目标文件还包含一个符号表，存储了程序中标签，由链接过程确定地址。其中包括了数据标签和代码标签。图 3.6 中有两个数据标签（string1 和 string2）和两个代码标签（main 和 printf）需要确定。由于在单个 32 位指令中很难指定一个 32 位的地址，RV32I 的链接器通常需要为每个标签调整两条指令。如图 3.6 所示：数据标签需要调整 lui 和 addi，代码标签需要调整 auipc 和 jalr。图 3.8 显示了图 3.7 中的目标文件链接后产生的 a.out 文件。

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

图 3.8 链接后的 RISC-V 机器语言 Hello World 程序。在 Unix 系统中，它的文件名是 a.out。

RISC-V 编译器支持多个 ABI，具体取决于 F 和 D 扩展是否存在。RV32 的 ABI 分别名为 ilp32，ilp32f 和 ilp32d。ilp32 表示 C 语言的整型（int），长整型（long）和指针（pointer）都是 32 位，可选后缀表示如何传递浮点参数。在 lip32 中，浮点参数在整数寄存器中传递；在 ilp32f 中，单精度浮点参数在浮点寄存器中传递；在 ilp32d 中，双精度浮点参数也在浮点寄存器中传递。

自然，如果想在浮点寄存中传递浮点参数，需要相应的浮点 ISA 添加 F 或 D 扩展（见第 5 章）。因此要编译 RV32I 的代码（GCC 选项-march=rv32i），必须使用 ilp32 ABI（GCC 选项-mabi=lib32）。反过来，调用约定并不要求浮点指令一定要使用浮点寄存器，因此 RV32IFD 与 ilp32，ilp32f 和 ilp32d 都兼容。

链接器检查程序的 ABI 是否和库匹配。尽管编译器本身可能支持多种 ABI 和 ISA 扩展的组合，但机器上可能只安装了特定的几种库。因此，一种常见的错误是在缺少合适的库的情况下链接程序。在这种情况下，链接器不会直接产生有用的诊断信息，它会尝试进行链接，然后提示不兼容。这种错误常常在从一台计算机上编译另一台计算机上运行的程序（交叉编译）时发生。

补充说明：链接器松弛（linker relaxation）

跳转并链接指令（jump and link）中有 20 位的相对地址域，因此一条指令就足够跳到很远的位置。尽管编译器为每个外部函数的跳转都生成了两条指令，很多时候其实一条就已经足够了。从两条指令到一条的优化同时节省了时间和空间开销，因此链接器会扫描几遍代码，尽可能地把两条指令替换为一条。每次替换会导致函数和调用它的位置之间的距离缩短，所以链接器会多次扫描替换，直到代码不再改变。这个过程称为链接器松弛，名字来源于求解方程组的松弛技术。除了过程调用之外，对于 gp 指针±2KiB 范围内的数据访问，RISC-V 链接器也会使用一个全局指针替换掉 lui 和 auipc 两条指令。对 tp 指针±2KiB 范围内的线程局部变量访问也有类似的处理。

指示符 Directive	描述 Description	
.text	Subsequent items are stored in the text section (machine code).	代码段（机器语言代码）。
.data	Subsequent items are stored in the data section (global variables).	数据段（全局变量）。
.bss	Subsequent items are stored in the bss section (global variables initialized to 0).	bss 段（初始化为 0 的全局变量）。
.section .foo	Subsequent items are stored in the section named .foo.	命名为.foo 的段。
.align n	Align the next datum on a 2^n-byte boundary. For example, .align 2 aligns the next value on a word boundary.	按 2n 字节对齐。如.align 2 是按字对齐。
.balign n	Align the next datum on a n-byte boundary. For example, .balign 4 aligns the next value on a word boundary.	按 n 字节对齐。如.balign 4 是按字对齐。
.globl sym	Declare that label sym is global and may be referenced from other files.	声明 sym 标签为全局的，可从其它文件访问。
.string "str"	Store the string str in memory and null-terminate it.	将字符串 str 存入内存，空字符结尾。
.byte b1,..., bn	Store the n 8-bit quantities in successive bytes of memory.	在内存中连续存储 n 个 8 位的量。
.half w1,...,wn	Store the n 16-bit quantities in successive memory halfwords.	在内存中连续存储 n 个 16 位的量。
.word w1,...,wn	Store the n 32-bit quantities in successive memory words.	在内存中连续存储 n 个 32 位的量。
.dword w1,...,wn	Store the n 64-bit quantities in successive memory doublewords.	在内存中连续存储 n 个 64 位的量。
.float f1,..., fn	Store the n single-precision floating-point numbers in successive memory words.	在内存中连续存储 n 个单精度浮点数。
.double d1,..., dn	Store the n double-precision floating-point numbers in successive memory doublewords.	在内存中连续存储 n 个双精度浮点数。
.option rvc	Compress subsequent instructions (see Chapter 7).	压缩指令（见第 7 章）。
.option norvc	Don't compress subsequent instructions.	不压缩指令。
.option relax	Allow linker relaxations for subsequent instructions.	允许链接器松弛（linker relaxation）。
.option norelax	Don't allow linker relaxations for subsequent instructions.	不允许链接器松弛。
.option pic	Subsequent instructions are position-independent code.	与位置无关代码段。
.option nopic	Subsequent instructions are position-dependent code.	与位置有关代码段。
.option push	Push the current setting of all .options to a stack, so that a subsequent .option pop will restore their value.	将所有的.option 设置存入栈。
.option pop	Pop the option stack, restoring all .options to their setting at the time of the last .option push.	从栈中弹出上次存入的.option 设置。

图 3.9 常见 RISC-V 汇编指示符。

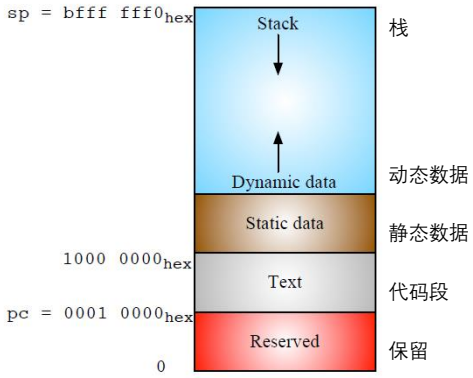


图 3.10 RV32I 为程序和数据分配内存。图中的顶部是高地址，底部是低地址。在 RISC-V 软件规范中，栈指针（sp）从 0xbffffff0 开始向下增长；程序代码段从 0x00010000 开始，包括静态链接库；程序代码段结束后是静态数据区，在这个例子中假设从 0x10000000 开始；然后是动态数据区，由 C 语言中的 malloc()函数分配，向上增长，其中包含动态链接库。

3.5 静态链接和动态链接

体系结构研究者常用静态链接的基准程序来测试处理器，尽管大多数实际的程序都有动态链接。他们说，关心性能的用户应该只使用静态链接，但其实这并不合理，因为加速实际的程序显然比加速基准程序更有意义。

上一节对静态链接（static linking）进行了说明，在程序运行前所有的库都进行了链接和加载。如果这样的库很大，链接一个库到多个程序中会十分占用内存。另外，链接时库是绑定的，即使它们后来的更新修复了 bug，强制的静态链接的代码仍然会使用旧的、有 bug 的版本。

为了解决这两个问题，现在的许多系统使用动态链接（dynamic linking），外部的函数在第一次被调用时才会加载和链接。后续所有调用都使用快速链接（fast linking），因此只会产生一次动态开销。每次程序开始运行，它都会按照需要链接最新版本的库函数。另外，如果多个程序使用了同一个动态链接库，库代码在内存中只会加载一次。

编译器产生的代码和静态链接的代码很相似。其不同之处在于，跳转的目标不是实际的函数，而是一个只有三条指令的存根函数（stub function）。存根函数会从内存中的一个表中加载实际的函数的地址并跳转。不过，在第一次调用时，表中还没有实际的函数的地址，只有一个动态链接的过程的地址。当这个动态链接过程被调用时，动态链接器通过符号表找到实际要调用的函数，复制到内存中，更新记录实际的函数地址的表。后续的每次调用的开销就是存根函数的三条指令的开销。

3.6 加载器

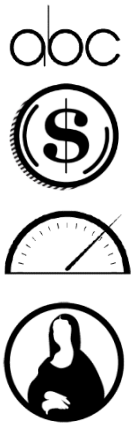
类似图 3.8 的程序以一个可执行文件的形式存储在计算机的存储设备上。运行时，加载器的作用是把这个程序加载到内存中，并跳转到它开始的地址。如今的“加载器”就是操作系统。换句话说，加载 a.out 是操作系统众多的任务之一。

动态链接程序的加载稍微有些复杂。操作系统不直接运行程序，而是运行一个动态链接器，再由动态链接器开始运行程序，并负责处理所有外部函数的第一次调用，把它们加载到内存中，并且修改程序，填入正确的调用地址。

3.7 结束语

保持简洁，保持功能单一。

——Kelly Johnson，提出“KISS 原则”的航空工程师，1960



汇编器向 RISC-V ISA 中增加了 60 条伪指令，使得 RISC-V 代码更易于读写，并且不增加硬件开销。将一个寄存器硬编码为 0 使得其中许多伪指令更容易实现。使用加载高位立即数（lui）和程序计数器与高位立即数相加（auipc）两条指令，简化了编译器和链接器寻找外部数据/函数的地址的过程。使用相对地址转移的代码与位置无关，减少了链接器的工作。大量的寄存器减少了寄存器保存和恢复的次数，加速函数调用和返回。

RISC-V 提供了一系列简单又有影响力的机制，降低成本，提高性能，并且使得编写程序更加容易。

3.8 扩展阅读

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

第四章 乘法和除法指令

若非必要，勿增实体。——奥卡姆的威廉（William of Occam），1320

奥卡姆的威廉（1452–1519）是一位英国神学家，他推广了现在所谓的“奥卡姆剃刀”原理，它意味着在科学方法中对简洁性的偏爱。

4.1 导言

RV32M 向 RV32I 中添加了整数乘法和除法指令。图 4.1 是 RV32M 扩展指令集的图形表示，图 4.2 列出了它们的操作码。

除法是直截了当的。可以回想起如下的式子：

$$\text{商} = (\text{被除数} - \text{余数}) \div \text{除数}$$

或者

$$\text{被除数} = \text{除数} \times \text{商} + \text{余数}$$

$$\text{余数} = \text{被除数} - (\text{商} \times \text{除数})$$

srl 可以做除数为 2^i 的无符号除法。例如，如果 a2=16（2^4），那么 srli t2,a1,4 这条指令和 divu t2,a1,a2 得到的结果相同。

RV32M 具有有符号和无符号整数的除法指令：divide(div)和 divide unsigned(divu)，它们将商放入目标寄存器。在少数情况下，程序员需要余数而不是商，因此 RV32M 提供 remainder(rem)和 remainder unsigned(remu)，它们在目标寄存器写入余数，而不是商。

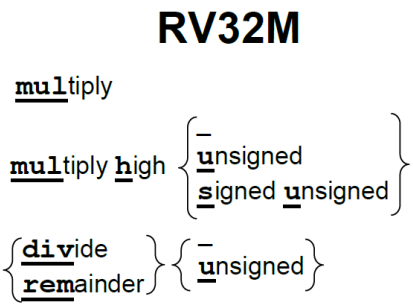


图 4.1：RV32M 指令的图示

31–25	24–20	19–15	14–12	11–7	6–0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

图 4.2：RV32M 操作码映射包含指令布局，操作码，指令格式类型和它们的名称（[Waterman and Asanovic 2017]的表 19.2 是此图的基础。）

乘法的式子很简单：

积 = 被乘数 × 乘数

它比除法要更为复杂，是因为积的长度是乘数和被乘数长度的和。将两个 32 位数相乘得到的是 64 位的乘积。为了正确地得到一个有符号或无符号的 64 位积，RISC-V 中带有四个乘法指令。要得到整数 32 位乘积（64 位中的低 32 位）就用 mul 指令。要得到高 32 位，如果操作数都是有符号数，就用 mulh 指令；如果操作数都是无符号数，就用 mulhu 指令；如果一个有符号一个无符号，可以用 mulhsu 指令。在一条指令中完成把 64 位积写入两个 32 位寄存器的操作会使硬件设计变得复杂，所以 RV32M 需要两条乘法指令才能得到一个完整的 64 位积。

sll 可以做乘数为 2^i 的无符号乘法。例如，如果 a2=16（2^4），那么 slli t2,a1,4 这条指令和 mul t2,a1,a2 得到的结果相同。

对许多微处理器来说，整数除法是相对较慢的操作。如前述，除数为 2 的幂次的无符号除法可以用右移来代替。事实证明，通过乘以近似倒数再修正积的高 32 位的方法，可以优化除数为其它数的除法。例如，图 4.3 显示了 3 为除数的无符号除法的代码。

```
# Compute unsigned division of a0 by 3 using multiplication.
   0: aaaab2b7    lui   t0,0xaaaab   # t0 = 0xaaaaaaab
   4: aab28293    addi  t0,t0,-1365 #    = ~ 2^32 / 1.5
   8: 025535b3    mulhu a1,a0,t0    # a1 = ~ (a0 / 1.5)
   c: 0015d593    srli  a1,a1,0x1   # a1 = (a0 / 3)
```

图 4.3：RV32M 中用乘法来实现除以常数操作的代码。要证明该算法适用于任何除数需要仔细的数值分析，而对于其它除数，其中的修正步骤更为复杂。算法正确性的证明以及产生倒数和修正步骤的算法在[Granlund and Montgomery 1994]中可以找到。

对于几乎所有的处理器，乘法比移位和加法慢很多，除法比乘法慢很多。

有什么不同之处？ 长期以来，ARM-32 只有乘法而无除法指令。直到第一台 ARM 处理器诞生的大约 20 年后（2005 年），除法指令才成为 ARM 的必要组成部分。MIPS-32 使用特殊寄存器（HI 和 LO）作为乘法和除法指令的唯一目标寄存器。虽然这种设计降低了早期 MIPS 处理器实现的复杂性，但它需要额外的移动指令以使用乘法或除法的结果，这可能会降低性能。HI 和 LO 寄存器也会增加架构状态，使得在任务之间切换的速度稍慢。

补充说明：mulhsu 对于多字有符号乘法很有用

当乘数有符号且被乘数无符号时，mulhsu 产生乘积的上半部分。当乘数的最高有效字（包含符号位）与被乘数的较低有效字（无符号）相乘时，它是多字有符号乘法的子步骤。该指令将多字乘法的性能提高了约 15%。

补充说明：检查是否除零也很简单

要测试除数是否为零，只需要在除法操作之前加入一条用于测试的 beqz 指令。RV32I 不会因为除零操作而 trap，因为极少数程序需要这种行为，而且在那些软件中可以很容易地检查是否除零。当然，除以其它常数永远不需要检查。

补充说明：mulh 和 mulhu 可以检查乘法的溢出

如果 mulhu 的结果为零，则在使用 mul 进行无符号乘法时不会溢出。类似地，如果 mulh 结果中的所有位与 mul 结果的符号位匹配（即当 mul 结果为正时 mulh 结果为 0，mul 结果为负时 mulh 结果为十六进制的 ffffffff），则使用 mul 进行有符号乘法时不会溢出。

4.2 结束语

最便宜，最快，并且最可靠的组件是那些没有出现的组件。

——C. Gordan Bell，著名小型计算机的架构师

为了为嵌入式应用提供最小的 RISC-V 处理器，乘法和除法被归入 RISC-V 的第一个可选标准扩展的一部分 RV32M。许多 RISC-V 处理器将包括 RV32M。

4.3 扩展阅读

T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN Notices*, volume 29, pages 61–72. ACM, 1994.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

第五章 RV32F 和 RV32D：单精度和双精度浮点数

Antoine de Saint Exup'ery, L'Avion（1900-1944）是法国作家和飞行员，以《小王子》一书而闻名。

只有当没有任何东西可以去除，而不是没有东西可以添加时，我们才最终达到了完美。

——Antoine de Saint Exup'ery, L'Avion, 1940

5.1 导言

尽管 RV32F 和 RV32D 是分开的，单独的可选指令集扩展，他们通常是包括在一起的。为简洁起见，我们在一章中介绍了几乎所有的单精度和双精度（32 位和 64 位）浮点指令。图 5.1 是一个 RV32F 和 RV32D 扩展指令集的图形表示。图 5.2 列出 RV32F 的操作码，图 5.3 列出了 RV32D 的操作码。和几乎所有其他现代 ISA 一样，RISC-V 服从 IEEE 754-2008 浮点标准[IEEE 标准委员会 2008]。

5.2 浮点寄存器

RV32F 和 RV32D 使用 32 个独立的 f 寄存器而不是 x 寄存器。使用两组寄存器的主要原因是：处理器在不增加 RISC-V 指令格式中寄存器描述符所占空间的情况下使用两组寄存器来将寄存器容量和带宽是乘 2，这可以提高处理器性能。使用两组寄存器对 RISC-V 指令集的主要影响是，必须要添加新的指令来加载和存储数据 f 寄存器，还需要添加新指令用于在 x 和 f 寄存器之间传递数据。图 5.4 列出了 RV32D 和 RV32F 寄存器及对应的由 RISC-V ABI 确定的寄存器名称。

如果处理器同时支持 RV32F 和 RV32D 扩展，则单精度数据仅使用 f 寄存器中的低 32 位。与 RV32I 中的 x0 不同，寄存器 f0 不是硬连线到常量 0，而是和所有其他 31 个 f 寄存器一样，是一个可变寄存器。

IEEE 754-2008 标准提供了几种浮点运算舍入的方法，这有助于确定误差范围和编写数值库。最准确且最常见的舍入模式是舍入到最近的偶数（RNE）。舍入模式可以通过浮点控制和状态寄存器 fcsr 进行设置。图 5.5 显示了 fcsr 并列出了舍入选项。它还包含标准所需的累积异常标志。

根据 MIPS 架构师之一 John Mashey 的说法，只有 16 个双精度寄存器是 MIPS 的 ISA 设计中犯过的最痛苦的错误。

有什么不同之处？ ARM-32 和 MIPS-32 都有 32 个单精度浮点寄存器但都只有 16 个双精度寄存器。它们都将两个单精度寄存器映射到双精度寄存器的左右两半。x86-32 浮点数算术没有任何寄存器，而是使用堆栈代替。堆栈条目是 80 位宽度提高精度，因此浮点数负载

补充说明：RV32FD 允许逐条指令设置舍入模式

这被称为静态舍入，当你只需要更改一条指令的舍入模式时，它可以帮助提高性能。默认设置是在 fcsr 中使用动态舍入模式。静态舍入所选择的模式是作为指令可选的最后一个参数，如 fadd.s ft0，ft1，ft2，rtz，将向零舍入，无论 fcsr 如何。图 5.5 的标题列出了不同舍入模式的名称。

将 32 位或 64 位操作数转换为 80 位，对于存储指令，反之亦然。x86-32 的一个后续版本增加了 8 个传统的 64 位浮点寄存器以及相关的操作指令。与 RV32FD 和 MIPS-32 不同，ARM-32 和 x86-32 忽视了在浮点和整数寄存器之间直接移动数据的指令。唯一的解决方案是先将浮点寄存器的内容存储在内存中，然后将其从内存加载到整数寄存器，反之亦然。

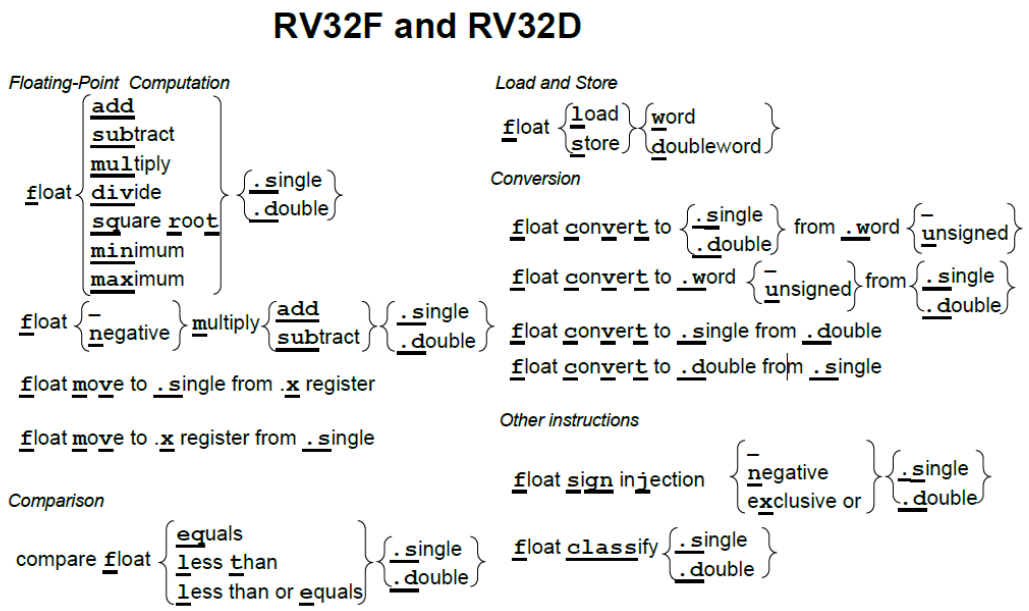


图 5.1：RV32F 和 RV32D 的指令图示。

5.3 浮点加载，存储和算术指令

对于 RV32F 和 RV32D，RISC-V 有两条加载指令（flw，fld）和两条存储指令（fsw，fsd）。他们和 lw 和 sw 拥有相同的寻址模式和指令格式。添加到标准算术运算中的指令有：（fadd.s，fadd.d，fsub.s，fsub.d，fmul.s，fmul.d，fdiv.s，fdiv.d），RV32F 和 RV32D 还包括平方根（fsqrt.s，fsqrt.d）指令。它们也有最小值和最大值指令（fmin.s，fmin.d，fmax.s，fmax.d），这些指令在不使用分支指令进行比较的情况下，将一对源操作数中的较小值或较大值写入目的寄存器。

不同于整数运算，浮点乘法的乘积大小与其操作数相同。此外，RV32F 和 RF32D 省略了浮点余数指令。

许多浮点算法（例如矩阵乘法）在执行完乘法运算后会立即执行一条加法或减法指令。因此，RISC-V 提供了指令用于先将两个操作数相乘然后将乘积加上（fmadd.s，fmadd.d）或减去（fmsub.s，fmsub.d）第三个操作数，最后再将结果写入目的寄存器。它还有在加上或减去第三个操作数之前对乘积取反的版本：fnmadd.s，fnmadd.d，fnmsub.s，fnmsub.d。这些融合的乘法 - 加法指令比单独的使用乘法及加法指令更准确，也更快，因为它们只（在加法之后）舍入过一次，而单独的乘法及加法指令则舍入了两次（先是在乘法之后，然后在加法之后）。这些指令需要一条新指令格式指定第 4 个寄存器，称为 R4。图 5.2 和 5.3 显示了 R4 格式，它是 R 格式的一个变种。

RV32F 和 RV32D 没有提供浮点分支指令，而是提供了浮点比较指令，这些根据两个浮点的比较结果将一个整数寄存器设置为 1 或 0：feq.s，feq.d，flt.s，flt.d，fle.s，fle.d。这些指令允许整数分支指令根据浮点数比较指令设置的条件进行分支跳转。例如，这段代码在 f1 <f2 时，则分支跳转到 Exit：

```
flt x5，f1，f2        # 如果 f1 < f2，则 x5 = 1;否则 x5 = 0
bne x5，x0，Exit  # 如果 x5！= 0，则跳转到 Exit
```

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	000011	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	010011	S fsw
rs3	00	rs2	rs1	rm	rd	100001	R4
rs3	00	rs2	rs1	rm	rd	100011	R4
rs3	00	rs2	rs1	rm	rd	100101	R4
rs3	00	rs2	rs1	rm	rd	100111	R4
0000000		rs2	rs1	rm	rd	101001	R fadd.s
0000100		rs2	rs1	rm	rd	101001	R fsub.s
0001000		rs2	rs1	rm	rd	101001	R fmul.s
0001100		rs2	rs1	rm	rd	101001	R fdiv.s
0001100		00000	rs1	rm	rd	101001	R fsqrt.s
0010000		rs2	rs1	000	rd	101001	R fsgnj.s
0010000		rs2	rs1	001	rd	101001	R fsgnjn.s
0010000		rs2	rs1	010	rd	101001	R fsgnjx.s
0010100		rs2	rs1	000	rd	101001	R fmin.s
0010100		rs2	rs1	001	rd	101001	R fmax.s
1100000		00000	rs1	rm	rd	101001	R fcvt.w.s
1100000		00001	rs1	rm	rd	101001	R
1110000		00000	rs1	000	rd	101001	R fmv.x.w
1010000		rs2	rs1	010	rd	101001	R feq.s
1010000		rs2	rs1	001	rd	101001	R flt.s
1010000		rs2	rs1	000	rd	101001	R fle.s
1110000		00000	rs1	001	rd	101001	R fclass.s
1101000		00000	rs1	rm	rd	101001	R fcvt.s.w
1101000		00001	rs1	rm	rd	101001	R
1111000		00000	rs1	000	rd	101001	R fmv.w.x

图 5.2：RV32F 操作码表包含了指令布局，操作码，格式类型和名称。这张表与下一张表在编码上的主要区别是：对于这张表，前两个指令第 12 位是 0，并且对于其余指令，第 25 位为 0，而在下一张表中，RV32D 中的这两个位均为 1（基于[Waterman and Asanovic 2017]的表 19.2）。

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	000011	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	010011	S fsd
rs3	01	rs2	rs1	rm	rd	100001	R4
rs3	01	rs2	rs1	rm	rd	100011	R4
rs3	01	rs2	rs1	rm	rd	100101	R4
rs3	01	rs2	rs1	rm	rd	100111	R4
0000001		rs2	rs1	rm	rd	101001	R fadd.d
0000101		rs2	rs1	rm	rd	101001	R fsub.d
0001001		rs2	rs1	rm	rd	101001	R fmul.d
0001101		rs2	rs1	rm	rd	101001	R fdiv.d
0001101		00000	rs1	rm	rd	101001	R fsqrt.d
0010001		rs2	rs1	000	rd	101001	R fsgnj.d
0010001		rs2	rs1	001	rd	101001	R fsgnjn.d
0010001		rs2	rs1	010	rd	101001	R fsgnjx.d
0010101		rs2	rs1	000	rd	101001	R fmin.d
0010101		rs2	rs1	001	rd	101001	R fmax.d
0100000		00001	rs1	rm	rd	101001	R fcvt.s.d
0100001		00000	rs1	rm	rd	101001	R fcvt.d.s
1010001		Rs2	rs1	010	rd	101001	R feq.d
1010001		rs2	rs1	001	rd	101001	R flt.d
1010001		rs2	rs1	000	rd	101001	R fle.d
1110001		00000	rs1	001	rd	101001	R fclass.d
1100001		00000	rs1	rm	rd	101001	R fcvt.w.d
1100001		00001	rs1	rm	rd	101001	R
1101001		00000	rs1	rm	rd	101001	R fmv.d.w
1101001		00001	rs1	rm	rd	101001	R

图 5.3：RV32D 操作码表包含了指令布局，操作码，格式类型和名称。这两个图中的一些指令并不仅仅是数据宽度不同。只有这张表有 fcvt.s.d 和 fcvt.d.s 指令，而只有另一张表有 fmv.x.w 和 fmv.w.x.指令（基于[Waterman and Asanovic 2017]的表 19.2）。

63 … 32	31 … 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	

图 5.4：RV32F 和 RV32D 的浮点寄存器。单精度寄存器占用了 32 个双精度寄存器中最右边的一半。第 3 章解释了 RISC-V 对于浮点寄存器的调用约定，阐述了 FP 参数寄存器（fa0-fa7），FP 保存寄存器（fs0-fs11）和 FP 临时寄存器（ft0-ft11）背后的基本原理（基于[Waterman and Asanovic 2017]的表 20.1）。

31 … 8	7 … 5	4	3	2	1	0
Reserved	Rounding Mode (`frm`)	Accrued Exceptions (`fflags`)				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

图 5.5：浮点控制和状态寄存器。它保存舍入模式和异常标志。舍入模式包括向最近的偶数舍入（frm 中的 rte，000）；向零舍入（rtz，001）；向下(−∞)舍入（rdn，010）；向上(+∞)舍入（rup，011）；以及向最近的最大值舍入（rmm，100）。 五个累积异常标志表示自上次由软件重置字段以来在任何浮点运算指令上出现的异常条件：NV 表示非法操作; DZ 表示除以零; OF 表示上溢; UF 表示下溢; NX 表示不精确（基于[Waterman andAsanovic 2017]的图 8.2）。

To	From			
	32b signed integer (`w`)	32b unsigned integer (`wu`)	32b floating point (`s`)	64b floating point (`d`)
32b signed integer (`w`)	–	–	`fcvt.w.s`	`fcvt.w.d`
32b unsigned integer (`wu`)	–	–	`fcvt.wu.s`	`fcvt.wu.d`
32b floating point (`s`)	`fcvt.s.w`	`fcvt.s.wu`	–	`fcvt.s.d`
64b floating point (`d`)	`fcvt.d.w`	`fcvt.d.wu`	`fcvt.d.s`	–

图 5.6：RV32F 和 RV32D 转换指令。在列中列出了源数据类型，在行中列出转换的目标数据类型。

5.4 浮点转换和搬运

RV32F 和 RV32D 支持在在 32 位有符号整数，32 位无符号整数，32 位浮点和 64 位之间浮点进行所有组合的转换（只要这个转换是有用，有意义的）。图 5.6 按源数据类型以及转换后的目的数据类型，罗列了这 10 条指令。

RV32F 还提供了将数据从 f 寄存器（fmv.x.w）移动到 x 寄存器的指令，以及反方向移动数据的指令（fmv.w.x）。

5.5 其他浮点指令

RV32F 和 RV32D 提供了不寻常的指令，有助于编写数学库以及提供有用的伪指令。（IEEE 754 浮点标准需要一种复制并且操作符号并对浮点数据进行分类的方式，这启发我们添加了这些指令。）

第一个是符号注入指令，它从第一个源操作数复制了除符号位之外的所有内容。符号位的取值取决于具体是什么指令：

1. 浮点符号注入（fsgnj.s，fsgnj.d）：结果的符号位是 rs2 的符号位。
2. 浮点符号取反注入（fsgnjn.s，fsgnjn.d）：结果的符号位与 rs2 的符号位相反。
3. 浮点符号异或注入（fsgnjx.s，fsgnjx.d）：结果符号位是 rs1 和 rs2 的符号位异或的结果。

除了有助于数学库中的符号操作，基于符号注入指令我们还提供了三种流行的浮点伪指令（参见第 37 页的图 3.4）：

1. 复制浮点寄存器：
   fmv.s rd，rs 事实上是 fsgnj.s rd，rs，rs
   fmv.d rd，rs 事实上是 sgnj.d rd，rs，rs。

```
void daxpy(size_t n, double a, const double x[], double y[])
{
  for (size_t i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```

图 5.7:用 C 编写的 浮点运算密集型的 DAXPY 程序

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
Instructions	10	10	12	12	16	11	11
Per Loop	6	6	7	7	6	7	7
Bytes	40	28	48	32	50	44	28

图 5.8：DAXPY 在四个 ISA 上生成的指令数和代码大小。它列出了每个循环的指令数量以及指令总数。第 7 章介绍 ARM Thumb-2，microMIPS 和 RV32C 指令集。

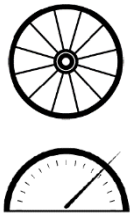
2. 否定：
   fneg.s rd，rs 映射到 fsgnjn.s rd，rs，rs
   fneg.d rd，rs 映射到 fsgnjn.d rd，rs，rs。
3. 绝对值（因为 0⊕0= 0 且 1⊕1= 0）：
   fabs.s rd，rs 变成了 fsgnjx.s rd，rs，rs
   fabs.d rd，rs 变成了 sgnjx.d rd，rs，rs。

第二个不常见的浮点指令是 classify 分类指令（fclass.s，fclass.d）。分类指令对数学库也很有帮助。他们测试一个源操作数来看源操作数满足下列 10 个浮点数属性中的哪些属性（参见下表），然后将测试结果的掩码写入目的整数寄存器的低 10 位。十位中仅有一位被设置为 1，其余为都设置为 0。

*x[rd]*位	含义
0	f [*rs1*]为−∞。
1	f [*rs1*]是负规格化数。
2	f [*rs1*]是负的非规格化数。
3	f [*rs1*]是-0。
4	f [*rs1*]是+0。
5	f [*rs1*]是正的非规格化数。
6	f [*rs1*]是正的规格化数。
7	f [*rs1*]为+∞。
8	f [*rs1*]是信号(signaling)NaN。
9	f [*rs1*]是一个安静(quiet)NaN。

5.6 使用 DAXPY 程序比较 RV32FD，ARM-32，MIPS-32 和 x86-32

DAXPY 这个名字来自公式本身：以双精度计算 A 乘上 X 加 Y (Double-precision A times X Plus Y)。此公式的单精度版本被称做 SAXPY。



我们现在将使用 DAXPY 作为我们的浮点基准对不同 ISA 进行比较（图 5.7）。它以双精度计算$Y = a \times X + Y$，其中 X 和 Y 是矢量，a 是标量。图 5.8 总结了 DAXPY 在四个不同的 ISA 下对应的指令数和字节数。他们的代码如图 5.9 至 5.12 所示。

与第 2 章中的插入排序一样，尽管 RISC-V 指令集强调本身的简单性，RISC-V 版本的不管是指令数量还是代码大小，都接近或者优于其他 ISA。在此示例中，RISC-V 的比较和执行分支指令和 ARM-32 和 x86-32 中更复杂的寻址模式，以及入栈、退栈指令节省了差不多数量的指令。

补充说明：16 位，128 位和十进制浮点运算

修订后的 IEEE 浮点标准（IEEE 754-2008）除了单精度和双精度之外，还描述了几种新的浮点数格式，它们称为 binary32 和 binary64。不出意料，修订后，新增了四倍精度，名为 binary128。 RISC-V 暂时计划用 RV32Q 扩展来支持新的浮点数格式（见第 11 章）。该标准还为二进制数据交换提供了两种新的数据尺寸，程序员可以会将这些浮点数以特定宽度存储在内存或存储中，但是或许不能以这种宽度进行计算计算。它们分别是半精度（binary16）和八元精度（binary256）。尽管标准对这两种新宽度的存储使用定义的，但 GPU 确实以半精度计算并将它们保存在内存中。 RISC-V 的计划在向量指令（第 8 章中的 RV32V）中包含半精度计算，但是前提是处理器如果支持向量半精度指令，则也必须支持半精度标量指令。令人惊讶的是，修订后标准还添加了十进制浮点数，新增的三种十进制格式分别是 decimal32，decimal64 和 decimal128。RISC-V 预留 RV32L 指令集扩展用于支持它（见第 11 章）。

5.7 结束语

少即是多。

——Robert Browning, 1855，极简主义（建筑）建筑学派在 20 世纪 80 年代采用这首诗作为公理。

IEEE 754-2008 浮点标准[IEEE Standards Committee 2008]定义了浮点数据类型，计算精度和所需操作。它的广泛流行大大降低了移植浮点程序的难度，这也意味着不同 ISA 中的浮点数部分可能比其他章节中描述的其他部分的指令更一致。

5.8 扩展阅读

IEEE Standards Committee. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

```
# RV32FD (7 insns in loop; 11 insns/44 bytes total; 28 bytes RVC)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0: 02050463 beqz    a0,28           # if n == 0, jump to Exit
  4: 00351513 slli    a0,a0,0x3       # a0 = n*8
  8: 00a60533 add     a0,a2,a0        # a0 = address of x[n] (last element)
Loop:
  c: 0005b787 fld     fa5,0(a1)       # fa5 = x[]
 10: 00063707 fld     fa4,0(a2)       # fa4 = y[]
 14: 00860613 addi    a2,a2,8         # a2++ (increment pointer to y)
 18: 00858593 addi    a1,a1,8         # a1++ (increment pointer to x)
 1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 20: fef63c27 fsd     fa5,-8(a2)      # y[i] = a*x[i] + y[i]
 24: fea614e3 bne     a2,a0,c         # if i != n, jump to Loop
Exit:
 28: 00008067         ret             # return
```

图 5.9：图 5.7 中 DAXPY 的 RV32D 代码。十六进制的地址位于机器的左侧，接下来是十六进制的语言代码，然后是汇编语言指令，最后是注释。比较和分支指令避免了 ARM-32 和 X86-32 代码中的两条比较指令。

```
# ARM-32 (6 insns in loop; 10 insns/40 bytes total; 28 bytes Thumb-2)
# r0 is n, d0 is a, r1 is pointer to x[0], r2 is pointer to y[0]
  0: e3500000 cmp      r0, #0             # compare n to 0
  4: 0a000006 beq      24 <daxpy+0x24>    # if n == 0, jump to Exit
  8: e0820180 add      r0, r2, r0, lsl #3 # r0 = address of x[n] (last element)
Loop:
  c: ecb16b02 vldmia   r1!,{d6}           # d6 = x[i], increment pointer to x
 10: ed927b00 vldr     d7,[r2]            # d7 = y[i]
 14: ee067b00 vmla.f64 d7, d6, d0         # d7 = a*x[i] + y[i]
 18: eca27b02 vstmia   r2!, {d7}          # y[i] = a*x[i] + y[i], incr. ptr to y
 1c: e1520000 cmp      r2, r0             # i vs. n
 20: 1afffff9 bne      c <daxpy+0xc>      # if i != n, jump to Loop
Exit:
 24: e12fff1e bx       lr                 # return
```

图 5.10：图 5.7 中 DAXPY 的 ARM-32 代码。 与 RISC-V 相比，ARM-32 的自动增量寻址模式可以节省两条指令。与插入排序不同，DAXPY 在 ARM-32 上不需要压栈和出栈寄存器。

```
# MIPS-32 (7 insns in loop; 12 insns/48 bytes total; 32 bytes microMIPS)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], f12 is a
  0: 10800009 beqz   a0,28 <daxpy+0x28>  # if n == 0, jump to Exit
  4: 000420c0 sll    a0,a0,0x3           # a0 = n*8 (filled branch delay slot)
  8: 00c42021 addu   a0,a2,a0            # a0 = address of x[n] (last element)
Loop:
  c: 24c60008 addiu  a2,a2,8             # a2++ (increment pointer to y)
 10: d4a00000 ldc1   $f0,0(a1)           # f0 = x[i]
 14: 24a50008 addiu  a1,a1,8             # a1++ (increment pointer to x)
 18: d4c2fff8 ldc1   $f2,-8(a2)          # f2 = y[i]
 1c: 4c406021 madd.d $f0,$f2,$f12,$f0    # f0 = a*x[i] + y[i]
 20: 14c4fffa bne    a2,a0,c <daxpy+0xc> # if i != n, jump to Loop
 24: f4c0fff8 sdc1   $f0,-8(a2)          # y[i] = a*x[i] + y[i] (filled delay slot)
Exit:
 28: 03e00008 jr     ra                  # return
 2c: 00000000 nop                        # (unfilled branch delay slot)
```

图 5.11：图 5.7 中 DAXPY 的 MIPS-32 代码。三个分支延迟槽中的两个填充了有用的指令。检查两个寄存器之间是否相等的指令避免了 ARM-32 和 x86-32 中的两条比较指令。与整数加载不同，浮点加载没有延迟槽。

```
# x86-32 (6 insns in loop; 16 insns/50 bytes total)
# eax is i, n is in memory at esp+0x8, a is in memory at esp+0xc
# pointer to x[0] is in memory at esp+0x14
# pointer to y[0] is in memory at esp+0x18
  0: 53                push       ebx                    # save ebx
  1: 8b 4c 24 08       mov        ecx,[esp+0x8]          # ecx has copy of n
  5: c5 fb 10 4c 24 0c vmovsd     xmm1,[esp+0xc]         # xmm1 has a copy of a
  b: 8b 5c 24 14       mov        ebx,[esp+0x14]         # ebx points to x[0]
  f: 8b 54 24 18       mov        edx,[esp+0x18]         # edx points to y[0]
 13: 85 c9             test       ecx,ecx                # compare n to 0
 15: 74 19             je         30 <daxpy+0x30>        # if n==0, jump to Exit
 17: 31 c0             xor        eax,eax                # i = 0 (since x^x==0)
Loop:
 19: c5 fb 10 04 c3    vmovsd     xmm0,[ebx+eax*8]       # xmm0 = x[i]
 1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
 24: c5 fb 11 04 c2    vmovsd     xmm0,xmm1,[edx+eax*8]  # y[i] = a*x[i] + y[i]
 29: 83 c0 01          add        eax,0x1                # i++
 2c: 39 c1             cmp        ecx,eax                # compare i vs n
 2e: 75 e9             jne        19 <daxpy+0x19>        # if i!=n, jump to Loop
Exit:
 30: 5b                pop        ebx                    # restore ebx
 31: c3                ret                               # return
```

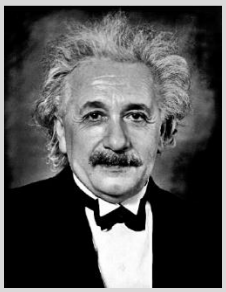
图 5.12：图 5.7 中 DAXPY 的 x86-32 代码。在这个例子中，x86-32 缺少寄存器的劣势在这里表现得很明显——有四个变量被分配到了内存，而在其他 ISA 中，这些变量是被存放在寄存器中的。它展示了 x86-32 中，如何将寄存器与零比较（test ecx，ecx）以及如何将一个寄存器清零（xor eax,eax）。

第六章 原子指令

所有的事物都应该尽量简单，但是不能太过简单。

——阿尔伯特·爱因斯坦（Albert Einstein），1933

阿尔伯特·爱因斯坦（1879-1955），20 世纪最著名的科学家。他提出了相对论，在第二次世界大战中提出制造原子弹。



6.1 导言

我们假定你已经了解了 ISA 对如何支持多进程，所以我们在这儿只对 RV32A 指令和它们的行为进行解释。如果你觉得需要一些背景知识补充，可以看一下维基百科上的“同步（计算机科学）”词条（英文维基地址：https://en.wikipedia.org/wiki/Synchronization 中文地址：https://zh.wikipedia.org/wiki/%E5%90%8C%E6%AD%A5 ）或者阅读《RISC-V 体系结构》2.1 节[Patterson and Hennessy 2017]。

RV32A 有两种类型的原子操作：

- 内存原子操作（AMO）
- 加载保留/条件存储（load reserved / store conditional）

图 6.1 是 RV32A 扩展指令集的示意图，图 6.2 列出了它们的操作码和指令格式。

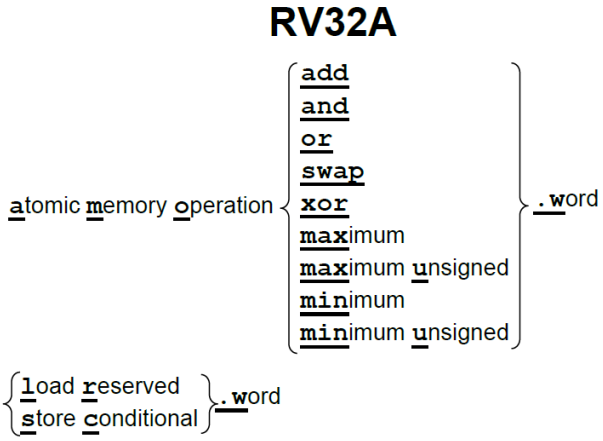


图 6.1 RV32A 指令图示

31		25	24 20	19 15	14 12	11 7	6 0	
00010	aq	rl	00000	rs1	010	rd	0101111	R lr.w
00011	aq	rl	rs2	rs1	010	rd	0101111	R sc.w
00001	aq	rl	rs2	rs1	010	rd	0101111	R amoswap.w
00000	aq	rl	rs2	rs1	010	rd	0101111	R amoadd.w
00100	aq	rl	rs2	rs1	010	rd	0101111	R amoxor.w
01100	aq	rl	rs2	rs1	010	rd	0101111	R amoand.w
01000	aq	rl	rs2	rs1	010	rd	0101111	R amoor.w
10000	aq	rl	rs2	rs1	010	rd	0101111	R amomin.w
10100	aq	rl	rs2	rs1	010	rd	0101111	R amomax.w
11000	aq	rl	rs2	rs1	010	rd	0101111	R amominu.w
11100	aq	rl	rs2	rs1	010	rd	0101111	R amomaxu.w

图 6.2 RV32A 指令格式、操作码、格式类型和名称。（这张图源于[Waterman and Asanovi´c 2017]的表 19.2。）

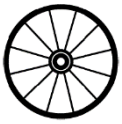
AMO 指令对内存中的操作数执行一个原子操作，并将目标寄存器设置为操作前的内存值。原子表示内存读写之间的过程不会被打断，内存值也不会被其它处理器修改。

加载保留和条件存储保证了它们两条指令之间的操作的原子性。加载保留读取一个内存字，存入目标寄存器中，并留下这个字的保留记录。而如果条件存储的目标地址上存在保留记录，它就把字存入这个地址。如果存入成功，它向目标寄存器中写入 0；否则写入一个非 0 的错误代码。

AMO 和 LR/SC 指令要求内存地址对齐，因为保证跨 cache 行的原子读写的难度很大。

为什么 RV32A 要提供两种原子操作呢？因为实际中存在两种不同的使用场景。

编程语言的开发者会假定体系结构提供了原子的比较-交换（compare-and-swap）操作：比较一个寄存器中的值和另一个寄存器中的内存地址指向的值，如果它们相等，将第三个寄存器中的值和内存中的值进行交换。这是一条通用的同步原语，其它的同步操作可以以它为基础来完成[Herlihy 1991]。



尽管将这样一条指令加入 ISA 看起来十分有必要，它在一条指令中却需要 3 个源寄存器和 1 个目标寄存器。源操作数从两个增加到三个，会使得整数数据通路、控制逻辑和指令格式都变得复杂许多。（RV32FD 的多路加法（multiply-add）指令有三个源操作数，但它影响的是浮点数据通路，而不是整数数据通路。）不过，加载保留和条件存储只需要两个源寄存器，用它们可以实现原子的比较交换（见图 6.3 的上半部分）。

用 lr/sc 实现内存字 M[a0]的比较-交换操作

```
# Compare-and-swap (CAS) memory word M[a0] using lr/sc.
# Expected old value in a1; desired new value in a2.
   0: 100526af    lr.w  a3,(a0)      # Load old value          #加载旧的值
   4: 06b69e63    bne   a3,a1,80     # Old value equals a1?    #比较旧的值与 a1 是否相等
   8: 18c526af    sc.w  a3,a2,(a0)   # Swap in new value if so #相等则存入新的值
   c: fe069ae3    bnez  a3,0         # Retry if store failed   #如果存入失败，重新尝试
        ... code following successful CAS goes here ...       …比较-交换成功之后的代码…
  80:                                # Unsuccessful CAS.       #比较-交换不成功
```

```
# Critical section guarded by test-and-set spinlock using an AMO.
   0: 00100293    li            t0,1         # Initialize lock value   #初始化锁
   4: 0c55232f    amoswap.w.aq  t1,t0,(a0)   # Attempt to acquire lock #尝试获取锁
   8: fe031ee3    bnez          t1,4         # Retry if unsuccessful   #如果失败，继续尝试
        ... critical section goes here ...                            …临界区代码…
  20: 0a05202f    amoswap.w.rl  x0,x0,(a0)   # Release lock.           #释放锁
```

图 6.3 同步的两个例子。第一个例子使用加载保留 lr.w/条件存储 sc.w 实现比较-交换操作；第二个例子使用原子交换 amoswap.w 实现互斥。

另外还提供 AMO 指令的原因是，它们在多处理器系统中拥有比加载保留/条件存储更好的可扩展性，例如可以用它们来实现高效的归约。AMO 指令在于 I/O 设备通信时也很有用，可以实现总线事务的原子读写。这种原子性可以简化设备驱动，并提高 I/O 性能。图 6.3 的下半部分展示了如何使用原子交换实现临界区。

补充说明： 内存一致性模型

RISC-V 具有宽松的内存一致性模型（relaxed memory consistency model），因此其他线程看到的内存访问可以是乱序的。图 6.2 中，所有的 RV32A 指令都有一个请求位（aq）和一个释放位（rl）。aq 被置位的原子指令保证其它线程在随后的内存访问中看到顺序的 AMO 操作；rl 被置位的原子指令保证其它线程在此之前看到顺序的原子操作。想要了解更详细的有关知识，可以查看[Adve and Gharachorloo 1996]。

有什么不同之处？ 原始的 MIPS-32 没有同步机制，设计者在后来的 MIPS ISA 中加入了加载保留/条件存储指令。

6.2 结束语

RV32A 是可选的，一个 RISC-V 处理器如果没有它就会更加简单。然而，正如爱因斯坦所言，一切事物都应该尽量简单，但不应该太过简单。RV32A 正是如此，许多的场景都离不开它。

6.3 扩展阅读

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

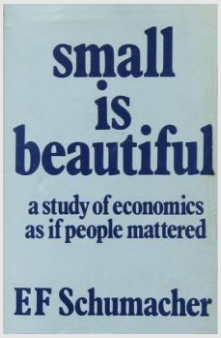
注记

http://parlab.eecs.berkeley.edu

第七章 压缩指令

小即是美。 ——E. F. Schumacher, 1973

E. F. Schumacher（1911-1977）撰写了一本经济学著作，主张人性化，分散化和适当的技术。它被翻译成多种语言，被评为第二次世界大战以来最具影响力的 100 本书之一。



7.1 导言

以前的 ISA 为了缩短代码长度而显著扩展了指令和指令格式的数量，比如添加了一些只有两个（而不是三个）操作数的指令，减小立即数域，等等。ARM 和 MIPS 为了能缩小代码，重新设计了两遍指令集，ARM 设计出了 ARM Thumb 和 Thumb 2，MIPS 先后设计出了 MIPS16 和 microMIPS。这些新的 ISA 为处理器和编译器增加了负担，同时也增加了汇编语言程序员的认知负担。

RV32C 采用了一种新颖的方法：每条短指令必须和一条标准的 32 位 RISC-V 指令一一对应。此外，16 位指令只对汇编器和链接器可见，并且是否以短指令取代对应的宽指令由它们决定。编译器编写者和汇编语言程序员可以幸福地忽略 RV32C 指令及其格式，他们能感知到的则是最后的程序大小小于大多数其它 ISA 的程序。图 7.1 是 RV32C 扩展指令集的图形化表示。

为了能在一系列的程序上得到良好的代码压缩效果，RISC-V 架构师精心挑选了 RVC 扩展中的指令。同时，基于以下的三点观察，架构师们成功地将指令压缩到了 16 位。第一，对十个常用寄存器（a0-a5，s0-s1，sp 以及 ra）访问的频率远超过其他寄存器；第二，许多指令的写入目标是它的源操作数之一；第三，立即数往往很小，而且有些指令比较喜欢某些特定的立即数。因此，许多 RV32C 指令只能访问那些常用寄存器；一些指令隐式写入源操作数的位置；几乎所有的立即数都被缩短了，load 和 store 操作只使用操作数整数倍尺寸的无符号数偏移量。

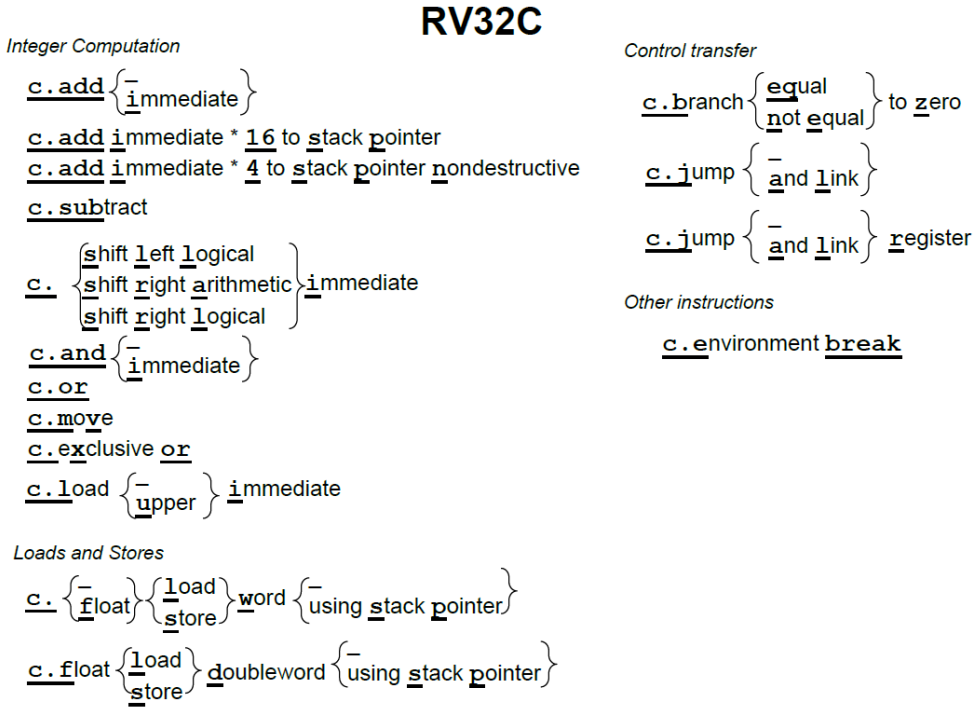


图 7.1：RV32C 的指令图示。移位指令的立即数域和 c.addi4spn 是零扩展的，其它指令采用符号位扩展。

Benchmark	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Insertion Sort	Instructions	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instructions	10	12	16	11
	Bytes	28	32	50	28

图 7.2：用压缩指令集写成的插入排序和 DAXPY 程序的指令数和代码长度。

图 7.3 和 7.4 列出了插入排序和 DAXPY 程序的 RV32C 代码。我们展示了这些 RV32C 指令，从而清楚地显示了这些压缩操作的效果，但是通常这些指令在汇编程序中是不可见的。注释中在括号内标出了与 RV32C 指令对应的等效 32 位指令。附录 A 中完整列出了 16 位 RV32C 指令和 32 位 RISC-V 指令的对应关系。

例如，在图 7.3 的插入排序程序中地址为 4 的地方，汇编器将如下的 32 位 RV32I 指令：

```
addi a4,x0,1    # i = 1
```

替换为了这条 16 位 RV32C 指令：

```
c.li a4,1       #（可扩展为 addi a4,x0,1） i = 1
```

RV32C 的 load 立即数指令比较短，是因为它只能指定一个寄存器和一个小的立即数。c.li 的机器码在图 7.3 中只有 4 个十六进制数，这表明 c.li 指令确实只有 2 字节长。

另一个例子在图 7.3 中地址为 10 的地方，汇编器将：

```
add a2,x0,a3    # a2 是指向 a[j]的指针
```

换成了这条 16 位 RV32C 指令：

```
c.mv a2,a3      #（可扩展为 add a2,x0,a3） a2 是指向 a[j]的指针
```

RV32C 的 move 指令只有 16 位长，因为它只指定两个寄存器。

尽管处理器的设计者们不能忽略 RV32C 的存在，但是有一个技巧可以让实现的代价变小：在执行之前用一个解码器将所有的 16 位指令转换为等价的 32 位指令。图 7.6 到 7.8 列出了解码器可以转换的 RV32C 指令的格式和操作码。最小的不支持任何扩展的 32 位 RISC-V 处理器要用到 8000 个门电路，而解码器只要 400 个门。如果它在这么小的设计中都只占 5%的体量，那么它在约有 100,000 个门的中等大小带有 cache 的处理器中相当于不占资源。

有什么不同之处？RV32C 中没有字节或半字指令，因为其他指令对代码长度的影响更大。第 9 页图 1.5 中 Thumb-2 相对于 RV32C，在代码长度上更有优势，这是由于 Load and Store Multiple 对于过程（函数、子例程）进入和退出时可以节省不少代码。为了保证能和 RV32G 中的指令一一对应，RV32C 中没有包括它们。而 RV32G 为了降低高端处理器的实现复杂性而省略了这些指令。由于 Thumb-2 是独立于 ARM-32 的 ISA，但是处理器可以在两个 ISA 间切换。为了支持两套 ISA，硬件必须有两个指令解码器，一个用于 ARM-32，一个用于 Thumb-2。RV32GC 是一个单独的 ISA，因此 RISC-V 处理器只需要一个解码器。

补充说明：为什么有些架构师不考虑 RV32C？

超标量处理器在一个时钟周期内同时取几条指令，因此译码阶段可能成为超标量处理器的瓶颈。macrofusion 是另一个例子，其中指令解码器把 RISC-V 指令组合为更加强大的指令来执行（参见第一章）。在这种情况下，16 位 RV32C 指令和 32 位 RV32I 指令混杂在一起增加了解码的复杂度，从而使得高性能处理器中在一个时钟周期内完成解码变得更难。

7.2 RV32GC，Thumb-2，microMIPS 和 x86-32 的比较

图 7.2 汇总了这四个 ISA 写成的插入排序和 DAXPY 程序的代码大小。

在插入排序的原始 19 条 RV32I 指令中，12 条被替换成了 RV32C 指令，所以代码长度从$19 \times 4 = 76$个字节变成了$12 \times 2 + 7 \times 4 = 52$个字节，节省了$24/76 = 32\%$。DAXPY 程序从$11 \times 4 = 44$个字节缩减到了$8 \times 2 + 3 \times 4 = 28$个字节，节省了$16/44 = 36\%$。

这两个小例子的结果与第二章第 9 页的图 1.5 惊人的一致，那里提到说，对于更多更复杂的程序，RV32G 代码比 RV32GC 代码长 37%。要达到这种程度的长度缩减，程序中必须有一半的指令可以被替换成 RV32C 指令。

补充说明：RV32C 真的是独一无二的吗？

RV32 指令在 RV32IC 中无法区分。Thumb-2 实际上是一个单独的 ISA，包含 16 位指令和 ARMv7 中大多数（但不是全部）的指令。例如，在 Thumb-2 中有 Compare and Branch on Zero，而 ARMv7 中没有，而对于 Reverse Subtarct with Carry 则正好相反。microMIPS 也不是 MIPS32 的超集。例如，microMIPS 计算分支偏移量的时候乘以 2，但在 MIPS32 中则为 4。RISC-V 中总是乘以 2。

7.3 结束语

我本可以把信写得更短，但我没有时间。——Blaise Pascal, 1656

他是建造了第一台机械计算器的数学家，因此图灵奖得主 Niklaus Wirth 用他的名字命名了一门编程语言。

RV32C 让 RISC-V 程序拥有了当今几乎最小的代码尺寸。你几乎可以将它们视为硬件协助的伪指令。但是，现在汇编器将它们在汇编语言程序员和编译器编写者面前隐藏起来。这里我们没有像第三章那样，将能提升 RISC-V 代码易用性与易读性的常用操作的组织成指令，来扩展真实的指令集。这两种方法都有助于提供程序员的工作效率。

RISC-V 提倡用一套简洁、有效的机制来提升性价比，RV32C 就是一个极佳的范例。

7.4 扩展阅读

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

```
# RV32C (19 instructions, 52 bytes)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
  0: 00450693 addi   a3,a0,4   # a3 is pointer to a[i]
  4: 4705     c.li   a4,1      # (expands to addi a4,x0,1) i = 1
Outer Loop:
  6: 00b76363 bltu   a4,a1,c   # if i < n, jump to Continue Outer loop
  a: 8082     c.ret            # (expands to jalr x0,ra,0) return from function
Continue Outer Loop:
  c: 0006a803 lw     a6,0(a3)  # x = a[i]
 10: 8636     c.mv   a2,a3     # (expands to add a2,x0,a3) a2 is pointer to a[j]
 12: 87ba     c.mv   a5,a4     # (expands to add a5,x0,a4) j = i
InnerLoop:
 14: ffc62883 lw     a7,-4(a2) # a7 = a[j-1]
 18: 01185763 ble    a7,a6,26  # if a[j-1] <= a[i], jump to Exit InnerLoop
 1c: 01162023 sw     a7,0(a2)  # a[j] = a[j-1]
 20: 17fd     c.addi a5,-1     # (expands to addi a5,a5,-1) j--
 22: 1671     c.addi a2,-4     # (expands to addi a2,a2,-4)decr a2 to point to a[j]
 24: fbe5     c.bnez a5,14     # (expands to bne a5,x0,14)if j!=0,jump to InnerLoop
Exit InnerLoop:
 26: 078a     c.slli a5,0x2    # (expands to slli a5,a5,0x2) multiply a5 by 4
 28: 97aa     c.add  a5,a0     # (expands to add a5,a5,a0)a5 = byte address of a[j]
 2a: 0107a023 sw     a6,0(a5)  # a[j] = x
 2e: 0705     c.addi a4,1      # (expands to addi a4,a4,1) i++
 30: 0691     c.addi a3,4      # (expands to addi a3,a3,4) incr a3 to point to a[i]
 32: bfd1     c.j    6         # (expands to jal x0,6) jump to Outer Loop
```

图 7.3：插入排序的 RV32C 代码。12 条 16 位指令使得代码长度缩减了 32%。每条指令的宽度可以很容易地得知。RV32C 指令（以 c.开头）在这个例子中显式出现，但通常汇编语言程序员和编译器无法看到它们。

```
# RV32DC (11 instructions, 28 bytes)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0: cd09     c.beqz a0,1a        # (expands to beq a0,x0,1a) if n==0, jump to Exit
  2: 050e     c.slli a0,a0,0x3    # (expands to slli a0,a0,0x3) a0 = n*8
  4: 9532     c.add a0,a2         # (expands to add a0,a0,a2) a0 = address of x[n]
Loop:
  6: 2218     c.fld fa4,0(a2)     # (expands to fld fa4,0(a2) ) fa5 = x[]
  8: 219c     c.fld fa5,0(a1)     # (expands to fld fa5,0(a1) ) fa4 = y[]
  a: 0621     c.addi a2,8         # (expands to addi a2,a2,8) a2++ (incr. ptr to y)
  c: 05a1     c.addi a1,8         # (expands to addi a1,a1,8) a1++ (incr. ptr to x)
  e: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 12: fef63c27 fsd fa5,-8(a2)      # y[i] = a*x[i] + y[i]
 16: fea618e3 bne a2,a0,6         # if i != n, jump to Loop
Exit:
 1a: 8082     ret                 # (expands to jalr x0,ra,0) return from function
```

图 7.4：DAXPY 的 RV32DC 代码。8 条十六位指令将代码长度缩减了 36%。每条指令的宽度见第二列的十六进制字符个数。RV32C 指令（以 c.开头）在这个例子中显式出现，但通常汇编语言程序员和编译器无法看到它们。

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzimm[5]	0	nzimm[4:0]	01	CI c.nop
000	nzimm[5]	rs1/rd≠0	nzimm[4:0]	01	CI c.addi
001	imm[11|4|9:8|10|6|7|3:1|5]			01	CJ c.jal
010	imm[5]	rd≠0	imm[4:0]	01	CI c.li
011	nzimm[9]	2	nzimm[4|6|8:7|5]	01	CI c.addi16sp
011	nzimm[17]	rd≠{0, 2}	nzimm[16:12]	01	CI c.lui
100	nzuimm[5]	00 rs1′/rd′	nzuimm[4:0]	01	CI c.srli
100	nzuimm[5]	01 rs1′/rd′	nzuimm[4:0]	01	CI c.srai
100	imm[5]	10 rs1′/rd′	imm[4:0]	01	CI c.andi
100	0	11 rs1′/rd′	00 rs2′	01	CR c.sub
100	0	11 rs1′/rd′	01 rs2′	01	CR c.xor
100	0	11 rs1′/rd′	10 rs2′	01	CR c.or
100	0	11 rs1′/rd′	11 rs2′	01	CR c.and
101	imm[11|4|9:8|10|6|7|3:1|5]			01	CJ c.j
110	imm[8|4:3]	rs1′	imm[7:6|2:1|5]	01	CB c.beqz
111	imm[8|4:3]	rs1′	imm[7:6|2:1|5]	01	CB c.bnez

图7.5：RV32C操作码映射（bits[1:0] = 01）列出了指令布局，操作码，指令格式和指令名称。rd’,rs1’和rs2’指的是10个常用的寄存器a0-a5，s0-s1，sp和ra。**（本图来源于[Waterman and Asanovi´c 2017]的表12.5。）**

15 14 13	12 11 10 9 8 7 6 5	4 3 2	1 0	
000	0	0	00	CIW *Illegal instruction*
000	nzuimm[5:4|9:6|2|3]	rd′	00	CIW c.addi4spn
001	uimm[5:3] rs1′ uimm[7:6]	rd′	00	CL c.fld
010	uimm[5:3] rs1′ uimm[2|6]	rd′	00	CL c.lw
011	uimm[5:3] rs1′ uimm[2|6]	rd′	00	CL c.flw
101	uimm[5:3] rs1′ uimm[7:6]	rs2′	00	CL c.fsd
110	uimm[5:3] rs1′ uimm[2|6]	rs2′	00	CL c.sw
111	uimm[5:3] rs1′ uimm[2|6]	rs2′	00	CL c.fsw

图7.6：RV32C操作码表（bits[1:0] = 00）列出了指令布局，操作码，指令格式和指令名称。rd’,rs1’和rs2’指的是10个常用的寄存器a0-a5，s0-s1，sp和ra。**（本图来源于[Waterman and Asanovi´c 2017]的表12.4。）**

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	CI c.slli
000	0	rs1/rd≠0	0	10	CI c.slli64
001	uimm[5]	rd	uimm[4:3|8:6]	10	CSS c.fldsp
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	CSS c.lwsp
011	uimm[5]	rd	uimm[4:2|7:6]	10	CSS c.flwsp
100	0	rs1≠0	0	10	CJ c.jr
100	0	rd≠0	rs2≠0	10	CR c.mv
100	1	0	0	10	CI c.ebreak
100	1	rs1≠0	0	10	CJ c.jalr
100	1	rs1/rd≠0	rs2≠0	10	CR c.add
101	uimm[5:3|8:6]		rs2	10	CSS c.fsdsp
110	uimm[5:2|7:6]		rs2	10	CSS c.swsp
111	uimm[5:2|7:6]		rs2	10	CSS c.fswsp

图7.7：RV32C操作码表（bits[1:0] = 10）列出了指令布局，操作码，指令格式和指令名称。（本图来源于[Waterman and Asanovi´c 2017]的表12.6。）

Format	Meaning	15 14 13	12	11 10 9	8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm	rs1′		imm	rd′	op
CS	Store	funct3	imm	rs1′		imm	rs2′	op
CB	Branch	funct3	offset	rs1′		offset		op
CJ	Jump	funct3	jump target					op

图7.8：16位RVC压缩指令的格式。rd',rs1'和rs2'指的是10个常用的寄存器a0-a5，s0-s1，sp和ra。（本图来源于[Waterman and Asanovi´c 2017]的表12.1。）

第八章 向量

我追求简洁。我理解不了那些复杂的东西。——Seymour Cray

Seymour Cray（1925–1996）是 1976 年第一个采用向量架构的，并且在商业上取得成功的超级计算机 Cray-1 的架构师。Cray-1 是一颗明珠，即使没有使用向量指令，它也是世界上最快的计算机。

8.1 导言

本章重点介绍数据并行，当存在大量数据可供应用程序同时计算时，我们称之为数据级并行性。数组是一个常见的例子。虽然它是科学应用的基础，但它也被多媒体程序使用。前者使用单精度和双精度浮点数据，后者通常使用 8 位和 16 位整数数据。

最著名的数据级并行架构是单指令多数据(SIMD，Single Instruction Multiple Data)。SIMD 最初的流行是因为它将 64 位寄存器的数据分成许多个 8 位、16 位或 32 位的部分，然后并行地计算它们。操作码提供了数据宽度和操作类型。数据传输只用单个（宽）SIMD 寄存器的 load 和 store 进行。

1997 年的英特尔多媒体扩展（MMX）使 SIMD 流行起来。它们通过 1999 年的流媒体 SIMD 扩展（SSE，Streaming SIMD Extensions）和 2010 年的高级向量扩展（AVX）得到了接受和扩展。MMX 的名声在英特尔的一则广告中得到了宣扬，该广告的内容是一种采用彩色干净套装的半导体产品线的光纤工作者在跳迪斯科(https://www.youtube.com/watch?v=paU16B-bZEA)。

把现有的 64 位寄存器进行拆分的做法由于其简单性而显得十分诱人。为了使 SIMD 更快，架构师随后加宽寄存器以同时计算更多部分。由于 SIMD ISA 属于增量设计阵营的一员，并且操作码指定了数据宽度，因此扩展 SIMD 寄存器也就意味着要同时扩展 SIMD 指令集。将 SIMD 寄存器宽度和 SIMD 指令数量翻倍的后续演进步骤都让 ISA 走上了复杂度逐渐提升的道路，这一后果由处理器设计者、编译器编写者和汇编语言程序员共同承担。

一个更老的，并且在我们看来更优雅的，利用数据级并行性的方案是向量架构。本章解释了我们在 RISC-V 中使用向量而不是 SIMD 的理由。

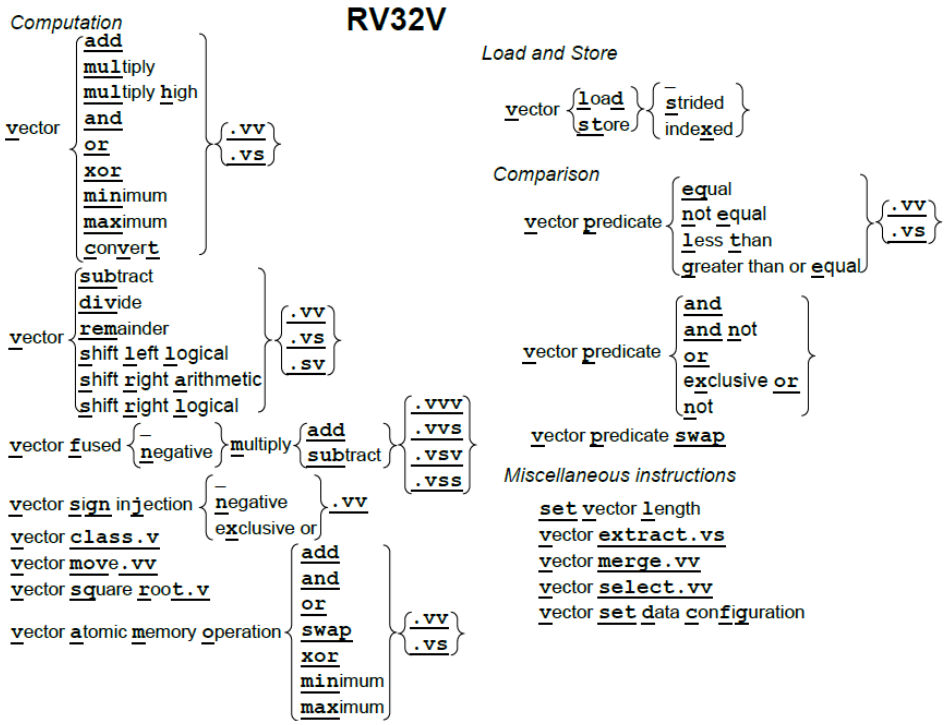


图 8.1：RV32V 的指令图示。由于采用了动态寄存器类型，这个指令图示也可以不加改变地用于第九章的 RV64V

向量计算机从内存中中收集数据并将它们放入长的，顺序的向量寄存器中。在这些向量寄存器上，流水线执行单元可以高效地执行运算。然后，向量架构将结果从向量寄存器中取出，并将其并分散地存回主存。向量寄存器的大小由实现决定，而不是像 SIMD 中那样嵌入操作码中。我们将会看到，*将向量的长度和每个时钟周期可以进行的最大操作数分离，是向量体系结构的关键所在*：向量微架构可以灵活地设计数据并行硬件而不会影响到程序员，程序员可以不用重写代码就享受到长向量带来的好处。此外，向量架构比 SIMD 架构拥有更少的指令数量。而且，与 SIMD 不同，向量架构有着完善的编译器技术。

向量架构比 SIMD 架构更少出现，因此知晓向量 ISA 的读者也更少。因此，本章会比前几章更加具有教程的风格。如果你想深入了解向量架构，请阅读[Hennessy and Patterson 2011]的第 4 章和附录 G。RV32V 还有一些简化了 ISA 的新颖功能。即使你已经熟悉了向量架构，也可能需要阅读我们的进一步解释。

8.2 向量计算指令

图 8.1 是 RV32V 扩展指令集的图形表示。RV32V 的编码尚未最终确定，所以本版不包含通常的指令布局图。

前面章节提到的每一个整数和浮点计算指令基本都有对应的向量版本：图 8.1 中的指令继承了来自 RV32I、RV32M、RV32F、RV32D 和 RV32A 的操作。每个向量指令都有几种类型，具体取决于源操作数是否都是向量（.vv 后缀），或者源操作数包含一个向量和一个标量（.vs 后缀）。一个标量后缀意味着有一个操作数来自 x 或 f 寄存器，另一个来自向量寄存器（v）。比方说，我们的 DAXPY 程序（见第 55 页第五章图 5.7）计算$Y = a \times X + Y$。其中X和Y是向量，a是标量。对于向量-标量操作，rs1 域指定了要访问的标量寄存器。

对诸如减法和除法之类的非对称运算，他们还会使用向量指令的第三种变体。其中第一个操作数是标量，第二个是向量（.sv 后缀）。像$Y = a - X$这样的操作就会使用这种变体。这种变体对于加法和乘法等对称运算来说是多余的，因此这些指令没有.sv 的版本。融合的（fused）乘法-加法指令有三个操作数，因此它们有着最多的向量和标量选项的组合：.vvv、.vvs，.vsv 和.vss。

读者可能会注意到，图 8.1 忽略了向量运算的数据类型和宽度。下一节解释了这么做的原因。

8.3 向量寄存器和动态类型

RV32V 添加了 32 个向量寄存器，它们的名称以 v 开头，但每个向量寄存器的元素个数不同。该数量取决于操作的宽度和专用于向量寄存器的存储大小，而这取决于处理器的设计者。比方说，如果处理器为向量寄存器分配了 4096 个字节，则这足以让这些 32 个向量寄存器中有 16 个 64 位元素，或者 32 个 32 位元素，或者 64 个 16 位元素，或 128 个 8 位元素。

为了在向量 ISA 中保持元素数量的灵活性，向量处理器会计算会*最大向量长度*（mvl），即在给定的容量限制下，向量程序使用这个向量寄存器可以运算的最大向量长度。向量长度寄存器（vl）为特定操作设定了向量中含有的元素数量，这有助于数组维度不是 mvl 的整数倍时的编程。我们将在下面的小节中更详细地演示 mvl，vl 和 8 个谓词寄

存器（vpi）的应用。

RV32V 采用了一种新颖的方法，即将数据类型和长度与向量寄存器而不是与指令操作码相关联。程序在执行向量计算指令之前用它们的数据类型和宽度标记向量寄存器。使用动态寄存器类型会减少向量指令的数量。这一点很重要，因为每个向量指令通常有六个整数版本和三个浮点版本，如图 8.1 所示。我们将在第 8.9 节看到，当我们面对众多的 SIMD 指令时，使用动态寄存器类型的向量架构减少了汇编语言程序员的认知负担以及编译器生成代码的难度。

动态类型的另一个优点是程序可以禁用未使用的向量寄存器。此功能可以将所有的向量存储器分配给已启用的向量寄存器。比如，假设只启用了两个 64 位浮点类型的向量寄存器，处理器有 1024 字节的向量寄存器空间。处理器将这些空间对半分，给每个向量寄存器 512 字节（512/8=64 个元素），因此将 mvl 设置位 64。因此我们可以看到，mvl 是动态的，但它的值由处理器设置，不能由软件直接改变。

源寄存器和目标寄存器决定了操作的类型和大小以及结果，因此动态类型隐含了转换。例如，处理器可以将双精度浮点数的向量乘以单精度标量，而无需先将操作数转换为相同的精度。这个额外的好处减少了向量指令的总数和实际执行的指令的数量。

可以用 vsetdcfg 指令来设置向量寄存器的类型。图 8.2 显示了 RV32V 可用的向量寄存器类型以及 RV64V 的更多类型（见第九章）。RV32V 要求向量浮点运算也有标量版本。因此，要使用 F32 类型，你也必须用到 RV32FV；要使用 F64 类型，你也必须用到 RV32FDV。RV32V 引入了 16 位浮点类型 F16。如果一个实现同时支持 RV32V 和 RV32F，则它必须支持 F16 和 F32 类型。

Type	Floating Point		Signed Integer		Unsigned Integer	
Width	Name	vetype	Name	vetype	Name	vetype
8 bits	–	–	X8	10 100	X8U	11 100
16 bits	F16	01 101	X16	10 101	X16U	11 101
32 bits	F32	01 110	X32	10 110	X32U	11 110
64 bits	F64	01 111	X64	10 111	X64U	11 111

图 8.2：RV32V 向量寄存器类型的编码。字段的最右边三位指示了数据的位宽，左边两位给出其类型。X64 和 U64 仅适用于 RV64V。F16 和 F32 需要 RV32F 扩展，F64 需要 RV32F 和 RV32D。F16 是 IEEE 754-2008 16 位浮点格式（binary16）。将 vetype 设置为 00000 会禁用向量寄存器。（本图基于[Waterman and Asanovic 2017]的表 17.4。）

为了避免上下文切换时间过慢，英特尔尽量避免在原始 MMX SIMD 扩展中添加寄存器。它只是重用现有的浮点寄存器，这意味着没有额外的上下文切换，但程序无法同时出现浮点和多媒体指令。

补充说明：RV32V 可以快速切换上下文

向量架构不如 SIMD 架构受欢迎的一个原因是：大家担心增加大型向量寄存器会延长中断时保存和恢复程序（上下文切换）的时间。动态寄存器类型对此很有帮助。程序员必须告诉处理器正在使用哪些向量寄存器，这意味着处理器需要在上下文切换中仅保存和恢复那些寄存器。RV32V 约定在不使用向量指令的时候禁用所有向量寄存器，这意味着处理器既可以具有向量寄存器的性能优势，又仅会在向量指令执行过程中发生中断时才会带来额外的上下文切换开销。早期的向量架构在发生中断时，不得不忍受保存和恢复全部向量寄存器的最大的上下文切换开销。

每个 load 和 store 指令都有一个 7 位的无符号立即数偏移量。它对于 load 按照目标寄存器的元素类型进行缩放，而对于 store 则按源寄存器缩放。

8.4 向量的 Load 和 Store 操作

向量 Load 和 Store 操作的最简单情况是处理按顺序存储在内存中的一维数组。向量 Load 用以 vld 指令中地址为起始地址的顺序存储的数据来填充向量寄存器。向量寄存器的数据类型确定数据元素的大小，向量长度寄存器 vl 中设置了要取的元素数量。向量 store 执行 vld 的逆操作。

例如，如果 a0 中存有 1024，且 v0 的类型是 X32，则 vld v0, 0(a0)会生成地址 1024，1028，1032，1036，……直到达到由 vl 设置的限制。

对于多维数组，某些访问不是顺序的。如果二维数组以行优先序存储，且对列元素进行顺序访问，则相邻列元素之间的地址差正好是行大小。向量架构通过跨步数据传输来支持 vlds 和 vsts 数据访问。对于 vlds 与 vsts，虽然可以通过将步长设置为元素大小来达到与 vld 和 vst 相同的效果，但 vld 和 vst 保证了所有的访问都是顺序的，这可以提供更高的内存带宽。另一个原因是，对于常见的按单位步长访问，使用 vld 和 vst 可以缩减代码长度，并减少执行的指令数。毕竟使用 vlds 和 vsts 指令来需要指定两个源寄存器，一个给出起始地址，另一个给出以字节为单位的步长，而对于单位步长的访问，多花指令来设置第二个寄存器，无遗是一种浪费。

例如，假设 a0 中的起始地址是地址 1024，且 a1 中行的长度是 64 字节。vlds v0, a0, a1 会将这个地址序列发送到内存：$1024, 1088(1024+1\times 64), 1152(1024+2\times 64), 1216(1024+3\times 64)$，以此类推，直到向量长度寄存器 vl 告诉它停止。返回的数据被顺序写入目标向量寄存器的各个元素。

到目前为止，我们都假设该程序在对密集数组进行操作。为了支持稀疏数组，向量架构用 vldx 和 vstx 提供索引数据传输。这些指令的一个源寄存器是向量寄存器，另一个是标量寄存器。标量寄存器具有稀疏数组的起始地址，向量寄存器的每个元素包含稀疏数组的非零元素的字节索引。

假设 a0 中的起始地址是地址 1024，向量寄存器 v1 在前四个元素中有这些字节索引：16，48，80，160。vldx v0, a0, v1 会将这个地址序列发送到内存：1040（$1024+16$）, $1072(1024+48), 1104(1024+80), 1184(1024+160)$。它将返回的数据顺序写入目标向量寄存器的元素中。

带索引的 load 也称为收集(*gather*)；带索引的 store 通常称为分散(*scatter*)。

以上我们把稀疏数组访问作为索引 Load 和 Store 操作的主要支持目标，但是还有许多其他算法通过索引表来间接访问数据。

8.5 向量操作期间的并行性

虽然简单的向量处理器一次操作一个向量元素，但由于元素操作根据定义是独立的，所以理论上处理器可以同时计算所有这些元素。RV32G 的数据位宽最大位 64 位，而如今的向量处理器通常在每个时钟周期内操作两个、四个或八个 64 位元素。当向量长度不是每个时钟周期执行的元素数量的倍数时，由硬件处理处理这些边缘情况。

与 SIMD 一样，对于较小数据的操作数量是较窄数据的位宽和较宽数据的位宽之比。因此，每个时钟周期计算 4 个 64 位操作的向量处理器通常每个时钟周期可以做 8 个 32 位，16 个 16 位或 32 个 8 位操作。

在 SIMD 中，ISA 架构师在设计过程中决定了每个时钟周期可以并行操作的最大数据数和每个寄存器的元素个数。相比之下，RV32V 处理器设计人员无需更改 ISA 或编译器就

可以选择它们的值，而对于 SIMD，寄存器每增加一倍都会使 SIMD 指令的数量翻倍，并且需要修改 SIMD 编译器。这种隐藏的灵活性意味着相同的 RV32V 程序不用改变，就可以在最简单或最复杂的向量处理器上运行。

8.6 向量运算的条件执行

当一个程序中的绝大部分操作都是用向量指令来实现的，那么这个程序被称为可向量化。Gather，scatter，以及谓词指令让更多的程序变得可向量化了。

一些向量计算包括 if 语句。向量架构不依赖于条件分支，而是包含了一个掩码，这个掩码禁止向量操作作用于某些元素。图 8.1 中的谓词指令在两个向量或向量和标量之间执行条件测试，如果条件成立则在掩码向量的每一个元素中写入一个 1，反之写入 0。（掩码向量必须和向量寄存器有相同的元素个数。）任何后续的向量指令都可以使用这个掩码。第 i 位为 1 表示元素 i 会被向量运算更改，为 0 表示该元素不会由向量运算改变。

RV32V 为掩码向量提供了 8 个向量谓词寄存器（vpi）。vpand，vpandn，vpor，vpxor 和 vpnot 指令在它们之间执行逻辑运算，从而有效处理嵌套条件语句。

RV32V 指定 vp0 或 vp1 作为控制向量操作的掩码。要对所有元素执行一个正常的操作，必须将这两个谓词寄存器中的一个设置为全 1。RV32V 中有一条 vpswap 指令，用于将其他六个谓词寄存器的一个快速交换到 vp0 或 vp1。谓词寄存器也是动态启用的，禁用它们可以快速清除所有谓词寄存器中的值。

例如，假设向量寄存器 v3 中的所有偶数元素都是负整数，所有奇数元素都是正整数。考虑如下的代码：

```
vplt.vs      vp0,v3,x0     # 将 v3 < 0 的掩码位置 1
add.vv,vp0   v0,v1,v2      # 将 v0 的掩码为 1 的对应元素替换为 v1+v2
```

这段代码将把 vp0 中所有的偶数位设为 1，奇数位设为 0，并且将把 v0 中所有的偶数元素替换为 v1 和 v2 中对应元素的和。v0 中的奇数元素不会改变。

8.7 其他向量指令

除了之前提到过的设置向量寄存器数据类型的指令（vsetdcfg），其他指令还有 setvl，它将向量长度寄存器（vl）设置为源操作数和最大向量长度（mvl）中的较小值。选择较小值的原因是，在循环中我们需要判断这些向量代码到底是可以按最大向量长度（mvl）运行，还是要以一个较小值运行，从而能处理循环尾部剩下的元素。因此，为了处理循环尾部的元素，每次循环迭代都执行 setvl。

RV32V 中还有三条指令可以操作向量寄存器中的元素。

向量选择（vselect）按第二个源向量（索引向量）指定的元素位置，从第一个源数据向量中取得元素，从而生成一个新的结果向量：

```
# vindices 存有 0 到 mvl-1 的值，它们用来从 vsrc 中选取元素
vselect vedst, vsrc, vindices
```

因此，如果 v2 的前四个元素是 8、0、4、2，那么 vselect v0, v1, v2 将用 v1 的第 8 个元素替换 v0 的第 0 个元素；v1 的第 0 个元素替换 v0 的第 1 个元素；v1 的第 4 个元素替换 v0 的第 2 个元素；v1 的第 2 个元素替换 v0 的第 3 个元素。

向量合并（vmerge）类似于向量选择，但它用向量谓词寄存器来选择源向量中要用到元素。新的结果向量由根据谓词寄存器从两个源寄存器之一取得元素产生。若谓词向量寄

存器元素为 0，则新元素来自 vsrc1；如果为 1，则来自 vsrc2。

```
# vpo 的第 i 位决定 vdest 中新元素 i 来自 vsrc1（若第 i 位是 0）
# 还是 vsrc2（第 i 位为 1）
vmerge,vpo vdest, vsrc1, vsrc2
```

因此，如果 vpo 的前四个元素是 1、0、0、1，v1 的前四个元素是 1、2、3、4，v2 的前四个元素是 10、20、30、40，那么 vmerge,vpo v0, v1, v2 将把 v0 的前四个元素变为 10、2、3、40。

向量提取指令从一个向量的中间开始取元素，并将它们放在第二个向量寄存器的开头：

```
# start 是一个标量寄存器，其中存储着从 vsrc 中取元素的起始位置
vextract vdest, vsrc, start
```

例如，如果向量长度 vl 是 64，而 a0 的值是 32，那么 vextract v0,v1,a0 会把 v1 中的后 32 个元素复制到 v0 的前三十二个元素。

对于任意的二元结合运算符，可以利用 vextract 指令以递归减半的方法进行缩减运算。例如，要对向量寄存器的所有元素求和，可以用 vextract 将向量的后半部分复制到另一个向量寄存器的前半部分，这就将向量长度缩短了一半。接下来，将这两个向量寄存器加到一起，并将它们的和作为新一轮递归的操作数，直到向量长度减少到 1。此时第零个元素中的结果就是原向量寄存器中所有元素的和。

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0              # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0            # vl = t0 = min(mvl, n)
  c:  vld    v0, a1            # load vector x
 10:  slli   t1, t0, 3         # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2            # load vector y
 18:  add    a1, a1, t1        # increment C pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1   # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0        # n -= vl (t0)
 24:  vst    v1, a2            # store Y
 28:  add    a2, a2, t1        # increment C pointer to y by vl*8
 2c:  bnez   a0, loop          # repeat if n != 0
 30:  ret                      # return
```

图 8.3：图 5.7 中 DAXPY 程序的 RV32V 代码。没有出现机器语言是因为 RV32V 的操作码还未定义。

8.8 例子：用 RV32V 写成的 DAXPY 程序

RISC-V 中的 V 也代表向量。RISC-V 架构师在向量架构方面拥有丰富的经验，并且对 SIMD 在微处理器中的主导地位感到沮丧。因此，V 不仅代表这是第五个伯克利 RISC 项目，也意味着这个 ISA 会突出向量。

图 8.3 显示了用 RV32V 汇编写成的 DAXPY 程序（见第五章第 55 页图 5.7），我们一次解释一个步骤。

RV32V DAXPY 程序做的第一件事是启用这个函数需要的向量寄存器。它只需要两个向量寄存器保存 x 和 y 的部分元素，这些元素一个个都是 8 字节宽的双精度浮点数。第一条指令生成一个常量，第二条指令将它写入配置向量寄存器的控制状态寄存器（vcfgd），

从而获得两个 F64 类型的寄存器（见图 8.2）。根据定义，硬件按数字顺序分配配置好的寄存器，这样便有了 v0 和 v1。

假设我们的 RV32V 处理器由 1024 字节的空间专门用于向量寄存器。硬件平均地给这两个双精度浮点型（8 字节）的向量寄存器分配空间。每个向量寄存器有$512/8 = 64$个元素，因此处理器将此函数的最大向量长度（mvl）设置为 64。

循环中的第一条指令为接下来的向量指令设置向量长度。setvl 指令把 mvl 和 n 中的小值写入 vl 和 t0。其中的深刻原因是，如果循环的迭代次数大于 n，那么这段代码最快可以一次处理 64 个值，所以把 mvl 的值写入 vl。如果 n 比 mvl 小，那么我们的读写不能超出 x 和 y 的范围，所以我们应该在循环最后一次迭代中只计算最后剩下的 n 个元素。setvl 还写入 t0，用于保存 vl 的值，在地址为 10 的循环控制变量中会用到。

没有 setvl 的向量架构具有额外的类似*露天采矿*（意为降低效率）的代码，用于将 vl 设置为循环的最后 n 个元素，并检查 n 是否为

地址 c 处的指令 vld 是一个向量 load 操作，按照标量寄存器 a1 中存储的变量 x 的地址从 x 中取值。它把 x 的 vl 个元素从内存传输到 v0。下面的移位指令 slli 将向量长度乘以数据的宽度（8 字节），以便稍后用于递增指向 x 和 y 的指针。

地址 14 处的指令（vld）将来自内存的 vl 个元素 load 到 v1 中，接下来的一条指令（add）将指向 x 的指针进行了递增。

地址 1c 处的指令是最重要的部分。vfmadd 将 x（v0）中的 vl 个元素乘以标量 a（f0）并将每个乘积加上 y（v1）中的 vl 个元素，最后将这 vl 个和存回 y（v1）。

剩下的就是将结果存到内存中以及一些必须的循环开销。在地址 20 处的指令（sub）将 n（a0）的减去 vl，以记录在本次迭代中完成的操作数。接下来的一条指令（vst）将 vl 个结果写入内存中 y 数组中。地址 28 处的指令（add）将指向 y 数组的指针递增。接下来的指令判断 n（a0）是否为 0，若不是则继续循环，反之执行最后一条 ret 指令返回调用点。

向量架构的强大之处在于，这个包含 10 条指令的循环的每次迭代都会进行$3 \times 64 = 192$次访存操作和$2 \times 64 = 128$个浮点乘加（假设 n 至少为 64）。这意味着每条指令平均有 19 次访存和 13 次运算。我们将在下一节看到，SIMD 的这些数据要差一个数量级。

8.9 RV32V，MIPS-32 MSA SIMD 和 x86-32 AVX SIMD 的比较

ARM-32 有一个名为 NEON 的 SIMD 扩展，但它不支持双精度浮点指令，所以它对 DAXPY 没有帮助。

我们即将看到 SIMD 和向量架构执行 DAXPY 程序的对比。如果你换一种角度来看，也可以把 SIMD 视为有着短向量寄存器（8 个 8 位“元素”）的受限向量架构，但它没有向量长度寄存器，也没有跨步或索引数据传输。

MIPS SIMD 第 83 页的图 8.5 显示了 DAXPY 程序的 MIPS SIMD 架构（MSA）版本。由于 MSA 寄存器为 128 位宽，所以每个 MSA SIMD 指令可以操作两个双精度浮点数。

这种簿记代码被认为是向量架构中露天采矿代码的一部分。如图 8.5 的标题所示，向量长度寄存器 vl 使得这样的 SIMD 簿记代码对于 RV32V 没有实际意义。传统的向量架构需要额外的代码来处理 n = 0 的极端情况。RV32V 只是在 n = 0 时使向量指令像 nops 一样。

与 RV32V 不同，由于没有向量长度寄存器，MSA 需要额外的指令来检查 n 的值是否有问题。当 n 为奇数时，计算单个浮点数的乘-加运算需要额外的代码，因为 MSA 必须对成对的操作数进行操作。该代码位于图 8.5 的地址 3c 到 4c 处。尽管概率不大，但 n 也有可能为 0。在这种情况下，地址为 10 处的分支将跳过主计算循环。

如果没有在循环附近执行分支跳转，则地址为 18 处的指令（splati.d）把 a 的值同时放入 SIMD 寄存器 w2 的两半中。在 SIMD 中，要加一个标量数据，我们需要将其复制拓宽到与 SIMD 寄存器等宽。

在循环内部，地址为 1c 处的 ld.d 指令将 y 的两个元素 load 到 SIMD 寄存器 w0 中，然后将指向 y 的指针进行递增。然后它将 x 的两个元素 load 到 SIMD 寄存器 w1 中。接下

来地址 28 处的指令执行将指向 x 的指针进行递增，紧接在后面的是地址为 2c 处的最重要的乘加指令。

循环结束时的分支指令（带延迟槽）判断指向 y 的指针是否已经超出了 y 的范围。如果没有，循环继续。地址 34 处的延迟槽中的 SIMD store 指令将结果写入 y 的两个元素。

主循环终止后，代码检查 n 是否是奇数。若 n 是奇数，用第五章的标量指令执行最后一次乘加操作。最后一条指令返回到调用点。

MIPS MSA DAXPY 代码核心的循环部分包含了 7 条指令，执行了 6 次双精度访存操作和 4 次浮点乘加。平均每个指令大约有 1 个访存和 0.5 个运算操作。

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instructions (static)	22	29	13
Bytes (static)	88	92	52
Instructions per Main Loop	7	6	10
Results per Main Loop	2	4	64
Instructions (dynamic, n=1000)	3511	1517	163

图 8.4：向量 ISA 的 DAXPY 指令数和代码大小。他列出了指令总数（静态），代码大小，每个循环的指令数和运算结果数，以及执行的指令数（n = 1000）。带 MSA 的 microMIPS 将代码大小缩减到 64 字节，RV32FDCV 将代码缩减到 40 字节。

x86 SIMD 在 84 页的图 8.6 的代码中我们可以看到，Intel 公司经历了多代 SIMD 扩展。SSE 扩展到了 128 位 SIMD，带来了 xmm 寄存器和可以使用这些寄存器的指令；AVX 的一部分带来了 256 位 SIMD，以及 ymm 寄存器及其指令。

地址 0 到 25 的第一组指令从内存中 load 变量，在 256 位 ymm 寄存器中创建 a 的四个副本，并在进入主循环之前进行测试，以确保 n 至少为 4。这用到了两条 SSE 指令和一条 AVX 指令。（图 8.6 的标题中有更详细的解释）

主循环是 DAXPY 计算的核心。地址 27 处的 AVX 指令 vmovapd 将 x 的 4 个元素 load 到 ymm0 中。地址 2c 处的 AVX 指令 vfmadd213pd 将 a（ymm2）乘以 x（ymm0）的 4 个元素的 4 个副本，加上 y 的四个元素（在内存中地址为 ecx+edx*8 处），并将 4 个和放入 ymm0。接下来地址 32 处的 AVX 指令 vmovapd 将 4 个结果存储到变量 y 中。随后的三条指令执行计数器的递增操作并在需要的时候重复循环。

与 MIPS MSA 的情况一样，地址 3e 和 57 之间的“边缘”代码处理了 n 不是 4 的倍数的情况。它用到了三个 SSE 指令。

x86-32 AVX2 DAXPY 代码中主循环的 6 条指令执行了 12 次双精度访存和 8 次浮点的乘法和加法操作。这样每条指令平均有约 2 次访存和 1 次运算。

补充说明：Illiac IV 最先显现了 SIMD 的编译复杂性

凭借 64 个并行的 64 位浮点单元（FPU），在摩尔定律发布之前，Illiac IV 计划拥有超过 100 万个逻辑门。它的架构师最初预测它每秒可以进行 10 亿次浮点运算（1000MFLOPS），但它的实际最好性能只有 15MFLOPS。它的成本从 1966 年估计的 800 万美元上升到了 1972 年的 3100 万美元（尽管只建造了计划的 256 个 FPU 中的 64 个）。该项目于 1965 年启动，但直到 1976 年（Cray-1 发布的那一年）才开始发挥实际作用。它可能是最臭名昭著的超级计算机，成为了十大工程灾难之一[Falk 1976]。

8.10 结束语

如果代码能向量化，最好的架构就是向量架构。

——Jim Smith 于 1994 年在国际计算机体系结构研讨会上的主旨演讲

图 8.4 总结了 RV32IFDV，MIPS-32 MSA 和 x86-32 AVX2 的 DAXPY 程序中的指令数和字节数。SIMD 架构程序中用于计算的代码比用于循环控制的代码要少不少。MIPS-32 MSA 和 x86-32 AVX2 代码中的三分之二到四分之三是 SIMD 开销：这些额外的代码要么是在为主 SIMD 循环准备数据，要么是在 n 不是 SIMD 寄存器中浮点数个数的倍数时处理那些边缘元素。

图 8.3 中的 RV32V 代码不需要这样的循环控制代码，因此它的指令数量少了一半。与 SIMD 不同，RV32V 有一个向量长度寄存器，使得不论 n 为何值，向量指令都可以工作。你可能会觉得 n 为 0 时 RV32V 会出现问题。实际上它不会，因为 RV32V 中的向量指令在 vl=0 时不会做出任何改变。

但是，SIMD 和向量处理之间的最为显著的区别不在于代码的长短。SIMD 执行的指令数比 RV32V 多 10 到 20 倍，因为每个 SIMD 循环在向量模式下只操作 2 到 4 个元素，而不是 RV32V 的 64 个元素。额外的取指和译码意味着在执行相同任务时要耗费更多的能量。

将图 8.4 中的结果与第五章中第 29 页的图 5.8 中的 DAXPY 的标量版本进行比较，我们发现 SIMD 大概使得代码的指令数和字节数加倍，但主循环的大小相同。执行的动态指令的数量以 2 或 4 的因子减少，这取决于 SIMD 寄存器的宽度。然而，RV32V 的向量代码大小变为原来的 1.2 倍（主循环 1.4 倍），但动态指令数是原来的 1/43！

即使动态指令的数量差别很大，但在我们看来，这并不是 SIMD 和向量架构的最主要的差异。没有向量长度寄存器会让指令数和循环控制代码暴增。像 MIPS-32 和 x86-32 这些遵循增量主义的 ISA 必须每次在将 SIMD 寄存器宽度翻倍时，都复制所有那些为较窄的 SIMD 寄存器定义的指令。于是不出意外地，在许多代 SIMD ISA 的传承中一共创造了数百条 MIPS-32 和 x86-32 指令，而且将来还会有数以百计的新指令出现。汇编语言程序员一定因这种粗暴的 ISA 演变方式而承担了难以承受的认知负担。像 vfmadd213pd 这样的指令，谁能记住它的含义并知道什么时候要用它？

相比之下，RV32V 代码不受向量寄存器的可用存储空间的大小影响。如果向量内存变大，不仅 RV32V 不会改变，而且你甚至不用重新编译。处理器提供了最大向量长度 mvl 的值，因此无论处理器将用于向量的存储空间从 1024 字节提升到了 4096 字节，还是将其降低到 256 字节，图 8.3 中的代码都不受影响。

不同于 SIMD 中由 ISA 指示所需的硬件，而且更改 ISA 意味着更改编译器那样，RV32V ISA 允许处理器设计人员为其应用分配合适资源用于数据并行，而不必影响程序员或编译器。可以说 SIMD 违反了第一章中将 ISA 架构和实现分离开来的 ISA 设计原则。

我们认为 RV32V 的模块化向量实现对比 ARM-32、MIPS-32 和 x86-32 的增量式 SIMD 架构在成本-能耗-性能、复杂度和编程简易性等方面的极大优势，可能是选用 RISC-V 的最有说服力的论据。

8.11 扩展阅读

H. Falk. What went wrong V: Reaching for a gigaflop: The fate of the famed Illiac IV was shaped by both research brilliance and real-world disasters. *IEEE spectrum*, 13(10):65–70, 1976.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

注记

http://parlab.eecs.berkeley.edu

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
00000000 <daxpy>:
  0: 2405fffe  li        a1,-2
  4: 00852824  and       a1,a0,a1         # a1 = floor(n/2)*2 (mask bit 0)
  8: 000540c0  sll       t0,a1,0x3        # t0 = byte address of a1
  c: 00e81821  addu      v1,a3,t0         # v1 = &y[a1]
 10: 10e30009  beq       a3,v1,38         # if y==&y[a1] goto Fringe (t0==0 so n is 0 | 1)
 14: 00c01025  move      v0,a2            # (delay slot) v0 = &x[0]
 18: 78786899  splati.d $w2,$w13[0]       # w2 = fill SIMD register with copies of a
Loop:
 1c: 78003823  ld.d      $w0,0(a3)         # w0 = 2 elements of y
 20: 24e70010  addiu     a3,a3,16          # increment C pointer to y by 2 Fl.Pt. numbers
 24: 78001063  ld.d      $w1,0(v0)         # w1 = 2 elements of x
 28: 24420010  addiu     v0,v0,16          # increment C pointer to x by 2 Fl.Pt. numbers
 2c: 7922081b  fmadd.d   $w0,$w1,$w2       # w0 = w0 + w1 * w2
 30: 1467fffa  bne       v1,a3,1c          # if (end of y != ptr to y) go to Loop
 34: 7bfe3827  st.d      $w0,-16(a3)       # (delay slot) store 2 elts of y
Fringe:
 38: 10a40005  beq       a1,a0,50          # if (n is even) goto Done
 3c: 00c83021  addu      a2,a2,t0          # (delay slot) a2 = &x[n-1]
 40: d4610000  ldc1      $f1,0(v1)         # f1 = y[n-1]
 44: d4c00000  ldc1      $f0,0(a2)         # f0 = x[n-1]
 48: 4c206b61  madd.d    $f13,$f1,$f13,$f0  # f13 = f1 + f0 * f13 (muladd if n is odd)
 4c: f46d0000  sdc1      $f13,0(v1)        # y[n-1] = f13 (store odd result)
Done:
 50: 03e00008  jr       ra                 # return
 54: 00000000  nop                         # (delay slot)
```

图 8.5：图 5.7 中 DAXPY 的 MIPS-32 MSA 代码。将此代码与图 8.3 中的 RV32V 代码进行比较时，SIMD 的循环控制开销显而易见。MIPS MSA 代码的第一部分（地址 0 到 18）复制了 SIMD 寄存器中的标量变量 a，并在进入主循环之前执行确保 n 至少为 2 的检查。当 n 不是 2 的倍数时，MIPS MSA 代码的第三部分（地址 38 到 4c）处理了这种边界情况情况。在 RV32V 中不需要这样的代码，因为向量长度寄存器 vl 和 setvl 指令使得该循环的代码适用于 n 的所有值，不论是奇数还是偶数。

```
# eax is i, n is esi, a is xmm1, pointer to x[0] is ebx, pointer to y[0] is ecx
 00000000 <daxpy>:
    0: 56                  push   esi
    1: 53                  push   ebx
    2: 8b 74 24 0c         mov    esi,[esp+0xc]   # esi = n
    6: 8b 5c 24 18         mov    ebx,[esp+0x18]  # ebx = x
    a: c5 fb 10 4c 24 10   vmovsd xmm1,[esp+0x10] # xmm1 = a
   10: 8b 4c 24 1c         mov    ecx,[esp+0x1c]  # ecx = y
   14: c5 fb 12 d1         vmovddup xmm2,xmm1     # xmm2 = {a,a}
   18: 89 f0               mov    eax,esi
   1a: 83 e0 fc            and    eax,0xfffffffc  # eax = floor(n/4)*4
   1d: c4 e3 6d 18 d2 01   vinsertf128 ymm2,ymm2,xmm2,0x1 # ymm2 = {a,a,a,a}
   23: 74 19               je     3e              # if n < 4 goto Fringe
   25: 31 d2               xor    edx,edx         # edx = 0
 Loop:
   27: c5 fd 28 04 d3      vmovapd ymm0,[ebx+edx*8] # load 4 elements of x
   2c: c4 e2 ed a8 04 d1   vfmadd213pd ymm0,ymm2,[ecx+edx*8] # 4 mul adds
   32: c5 fd 29 04 d1      vmovapd [ecx+edx*8],ymm0 # store into 4 elements of y
   37: 83 c2 04            add    edx,0x4
   3a: 39 c2               cmp    edx,eax          # compare to n
   3c: 72 e9               jb     27               # repeat loop if < n
 Fringe:
   3e: 39 c6               cmp    esi,eax          # any fringe elements?
   40: 76 17               jbe    59               # if (n mod 4) == 0 goto Done
 FringeLoop:
   42: c5 fb 10 04 c3      vmovsd xmm0,[ebx+eax*8] # load element of x
   47: c4 e2 f1 a9 04 c1   vfmadd213sd xmm0,xmm1,[ecx+eax*8] # 1 mul add
   4d: c5 fb 11 04 c1      vmovsd [ecx+eax*8],xmm0 # store into element of y
   52: 83 c0 01            add    eax,0x1          # increment Fringe count
   55: 39 c6               cmp    esi,eax          # compare Loop and Fringe counts
   57: 75 e9               jne    42 <daxpy+0x42>  # repeat FringeLoop if != 0
 Done:
   59: 5b                  pop    ebx              # function epilogue
   5a: 5e                  pop    esi
   5b: c3                  ret
```

图 8.6：图 5.7 中 DAXPY 的 x86-32 AVX2 代码。地址 a 处的 SSE 指令 vmovsd **把 a load 到 128 位** xmm1 **寄存器的一半。地址 14 处的 SSE 指令** vmovddup **将 a 复制到** xmm1 **的全部两半，以用于接下来的 SIMD 计算。地址 1d 处的 AVX 指令** vinsertf128 **从** xmm1 **中的 a 的两个副本，在** ymm2 **中生成 a 的四个副本。地址 42 到 4d 的三个 AVX 指令（**vmovsd，vfmadd213sd，vmovsd**）处理mod（n，4）≠ 0 的情况。它们以一次一个元素的方式执行 DAXPY 操作，循环在这个函数正好进行了 n 次乘-加操作的时候停止。再提一次，RV32V 不需要这样的代码，因为向量长度寄存器** vl **和** setvl **指令使得那些循环代码适用于 n 为任意值的情况。**

第九章 RV64：64 位地址指令

在计算机设计中只能出现一个错误是难以恢复的——没有足够的地址位用于存储器寻址和存储器管理。

——C. Gordon Bell, 1976

C. Gordon Bell（1934-）是当时最受欢迎的两种小型机架构的首席架构师之一：1970 年宣布的数字设备公司 PDP-11（16 位地址）及其七年后的继任者，数字设备公司 32 位地址 VAX-11（虚拟地址扩展）。

9.1 导言

图 9.1 至 9.4 是 RV32G 指令集的 64 位版本 RV64G 指令集的图示。由图可见，要切换到 64 位 ISA，ISA 只添加了少数指令。指令集只添加了 32 位指令对应的字(word)，双字(doubleword)和长整数(long)版本的指令，并将所有寄存器（包括 PC）扩展为 64 位。因此，RV64I 中的 sub 操作的是两个 64 位数字而不是 RV32I 中的 32 位数字。RV64 很接近 RV32 但实际上又有所不同;它添加了少量指令同时基础指令做的事情与 RV32 中稍有不同。

例如，图 9.8 中 RV64I 版本的插入排序与第 2 章第 27 页的图 2.8 中 RV32I 版本的插入排序非常相似。它们指令数量和大小都相同。唯一的变化是加载和存储字指令变为加载并存储双字，地址增量从对应字的 4（4 字节）变为对应双字的 8（8 字节）。图 9.5 列出了图 9.1 到 9.4 中的 RV64GC 指令的操作码。

尽管 RV64I 有 64 位地址且默认数据大小为 64 位，32 位字仍然是程序中的有效数据类型。因此，RV64I 需要支持字，就像 RV32I 需要支持字节和半字一样。更具体地说，由于寄存器现在是 64 位宽，RV64I 添加字版本的加法和减法指令：addw，addiw，subw。这些指令将计算结果截断为 32 位，结果符号扩展后再写入目标寄存器。 RV64I 也包括字版本的移位指令（sllw，slliw，srlw，srliw，sraw，sraiw），以获得 32 位移位结果而不是 64 位移位结果。要进行 64 位数据传输，RV64 提供了加载和存储双字指令：ld，sd。最后，就像 RV32I 中有无符号版本的加载单字节和加载半字的指令，RV64I 也有一个无符号版本的加载字：lwu。

出于类似的原因，RV64 需要添加字版本的乘法，除法和取余指令：mulw，divw，divuw，remw，remuw。为了支持对单字及双字的同步操作，RV64A 为其所有的 11 条指令都添加了双字版本。

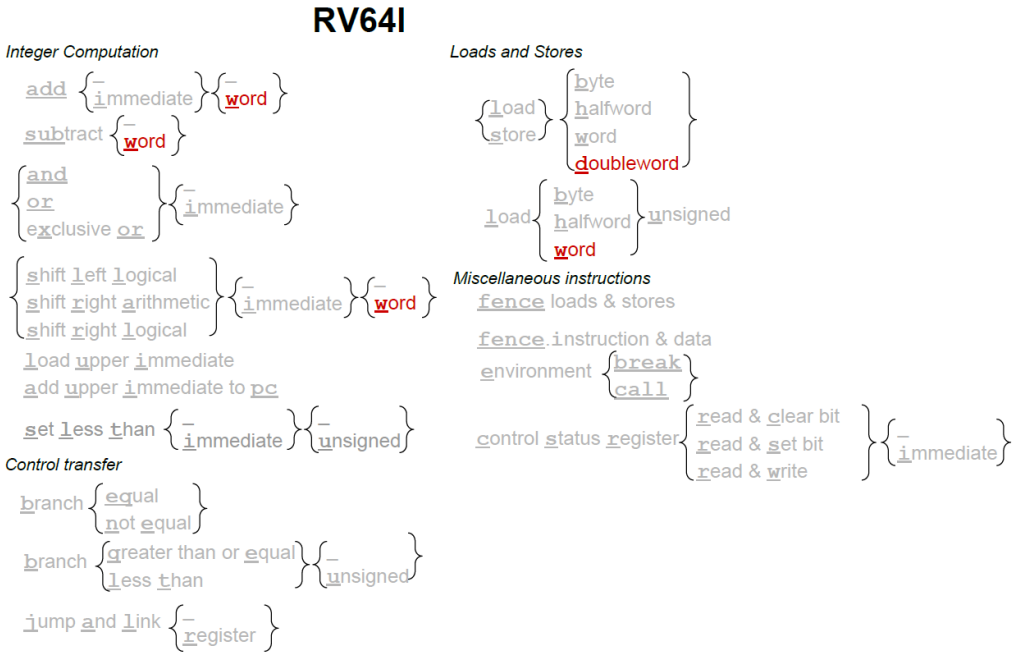


图 9.1：RV64I 指令图示。带下划线的字母从左到右连起来构成 RV64I 指令。灰色部分是扩展到 64 位寄存器的旧 RV64I 指令，而暗（红色）部分是 RV64I 的新指令。

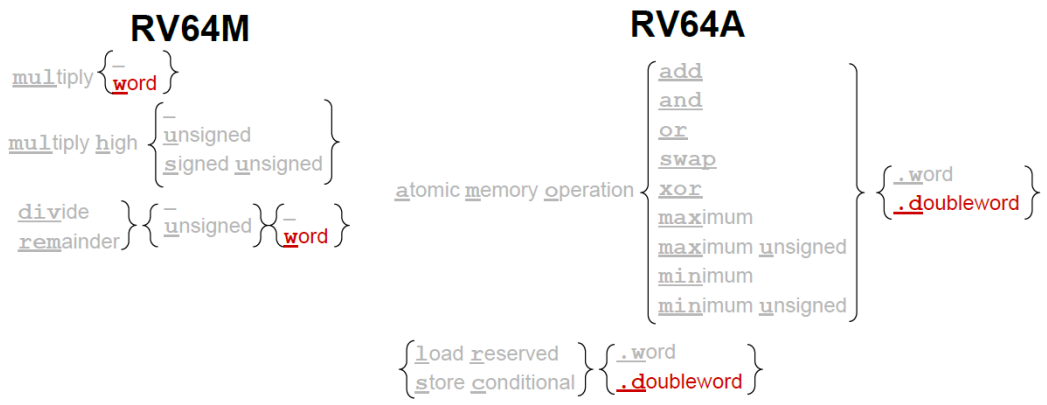


图 9.2：RV64M 和 RV64A 指令图示

RV64F and RV64D

Floating-Point Computation

float { add / subtract / multiply / divide / square root / minimum / maximum } { .single / .double }

float { – / negative } multiply { add / subtract } { .single / .double }

float move to { .single / .double } from .x register

float move to .x register from { .single / .double }

Comparison

compare float { equals / less than / less than or equals } { .single / .double }

Load and Store

float { load / store } { word / doubleword }

Conversion

float convert to { .single / .double } from { .word / .long } { – / unsigned }

float convert to { .word / .long } { – / unsigned } from { .single / .double }

float convert to .single from .double

float convert to .double from .single

Other instructions

float sign injection { – / negative / exclusive or } { .single / .double }

float classify { .single / .double }

图 9.3：RV64F 和 RV64D 指令图

RV64C

Integer Computation

c.add { – / immediate } { – / word }

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract { – / word }

c. { shift left logical / shift right arithmetic / shift right logical } immediate

c.and { – / immediate }

c.or

c.move

c.exclusive or

c.load { – / upper } immediate

Loads and Stores

c. { – / ~~float~~ } { load / store } { ~~word~~ / doubleword } { – / using stack pointer }

c.float { load / store } doubleword { – / using stack pointer }

Control transfer

c.branch { equal / not equal } to zero

c.jump { – / ~~and link~~ }

c.jump { – / and link } register

Other instructions

c.environment break

图 9.4L：RV64C 指令图

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	110	rd	0000011	I lwu
imm[11:0]		rs1	011	rd	0000011	I ld
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	S sd
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
imm[11:0]		rs1	000	rd	0011011	I addiw
0000000	shamt	rs1	001	rd	0011011	I slliw
0000000	shamt	rs1	101	rd	0011011	I srliw
0100000	shamt	rs1	101	rd	0011011	I sraiw
0000000	rs2	rs1	000	rd	0111011	R addw
0100000	rs2	rs1	000	rd	0111011	R subw
0000000	rs2	rs1	001	rd	0111011	R sllw
0000000	rs2	rs1	101	rd	0111011	R srlw
0100000	rs2	rs1	101	rd	0111011	R sraw

RV64M Satndard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	R mulw
0000001	rs2	rs1	100	rd	0111011	R divw
0000001	rs2	rs1	101	rd	0111011	R divuw
0000001	rs2	rs1	110	rd	0111011	R remw
0000001	rs2	rs1	111	rd	0111011	R remuw

RV64A Satndard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	R lr.d
00011	aq	rl	rs2	rs1	011	rd	0101111	R sc.d
00001	aq	rl	rs2	rs1	011	rd	0101111	R amoswap..d
00000	aq	rl	rs2	rs1	011	rd	0101111	R amoadd.d
00100	aq	rl	rs2	rs1	011	rd	0101111	R amoxor.d
01100	aq	rl	rs2	rs1	011	rd	0101111	R amoand.d
01000	aq	rl	rs2	rs1	011	rd	0101111	R amoor.d
10000	aq	rl	rs2	rs1	011	rd	0101111	R amomin.d
10100	aq	rl	rs2	rs1	011	rd	0101111	R amomax.d
11000	aq	rl	rs2	rs1	011	rd	0101111	R amominu.d
11100	aq	rl	rs2	rs1	011	rd	0101111	R amomaxu.d

RV64F Satndard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	R fcvt.l.s
1100000	00011	rs1	rm	rd	1010011	R fcvt.lu.s
1101000	00010	rs1	rm	rd	1010011	R fcvt.s.l
1101000	00011	rs1	rm	rd	1010011	R fcvt.s.lu

RV64D Satndard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	R fcvt.l.d
1100001	00011	rs1	rm	rd	1010011	R fcvt.lu.d
1110001	00000	rs1	000	rd	1010011	R fmv.x.d
1101001	00010	rs1	rm	rd	1010011	R fcvt.d.l
1101001	00011	rs1	rm	rd	1010011	R fcvt.d.lu
1111001	00000	rs1	000	rd	1010011	R fmv.d.x

图 9.5：RV64 基本指令和可选扩展指令的的操作码表。这张图包含了指令布局，操作码，格式类型和名称(基于[Waterman and Asanovic 2017]的表 19.2）。

RV64F 和 RV64D 添加了整数双字转换指令，并称它们为长整数，以避免与双精度浮点数据混淆：fcvt.l.s，fcvt.l.d，fcvt.lu.s，fcvt.lu.d，fcvt.s.l，fcvt.s.lu，fcvt.d.l，fcvt.d.lu. 由于整数 x 寄存器现在是 64 位宽，它们现在可以保存双精度浮点数据，因此 RV64D 增加了两个浮点指令：fmv.x.w 和 fmv.w.x.

RV64 和 RV32 之间基本是超集关系，但是有一个例外是压缩指令。 RV64C 取代了一些 RV32C 指令，因为其他一些指令对于 64 位地址可以取得更好的代码压缩效果。RV64C 放弃了压缩跳转并链接（c.jal）和整数和浮点加载和存储字指令（c.lw，c.sw，c.lwsp，c.swsp，c.flw，c.fsw，c.flwsp 和 c.fswsp）。在他们的位置，RV64C 添加了更受欢迎的字加减指令（c.addw，c.addiw，c.subw）以及加载和存储双字指令（c.ld，c.sd，c.ldsp，c.sdsp）。

补充说明：RV64 ABI 包括 lp64，lp64f 和 lp64d
lp64 表示 C 语言中的长整型以及指针类型为 64 位；整型仍然是 32 位。与 RV32（见第 3 章）相同，后缀 f 和 d 表示如何传递浮点参数。

补充说明：RV64V 没有指令图示
是因为有动态寄存器类型的存在，它与 RV32V 的完全一致。唯一的变化是第 75 页的图 8.2 中的 X64 和 X64U 动态寄存器类型仅在 RV64V 中出现，RV32V 并不支持。

9.2 使用插入排序来比较 RV64 与其他 64 位 ISA

正如 Gordon Bell 在本章开头所说，一个架构致命的缺陷是用光了地址位。随着程序使用的内存大小逐渐逼近 32 位地址空间的极限，不同指令集的架构师开始了设计他们指令集的 64 位地址版本[Mashey 2009]。

最早的是 MIPS，在 1991 年，它将所有寄存器以及程序计数器从 32 扩展至 64 位并添加了新的 64 位版本的 MIPS-32 指令。 MIPS-64 汇编语言指令都以字母“d”开头，例如 daddu 或 dsll（参见图 9.10）。程序员可以在同一个程序中混合使用 MIPS-32 和 MIPS-64 指令。MIPS-64 删除了 MIPS-32 中的加载延迟槽（流水线在侦测到写后读相关时会停止）。

十年之后，是 x86-32 指令集也迎来了 64 位。架构师们在拓展地址空间的同时，也借机在 x86-64 中进行了一系列改进：

- 整数寄存器的数量从 8 增加到 16（r8-r15）；
- 将 SIMD 寄存器的数量从 8 增加到 16（xmm8-xmm15）并且添加了 PC 相关数据寻址，以更好地支持与位置无关的代码。
- 添加了 PC 相关数据寻址，以更好地支持与位置无关的代码。
  这些改进部分缓和了 x86-32 指令集长久以来的一些弊端。

通过比较插入排序的 x86-32 版本（第 2 章第 30 页上图 2.11 中）和 x86-64 版本（图 9.11 中）的指令，我们可以发现 x86-64 指令集的优势。新的 64 位 ISA 将所有变量分配在寄存器中，而不是像 x86-32 一样，要将多个变量保存到内存中，这将指令的数量从 20 条减少到了 15 条。尽管 64 位代码指令数量比 32 位少，但是代码大小实际上要大一个字节，从 45 变成

了 46 字节。原因是为了挤进新的操作码以便操作更多的寄存器，x86-64 添加了一个前缀字

ISA	ARM-64	MIPS-64	x86-64	RV64I	RV64I+RV64C
Instructions	16	24	15	19	19
Bytes	64	96	46	76	52

图 9.6：四个 ISA 的插入排序的指令数和代码大小。 ARM Thumb-2 和 microMIPS 是 32 位地址 ISA，因此不适用于 ARM-64 和 MIPS-64。

节来识别新指令。从 x86-32 到 x86-64 平均指令长度增长了。

又过了十年，ARM 也遇到了同样的地址问题。但是他们没有像 x86-64 那样，把旧的 ISA 扩展到支持 64 位地址。他们利用这个机会发明了一个全新的 ISA。从头设计一个新 ISA，使得他们不必继承 ARM-32 的许多尴尬特性，他们重新设计了一个现代 ISA：

- 将整数寄存器的数量从 15 增加到 31;
- 从寄存器组中删除 PC;
- 为大多数指令提供硬连线到零的寄存器（r31）;
- 与 ARM-32 不同，ARM-64 的所有数据寻址模式都适用于所有数据大小和类型;
- ARM-64 去除了 ARM-32 的加载存储多个数据的指令
- ARM-64 去除了 ARM-32 指令的条件执行选项。

ARM-32 的一些弱点依然存在于 ARM-64 指令集中：分支指令使用的条件码，指令中源和目标寄存器字段并不固定，条件移动指令，复杂寻址模式，不一致的性能计数器，以及只支持 32 位长度的指令。另外 ARM-64 无法切换到 Thumb-2 ISA，因为 Thumb-2 仅适用于 32 位地址。

与 RISC-V 不同，ARM 决定采用最大主义的方法来设计 ISA。虽然 ISA 比 ARM-32 更好，但它也更大。例如，它有超过 1000 条指令并且 ARM-64 手册长 3185 页[ARM 2015]。而且，它的指令数仍然在增长。自公布几年以来，ARM-64 已经经历了三次扩展。

图 9.9 中插入排序的 ARM-64 代码看起来更接近 RV64I 代码或 x86-64 代码，而不太像 ARM-32 代码。例如，因为有 31 个寄存器可用，就没有必要从堆栈中保存和恢复寄存器。而且由于 PC 不再存放于通用寄存器中，ARM-64 单独增加了一条返回指令。

图 9.6 总结了插入排序在不同 ISA 下的指令数和字节数。图 9.8 到 9.11 显示了 RV64I，ARM-64，MIPS-64 和 x86-64 的代码。这四段代码注释中括号内的短语阐明了第 2 章中的 RV32I 版本与这些 RV64I 版本之间的差异。

MIPS-64 用到了最多的指令，主要是因为它需要使用 nop 指令来填充无法有效利用起来的分支延迟槽。由于比较和分支用一条指令完成，而且分支指令没有延迟槽，RV64I 需要的指令更少。虽然相较于 RV64I，对于每个分支，ARM-64 和 x86-64 需要多使用两条指令，但它们的缩放寻址模式避免了 RV64I 中所需的地址算术指令，可以让它们少使用一些指令。但是总的而言，RV64I + RV64C 代码大小要小得多，具体原因会在下一节阐述。

英特尔没有发明 x86-64 ISA。当向 64 位地址转换时，英特尔发明了一种名为 Itanium 的新 ISA，它与 x86-32 不兼容。x86-32 处理器的竞争对手被拦在了 Itanium 的门外，因此 AMD 发明了一款名为 AMD64 的 64 位版 x86-32。Itanium 最终失败了，因此英特尔被迫采用 AMD64 ISA 作为 x86-32 的 64 位地址继承者，我们称之为 x86-64 [Kerner & Padgett 2007]。

补充说明：ARM-64，MIPS-64 和 x86-64 不是官方名称

他们的官方名称是：我们所说的 ARM-64 实则是 ARMv8，MIPS-64 是 MIPS-IV 和 x86-64 是 AMD64（有关 x86-64 的历史记录，请参见上一页的侧栏）。

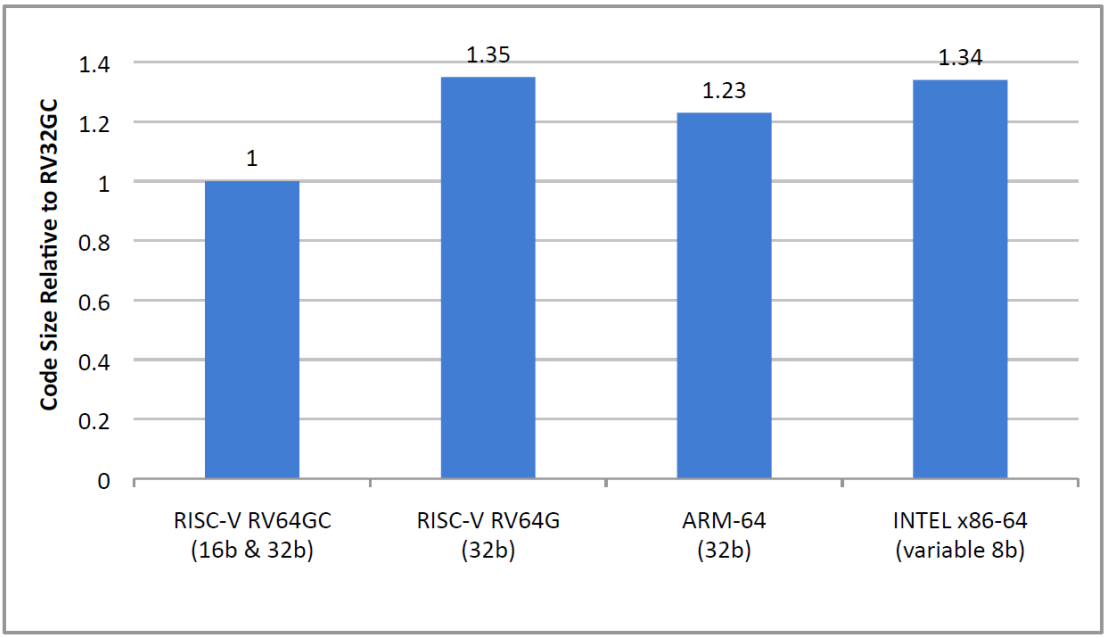


图 9.7：RV64G，ARM-64 和 x86-64 与 RV64GC 的相对程序大小比较。我们使用了比图 9.6 中更大的程序来进行对比。该图第 2 章中第 9 页的图 1.5 中的 32 位 ISA 比较的对应的 64 位 ISA 比较 .RV32C 代码大小与 RV64C 几乎一致;仅比 RV64C 小 1%。 ARM-64 没有 Thumb-2 选项，因此其他 64 位 ISA 的代码大小明显大于 RV64GC 代码。测量的程序是 SPEC CPU2006 基准测试，使用的 GCC 编译器[Waterman 2016]。

9.3 程序大小

图 9.7 比较了 RV64，ARM-64 和 x86-64 的平均相对代码大小。将这个图见第 1 章第 9 页的图 1.5 比较。首先，RV32GC 代码的大小与 RV64GC 几乎相同;它只比 RV64GC 小 1%。RV32I 和 RV64I 的代码大小也很接近。而 ARM-64 代码比 ARM-32 代码小 8%，由于没有 64 位地址版本的 Thumb-2，所以所有指令都保持 32 位长。因此，ARM-64 代码比 ARMThumb-2 代码大 25%。由于添加了前缀操作码以装下新的指令以及扩展的寄存器，x86-64 的代码比 x86-32 代码大 7%。因此，就程序大小而言，RV64GC 更优秀，因为 ARM-64 代码比 RV64GC 大 23%，x86-64 代码比 RV64GC 大 34%。程序大小的差异如此得大，以至于 RV64 可以较低的指令高速缓存缺失率来提供更高的性能，或者可以使用更小的指令缓存来降低成本，但依然能提供令人满意的缺失率。

9.4 结束语

成为先驱者的一个问题是你总是犯错误，而我永远不会想成为先驱者。最好是在看到先驱者所犯的错误后，赶紧来做这件事情，成为第二个做这件事情的人。

——Seymour Cray，第一台超级计算机的架构师，1976 年

耗尽地址位是计算机体系结构的致命弱点，许多架构因为这个缺点而消亡。ARM-32 和

Thumb-2 仍然是 32 位架构，所以他们对大型程序没有帮助。像 MIPS-64 和 x86-64 这样的一些 ISA 在转型中幸存下来，但 x86-64 并不是 ISA 设计的典范，而写这篇文章的时候，MIPS-64 的前路依然迷茫。ARM-64 是一个新的大型 ISA，时间会告诉我们它会有多成功。

MIPS 正在出售。Imagination Technologies 于 2012 年以 1 亿美元收购了 MIPS ISA，最近宣布将其出售；还没有买家。

RISC-V 受益于同时设计 32 位和 64 位架构，而较老的 ISA 必须依次设计它们。不出所料，对于 RISC-V 程序员和编译器编写者来说，32 位到 64 位之间的过渡是最简单的; RV64I ISA 几乎包含了所有 RV32I 指令。这也就是为什么我们只用两页参考卡片，就可以列出 RV32GCV 和 RV64GCV 指令集。更重要的是，同步设计意味着 64 位架构指令集不必被狭窄的 32 位操作码空间限制。 RV64I 有足够的空间用于可选的指令扩展，特别是 RV64C，这使它成为代码大小比其他所有 64 位 ISA 都要小。

我们认为 64 位架构更能体现 RISC-V 设计上的优越性，毕竟我们设计 64 位 ISA 比先行者们晚了 20 年，这样我们可以可以学习先行者们的好的设计并从他们的错误中吸取教训。

补充说明：RV128

RV128 最初是作为 RISC-V 架构师内部的一个玩笑话，只是为了显示 128 位地址的 ISA 是可能的。但是，仓库规模的计算机可能很快就会拥有超过 2^{64} 字节存储容量的半导体存储器（DRAM 和闪存），同时程序员可能会想要用访问内存的方式来访问这些存储。同时还有人[伍德拉夫等人 2014]提议使用 128 位地址来提高安全性。 RISC-V 手册确实指定了一个完整的 128 位 ISA 叫做 RV128G [Waterman and Asanovic 2017]。如图 9.1 至 9.4 所示，新增的指令基本上是与从 RV32 切换到 RV64 新增的指令类似。所有的寄存器也增长到 128 位，新的 RV128 指令要么指定了 128 位版本的指令（指令名称中使用 Q，意为四字（quadword））或其他指令的 64 位版本（指令名称中使用 D，意为双字（double word）的）。

9.5 扩展阅读

I. ARM. Armv8-a architecture reference manual. 2015.

M. Kerner and N. Padgett. A history of modern 64-bit computing. Technical report, CS Department,University of Washington, Feb 2007. URL http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf.

J. Mashey. The long road to 64 bits. *Communications of the ACM*, 52(1):45–53, 2009.

A.Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.

```
# RV64I (19 instructions, 76 bytes, or 52 bytes with RV64C)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00850693  addi  a3,a0,8    # (8 vs 4) a3 is pointer to a[i]
   4: 00100713  li    a4,1       # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  ret              # return from function
Continue Outer Loop:
  10: 0006b803  ld    a6,0(a3)   # (ld vs lw) x = a[i]
  14: 00068613  mv    a2,a3      # a2 is pointer to a[j]
  18: 00070793  mv    a5,a4      # j = i
Inner Loop:
  1c: ff863883  ld    a7,-8(a2)  # (ld vs lw, 8 vs 4) a7 = a[j-1]
  20: 01185a63  ble   a7,a6,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01163023  sd    a7,0(a2)   # (sd vs sw) a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ff860613  addi  a2,a2,-8   # (8 vs 4) decrement a2 to point to a[j]
  30: fe0796e3  bnez  a5,1c      # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00379793  slli  a5,a5,0x3  # (8 vs 4) multiply a5 by 8
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107b023  sd    a6,0(a5)   # (sd vs sw) a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00868693  addi  a3,a3,8    # increment a3 to point to a[i]
  48: fc1ff06f  j     8          # jump to Outer Loop  # continue outer loop
```

图 9.8：图 2.5 中插入排序的 RV64I 代码。 RV64I 汇编语言程序与第 2 章中第 27 页的图 2.8 中的 RV32I 汇编语言程序类似。我们在注释中的括号内列出了差异。数据的大小现在是 8 个字节而不是 4 个，所以三条指令中的常数从 4 变到了 8.由于数据宽度的变化，两个加载字（lw）相应变成了加载双字（ld）和两个存储字（sw）相应变成了存储双字（sd）。

```
# ARM-64 (16 instructions, 64 bytes)
# x0 points to a[0], x1 is n, x2 is j, x3 is i, x4 is x
  0: d2800023  mov  x3, #0x1             # i = 1
Outer Loop:
  4: eb01007f  cmp  x3, x1               # compare i vs n
  8: 54000043  b.cc 10                   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
  c: d65f03c0  ret                       # return from function
Continue Outer Loop:
 10: f8637804  ldr  x4, [x0, x3, lsl #3] # (x4 ca r4) vs x = a[i]
 14: aa0303e2  mov  x2, x3               # (x2 vs r2) j = i
Inner Loop:
 18: 8b020c05  add  x5, x0, x2, lsl #3   # x5 is pointer to a[j]
 1c: f85f80a5  ldur x5, [x5, #-8]        # x5 = a[j]
 20: eb0400bf  cmp  x5, x4               # compare a[j-1] vs. x
 24: 5400008d  b.le 34                   # if a[j-1]<=a[i], jump to Exit Inner Loop

 28: f8227805  str  x5, [x0, x2, lsl #3] # a[j] = a[j-1]
 2c: f1000442  subs x2, x2, #0x1         # j--
 30: 54ffff41  b.ne 18                   # if j != 0, jump to Inner Loop
Exit Inner Loop:
 34: f8227804  str  x4, [x0, x2, lsl #3] # a[j] = x
 38: 91000463  add  x3, x3, #0x1         # i++
 3c: 17fffff2  b    4                    # jump to Outer Loop
```

图 9.9：图 2.5 所示的插入排序的 ARM-64 代码。 ARM-64 汇编语言程序是第 2 章中第 30 页图 2.11 中的 ARM-32 汇编语言不同，它是一套全新的指令系统。寄存器以 x 开始而不是以 a 开头。数据寻址模式支持将寄存器移位 3 位用于将索引缩放为字节地址。使用 31 个寄存器，所以无需保存和恢复寄存器堆栈。由于 PC 不是寄存器之一，因此它使用了单独的返回指令。事实上，代码看起来更接近 RV64I 代码或 x86-64 代码而不是 ARM-32 代码。

```
# MIPS-64 (24 instructions, 96 bytes)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
  0: 64860008 daddiu a2,a0,8   # (daddiu vs addiu, 8 vs 4) a2 is pointer to a[i]
  4: 24030001 li     v1,1      # i = 1
Outer Loop:
  8: 0065102b sltu   v0,v1,a1  # set on i < n
  c: 14400003 bnez   v0,1c     # if i < n, jump to Continue Outer Loop
 10: 00c03825 move   a3,a2     # a3 is pointer to a[j] (slot filled)
 14: 03e00008 jr     ra        # return from function
 18: 00000000 nop              # branch delay slot unfilled
Continue Outer Loop:
 1c: dcc80000 ld     a4,0(a2)  # (ld vs lw) x = a[i]
 20: 00601025 move   v0,v1     # j = i
Inner Loop:
 24: dce9fff8 ld     a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
 28: 0109502a slt    a6,a4,a5  # (no load delay slot) set a[i] < a[j-1]
 2c: 11400005 beqz   a6,44     # if a[j-1] <= a[i], jump to Exit Inner Loop
 30: 00000000 nop              # branch delay slot unfilled
 34: 6442ffff daddiu v0,v0,-1  # (daddiu vs addiu) j--
 38: fce90000 sd     a5,0(a3)  # (sd vs sw, a5 vs t1) a[j] = a[j-1]
 3c: 1440fff9 bnez   v0,24     # if j != 0, jump to Inner Loop (next slot filled)
 40: 64e7fff8 daddiu a3,a3,-8  # (daddiu vs addiu, 8 vs 4) decr a2 pointer to a[j]
Exit Inner Loop:
 44: 000210f8 dsll   v0,v0,0x3 # (dsll vs sll)
 48: 0082102d daddu  v0,a0,v0  # (daddu vs addu) v0 now byte address oi a[j]
 4c: fc480000 sd     a4,0(v0)  # (sd vs sw) a[j] = x
 50: 64630001 daddiu v1,v1,1   # (daddiu vs addiu) i++
 54: 1000ffec b      8         # jump to Outer Loop (next delay slot filled)
 58: 64c60008 daddiu a2,a2,8   # (daddiu vs addiu, 8 vs 4) incr a2 pointer to a[i]
 5c: 00000000 nop              # Unncessary(?)
```

图 9.10：图 2.5 中所示的插入排序的 MIPS-64 代码。MIPS-64 汇编语言程序与第 2 章第 29 页图 2.10 中的 MIPS-32 汇编语言有一些不同之处。首先，对于 64 位数据的大多数操作都在其名称前加上“d”：daddiu，daddu，dsll。如图 9.8，由于数据大小从 4 字节增加到 8 字节，因此有三条指令将常量从 4 更改为 8。再次与 RV64I 类似，增加的数据宽度，使得两个加载字（lw）变成了加载双字（ld），两个存储字（sw）变成了存储双字（sd）。最后，MIPS-64 没有了 MIPS-32 中的加载延迟槽;当出现写后读依赖时，流水线会阻塞。

```
# x86-64 (15 instructions, 46 bytes)
# rax is j, rcx is x, rdx is i, rsi is n, rdi is pointer to a[0]
  0: ba 01 00 00 00 mov edx,0x1
Outer Loop:
  5: 48 39 f2       cmp rdx,rsi            # compare i vs. n
  8: 73 23          jae 2d <Exit Loop>     # if i >= n, jump to Exit Outer Loop
  a: 48 8b 0c d7    mov rcx,[rdi+rdx*8]    # x = a[i]
  e: 48 89 d0       mov rax,rdx            # j = i
Inner Loop:
 11: 4c 8b 44 c7 f8 mov r8,[rdi+rax*8-0x8] # r8 = a[j-1]
 16: 49 39 c8       cmp r8,rcx             # compare a[j-1] vs. x
 19: 7e 09          jle 24 <Exit Loop>     # if a[j-1]<=a[i],jump to Exit InnerLoop
 1b: 4c 89 04 c7    mov [rdi+rax*8],r8     # a[j] = a[j-1]
 1f: 48 ff c8       dec rax                # j--
 22: 75 ed          jne 11 <Inner Loop>    # if j != 0, jump to Inner Loop
Exit InnerLoop:
 24: 48 89 0c c7    mov [rdi+rax*8],rcx    # a[j] = x
 28: 48 ff c2       inc rdx                # i++
 2b: eb d8          jmp 5 <Outer Loop>     # jump to Outer Loop
Exit Outer Loop:
 2d: c3             ret                    # return from function
```

图 9.11：图 2.5 中插入排序的 x86-64 代码。 x86-64 汇编语言程序与第 2 章中第 30 页图 2.11 中的 x86-32 汇编语言非常不同。首先，与 RV64I 不同，较宽的寄存器有不同的名称 rax，rcx，rdx，rsi，rdi，r8。第二，因为 x86-64 增加了 8 个寄存器，现在可以将所有变量保存在寄存器而不是内存中。第三，x86-64 指令比 x86-32 更长，因为许多指令需要预先添加 8 位或 16 位前缀码，才能使得操作码空间中放得下这些新指令。例如，递增或递减寄存器（inc，dec）在 x86-32 中只需要 1 字节，但 x86-64 中需要 3 个字节。因此，对于 Insertion Sort，虽然 x86-64 指令数比 x86-32 少，但代码大小为几乎与 x86-32 相同：45 个字节对 46 个字节。

第十章 RV32/64 特权架构

简洁性是可靠性的前提。——Edsger W. Dijkstra

Edsger W. Dijkstra（1930-2002）因开发编程语言的基础性贡献而获得 1972 年图灵奖。

10.1 导言

到目前为止，本书主要关注 RISC-V 对通用计算的支持：我们引入的所有指令都在用户模式（应用程序的代码在此模式下运行）下可用。本章介绍两种新的权限模式：运行最可信的代码的机器模式（machine mode），以及为 Linux，FreeBSD 和 Windows 等操作系统提供支持的监管者模式（supervisor mode）。这两种新模式都比用和模式有着更高的权限，这也是本章标题的来源。有更多权限的模式通常可以使用权限较低的模式的所用功能，并且它们还有一些低权限模式下不可用的额外功能，例如处理中断和执行 I/O 的功能。处理器通常大部分时间都运行在权限最低的模式下，处理中断和异常时会将控制权移交到更高权限的模式。

嵌入式系统运行时（runtime）和操作系统用这些新模式的功能来响应外部事件，如网络数据包的到达；支持多任务处理和任务间保护；抽象和虚拟化硬件功能等。鉴于这些主题的广度，为此而编撰的全面的程序员指南会是另外一本完整的书。但我们的这一章节旨在强调 RISC-V 这部分功能的亮点。对嵌入式系统运行时和操作系统不感兴趣的程序员可以跳过或略读本章。

图 10.1 是 RISC-V 特权指令的图形表示，图 10.2 列出了这些指令的操作码。显然，特权架构添加的指令非常少。作为替代，几个新的控制状态寄存器（CSR）显示了附加的功能。

本章将 RV32 和 RV64 特权架构一并介绍。一些概念仅在整数寄存器的大小上有所不同，因此为了描述简洁，我们引入术语 XLEN 来指代整数寄存器的宽度（以位为单位）。对于 RV32，XLEN 为 32；对 RV64，XLEN 则是 64。

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
0001000	00010	00000	000	00000	1110011	R sret
0011000	00010	00000	000	00000	1110011	R mret
0001000	00101	00000	000	00000	1110011	R wfi
0001001	rs2	rs1	000	00000	1110011	R sfence.vma

图 10.1：RISC-V 特权指令的指令图示。

RV32/64 Privileged Instructions

machine-mode
supervisor-mode } trap return
supervisor-mode fence virtual memory address
wait for interrupt

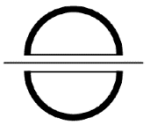
图 10.2：RISC-V 特权指令的指令布局、操作码、指令类型和名称。（来自[Waterman and Asanovic 2017]中的表 6.1。）

10.2 简单嵌入式系统的机器模式

机器模式（缩写为 M 模式，M-mode）是 RISC-V 中 *hart*（hardware thread，硬件线程）可以执行的最高权限模式。在 M 模式下运行的 hart 对内存，I/O 和一些对于启动和配置系统来说必要的底层功能有着完全的使用权。因此它是唯一所有标准 RISC-V 处理器都必须实现的权限模式。实际上简单的 RISC-V 微控制器仅支持 M 模式。这类系统是本节的重点。

hart 是硬件线程(hardware thread)的缩略形式。我们用该术语将它们与大多数程序员熟悉的软件线程区分开来。软件线程在 harts 上进行分时复用。大多数处理器核都只有一个 hart。

机器模式最重要的特性是拦截和处理异常（不寻常的运行时事件）的能力。RISC-V 将异常分为两类。一类是同步异常，这类异常在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令时。另一类是中断，它是与指令流异步的外部事件，比如鼠标的单击。RISC-V 中实现精确例外：保证异常之前的所有指令都完整地执行了，而后续的指令都没有开始执行（或等同于没有执行）。图 10.3 列出了触发标准例外的原因。



C 扩展不会发生非对齐的指令地址异常，因为它永远不可能跳转到奇数地址：分支和 JAL 的立即数总是偶数，JALR 屏蔽其有效地址的最低有效位。如果没有 C 扩展，跳转到被 4 除余 2 的地址时会发生此异常。

在 M 模式运行期间可能发生的同步例外有五种：

- *访问错误异常* 当物理内存的地址不支持访问类型时发生（例如尝试写入 ROM）。
- *断点异常* 在执行 ebreak 指令，或者地址或数据与调试触发器匹配时发生。
- *环境调用异常* 在执行 ecall 指令时发生。
- *非法指令异常* 在译码阶段发现无效操作码时发生。
- *非对齐地址异常* 在有效地址不能被访问大小整除时发生，例如地址为 0x12 的 amoadd.w。

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3：RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常（参见第 10.5 节）。（来自[Waterman and Asanovic 2017]中的表 3.6。）

如果你还记得第二章声明允许非对齐的 load 和 store，你可能会问为什么图 10.3 中还会有非对齐的 load 和 store 地址异常。原因有两个，首先，第六章的原子内存操作需要自然对齐的地址；其次，一些实现者选择省略对于非对齐的常规 load 和 store 的硬件支持，因为它是一个难以实现的不常用特性。没有这种硬件的处理器依赖于异常处理程序，用一系列较小的对齐 load 和 store 来模拟软件中非对齐的 load 和 store。应用程序代码并没有变得更好：虽然速度很慢，非对齐访存操作仍按预期进行，而硬件仍然很简单。或者，在更高性能的处理器中可以硬件实现非对齐的 load 和 store。这种实现上的灵活性归功于 RISC-V 允许非对齐 load 和 store 与常规 load 和 store 共用操作码。这遵照了第一章将架构和具体实现隔离开的准则。

有三种标准的中断源：软件、时钟和外部来源。软件中断通过向内存映射寄存器中存数来触发，并通常用于由一个 hart 中断另一个 hart（在其他架构中称为处理器间中断机制）。当 hart 的时间比较器（一个名为 mtimecmp 的内存映射寄存器）大于实时计数器 mtime 时，会触发时钟中断。外部中断由平台级中断控制器（大多数外部设备连接到这个中断控制器）引发。不同的硬件平台具有不同的内存映射并且需要中断控制器的不同特性，因此用于发出和消除这些中断的机制因平台而异。所有 RISC-V 系统的共同问题是如何处理异常和屏蔽中断，这是下一节的主题。

10.3 机器模式下的异常处理

八个控制状态寄存器（CSR）是机器模式下异常处理的必要部分：

- mtvec（Machine Trap Vector）它保存发生异常时处理器需要跳转到的地址。
- mepc（Machine Exception PC）它指向发生异常的指令。
- mcause（Machine Exception Cause）它指示发生异常的种类。
- mie（Machine Interrupt Enable）它指出处理器目前能处理和必须忽略的中断。
- mip（Machine Interrupt Pending）它列出目前正准备处理的中断。
- mtval（Machine Trap Value）它保存了陷入（trap）的附加信息：地址例外中出错的地址、发生非法指令例外的指令本身，对于其他异常，它的值为 0。
- mscratch（Machine Scratch）它暂时存放一个字大小的数据。
- mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态，如图 10.4 所示。

XLEN-1	XLEN-2 … 23	22	21	20	19	18	17
SD	0	TSR	TW	TVM	MXR	SUM	MPRV
1	XLEN-24	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	0	SPP	MPIE	0	SPIE	UPIE	MIE	0	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

图 10.4：mstatus 控制状态寄存器。在仅有机器模式且没有 F 和 V 扩展的简单处理中，有效的域只有全局中断使能、MIE 和 MPIE（它在异常发生后保存 MIE 的旧值）。RV32 的 XLEN 时 32，RV64 是 40。（来自[Waterman and Asanovic 2017]中的表 3.6；有关其他域的说明请参见该文档的第 3.1 节。）

处理器在 M 模式下运行时，只有在全局中断使能位 mstatus.MIE 置 1 时才会产生中断.此外，每个中断在控制状态寄存器 mie 中都有自己的使能位。这些位在 mie 中的位置

对应于图 10.3 中的中断代码。例如，mie[7]对应于 M 模式中的时钟中断。控制状态寄存器 mip

Encoding	Name	Abbreviation
00	User	U
01	Supervisor	S
11	Machine	M

图 10.5：RISC-V 的权限模式和它们的编码

具有相同的布局，并且它指示当前待处理的中断。将所有三个控制状态寄存器合在一起考虑，如果 mstatus.MIE = 1，mie[7] = 1，且 mip[7] = 1，则可以处理机器的时钟中断。

当一个 hart 发生异常时，硬件自动经历如下的状态转换：

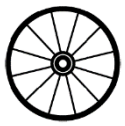
- 异常指令的 PC 被保存在 mepc 中，PC 被设置为 mtvec。（对于同步异常，mepc 指向导致异常的指令；对于中断，它指向中断处理后应该恢复执行的位置。）
- 根据异常来源设置 mcause（如图 10.3 所示），并将 mtval 设置为出错的地址或者其它适用于特定异常的信息字。
- 把控制状态寄存器 mstatus 中的 MIE 位置零以禁用中断，并把先前的 MIE 值保留到 MPIE 中。
- 发生异常之前的权限模式保留在 mstatus 的 MPP 域中，再把权限模式更改为 M。图 10.5 显示了 MPP 域的编码（如果处理器仅实现 M 模式，则有效地跳过这个步骤）。

RISC-V 还支持向量中断，其中处理器跳转到各类中断各自对应的地址，而不是一个统一的入口点。这种寻址消除了读取和解码 mcause 的需要，加快了中断处理速度。将 mtval [0]设置为 1 可启用此功能；然后根据中断原因 x 将 PC 设置为（mtval-1 + 4x），而不是通常的 mtval。

为避免覆盖整数寄存器中的内容，中断处理程序先在最开始用 mscratch 和整数寄存器（例如 a0）中的值交换。通常，软件会让 mscratch 包含指向附加临时内存空间的指针，处理程序用该指针来保存其主体中将会用到的整数寄存器。在主体执行之后，中断程序会恢复它保存到内存中的寄存器，然后再次使用 mscratch 和 a0 交换，将两个寄存器恢复到它们在发生异常之前的值。最后，处理程序用 mret 指令（M 模式特有的指令）返回。mret 将 PC 设置为 mepc，通过将 mstatus 的 MPIE 域复制到 MIE 来恢复之前的中断使能设置，并将权限模式设置为 mstatus 的 MPP 域中的值。这基本是前一段中描述的逆操作。

图 10.6 展示了遵循此模式的基本时钟中断处理程序的 RISC-V 汇编代码。它只对时间比较器执行了递增操作，然后继续执行之前的任务。更实际的时钟中断处理程序可能会调用调度程序，从而在任务之间切换。它是非抢占的，因此在处理程序的过程中中断会被禁用。不考虑这些限制条件的话，它就是一个只有一页的 RISC-V 中断处理程序的完整示例！

有时需要在处理异常的过程中转到处理更高优先级的中断。唉，mepc，mcause，mtval 和 mstatus 这些控制寄存器只有一个副本，处理第二个中断的时候如果软件不进行一些帮助的话，这些寄存器中的旧值会被破坏，导致数据丢失。可抢占的中断处理程序可以在启用中断之前把这些寄存器保存到内存中的栈，然后在退出之前，禁用中断并从栈中恢复寄存器。



除了上面介绍的 mret 指令之外，M 模式还提供了另外一条指令：wfi（Wait For Interrupt）。wfi 通知处理器目前没有任何有用的工作，所有它应该进入低功耗模式，直到任何使能有效的中断等待处理，即mie&mip ≠ 0。RISC-V 处理器以多种方式实现该指令，包括到中断待处理之前都停止时钟。有的时候只把这条指令当作 nop 来执行。因此，wfi 通常在循环内使用。

```
# save registers
csrrw a0, mscratch, a0  # save a0; set a0 = &temp storage
sw a1, 0(a0)            # save a1
sw a2, 4(a0)            # save a2
sw a3, 8(a0)            # save a3
sw a4, 12(a0)           # save a4

# decode interrupt cause
csrr a1, mcause         # read exception cause
bgez a1, exception      # branch if not an interrupt
andi a1, a1, 0x3f       # isolate interrupt cause
li a2, 7                # a2 = timer interrupt cause
bne a1, a2, otherInt    # branch if not a timer interrupt

# handle timer interrupt by incrementing time comparator
la a1, mtimecmp         # a1 = &time comparator
lw a2, 0(a1)            # load lower 32 bits of comparator
lw a3, 4(a1)            # load upper 32 bits of comparator
addi a4, a2, 1000       # increment lower bits by 1000 cycles
sltu a2, a4, a2         # generate carry-out
add a3, a3, a2          # increment upper bits
sw a3, 4(a1)            # store upper 32 bits
sw a4, 0(a1)            # store lower 32 bits

# restore registers and return
lw a4, 12(a0)           # restore a4
lw a3, 4(a0)            # restore a3
lw a2, 4(a0)            # restore a2
lw a1, 0(a0)            # restore a1
csrrw a0, mscratch, a0  # restore a0; mscratch = &temp storage
mret                    # return from handler
```

图 10.6；简单的 RISC-V 时钟中断处理程序代码。代码中假定了全局中断已通过置位 mstatus.MIE 启用；时钟中断已通过置位 mie[7]启用；mtvec CSR 已设置为此处理程序的入口地址；而且 mscratch CSR 已经设置为有 16 个字节用于保存寄存器的临时空间的地址。第一部分保存了五个寄存器，把 a0 保存在 mscratch 中，a1 到 a4 保存在内存中。然后它检查 mcause 来读取异常的类别：如果 mcause<0 则是中断，反之则是同步异常。如果是中断，就检查 mcause 的低位是否等于 7，如果是，就是 M 模式的时钟中断。如果确定是时钟中断，就给时间比较器加上 1000 个时钟周期，于是下一个时钟中断会发生在大约 1000 个时钟周期之后。最后一段恢复了 a0 到 a4 和 mscratch，然后用 mret 指令返回。

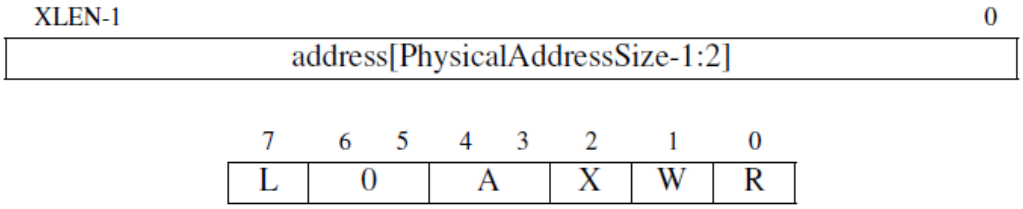


图 10.7：PMP 地址和配置寄存器。地址寄存器右移两位，如果物理地址位宽小于 XLEN-2，则高位为 0。R、W 和 X 域分别对应读、写和执行权限。A 域设置是否启用此 PMP，L 域锁定了 PMP 和对应的地址寄存器。

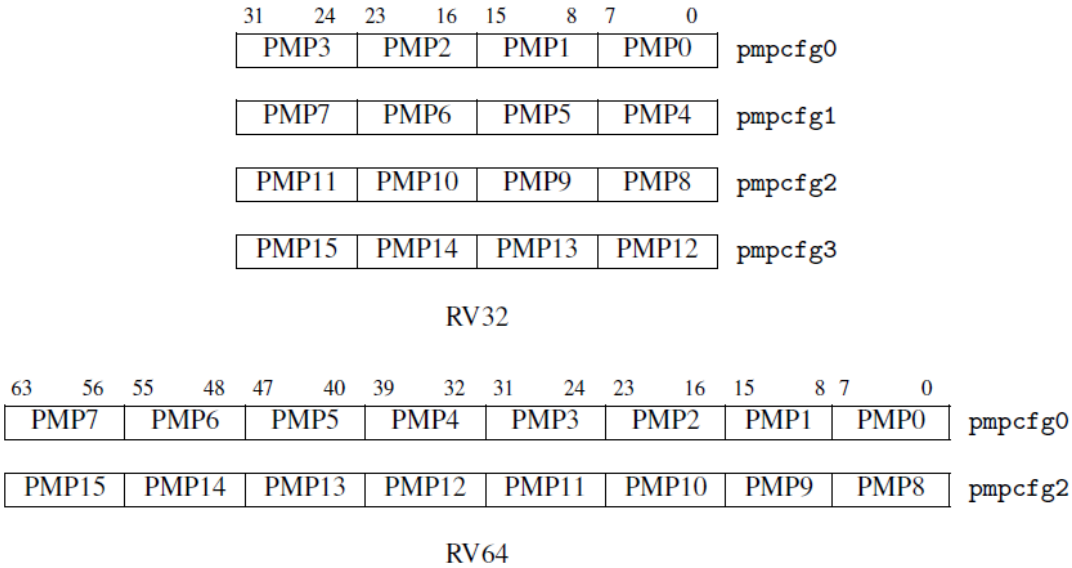


图 10.8：pmpcfg **CSR 中 PMP 配置的布局。对于 RV32（上半部分），16 个配置寄存器被分配到 4 个 CSR 中。对于 RV64（下半部分），它们则分配到了两个偶数编号的 CSR 中。**

补充说明：wfi 不论全局中断使能有效与否都有用

如果在全局中断使能有效（mstatus.MIE = 1）时执行 wfi，然后有一个使能有效的中断等待执行，则处理器跳转到异常处理程序。另一方面，如果在全局禁用中断时执行 wfi，接着一个使能有效的中断等待执行，那么处理器继续执行 wfi 之后的代码。这些代码通常会检查控制状态寄存器 mip，以决定下一步该做什么。与跳转到异常处理程序相比，这个策略可以减少中断延迟，因为不需要保存和恢复整数寄存器。

10.4 嵌入式系统中的用户模式和进程隔离

虽然机器模式对于简单的嵌入式系统已经足够，但它仅适用于那些整个代码库都可信的情况，因为 M 模式可以自由地访问硬件平台。更常见的情况是，不能信任所有的应用程序代码，因为不能事先得知这一点，或者它太大，难以证明正确性。因此，RISC-V 提供了保护系统免受不可信的代码危害的机制，并且为不受信任的进程提供隔离保护。

必须禁止不可信的代码执行特权指令（如 mret）和访问特权控制状态寄存器（如 mstatus），因为这将允许程序控制系统。这样的限制很容易实现，只要加入一种额外的权限模式：*用户模式*（U 模式）。这种模式拒绝使用这些功能，并在尝试执行 M 模式指令或访问 CSR 的时候产生非法指令异常。其它时候，U 模式和 M 模式的表现十分相似。通过将 mstatus.MPP 设置为 U（如图 10.5 所示，编码为 0），然后执行 mret 指令，软件可以从 M 模式进入 U 模式。如果在 U 模式下发生异常，则把控制移交给 M 模式。

这些不可信的代码还必须被限制只能访问自己那部分内存。实现了 M 和 U 模式的处理器具有一个叫做*物理内存保护*（PMP，Physical Memory Protection）的功能，允许 M 模式指定 U 模式可以访问的内存地址。PMP 包括几个地址寄存器（通常为 8 到 16 个）和相应的配置寄存器。这些配置寄存器可以授予或拒绝读、写和执行权限。当处于 U 模式的处理器尝试取指或执行 load 或 store 操作时，将地址和所有的 PMP 地址寄存器比较。如果地址大于等于 PMP 地址 i，但小于 PMP 地址 i+1，则 PMP i+1 的配置寄存器决定该访问是否可以继续，如果不能将会引发访问异常。

图 10.7 显示了 PMP 地址寄存器和配置寄存器的布局。两者都是 CSR，地址寄存器名为 pmpaddro 到 pmpaddrN，其中 N+1 是实现的 PMP 个数。地址寄存器右移两位，因为 PMP 以四字节为单位。配置寄存器密集地填充在 CSR 中以加速上下文切换，如图 10.8 所示。PMP 的配置由 R、W 和 X 位组成，他们分别对于 load，store 和 fetch 操作，还有另一个域 A，当它为 0 时禁用此 PMP，当它为 1 时启用。PMP 配置还支持其他模式，还可以加锁，[Waterman and Asanovic 2017]中描述了这些功能。

10.5 现代操作系统的监管者模式

上一节中描述的 PMP 方案对嵌入式系统的实现很有吸引力，因为它以相对较低的成本提供了内存保护，但它的一些缺点限制了它在通用计算中的使用。由于 PMP 仅支持固定数量的内存区域，因此无法对它进行扩展从而适应复杂的应用程序。而且由于这些区域必须在物理存储中连续，因此系统可能产生存储碎片化的问题。另外，PMP 不能有效地支持对辅存的分页。

存储碎片化的问题发生在内存可用的时候，但不包括足够大的连续块的情况。

更复杂的 RISC-V 处理器用和几乎所有通用架构相同的方式处理这些问题：使用基于页面的虚拟内存。这个功能构成了*监管者模式*（S 模式）的核心，这是一种可选的权限模式，旨在支持现代类 Unix 操作系统，如 Linux，FreeBSD 和 Windows。S 模式比 U 模式权限更高，但比 M 模式低。与 U 模式一样，S 模式下运行的软件不能使用 M 模式的 CSR 和指令，并且受到 PMP 的限制。本届介绍 S 模式的中断和异常，下一节将详细介绍 S 模式下的虚拟内存系统。

为什么不无条件地将中断委托给 S 模式？ 一个原因是虚拟化：如果 M 模式想要虚拟一个 S 模式的设备，其中断应该转到 M 模式，而不是 S 模式。

默认情况下，发生所有异常（不论在什么权限模式下）的时候，控制权都会被移交到 M 模式的异常处理程序。但是 Unix 系统中的大多数例外都应该进行 S 模式下的系统调用。M 模式的异常处理程序可以将异常重新导向 S 模式，但这些额外的操作会减慢大多数异常的处理速度。因此，RISC-V 提供了一种异常委托机制。通过该机制可以选择性地将中断和同步异常交给 S 模式处理，而完全绕过 M 模式。

mideleg（Machine Interrupt Delegation，机器中断委托）CSR 控制将哪些中断委托给 S 模式。与 mip 和 mie 一样，mideleg 中的每个位对应于图 10.3 中相同的异常。例如，mideleg[5]对应于 S 模式的时钟中断，如果把它置位，S 模式的时钟中断将会移交 S 模式的异常处理程序，而不是 M 模式的异常处理程序。

委托给 S 模式的任何中断都可以被 S 模式的软件屏蔽。sie（Supervisor Interrupt

XLEN-1	XLEN-2	20	19	18	17
SD	0		MXR	SUM	0
1	XLEN-21		1	1	1

16 15	14 13	12 9	8	7 6	5	4	3 2	1	0
XS[1:0]	FS[1:0]	0	SPP	0	SPIE	UPIE	0	SIE	UIE
2	2	4	1	2	1	1	2	1	1

图 10.9：sstatus CSR。sstatus 是 mstatus（图 10.4）的一个子集，因此它们的布局类似。SIE 和 SPIE 中分别保存了当前的和异常发生之前的中断使能，类似于 mstatus 中的 MIE 和 MPIE。RV32 的 XLEN 为 32，RV64 为 40。（来自[Waterman and Asanovic 2017]中的图 4.2；有关其他域的说明请参见该文档的第 4.1 节。）

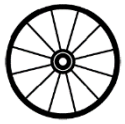
Enable，监管者中断使能）和 sip（Supervisor Interrupt Pending，监管者中断待处理）CSR 是 S 模式的控制状态寄存器，他们是 mie 和 mip 的子集。它们有着和 M 模式下相同的布局，但在 sie 和 sip 中只有与由 mideleg 委托的中断对应的位才能读写。那些没有被委派的中断对应的位始终为零。

S 模式不直接控制时钟中断和软件中断，而是使用 ecall 指令请求 M 模式设置定时器或代表它发送处理器间中断。该软件约定是监管者二进制接口 (Supervisor Binary Interface)的一部分。

M 模式还可以通过 medeleg CSR 将同步异常委托给 S 模式。该机制类似于刚才提到的中断委托，但 medeleg 中的位对应的不再是中断，而是图 10.3 中的同步异常编码。例如，置上 medeleg[15]便会把 store page fault（store 过程中出现的缺页）委托给 S 模式。

请注意，无论委派设置是怎样的，发生异常时控制权都不会移交给权限更低的模式。在 M 模式下发生的异常总是在 M 模式下处理。在 S 模式下发生的异常，根据具体的委派设置，可能由 M 模式或 S 模式处理，但永远不会由 U 模式处理。

S 模式有几个异常处理 CSR：sepc、stvec、scause、sscratch、stval 和 sstatus，它们执行与 10.2 中描述的 M 模式 CSR 相同的功能。图 10.9 显示了 sstatus 寄存器的布局。监管者异常返回指令 sret 与 mret 的行为相同，但它作用于 S 模式的异常处理 CSR，而不是 M 模式的 CSR。



S 模式处理例外的行为已和 M 模式非常相似。如果 hart 接受了异常并且把它委派给了 S 模式，则硬件会原子地经历几个类似的状态转换，其中用到了 S 模式而不是 M 模式的 CSR：

- 发生例外的指令的 PC 被存入 sepc，且 PC 被设置为 stvec。
- scause 按图 10.3 根据异常类型设置，stval 被设置成出错的地址或者其它特定异常的信息字。
- 把 sstatus CSR 中的 SIE 置零，屏蔽中断，且 SIE 之前的值被保存在 SPIE 中。
- 发生例外时的权限模式被保存在 sstatus 的 SPP 域，然后设置当前模式为 S 模式。

10.6 基于页面的虚拟内存

S 模式提供了一种传统的虚拟内存系统，它将内存划分为固定大小的页来进行地址转换和对内存内容的保护。启用分页的时候，大多数地址（包括 load 和 store 的有效地址和 PC 中的地址）都是*虚拟地址*。要访问物理内存，它们必须被转换为真正的*物理地址*，这通过遍历一种称为*页表*的高基数树实现。页表中的叶节点指示虚地址是否已经被映射到了真正的物理页面，如果是，则指示了哪些权限模式和通过哪种类型的访问可以操作这个页。

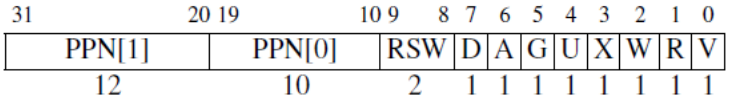


图 10.10：一个 RV32 Sv32 页表项（PTE）。

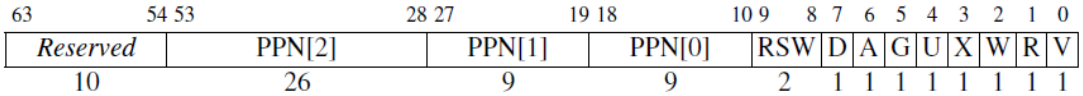


图 10.11：一个 RV64 Sv39 页表项（PTE）。

从 IBM 360 模型 67 开始，4 KiB 大小的页面已经流行了五十年。Atlas 是第一台带分页的计算机，有 3 KiB 大小的页（字长 6 个字节）。 我们发现，在计算机性能和内存容量呈指数增长半个世纪后，页面大小基本保持不变，这一点值得注意。

访问未被映射的页或访问权限不足会导致页错误例外（page fault exception）。

RISC-V 的分页方案以 SvX 的模式命名,其中 X 是以位为单位的虚拟地址的长度。RV32 的分页方案 Sv32 支持 4GiB 的虚址空间，这些空间被划分为 2^{10} 个 4 MiB 大小的巨页。每个巨页被进一步划分为 2^{10} 个 4 KiB 大小的基页（分页的基本单位）。因此，Sv32 的页表是基数为 2^{10} 的两级树结构。页表中每个项的大小是四个字节，因此页表本身的大小是 4 KiB。页表的大小和每个页的大小完全相同，这样的设计简化了操作系统的内存分配。

图 10.10 显示了 Sv32 页表项（page-table entry，PTE）的布局，从左到右分别包含如下所述的域：

- V 位决定了该页表项的其余部分是否有效（V = 1 时有效）。若 V = 0，则任何遍历到此页表项的虚址转换操作都会导致页错误。
- R、W 和 X 位分别表示此页是否可以读取、写入和执行。如果这三个位都是 0，那么这个页表项是指向下一级页表的指针，否则它是页表树的一个叶节点。
- U 位表示该页是否是用户页面。若 U = 0，则 U 模式不能访问此页面，但 S 模式可以。若 U = 1，则 U 模式下能访问这个页面，而 S 模式不能。
- G 位表示这个映射是否对所有虚址空间有效，硬件可以用这个信息来提高地址转换的性能。这一位通常只用于属于操作系统的页面。
- A 位表示自从上次 A 位被清除以来，该页面是否被访问过。
- D 位表示自从上次清除 D 位以来页面是否被弄脏（例如被写入）。
- RSW 域留给操作系统使用，它会被硬件忽略。
- PPN 域包含物理页号，这是物理地址的一部分。若这个页表项是一个叶节点，那么 PPN 是转换后物理地址的一部分。否则 PPN 给出下一节页表的地址。（图 10.10 将 PPN 划分为两个子域，以简化地址转换算法的描述。）

操作系统依赖于 A 位和 D 位来决定将哪些页面交换到辅存。定期清除 A 位有助于 OS 判断哪些页面是最近最少使用的。 置上 D 位表示换出该页面的成本更高，因为它必须写回辅存。

其他 RV64 分页方案只是向页表添加更多级别。Sv48 与 Sv39 几乎相同，但其虚拟地址空间大 2^9 倍，页表更深一层。

RV64 支持多种分页方案，但我们只介绍最受欢迎的一种，Sv39。Sv39 使用和 Sv32 相同的 4 KiB 大的基页。页表项的大小变成 8 个字节，所以它们可以容纳更大的物理地址。为了保证页表大小和页面大小一致，树的基数相应地降到 2^9，树也变为三层。Sv39 的 512 GiB 地址空间划分为 2^9 个 1 GiB 大小的吉页。每个吉页被进一步划分为 2^9 个巨页。在 Sv39 中这些巨页大小为 2 MiB，比 Sv32 中略小。每个巨页再进一步分为 2^9 个 4 KiB 大小的基页。

图 10.11 显示了 Sv39 页表项的布局。它和 Sv32 完全相同，只是 PPN 字段被扩展到了 44 位，以支持 56 位的物理地址，或者说 2^{26} GiB 大小的物理地址空间。

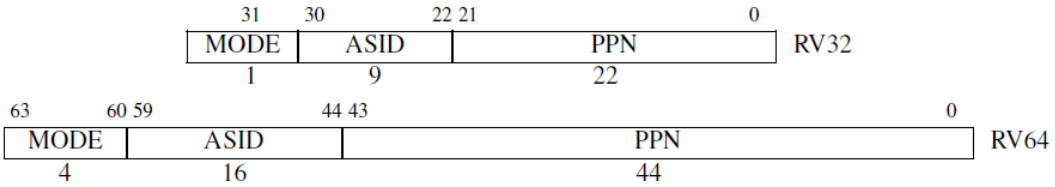


图 10.12：satp CSR。来自[Waterman and Asanovic 2017]中的图 4.11 和 4.12。

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

图 10.13：satp CSR 中 MODE 域的编码。来自[Waterman and Asanovic 2017]中的表 4.3。

补充说明：未被使用的地址位

由于 Sv39 的虚拟地址比 RV64 整数寄存器要短，可能你想知道剩下的 35 位是什么。Sv39 要求地址位 63-39 是第 38 位的副本。因此有效的虚拟地址是 $0000_0000_0000_0000_{hex}$-$0000_003f_ffff_ffff_{hex}$ 和 $ffff_ffc0_0000_0000_{hex}$-$ffff_ffff_ffff_ffff_{hex}$。这两个区间之间间隔的大小是两个区间长度大小的 2^{25} 倍，看上去似乎浪费了 64 位寄存器可以表达范围的 99.999997%。为什么不充分地利用这额外的 25 位空间呢？答案是，随着程序的增长，它们可能会需要大于 512 GiB 的虚址空间。而架构师希望再不破坏向后兼容性的前提下增加地址空间。如果我们允许程序在高 25 位中存储额外的数据，那么以后就不可能把这些位回收从而存储更大的地址。像这样允许在未使用的地址位中存储数据的严重错误，在计算机的历史中已经重复出现了多次。

一个叫 satp（Supervisor Address Translation and Protection，监管者地址转换和保护）的 S 模式控制状态寄存器控制了分页系统。如图 10.12 所示，satp 有三个域。MODE 域可以开启分页并选择页表级数，图 10.13 展示了它的编码。ASID（Address Space Identifier，地址空间标识符）域是可选的，它可以用来降低上下文切换的开销。最后，PPN 字段保存了根页表的物理地址，它以 4 KiB 的页面大小为单位。通常 M 模式的程序在第一次进入 S 模式之前会把零写入 satp 以禁用分页，然后 S 模式的程序在初始化页表以后会再次进行 satp 寄存器的写操作。

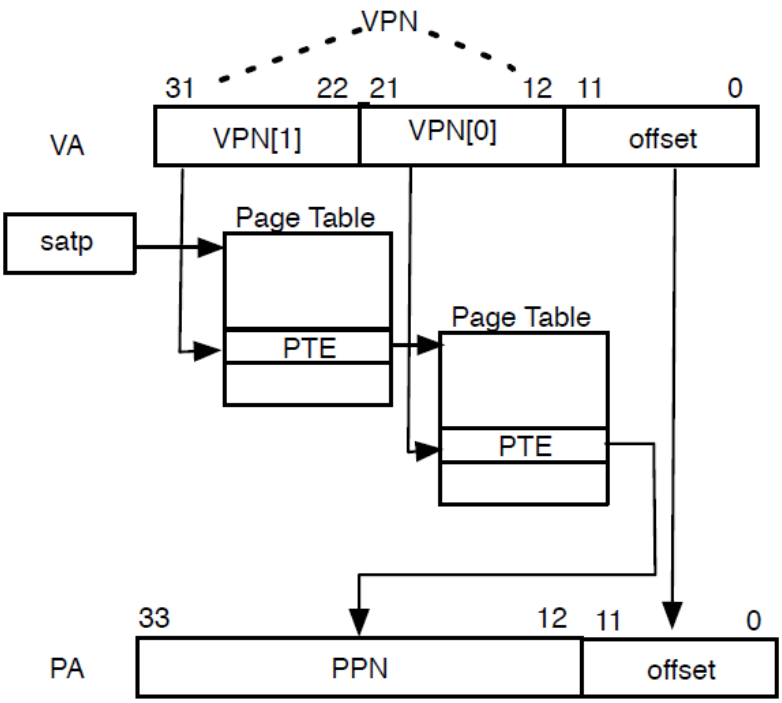


图 10.14：Sv32 中地址转换过程的图示。

当在 satp 寄存器中启用了分页时，S 模式和 U 模式中的虚拟地址会以从根部遍历页表的方式转换为物理地址。图 10.14 描述了这个过程：

1. satp.PPN 给出了一级页表的基址，VA[31:22]给出了一级页号，因此处理器会读取位于地址$(\mathrm{satp.PPN} \times 4096 + \mathrm{VA}[31:22] \times 4)$的页表项。
2. 该 PTE 包含二级页表的基址，VA[21:12]给出了二级页号，因此处理器读取位于地址$(\mathrm{PTE.PPN} \times 4096 + \mathrm{VA}[21:12] \times 4)$的叶节点页表项。
3. 叶节点页表项的 PPN 字段和页内偏移（原始虚址的最低 12 个有效位）组成了最终结果：物理地址就是$(\mathrm{LeafPTE.PPN} \times 4096 + \mathrm{VA}[11:0])$

随后处理器会进行物理内存的访问。Sv39 的转换过程几乎和 Sv32 相同，区别在于其具有较大的 PTE 和更多级页表。本章末尾的图 10.19 给出了页表遍历算法的完整描述，详细说明了例外条件和超页面转换的特殊情况。

除了一点以外，我们几乎讲完了 RISC-V 分页系统的所有内容。如果所有取指，load 和 store 操作都导致多次页表访问，那么分页会大大地降低性能！所有现代的处理器都用地址转换缓存（通常称为 TLB，全称为 Translation Lookaside Buffer）来减少这种开销。为了降低这个缓存本身的开销，大多数处理器不会让它时刻与页表保持一致。这意味着如果操作系统修改了页表，那么这个缓存会变得陈旧而不可用。S 模式添加了另一条指令来解决这个问题。这条 sfence.vma 会通知处理器，软件可能已经修改了页表，于是处理器可以相应地刷新转换缓存。它需要两个可选的参数，这样可以缩小缓存刷新的范围。一个位于 rs1，它指示了页表哪个虚址对应的转换被修改了；另一个位于 rs2，它给出了被修改页表的进程的地址空间标识符（ASID）。如果两者都是 x0，便会刷新整个转换缓存。

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	WIRI	SEIP	UEIP	MTIP	WIRI	STIP	UTIP	MSIP	WIRI	SSIP	USIP
WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

图 10.15：机器中断寄存器。它们是宽为 XLEN 位的读/写寄存器，用于保存待处理的中断（mip）和中断使能位（mie）CSR。只有与 mip 中的位对应的低权限软件中断（USIP，SSIP）、时钟中断（UTIP，STIP）和外部中断（UEIP，SEIP）的位才能通过该 CSR 的地址写入；其余的位是只读的。

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
WIRI	SEIP	UEIP	WIRI	STIP	UTIP	WIRI	SSIP	USIP
WPRI	SEIE	UEIE	WPRI	STIE	UTIE	WPRI	SSIE	USIE
XLEN-10	1	1	2	1	1	2	1	1

图 10.16：监管者中断寄存器。它们是宽为 XLEN 位的读/写寄存器，用于保存待处理的中断（sip）和中断使能位（sie）CSR。

XLEN-1 2	1 0
BASE[XLEN-1:2]	MODE
XLEN-2	2

图 10.17：机器和监管者自陷向量（trap-vector）基地址寄存器（mtvec 和 stvec）CSR。他们是位宽为 XLEN 的读/写寄存器，用于保存自陷向量的配置，包括向量基址（BASE）和向量模式（MODE）。BASE 域中的值必须按 4 字节对齐。MODE = 0 表示所有异常都把 PC 设置为 BASE。MODE = 1 会在一部中断时将 PC 设置为$(\boldsymbol{BASE} + (\mathbf{4} \times \boldsymbol{cause}))$。

XLEN-1	XLEN-2 0
Interrupt	Exception Code
1	XLEN-1

图 10.18：机器和监管者 cause（mcause 和 scause）CSR。当处理自陷时，CSR 中被写入一个指示导致自陷的事件的代码。如果自陷由中断引起，则置上中断位。“异常代码”字段包含指示最后一个异常的代码。[Waterman and Asanovic 2017]中的表 3.6 中包含自陷的来源到自陷代码的映射。

补充说明： 多处理器中的地址转换缓存一致性

sfence.vma 仅影响执行当前指令的 hart 的地址转换硬件。当 hart 更改了另一个 hart 正在使用的页表时，前一个 hart 必须用处理器间中断来通知后一个 hart，他应该执行 sfence.vma 指令。这个过程通常被称为 *TLB* 击落。

10.7 结束语

研究表明，最优秀的设计师会设计出更快、更小、更简单的结构，而且设计过程也更轻松。伟大的结构和一般的结构之间差了一个数量级。

——Fred Brooks,Jr.,1986

Brooks 是图灵奖获得者，还是 IBM System/360 系列计算机的架构师（这些计算机说明了将架构和实现区分开来的重要性）。这个 1964 年诞生的架构的继承者至今仍在销售。

RISC-V 特权架构的模块化特性满足了各种系统的需求。十分精简的机器模式以低成本的特征支持裸机嵌入式应用。附加的用户模式和物理内存保护功能共同支持了更复杂的嵌入式系统中的多任务处理。最后，监管者模式和基于页面的虚拟内存提供了运行现代操作系统所必需的灵活性。

10.8 扩展阅读

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017. URL https://riscv.org/specifications/privileged-isa/.

注记

http://parlab.eecs.berkeley.edu

1. Let a be `satp`.$ppn \times$ PAGESIZE, and let $i =$ LEVELS $- 1$.
2. Let pte be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, $pte.x$, and $pte.u$ bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If $i > 0$ and $pa.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, then either:
   - Raise a page-fault exception, or:
   - Set $pte.a$ to 1 and, if the memory access is a store, also set $pte.d$ to 1.
8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

图 10.19：虚址到物理地址转换的完整算法。va 是输入的虚拟地址，pa 是输出的物理地址。PAGESIZE 是常数 212。在 Sv32 中，LEVELS = 2 且 PTESIZE = 4；而在 Sv39 中，LEVELS = 3 且 PTESIZE = 8。来自[Waterman and Asanovic 2017]中的第 4.3.2 节。

第十一章 RISC-V 未来的可选扩展

对于复杂性，傻子无视它，实用主义者忍受它，有的人有时能规避它，而天才则解决它。

——Alan Perlis, 1982

Alan Perlis（1922-1900），因为在高级编程语言和编译器领域的贡献而成为第一位图灵奖得主（1966）。1958 年参与设计了 ALGOL，对后来所有的命令式语言，包括 C 和 Java 都有巨大影响。

以 RISC-V 为基础，至少可以有 8 种可选的扩展。

11.1 “B”标准扩展：位操作

B 扩展提供位操作，包括插入、提取和测试位字段（insert, extract, and test bit fields），旋转（rotations），漏斗位移（funnel shifts），位置换和字节置换（bit and byte permutations），计算前导 0 和尾随 0（count leading and trailing zeros）和计算置位数（count bits set）等。

11.2 “E”标准扩展：嵌入式

为了降低对低端核心的开销，这个扩展少了 16 个寄存器。正是因为 RV32E，被保存寄存器和临时寄存器都是在 0-15 号和 16-31 号这两部分之间分开的（图 3.2）。

11.3 “H”特权态架构扩展：支持管理程序（Hypervisor）

H 特权架构扩展加入了管理程序模式和基于内存页的二级地址翻译机制，提高在同一台计算机上运行多个操作系统的效率。

11.4 “J”标准扩展：动态翻译语言

有许多常用的语言使用了动态翻译，比如 Java 和 Javascript。这些语言的动态检查和垃圾回收可以得到 ISA 的支持。（J 表示即时（Just-In-Time）编译。）

11.5 “L”标准扩展：十进制浮点

L 扩展的目的是支持 IEEE 754-2008 标准规定的十进制浮点算术运算。二进制数的问题在于无法表示出一些常用的十进制小数，如 0.1。RV32L 使得计算基数可以和输入输出的基数相同。

11.6 “N”标准扩展：用户态中断

N 扩展允许用户态程序发生中断和例外后，直接进入用户态的处理程序，不触发外层运

行环境响应。用户态中断主要用于支持存在 M 模式和 U 模式的安全嵌入式系统（见第 10 章）。不过，它也能支持类 Unix 操作系统中的用户态中断。当在 Unix 环境中使用时，传统的信号处理机制依然保留，而用户态中断可以用来做未来的扩展，产生诸如垃圾回收屏障（garbage collection barriers）、整数溢出（integer overflow）、浮点陷入（floating-point traps）等用户态事件。

11.7 “P”标准扩展：封装的单指令多数据（Packed-SIMD）指令

P 扩展细分了现有的寄存器架构，提供更小数据类型上的并行计算。封装的单指令多数据指令代表了一种合理复用现有宽数据通路的设计。不过，如果有额外的资源来进行并行计算，第 8 章的向量架构通常是更好的选择，设计者更应使用 RVV 扩展。

11.8 “Q”标准扩展：四精度浮点

Q 扩展增加了符合 IEEE 754-2008 标准的 128 位的四精度浮点指令。扩展后的浮点寄存器可以存储一个单精度、双精度或者四精度的浮点数。四精度浮点扩展要求 RV64IFD。

11.9 结束语

简化，简化。

——亨利·大卫·梭罗（Henry David Thoreau），19 世纪著名作家，1854

RISC-V 具有开放、标准的扩展方式，可能意味着可以在指令集最终确定之前得到反馈和争论，使得进一步的修改为时未晚。理想情况下，一小部分成员将先把一个提议实现出来，然后再提交通过，而 FPGA 上让实现的过程变得很容易。通过 RISC-V 基础委员会提交指令扩展所需工作量比较适中，他们将努力控制 ISA 变动的速度，至少不会像 x86-32 那样有太快的变化（见第[1]章的图 1.2）。另外别忘了，不管有多少扩展被应用了，这一章提到的这些东西都是可选的。

我们希望 RISC-V 可以在保持简洁和高效的同时适应技术需求的发展。如果 RISC-V 成功了，它将成为以往增量式 ISA 上的一次革命性突破。

附录 A　RISC-V 指令列表

Coco Chanel（1883-1971）香奈儿时装品牌的创始人，她对昂贵的简约的追求塑造了 20 世纪的时尚。

简约是一切真正优雅的要义。——Coco Chanel，1923

本附录列出了 RV32/64I 的所有指令、本书中涵盖的所有扩展（RVM、RVA、RVF、RVD、RVC 和 RVV）以及所有伪指令。每个条目都包括指令名称、操作数、寄存器传输级定义、指令格式类型、中文描述、压缩版本（如果存在），以及一张带有操作码的指令布局图。我们认为这些摘要对于您了解所有的指令已经足够，但如果您想了解更多细节，请参阅 RISC-V 官方规范[Waterman and Asanovic 2017]。

为了帮助读者在本附录中找到所需的指令，左侧（奇数）页面的标题包含该页顶部的第一条指令，右侧（偶数）页面的标题包含该页底部的最后一条指令。格式类似于字典的标题，有助于您搜索单词所在的页面。例如，下一个偶数页的标题是 **AMOADD.W**，这是该页的第一条指令；下一个奇数页的标题是 **AMOMINU.D**，这是该页的最后一条指令。如下是你能在这两页中找到的指令：amoadd.w、adoand.d、amoadn.w、amomax.d、amomax.w、amomaxu.d、amomaxu.w、amomin.d、amomin.w 和 amominu.d。

add rd, rs1, rs2 x[rd] = x[rs1] + x[rs2]

加 *(Add)*. R-type, RV32I and RV64I.

把寄存器 x[*rs2*]加到寄存器 x[*rs1*]上，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.add** rd, rs2; **c.mv** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	Rd	0110011

---

addi rd, rs1, immediate x[rd] = x[rs1] + sext(immediate)

加立即数*(Add Immediate)*. I-type, RV32I and RV64I.

把符号位扩展的立即数加到寄存器 x[*rs1*]上，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0010011

---

addiw rd, rs1, immediate x[rd] = sext((x[rs1] + sext(immediate))[31:0])

加立即数字*(Add Word Immediate)*. I-type, RV64I.

把符号位扩展的立即数加到 x[*rs1*]，将结果截断为 32 位，把符号位扩展的结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.addiw** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0011011

---

addw rd, rs1, rs2 x[rd] = sext((x[rs1] + x[rs2])[31:0])

加字*(Add Word)*. R-type, RV64I.

把寄存器 x[*rs2*]加到寄存器 x[*rs1*]上，将结果截断为 32 位，把符号位扩展的结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.addw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0111011

---

amoadd.d rd, rs2, (rs1) x[rd] = AMO64(M[x[rs1]] + x[rs2])

原子加双字*(Atomic Memory Operation: Add Doubleword)*. R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 *t*+x[*rs2*]，把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00000	aq	rl	rs2	rs1	011	rd	0101111

---

amoadd.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] + x[rs2])

原子加字(Atomic Memory Operation: Add Word). R-type, RV32A and RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 *t*+x[*rs2*]，把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00000	aq	rl	rs2	rs1	010	rd	0101111

amoand.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] & x[rs2])

原子双字与 (Atomic Memory Operation: AND Doubleword). R-type, RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]位与的结果，把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01100	aq	rl	rs2	rs1	011	rd	0101111

amoand.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] & x[rs2])

原子字与 (Atomic Memory Operation: AND Word). R-type, RV32A and RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]位与的结果，把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01100	aq	rl	rs2	rs1	010	rd	0101111

amomax.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] MAX x[rs2])

原子最大双字(Atomic Memory Operation: Maximum Doubleword). R-type, RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]中较大的一个（用二进制补码比较），把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10100	aq	rl	rs2	rs1	011	rd	0101111

amomax.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MAX x[rs2])

原子最大字(Atomic Memory Operation: Maximum Word). R-type, RV32A and RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]中较大的一个（用二进制补码比较），把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10100	aq	rl	rs2	rs1	010	rd	0101111

amomaxu.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])

原子无符号最大双字(Atomic Memory Operation: Maximum Doubleword, Unsigned). R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]中较大的一个（用无符号比较），把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11100	aq	rl	rs2	rs1	011	rd	0101111

---

amomaxu.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])

原子无符号最大字(Atomic Memory Operation: Maximum Word, Unsigned). R-type, RV32A and RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]中较大的一个（用无符号比较），把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11100	aq	rl	rs2	rs1	010	rd	0101111

---

amomin.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] MIN x[rs2])

原子最小双字(Atomic Memory Operation: Minimum Doubleword). R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]中较小的一个（用二进制补码比较），把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10000	aq	rl	rs2	rs1	011	rd	0101111

---

amomin.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MIN x[rs2])

原子最小字(Atomic Memory Operation: Minimum Word). R-type, RV32A and RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]中较小的一个（用二进制补码比较），把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10000	aq	rl	rs2	rs1	010	rd	0101111

---

amominu.d rd, rs2,(rs1)

x[rd] = AMO64(M[x[rs1]] MINU x[rs2])

原子无符号最小双字(Atomic Memory Operation: Minimum Doubleword, Unsigned). R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]中较小的一个（用无符号比较），把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11000	aq	rl	rs2	rs1	011	rd	0101111

---

amominu.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MINU x[rs2])

原子无符号最大字(Atomic Memory Operation: Minimum Word, Unsigned). R-type, RV32A and RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]中较小的一个（用无符号比较），把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11000	aq	rl	rs2	rs1	010	rd	0101111

amoor.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] | x[rs2])

原子双字或 (Atomic Memory Operation: OR Doubleword). R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]位或的结果，把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01000	aq	rl	rs2	rs1	011	rd	0101111

amoor.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] | x[rs2])

原子字或 (Atomic Memory Operation: OR Word). R-type, RV32A and RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]位或的结果，把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01000	aq	rl	rs2	rs1	010	rd	0101111

amoswap.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])

原子双字交换 (Atomic Memory Operation: Swap Doubleword). R-type, RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 x[*rs2*]的值，把 x[*rd*]设为 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00001	aq	rl	rs2	rs1	011	rd	0101111

amoor.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])

原子字交换 (Atomic Memory Operation: Swap Word). R-type, RV32A and RV64A.

进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 x[*rs2*]的值，把 x[*rd*]设为符号位扩展的 *t*。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00001	aq	rl	rs2	rs1	010	rd	0101111

amoxor.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] ^ x[rs2])

原子双字异或 (Atomic Memory Operation: XOR Doubleword). R-type, RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的双字记为 *t*，把这个双字变为 t 和 x[*rs2*]按位异或的结果，把 x[*rd*]设为 *t*。

31–27	26	25	24–20	19–15	14–12	11–7	6–0
00100	aq	rl	rs2	rs1	011	rd	0101111

amoxor.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] ^ x[rs2])

原子字异或 (Atomic Memory Operation: XOR Word). R-type, RV32A and RV64A.
进行如下的原子操作：将内存中地址为 x[*rs1*]中的字记为 *t*，把这个字变为 t 和 x[*rs2*]按位异或的结果，把 x[*rd*]设为符号位扩展的 *t*。

31–27	26	25	24–20	19–15	14–12	11–7	6–0
00100	aq	rl	rs2	rs1	010	rd	0101111

and rd, rs1, rs2

x[rd] = x[rs1] & x[rs2]

与 (And). R-type, RV32I and RV64I.
将寄存器 x[*rs1*]和寄存器 x[*rs2*]位与的结果写入 x[*rd*]。
压缩形式：**c.and** rd, rs2

31–25	24–20	19–15	14–12	11–7	6–0
0000000	rs2	rs1	111	rd	0110011

andi rd, rs1, immediate

x[rd] = x[rs1] & sext(immediate)

与立即数 (And Immediate). I-type, RV32I and RV64I.
把符号位扩展的立即数和寄存器 x[*rs1*]上的值进行位与，结果写入 x[*rd*]。
压缩形式：**c.andi** rd, imm

31–20	19–15	14–12	11–7	6–0
immediate[11:0]	rs1	111	rd	0010011

auipc rd, immediate

x[rd] = pc + sext(immediate[31:12] << 12)

PC 加立即数 (Add Upper Immediate to PC). U-type, RV32I and RV64I.
把符号位扩展的 20 位（左移 12 位）立即数加到 *pc* 上，结果写入 x[*rd*]。

31–12	11–7	6–0
immediate[31:12]	rd	0010111

beq rs1, rs2, offset

if (rs1 == rs2) pc += sext(offset)

相等时分支 *(Branch if Equal)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]和寄存器 x[*rs2*]的值相等，把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

压缩形式：**c.beqz** rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

---

beqz rs1, offset

if (rs1 == 0) pc += sext(offset)

等于零时分支 *(Branch if Equal to Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

可视为 **beq** rs1, x0, offset.

---

bge rs1, rs2, offset

if (rs1 $\geqslant_s$ rs2) pc += sext(offset)

大于等于时分支 *(Branch if Greater Than or Equal)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]的值大于等于寄存器 x[*rs2*]的值（均视为二进制补码），把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	101	offset[4:1|11]	1100011

---

bgeu rs1, rs2, offset

if (rs1 $\geqslant_u$ rs2) pc += sext(offset)

无符号大于等于时分支 *(Branch if Greater Than or Equal, Unsigned)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]的值大于等于寄存器 x[*rs2*]的值（均视为无符号数），把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	111	offset[4:1|11]	1100011

---

bgez rs1, offset

if (rs1 $\geqslant_s$ 0) pc += sext(offset)

大于等于零时分支 *(Branch if Greater Than or Equal to Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

可视为 **bge** rs1, x0, offset.

---

bgt rs1, rs2, offset

if (rs1 $>_s$ rs2) pc += sext(offset)

大于时分支 *(Branch if Greater Than)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

可视为 **blt** rs2, rs1, offset.

---

bgtu rs1, rs2, offset

if (rs1 $>_u$ rs2) pc += sext(offset)

无符号大于时分支 *(Branch if Greater Than, Unsigned)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

可视为 **bltu** rs2, rs1, offset.

---

bgtz rs1, offset　　if (rs2 $>_s$ 0) pc += sext(offset)

大于零时分支 *(Branch if Greater Than Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.
可视为 **blt** x0, rs2, offset.

---

ble rs1, rs2, offset　　if (rs1 $\leqslant_s$ rs2) pc += sext(offset)

小于等于时分支 *(Branch if Less Than or Equal)*. 伪指令(Pesudoinstruction), RV32I and RV64I.
可视为 **bge** rs2, rs1, offset.

---

bleu rs1, rs2, offset　　if (rs1 $\leqslant_u$ rs2) pc += sext(offset)

小于等于时分支 *(Branch if Less Than or Equal, Unsigned)*. 伪指令(Pesudoinstruction), RV32I and RV64I.
可视为 **bgeu** rs2, rs1, offset.

---

blez rs2, offset　　if (rs2 $\leqslant_s$ 0) pc += sext(offset)

小于等于零时分支 *(Branch if Less Than or Equal to Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.
可视为 **bge** x0, rs2, offset.

---

blt rs1, rs2, offset　　if (rs1 $<_s$ rs2) pc += sext(offset)

小于时分支 *(Branch if Less Than)*. B-type, RV32I and RV64I.
若寄存器 x[*rs1*]的值小于寄存器 x[*rs2*]的值（均视为二进制补码），把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	100	offset[4:1|11]	1100011

---

bltz rs2, offset　　if (rs1 $<_s$ 0) pc += sext(offset)

小于零时分支 *(Branch if Less Than Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.
可视为 **blt** rs1, x0, offset.

---

bltu rs1, rs2, offset　　if (rs1 $<_u$ rs2) pc += sext(offset)

无符号小于时分支 *(Branch if Less Than, Unsigned)*. B-type, RV32I and RV64I.
若寄存器 x[*rs1*]的值小于寄存器 x[*rs2*]的值（均视为无符号数），把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	110	offset[4:1|11]	1100011

---

bne rs1, rs2, offset

if (rs1 ≠ rs2) pc += sext(offset)

不相等时分支 *(Branch if Not Equal)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]和寄存器 x[*rs2*]的值不相等，把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

压缩形式：**c.bnez** rs1, offset

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[12|10:5]	rs2	rs1	001	offset[4:1|11]	1100011

---

bnez rs1, offset

if (rs1 ≠ 0) pc += sext(offset)

不等于零时分支 *(Branch if Not Equal to Zero)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

可视为 **bne** rs1, x0, offset.

---

c.add rd, rs2

x[rd] = x[rd] + x[rs2]

加 *(Add)*. RV32IC and RV64IC.

扩展形式为 **add** rd, rd, rs2. rd=x0 或 rs2=x0 时非法。

15 – 13	12	11 – 7	6 – 2	1 – 0
100	1	rd	rs2	10

---

c.addi rd, imm

x[rd] = x[rd] + sext(imm)

加立即数 *(Add Immediate)*. RV32IC and RV64IC.

扩展形式为 **addi** rd, rd, imm.

15 – 13	12	11 – 7	6 – 2	1 – 0
000	imm[5]	rd	imm[4:0]	01

---

c.addi16sp imm

x[2] = x[2] + sext(imm)

加 16 倍立即数到栈指针 *(Add Immediate, Scaled by 16, to Stack Pointer)*. RV32IC and RV64IC.

扩展形式为 **addi** x2, x2, imm. imm=0 时非法。

15 – 13	12	11 – 7	6 – 2	1 – 0
011	imm[9]	00010	imm[4|6|8:7|5]	01

---

c.addi4spn rd', uimm

x[8+rd'] = x[2] + uimm

加 4 倍立即数到栈指针 *(Add Immediate, Scaled by 4, to Stack Pointer, Nondestructive)*. RV32IC and RV64IC.

扩展形式为 **addi** rd, x2, uimm, 其中 rd=8+rd'. uimm=0 时非法。

15 – 13	12 – 5	4 – 2	1 – 0
000	uimm[5:4|9:6|2|3]	rd'	00

---

c.addiw rd, imm

x[rd] = sext((x[rd] + sext(imm))[31:0])

加立即数字 *(Add Word Immediate)*. RV64IC.

扩展形式为 **addiw** rd, rd, imm. rd=x0 时非法。

15 13	12	11 7	6 2	1 0
001	imm[5]	rd	imm[4:0]	01

---

c.and rd', rs2'

x[8+rd'] = x[8+rd'] & x[8+rs2']

与 *(AND)*. RV32IC and RV64IC.

扩展形式为 **and** rd, rd, rs2, 其中 rd=8+rd', rs2=8+rs2'.

15 10	9 7	6 5	4 2	1 0
100011	rd'	11	rs2'	01

---

c.addw rd', rs2'

x[8+rd'] = sext((x[8+rd'] + x[8+rs2'])[31:0])

加字 *(Add Word)*. RV64IC.

扩展形式为 **addw** rd, rd, rs2, 其中 rd=8+rd', rs2=8+rs2'.

15 10	9 7	6 5	4 2	1 0
100111	rd'	01	rs2'	01

---

c.andi rd', imm

x[8+rd'] = x[8+rd'] & sext(imm)

与立即数 *(AND Immediate)*. RV32IC and RV64IC.

扩展形式为 **andi** rd, rd, imm, 其中 rd=8+rd'.

15 13	12	11	10 9	7 6	2 1	0
100	imm[5]	10	rd'	imm[4:0]	01	

---

c.beqz rs1', offset

if (x[8+rs1'] == 0) pc += sext(offset)

等于零时分支 *(Branch if Equal to Zero)*. RV32IC and RV64IC.

扩展形式为 **beq** rs1, x0, offset, 其中 rs1=8+rs1'.

15 13	12 10	9 7	6 2	1 0
110	offset[8|4:3]	rs1'	offset[7:6|2:1|5]	01

---

c.bnez rs1', offset

if (x[8+rs1'] ≠ 0) pc += sext(offset)

不等于零时分支 *(Branch if Not Equal to Zero)*. RV32IC and RV64IC.

扩展形式为 **bne** rs1, x0, offset, 其中 rs1=8+rs1'.

15 13	12 10	9 7	6 2	1 0
111	offset[8|4:3]	rs1'	offset[7:6|2:1|5]	01

---

c.ebreak

RaiseException(Breakpoint)

环境断点 *(Environment Breakpoint)*. RV32IC and RV64IC.

扩展形式为 **ebreak**.

15 13	12	11 7	6 2	1 0
100	1	00000	00000	10

---

c.fld rd', uimm(rs1')

f[8+rd'] = M[x[8+rs1'] + uimm][63:0]

浮点双字加载 *(Floating-point Load Doubleword)*. RV32DC and RV64DC.

扩展形式为 **fld** rd, uimm(rs1), 其中 rd=8+rd', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
001	uimm[5:3]	rs1'	uimm[7:6]	rd'	00

---

c.fldsp rd, uimm(x2)

f[rd] = M[x[2] + uimm][63:0]

栈指针相关浮点双字加载 *(Floating-point Load Doubleword, Stack-Pointer Relative)*. RV32DC and RV64DC.

扩展形式为 **fld** rd, uimm(x2).

15 13	12	11 7	6 2	1 0
001	uimm[5]	rd	uimm[4:3|8:6]	10

---

c.flw rd', uimm(rs1')

f[8+rd'] = M[x[8+rs1'] + uimm][31:0]

浮点字加载 *(Floating-point Load Word)*. RV32FC.

扩展形式为 **flw** rd, uimm(rs1), 其中 rd=8+rd', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
011	uimm[5:3]	rs1'	uimm[2|6]	rd'	00

---

c.flwsp rd, uimm(x2)

f[rd] = M[x[2] + uimm][31:0]

栈指针相关浮点字加载 *(Floating-point Load Word, Stack-Pointer Relative)*. RV32FC.

扩展形式为 **flw** rd, uimm(x2).

15 13	12	11 7	6 2	1 0
011	uimm[5]	rd	uimm[4:2|7:6]	10

---

c.fsd rs2', uimm(rs1')

M[x[8+rs1'] + uimm][63:0] = f[8+rs2']

浮点双字存储 *(Floating-point Store Doubleword)*. RV32DC and RV64DC.

扩展形式为 **fsd** rs2, uimm(rs1), 其中 rs2=8+rs2', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
101	uimm[5:3]	rs1'	uimm[7:6]	rs2'	00

---

c.fsdsp rs2, uimm(x2)

M[x[2] + uimm][63:0] = f[rs2]

栈指针相关浮点双字存储 *(Floating-point Store Doubleword, Stack-Pointer Relative)*. RV32DC and RV64DC.

扩展形式为 **fsd** rs2, uimm(x2).

15 13	12 7	6 2	1 0
101	uimm[5:3|8:6]	rs2	10

---

c.fsw rs2', uimm(rs1')

M[x[8+rs1'] + uimm][31:0] = f[8+rs2']

浮点字存储 *(Floating-point Store Word)*. RV32FC.

扩展形式为 **fsw** rs2, uimm(rs1), 其中 rs2=8+rs2', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
111	uimm[5:3]	rs1'	uimm[2|6]	rs2'	00

---

c.fswsp rs2, uimm(x2)

M[x[2] + uimm][31:0] = f[rs2]

栈指针相关浮点字存储 *(Floating-point Store Word, Stack-Pointer Relative)*. RV32FC.

扩展形式为 **fsw** rs2, uimm(x2).

15 13	12 7	6 2	1 0
111	uimm[5:2|7:6]	rs2	10

---

c.j offset

pc += sext(offset)

跳转 *(Jump)*. RV32IC and RV64IC.

扩展形式为 **jal** x0, offset.

15 13	12 2	1 0
101	offset[11|4|9:8|10|6|7|3:1|5]	01

---

c.jal offset

x[1] = pc+2; pc += sext(offset)

链接跳转 *(Jump and Link)*. RV32IC.

扩展形式为 **jal** x1, offset.

15 13	12 2	1 0
001	offset[11|4|9:8|10|6|7|3:1|5]	01

---

c.jalr rs1

t = pc+2; pc = x[rs1]; x[1] = t

寄存器链接跳转 *(Jump and Link Register)*. RV32IC and RV64IC.

扩展形式为 **jalr** x1, 0(rs1). 当 rs1=x0 时非法。

15 13	12	11 7	6 2	1 0
100	1	rs1	00000	10

---

c.jr rs1

pc = x[rs1]

寄存器跳转 *(Jump Register)*. RV32IC and RV64IC.

扩展形式为 **jalr** x0, 0(rs1). 当 rs1=x0 时非法。

15 13	12	11 7	6 2	1 0
100	0	rs1	00000	10

---

c.ld rd', uimm(rs1')

x[8+rd'] = M[x[8+rs1'] + uimm][63:0]

双字加载 *(Load Doubleword)*. RV64IC.

扩展形式为 **ld** rd, uimm(rs1), 其中 rd=8+rd', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
011	uimm[5:3]	rs1'	uimm[7:6]	rd'	00

---

c.ldsp rd, uimm(x2)

x[rd] = M[x[2] + uimm][63:0]

栈指针相关双字加载 *(Load Doubleword, Stack-Pointer Relative)*. RV64IC.

扩展形式为 **ld** rd, uimm(x2). rd=x0 时非法。

15 13	12	11 7	6 2	1 0
011	uimm[5]	rd	uimm[4:3|8:6]	10

---

c.li rd, imm

x[rd] = sext(imm)

立即数加载 *(Load Immediate)*. RV32IC and RV64IC.

扩展形式为 **addi** rd, x0, imm.

15 13	12	11 7	6 2	1 0
010	imm[5]	rd	imm[4:0]	01

---

c.lui rd, imm

x[rd] = sext(imm[17:12] << 12)

高位立即数加载 *(Load Upper Immediate)*. RV32IC and RV64IC.

扩展形式为 **lui** rd, imm. 当 rd=x2 或 imm=0 时非法。

15 13	12	11 7	6 2	1 0
011	imm[17]	rd	imm[16:12]	01

---

c.lw rd', uimm(rs1')

x[8+rd'] = sext(M[x[8+rs1'] + uimm][31:0])

字加载 *(Load Word)*. RV32IC and RV64IC.

扩展形式为 **lw** rd, uimm(rs1), 其中 rd=8+rd', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
010	uimm[5:3]	rs1'	uimm[2|6]	rd'	00

---

c.lwsp rd, uimm(x2)

x[rd] = sext(M[x[2] + uimm][31:0])

栈指针相关字加载 *(Load Word, Stack-Pointer Relative)*. RV32IC and RV64IC.

扩展形式为 **lw** rd, uimm(x2). rd=x0 时非法。

15 13	12	11 7	6 2	1 0
010	uimm[5]	rd	uimm[4:2|7:6]	10

---

c.mv rd, rs2

x[rd] = x[rs2]

移动 *(Move)*. RV32IC and RV64IC.

扩展形式为 **add** rd, x0, rs2. rs2=x0 时非法。

15 13	12	11 7	6 2	1 0
100	0	rd	rs2	10

---

c.or rd', rs2'

x[8+rd'] = x[8+rd'] | x[8+rs2']

或 *(OR)*. RV32IC and RV64IC.

扩展形式为 **or** rd, rd, rs2，其中 rd=8+rd', rs2=8+rs2'.

15 10	9 7	6 5	4 2	1 0
100011	rd'	10	rs2'	01

---

c.sd rs2', uimm(rs1')

M[x[8+rs1'] + uimm][63:0] = x[8+rs2']

双字存储*(Store Doubleword)*. RV64IC.

扩展形式为 **sd** rs2, uimm(rs1)，其中 rs2=8+rs2', rs1=8+rs1'.

15 13	12 10	9 7	6 5	4 2	1 0
111	uimm[5:3]	rs1'	uimm[7:6]	rs2'	00

---

c.sdsp rs2, uimm(x2)

M[x[2] + uimm][63:0] = x[rs2]

栈指针相关双字存储 *(Store Doubleword, Stack-Pointer Relative)*. RV64IC.

扩展形式为 **sd** rs2, uimm(x2).

15 13	12 7	6 2	1 0
111	uimm[5:3|8:6]	rs2	10

---

c.slli rd, uimm

x[rd] = x[rd] << uimm

立即数逻辑左移 *(Shift Left Logical Immediate)*. RV32IC and RV64IC.

扩展形式为 **slli** rd, rd, uimm.

15 13	12	11 7	6 2	1 0
000	uimm[5]	rd	uimm[4:0]	10

---

c.srai rd', uimm

x[8+rd'] = x[8+rd'] >>$_s$ uimm

立即数算术右移 *(Shift Right Arithmetic Immediate)*. RV32IC and RV64IC.

扩展形式为 **srai** rd, rd, uimm, 其中 rd=8+rd'.

15 13	12	11 10 9	7 6	2 1	0
100	uimm[5]	01	rd'	uimm[4:0]	01

---

c.srli rd', uimm

x[8+rd'] = x[8+rd'] >>$_u$ uimm

立即数逻辑右移 *(Shift Right Logical Immediate)*. RV32IC and RV64IC.

扩展形式为 **srli** rd, rd, uimm, 其中 rd=8+rd'.

15 13	12	11 10 9	7 6	2 1	0
100	uimm[5]	00	rd'	uimm[4:0]	01

---

c.sub rd', rs2'

x[8+rd'] = x[8+rd'] - x[8+rs2']

减 *(Subtract)*. RV32IC and RV64IC.

扩展形式为 **sub** rd, rd, rs2. 其中 rd=8+rd', rs2=8+rs2'. .

15 10 9	7 6	5 4	2 1	0
100011	rd'	00	rs2'	01

---

c.subw rd', rs2'

x[8+rd'] = sext((x[8+rd'] - x[8+rs2'])[31:0])

减字 *(Subtract Word)*. RV64IC.

扩展形式为 **subw** rd, rd, rs2. 其中 rd=8+rd', rs2=8+rs2'. .

15 10 9	7 6	5 4	2 1	0
100111	rd'	00	rs2'	01

---

c.sw rs2', uimm(rs1')

M[x[8+rs1'] + uimm][31:0] = x[8+rs2']

字存储 *(Store Word)*. RV32IC and RV64IC.

扩展形式为 **sw** rs2, uimm(rs1), 其中 rs2=8+rs2', rs1=8+rs1'.

15 13	12 10 9	7 6	5 4	2 1	0
110	uimm[5:3]	rs1'	uimm[2|6]	rs2'	00

---

c.swsp rs2, uimm(x2)

M[x[2] + uimm][31:0] = x[rs2]

栈指针相关字存储 *(Store Word, Stack-Pointer Relative)*. RV32IC and RV64IC.

扩展形式为 **sw** rs2, uimm(x2).

15 13	12 7 6	2 1	0
110	uimm[5:2|7:6]	rs2	10

---

c.xor rd', rs2' x[8+rd'] = x[8+rd'] ^ x[8+rs2']

异或 *(Exclusive-OR)*. RV32IC and RV64IC.

扩展形式为 **xor** rd, rd, rs2，其中 rd=8+rd', rs2=8+rs2'.

15	10 9	7 6	5 4	2 1	0
100011	rd'	01	rs2'	01	

---

call rd, symbol x[rd] = pc+8; pc = &symbol

调用 *(Call)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

把下一条指令的地址（*pc*+8）写入 x[*rd*]，然后把 *pc* 设为 *symbol*。等同于 **auipc** rd, offestHi, 再加上一条 **jalr** rd, offsetLo(rd). 若省略了 *rd*，默认为 x1.

---

csrr rd, csr x[rd] = CSRs[csr]

读控制状态寄存器 *(Control and Status Register Read)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

把控制状态寄存器 *csr* 的值写入 x[*rd*]，等同于 **csrrs** rd, csr, x0.

---

csrc csr, rs1 CSRs[csr] &= ~x[rs1]

清除控制状态寄存器 *(Control and Status Register Clear)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

对于 x[*rs1*]中每一个为 1 的位，把控制状态寄存器 *csr* 的的对应位清零，等同于 **csrrc** x0, csr, rs1.

---

csrci csr, zimm[4:0] CSRs[csr] &= ~zimm

立即数清除控制状态寄存器 *(Control and Status Register Clear Immediate)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

对于五位的零扩展的立即数中每一个为 1 的位，把控制状态寄存器 *csr* 的的对应位清零，等同于 **csrrci** x0, csr, zimm.

---

csrrc rd, csr, rs1 t = CSRs[csr]; CSRs[csr] = t &~x[rs1]; x[rd] = t

读后清除控制状态寄存器 *(Control and Status Register Read and Clear)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和寄存器 x[*rs1*]取反后按位与的结果写入 *csr*，再把 *t* 写入 x[*rd*]。

31	20 19	15 14	12 11	7 6	0
csr	rs1	011	rd	1110011	

---

csrrci rd, csr, zimm[4:0] `t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t`

立即数读后清除控制状态寄存器 *(Control and Status Register Read and Clear Immediate)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和五位的零扩展的立即数 *zimm* 取反后按位与的结果写入 *csr*，再把 *t* 写入 x[*rd*]（*csr* 寄存器的第 5 位及更高位不变）。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	zimm[4:0]	111	rd	1110011

---

csrrs rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t`

读后置位控制状态寄存器 *(Control and Status Register Read and Set)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和寄存器 x[*rs1*]按位或的结果写入 *csr*，再把 *t* 写入 x[*rd*]。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	rs1	010	rd	1110011

---

csrrsi rd, csr, zimm[4:0] `t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t`

立即数读后设置控制状态寄存器 *(Control and Status Register Read and Set Immediate)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把 *t* 和五位的零扩展的立即数 *zimm* 按位或的结果写入 *csr*，再把 *t* 写入 x[*rd*]（*csr* 寄存器的第 5 位及更高位不变）。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	zimm[4:0]	110	rd	1110011

---

csrrw rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t`

读后写控制状态寄存器 *(Control and Status Register Read and Write)*. I-type, RV32I and RV64I.

记控制状态寄存器 *csr* 中的值为 *t*。把寄存器 x[*rs1*]的值写入 *csr*，再把 *t* 写入 x[*rd*]。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	rs1	001	rd	1110011

---

csrrwi rd, csr, zimm[4:0] `x[rd] = CSRs[csr]; CSRs[csr] = zimm`

立即数读后写控制状态寄存器 *(Control and Status Register Read and Write Immediate)*. I-type, RV32I and RV64I.

把控制状态寄存器 *csr* 中的值拷贝到 x[*rd*]中，再把五位的零扩展的立即数 *zimm* 的值写入 *csr*。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
csr	zimm[4:0]	101	rd	1110011

---

csrs csr, rs1

CSRs[csr] |= x[rs1]

置位控制状态寄存器 *(Control and Status Register Set)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

对于 x[*rs1*]中每一个为 1 的位，把控制状态寄存器 *csr* 的的对应位置位，等同于 **csrrs** x0, csr, rs1.

---

csrsi csr, zimm[4:0]

CSRs[csr] |= zimm

立即数置位控制状态寄存器 *(Control and Status Register Set Immediate)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

对于五位的零扩展的立即数中每一个为 1 的位，把控制状态寄存器 *csr* 的的对应位清零，等同于 **csrrsi** x0, csr, zimm.

---

csrw csr, rs1

CSRs[csr] = x[rs1]

写控制状态寄存器 *(Control and Status Register Set)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

对于 x[*rs1*]中每一个为 1 的位，把控制状态寄存器 *csr* 的的对应位置位，等同于**csrrw** x0, csr, rs1.

---

csrwi csr, zimm[4:0]

CSRs[csr] = zimm

立即数写控制状态寄存器 *(Control and Status Register Write Immediate)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

把五位的零扩展的立即数的值写入控制状态寄存器 *csr* 的，等同于 **csrrwi** x0, csr, zimm.

---

div rd, rs1, rs2

x[rd] = x[rs1] $\div_s$ x[rs2]

除法*(Divide)*. R-type, RV32M and RV64M.

用寄存器 x[*rs1*]的值除以寄存器 x[*rs2*]的值，向零舍入，将这些数视为二进制补码，把商写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0110011

---

divu rd, rs1, rs2

x[rd] = x[rs1] $\div_u$ x[rs2]

无符号除法*(Divide, Unsigned)*. R-type, RV32M and RV64M.

用寄存器 x[*rs1*]的值除以寄存器 x[*rs2*]的值，向零舍入，将这些数视为无符号数，把商写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0110011

---

divuw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\div_u$ x[rs2][31:0])

无符号字除法(Divide Word, Unsigned). R-type, RV64M.

用寄存器 x[*rs1*]的低 32 位除以寄存器 x[*rs2*]的低 32 位，向零舍入，将这些数视为无符号数，把经符号位扩展的 32 位商写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0111011

---

divw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\div_s$ x[rs2][31:0])

字除法(Divide Word). R-type, RV64M.

用寄存器 x[*rs1*]的低 32 位除以寄存器 x[*rs2*]的低 32 位，向零舍入，将这些数视为二进制补码，把经符号位扩展的 32 位商写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0111011

Ebreak

RaiseException(Breakpoint)

环境断点 (Environment Breakpoint). I-type, RV32I and RV64I.

通过抛出断点异常的方式请求调试器。

31 20	19 15	14 12	11 7	6 0
000000000001	00000	000	00000	1110011

---

ecall

RaiseException(EnvironmentCall)

环境调用 (Environment Call). I-type, RV32I and RV64I.

通过引发环境调用异常来请求执行环境。

31 20	19 15	14 12	11 7	6 0
000000000000	00000	000	00000	1110011

---

fabs.d rd, rs1

f[rd] = |f[rs1]|

浮点数绝对值 (Floating-point Absolute Value). *伪指令*(Pesudoinstruction), RV32D and RV64D.

把双精度浮点数 f[*rs1*]的绝对值写入 f[*rd*]。

等同于 **fsgnjx.d** rd, rs1, rs1.

---

fabs.s rd, rs1

f[rd] = |f[rs1]|

浮点数绝对值 (Floating-point Absolute Value). *伪指令*(Pesudoinstruction), RV32F and RV64F.

把单精度浮点数 f[*rs1*]的绝对值写入 f[*rd*]。

等同于 **fsgnjx.s** rd, rs1, rs1.

---

fadd.d rd, rs1, rs2

f [rd] = f [rs1] + f [rs2]

双精度浮点加*(Floating-point Add, Double-Precision)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相加，并将舍入后的和写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	rm	rd	1010011

fadd.s rd, rs1, rs2

f [rd] = f [rs1] + f [rs2]

单精度浮点加*(Floating-point Add, Single-Precision)*. R-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相加，并将舍入后的和写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	rm	rd	1010011

fclass.d rd, rs1, rs2

x[rd] = classify_d(f[rs1])

双精度浮点分类*(Floating-point Classify, Double-Precision)*. R-type, RV32D and RV64D.
把一个表示寄存器 f[*rs1*]中双精度浮点数类别的掩码写入 x[*rd*]中。关于如何解释写入 x[*rd*]的值，请参阅指令 **fclass.s** 的介绍。

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	rs1	001	rd	1010011

fclass.s rd, rs1, rs2

x[rd] = classify_s(f[rs1])

单精度浮点分类*(Floating-point Classify, Single-Precision)*. R-type, RV32F and RV64F.
把一个表示寄存器 f[*rs1*]中单精度浮点数类别的掩码写入 x[*rd*]中。x[*rd*]中有且仅有一位被置上，见下表。

x*[rd]*位	含义
0	f [*rs1*]为−∞。
1	f [*rs1*]是负规格化数。
2	f [*rs1*]是负的非规格化数。
3	f [*rs1*]是-0。
4	f [*rs1*]是+0。
5	f [*rs1*]是正的非规格化数。
6	f [*rs1*]是正的规格化数。
7	f [*rs1*]为+∞。
8	f [*rs1*]是信号(signaling)NaN。
9	f [*rs1*]是一个安静(quiet)NaN。

31 25	24 20	19 15	14 12	11 7	6 0
1110000	00000	rs1	001	rd	1010011

fcvt.d.l rd, rs1, rs2

$f[rd] = f64_{s64}(x[rs1])$

长整型向双精度浮点转换*(Floating-point Convert to Double from Long)*. R-type, RV64D.
把寄存器 x[*rs1*]中的 64 位二进制补码表示的整数转化为双精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00010	rs1	rm	rd	1010011

---

fcvt.d.lu rd, rs1, rs2

$f[rd] = f64_{u64}(x[rs1])$

无符号长整型向双精度浮点转换*(Floating-point Convert to Double from Unsigned Long)*. R-type, RV64D.
把寄存器 x[*rs1*]中的 64 位无符号整数转化为双精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00011	rs1	rm	rd	1010011

---

fcvt.d.s rd, rs1, rs2

$f[rd] = f64_{f32}(f[rs1])$

单精度向双精度浮点转换*(Floating-point Convert to Double from Single)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]中的单精度浮点数转化为双精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0100001	00000	rs1	rm	rd	1010011

---

fcvt.d.w rd, rs1, rs2

$f[rd] = f64_{s32}(x[rs1])$

字向双精度浮点转换*(Floating-point Convert to Double from Word)*. R-type, RV32D and RV64D.
把寄存器 x[*rs1*]中的 32 位二进制补码表示的整数转化为双精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00000	rs1	rm	rd	1010011

---

fcvt.d.wu rd, rs1, rs2

$f[rd] = f64_{u32}(x[rs1])$

无符号字向双精度浮点转换*(Floating-point Convert to Double from Unsigned Word)*. R-type, RV32D and RV64D.
把寄存器 x[*rs1*]中的 32 位无符号整数转化为双精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00001	rs1	rm	rd	1010011

---

fcvt.l.d rd, rs1, rs2

$x[rd] = s64_{f64}(f[rs1])$

双精度浮点向长整型转换*(Floating-point Convert to Long from Double)*. R-type, RV64D.
把寄存器 f[*rs1*]中的双精度浮点数转化为 64 位二进制补码表示的整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00010	rs1	rm	rd	1010011

---

fcvt.l.s rd, rs1, rs2

$x[rd] = s64_{f32}(f[rs1])$

单精度浮点向长整型转换*(Floating-point Convert to Long from Single)*. R-type, RV64F.
把寄存器 f[*rs1*]中的单精度浮点数转化为 64 位二进制补码表示的整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00010	rs1	rm	rd	1010011

---

fcvt.lu.d rd, rs1, rs2

$x[rd] = u64_{f64}(f[rs1])$

双精度浮点向无符号长整型转换*(Floating-point Convert to Unsigned Long from Double)*. R-type, RV64D.
把寄存器 f[*rs1*]中的双精度浮点数转化为 64 位无符号整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00011	rs1	rm	rd	1010011

---

fcvt.lu.s rd, rs1, rs2

$x[rd] = u64_{f32}(f[rs1])$

单精度浮点向无符号长整型转换*(Floating-point Convert to Unsigned Long from Single)*. R-type, RV64F.
把寄存器 f[*rs1*]中的单精度浮点数转化为 64 位二进制补码表示的整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	rs1	rm	rd	1010011

---

fcvt.s.d rd, rs1, rs2

$f[rd] = f32_{f64}(f[rs1])$

双精度向单精度浮点转换*(Floating-point Convert to Single from Double)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]中的双精度浮点数转化为单精度浮点数，再写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0100000	00001	rs1	rm	rd	1010011

---

fcvt.s.l rd, rs1, rs2

f[rd] = $f32_{s64}$(x[rs1])

长整型向单精度浮点转换*(Floating-point Convert to Single from Long)*. R-type, RV64F.
把寄存器 x[*rs1*]中的 64 位二进制补码表示的整数转化为单精度浮点数，再写入 f[*rd*]中。

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00010	rs1	rm	rd	1010011	

---

fcvt.s.lu rd, rs1, rs2

f[rd] = $f32_{u64}$(x[rs1])

无符号长整型向单精度浮点转换*(Floating-point Convert to Single from Unsigned Long)*. R-type, RV64F.
把寄存器 x[*rs1*]中的 64 位的无符号整数转化为单精度浮点数，再写入 f[*rd*]中。

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00011	rs1	rm	rd	1010011	

---

fcvt.s.w rd, rs1, rs2

f[rd] = $f32_{s32}$(x[rs1])

字向单精度浮点转换*(Floating-point Convert to Single from Word)*. R-type, RV32F and RV64F.
把寄存器 x[*rs1*]中的 32 位二进制补码表示的整数转化为单精度浮点数，再写入 f[*rd*]中。

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00000	rs1	rm	rd	1010011	

---

fcvt.s.wu rd, rs1, rs2

f[rd] = $f32_{u32}$(x[rs1])

无符号字向单精度浮点转换*(Floating-point Convert to Single from Unsigned Word)*. R-type, RV32F and RV64F.
把寄存器 x[*rs1*]中的 32 位无符号整数转化为单精度浮点数，再写入 f[*rd*]中。

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00001	rs1	rm	rd	1010011	

---

fcvt.w.d rd, rs1, rs2

x[rd] = sext($s32_{f64}$(f[rs1]))

双精度浮点向字转换*(Floating-point Convert to Word from Double)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]中的双精度浮点数转化为 32 位二进制补码表示的整数，再写入 x[*rd*]中。

31	25 24	20 19	15 14	12 11	7 6	0
1100001	00000	rs1	rm	rd	1010011	

---

fcvt.wu.d rd, rs1, rs2

$x[rd] = sext(u32_{f64}(f[rs1]))$

双精度浮点向无符号字转换*(Floating-point Convert to Unsigned Word from Double)*. R-type, RV32D and RV64D.

把寄存器 f[*rs1*]中的双精度浮点数转化为 32 位无符号整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00001	rs1	rm	rd	1010011

---

fcvt.w.s rd, rs1, rs2

$x[rd] = sext(s32_{f32}(f[rs1]))$

单精度浮点向字转换*(Floating-point Convert to Word from Single)*. R-type, RV32F and RV64F.

把寄存器 f[*rs1*]中的单精度浮点数转化为 32 位二进制补码表示的整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00000	rs1	rm	rd	1010011

---

fcvt.wu.s rd, rs1, rs2

$x[rd] = sext(u32_{f32}(f[rs1]))$

单精度浮点向无符号字转换*(Floating-point Convert to Unsigned Word from Single)*. R-type, RV32F and RV64F.

把寄存器 f[*rs1*]中的单精度浮点数转化为 32 位无符号整数，再写入 x[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00001	rs1	rm	rd	1010011

---

fdiv.d rd, rs1, rs2

$f[rd] = f[rs1] \div f[rs2]$

双精度浮点除法*(Floating-point Divide, Double-Precision)*. R-type, RV32D and RV64D.

把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相除，并将舍入后的商写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0001101	rs2	rs1	rm	rd	1010011

---

fdiv.s rd, rs1, rs2

$f[rd] = f[rs1] \div f[rs2]$

单精度浮点除法*(Floating-point Divide, Single-Precision)*. R-type, RV32F and RV64F.

把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相除，并将舍入后的商写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0001100	rs2	rs1	rm	rd	1010011

---

fence pred, succ

Fence(pred, succ)

同步内存和 I/O*(Fence Memory and I/O)*. I-type, RV32I and RV64I.

在后续指令中的内存和 I/O 访问对外部（例如其他线程）可见之前，使这条指令之前的内存及 I/O 访问对外部可见。比特中的第 3,2,1 和 0 位分别对应于设备输入，设备输出，内存读写。例如 **fence** r, rw，将前面读取与后面的读取和写入排序，使用 *pred* = 0010 和 *succ* = 0011 进行编码。如果省略了参数，则表示 **fence** iorw, iorw，即对所有访存请求进行排序。

31 28	27 24	23 20	19 15	14 12	11 7	6 0
0000	pred	succ	00000	000	00000	0001111

---

fence.i

Fence(Store, Fetch)

同步指令流*(Fence Instruction Stream)*. I-type, RV32I and RV64I.

使对内存指令区域的读写，对后续取指令可见。

31 20	19 15	14 12	11 7	6 0
000000000000	00000	001	00000	0001111

---

feq.d rd, rs1, rs2

x[rd] = f[rs1] == f[rs2]

双精度浮点相等*(Floating-point Equals, Double-Precision)*. R-type, RV32D and RV64D.

若寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相等，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	010	rd	1010011

---

feq.s rd, rs1, rs2

x[rd] = f[rs1] == f[rs2]

单精度浮点相等*(Floating-point Equals, Single-Precision)*. R-type, RV32F and RV64F.

若寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相等，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	010	rd	1010011

---

fld rd, offset(rs1)

f[rd] = M[x[rs1] + sext(offset)][63:0]

浮点加载双字*(Floating-point Load Doubleword)*. I-type, RV32D and RV64D.

从内存地址 x[*rs1*] + *sign-extend*(*offset*)中取双精度浮点数，并写入 f[*rd*]。

压缩形式：**c.fldsp** rd, offset; **c.fld** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	011	rd	0000111

---

fle.d rd, rs1, rs2 x[rd] = f[rs1] ≤ f[rs2]

双精度浮点小于等于*(Floating-point Less Than or Equal, Double-Precision)*. R-type, RV32D and RV64D.

若寄存器 f[*rs1*]中的双精度浮点数小于等于 f[*rs2*]中的双精度浮点数，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	000	rd	1010011

---

fle.s rd, rs1, rs2 x[rd] = f[rs1] ≤ f[rs2]

单精度浮点小于等于*(Floating-point Less Than or Equal, Single-Precision)*. R-type, RV32F and RV64F.

若寄存器 f[*rs1*]中的单精度浮点数小于等于 f[*rs2*]中的单精度浮点数，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	000	rd	1010011

---

fle.d rd, rs1, rs2 x[rd] = f[rs1] < f[rs2]

双精度浮点小于 *(Floating-point Less Than, Double-Precision)*. R-type, RV32D and RV64D.

若寄存器 f[*rs1*]中的双精度浮点数小于 f[*rs2*]中的双精度浮点数，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	001	rd	1010011

---

fle.s rd, rs1, rs2 x[rd] = f[rs1] < f[rs2]

单精度浮点小于 *(Floating-point Less Than, Single-Precision)*. R-type, RV32F and RV64F.

若寄存器 f[*rs1*]中的单精度浮点数小于 f[*rs2*]中的单精度浮点数，则在 x[*rd*]中写入 1，反之写 0。

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	001	rd	1010011

---

flw rd, offset(rs1) f[rd] = M[x[rs1] + sext(offset)][31:0]

浮点加载字*(Floating-point Load Word)*. I-type, RV32F and RV64F.

从内存地址 x[*rs1*] + *sign-extend*(*offset*)中取单精度浮点数，并写入 f[*rd*]。

压缩形式：**c.flwsp** rd, offset; **c.flw** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	010	rd	0000111

---

fmadd.d rd, rs1, rs2, rs3

f[rd] = f[rs1]×f[rs2]+f[rs3]

双精度浮点乘加*(Floating-point Fused Multiply-Add, Double-Precision)*. R4-type, RV32D and RV64D.

把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相乘，并将未舍入的积和寄存器 f[*rs3*]中的双精度浮点数相加，将舍入后的双精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1000011

---

fmadd.s rd, rs1, rs2, rs3

f[rd] = f[rs1]×f[rs2]+f[rs3]

单精度浮点乘加*(Floating-point Fused Multiply-Add, Single-Precision)*. R4-type, RV32F and RV64F.

把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相乘，并将未舍入的积和寄存器 f[*rs3*]中的单精度浮点数相加，将舍入后的单精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1000011

---

fmax.d rd, rs1, rs2

f[rd] = max(f[rs1], f[rs2])

双精度浮点最大值*(Floating-point Maximum, Double-Precision)*. R-type, RV32D and RV64D.

把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数中的较大值写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0010101	rs2	rs1	001	rd	1010011

---

fmax.s rd, rs1, rs2

f[rd] = max(f[rs1], f[rs2])

单精度浮点最大值*(Floating-point Maximum, Single-Precision)*. R-type, RV32F and RV64F.

把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数中的较大值写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	001	rd	1010011

---

fmin.d rd, rs1, rs2

f[rd] = min(f[rs1], f[rs2])

双精度浮点最小值*(Floating-point Minimum, Double-Precision)*. R-type, RV32D and RV64D.

把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数中的较小值写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0010101	rs2	rs1	000	rd	1010011

---

fmin.s rd, rs1, rs2

f[rd] = min(f[rs1], f[rs2])

单精度浮点最小值*(Floating-point Minimum, Single-Precision)*. R-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数中的较小值写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	000	rd	1010011

---

fmsub.d rd, rs1, rs2, rs3

f[rd] = f[rs1]×f[rs2]-f[rs3]

双精度浮点乘减*(Floating-point Fused Multiply-Subtract, Double-Precision)*. R4-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相乘，并将未舍入的积减去寄存器 f[*rs3*]中的双精度浮点数，将舍入后的双精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1000111

---

fmsub.s rd, rs1, rs2, rs3

f[rd] = f[rs1]×f[rs2]-f[rs3]

单精度浮点乘减*(Floating-point Fused Multiply-Subtarct, Single-Precision)*. R4-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相乘，并将未舍入的积减去寄存器 f[*rs3*]中的单精度浮点数，将舍入后的单精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1000111

---

fmul.d rd, rs1, rs2

f[rd] = f[rs1] × f[rs2]

双精度浮点乘*(Floating-point Multiply, Double-Precision)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相乘，将舍入后的双精度结果写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	rm	rd	1010011

---

fmul.s rd, rs1, rs2

f[rd] = f[rs1] × f[rs2]

单精度浮点乘*(Floating-point Multiply, Single-Precision)*. R-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相乘，将舍入后的单精度结果写入 f[*rd*]中。

31 25	24 20	19 15	14 12	11 7	6 0
0001000	rs2	rs1	rm	rd	1010011

---

fmv.d rd, rs1 f[rd] = f[rs1]

双精度浮点移动 (Floating-point Move). 伪指令(Pesudoinstruction), RV32D and RV64D.
把寄存器 f[*rs1*]中的双精度浮点数复制到 f[*rd*]中，等同于 **fsgnj.d** rd, rs1, rs1.

---

fmv.d.x rd, rs1, rs2 f[rd] = x[rs1][63:0]

双精度浮点移动(Floating-point Move Doubleword from Integer). R-type, RV64D.
把寄存器 x[*rs1*]中的双精度浮点数复制到 f[*rd*]中。

31 25 24	20 19	15 14	12 11	7 6	0
1111001	00000	rs1	000	rd	1010011

---

fmv.s rd, rs1 f[rd] = f[rs1]

单精度浮点移动 (Floating-point Move). 伪指令(Pesudoinstruction), RV32F and RV64F.
把寄存器 f[*rs1*]中的单精度浮点数复制到 f[*rd*]中，等同于 **fsgnj.s** rd, rs1, rs1.

---

fmv.d.x rd, rs1, rs2 f[rd] = x[rs1][31:0]

单精度浮点移动(Floating-point Move Word from Integer). R-type, RV32F and RV64F.
把寄存器 x[*rs1*]中的单精度浮点数复制到 f[*rd*]中。

31 25 24	20 19	15 14	12 11	7 6	0
1111000	00000	rs1	000	rd	1010011

---

fmv.x.d rd, rs1, rs2 x[rd] = f[rs1][63:0]

双精度浮点移动(Floating-point Move Doubleword to Integer). R-type, RV64D.
把寄存器 f[*rs1*]中的双精度浮点数复制到 x[*rd*]中。

31 25 24	20 19	15 14	12 11	7 6	0
1110001	00000	rs1	000	rd	1010011

---

fmv.x.w rd, rs1, rs2 x[rd] = sext(f[rs1][31:0])

单精度浮点移动(Floating-point Move Word to Integer). R-type, RV32F and RV64F.
把寄存器 f[*rs1*]中的单精度浮点数复制到 x[*rd*]中，对于 RV64F，将结果进行符号扩展。

31 25 24	20 19	15 14	12 11	7 6	0
1110000	00000	rs1	000	rd	1010011

---

fneg.d rd, rs1 f[rd] = -f[rs1]

双精度浮点取反 *(Floating-point Negate)*. 伪指令(Pesudoinstruction), RV32D and RV64D.
把寄存器 f[*rs1*]中的双精度浮点数取反后写入 f[*rd*]中，等同于 **fsgnjn.d** rd, rs1, rs1.

---

fneg.s rd, rs1 f[rd] = -f[rs1]

单精度浮点取反 *(Floating-point Negate)*. 伪指令(Pesudoinstruction), RV32F and RV64F.
把寄存器 f[*rs1*]中的单精度浮点数取反后写入 f[*rd*]中，等同于 **fsgnjn.s** rd, rs1, rs1.

---

fnmadd.d rd, rs1, rs2, rs3 f[rd] = f[rs1]×f[rs2]+f[rs3]

双精度浮点乘取反加*(Floating-point Fused Negative Multiply-Add, Double-Precision)*. R4-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相乘，将结果取反，并将未舍入的积和寄存器 f[*rs3*]中的双精度浮点数相加，将舍入后的双精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1001111

---

fnmadd.s rd, rs1, rs2, rs3 f[rd] = -f[rs1]_f[rs2]-f[rs3]

单精度浮点乘取反加*(Floating-point Fused Negative Multiply-Add, Single-Precision)*. R4-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相乘，将结果取反，并将未舍入的积和寄存器 f[*rs3*]中的单精度浮点数相加，将舍入后的单精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1001111

---

fnmsub.d rd, rs1, rs2, rs3 f[rd] = -f[rs1]_f[rs2]+f[rs3]

双精度浮点乘取反减*(Floating-point Fused Negative Multiply-Subtract, Double-Precision)*. R4-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相乘，将结果取反，并将未舍入的积减去寄存器 f[*rs3*]中的双精度浮点数，将舍入后的双精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1001011

---

fnmsub.s rd, rs1, rs2, rs3 f[rd] = -f[rs1]×f[rs2]+f[rs3]

单精度浮点乘取反减*(Floating-point Fused Negative Multiply-Subtract, Single-Precision)*. R4-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相乘，将结果取反，并将未舍入的积减去寄存器 f[*rs3*]中的单精度浮点数，将舍入后的单精度浮点数写入 f[*rd*]。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1001011

---

frcsr rd x[rd] = CSRs[fcsr]

浮点读控制状态寄存器 *(Floating-point Read Control and Status Register)*. 伪指令(Pseudoinstruction), RV32F and RV64F.
把浮点控制状态寄存器的值写入 x[*rd*]，等同于 **csrrs** rd, fcsr, x0.

---

frflags rd x[rd] = CSRs[fflags]

浮点读异常标志 *(Floating-point Read Exception Flags)*. 伪指令(Pseudoinstruction), RV32F and RV64F.
把浮点异常标志的值写入 x[*rd*]，等同于 **csrrs** rd, fflags, x0.

---

frrm rd x[rd] = CSRs[frm]

浮点读舍入模式 *(Floating-point Read Rounding Mode)*. 伪指令(Pseudoinstruction), RV32F and RV64F.
把浮点舍入模式的值写入 x[*rd*]，等同于 **csrrs** rd, frm, x0.

---

fscsr rd, rs1 t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t

浮点换出控制状态寄存器 *(Floating-point Swap Control and Status Register)*. 伪指令(Pseudoinstruction), RV32F and RV64F.
把寄存器 x[*rs1*]的值写入浮点控制状态寄存器，并将浮点控制状态寄存器的原值写入 x[*rd*]，等同于 **csrrw** rd, fcsr, rs1。*rd* 默认为 x0。

---

fsd rs2, offset(rs1) M[x[rs1] + sext(offset)] = f[rs2][63:0]

双精度浮点存储*(Floating-point Store Doubleword)*. S-type, RV32D and RV64D.
将寄存器 f[*rs2*]中的双精度浮点数存入内存地址 x[*rs1*] + *sign-extend*(*offset*)中。
压缩形式：**c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100111

---

fsflags rd, rs1 t = CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd] = t

浮点换出异常标志 *(Floating-point Swap Exception Flags)*. 伪指令(Pseudoinstruction), RV32F and RV64F.

把寄存器 x[*rs1*]的值写入浮点异常标志寄存器，并将浮点异常标志寄存器的原值写入 x[*rd*]，等同于 **csrrw** rd, fflags, rs1。*rd* 默认为 x0。

---

fsgnj.d rd, rs1, rs2 f[rd] = {f[rs2][63], f[rs1][62:0]}

双精度浮点符号注入*(Floating-point Sign Inject, Double-Precision)*. R-type, RV32D and RV64D.

用 f[*rs1*]指数和有效数以及 f[*rs2*]的符号的符号位，来构造一个新的双精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	000	rd	1010011

---

fsgnj.s rd, rs1, rs2 f[rd] = {f[rs2][31], f[rs1][30:0]}

单精度浮点符号注入*(Floating-point Sign Inject, Single-Precision)*. R-type, RV32F and RV64F.

用 f[*rs1*]指数和有效数以及 f[*rs2*]的符号的符号位，来构造一个新的单精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	000	rd	1010011

---

fsgnjn.d rd, rs1, rs2 f[rd] = {~f[rs2][63], f[rs1][62:0]}

双精度浮点符号取反注入*(Floating-point Sign Inject-Negate, Double-Precision)*. R-type, RV32D and RV64D.

用 f[*rs1*]指数和有效数以及 f[*rs2*]的符号的符号位取反，来构造一个新的双精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	001	rd	1010011

---

fsgnjn.s rd, rs1, rs2 f[rd] = {~f[rs2][31], f[rs1][30:0]}

单精度浮点符号取反注入*(Floating-point Sign Inject-Negate, Single-Precision)*. R-type, RV32F and RV64F.

用 f[*rs1*]指数和有效数以及 f[*rs2*]的符号的符号位取反，来构造一个新的单精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	001	rd	1010011

---

fsgnjx.d rd, rs1, rs2

f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}

双精度浮点符号异或注入(Floating-point Sign Inject-XOR, Double-Precision). R-type, RV32D and RV64D.

用 f[*rs1*]指数和有效数以及 f[*rs1*]和 f[*rs2*]的符号的符号位异或，来构造一个新的双精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	010	rd	1010011

---

fsgnjx.s rd, rs1, rs2

f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}

单精度浮点符号异或注入(Floating-point Sign Inject-XOR, Single-Precision). R-type, RV32F and RV64F.

用 f[*rs1*]指数和有效数以及 f[*rs1*]和 f[*rs2*]的符号的符号位异或，来构造一个新的单精度浮点数，并将其写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	010	rd	1010011

---

fsqrt.d rd, rs1, rs2

f[rd] =$\sqrt{\text{f[rs1]}}$

双精度浮点平方根(Floating-point Square Root, Double-Precision). R-type, RV32D and RV64D.

将 f[*rs1*]中的双精度浮点数的平方根舍入和写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0101101	00000	rs1	rm	rd	1010011

---

fsqrt.s rd, rs1, rs2

f[rd] =$\sqrt{\text{f[rs1]}}$

单精度浮点平方根(Floating-point Square Root, Single-Precision). R-type, RV32F and RV64F.

将 f[*rs1*]中的单精度浮点数的平方根舍入和写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0101100	00000	rs1	rm	rd	1010011

---

fsrm rd, rs1

t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t

浮点换出舍入模式 (Floating-point Swap Rounding Mode). 伪指令(Pseudoinstruction), RV32F and RV64F.

把寄存器 x[*rs1*]的值写入浮点舍入模式寄存器，并将浮点舍入模式寄存器的原值写入 x[*rd*]，等同于 **csrrw** rd, frm, rs1。*rd* 默认为 x0。

---

fsub.d rd, rs1, rs2 f[rd] = f[rs1] - f[rs2]

双精度浮点减*(Floating-point Subtract, Double-Precision)*. R-type, RV32D and RV64D.
把寄存器 f[*rs1*]和 f[*rs2*]中的双精度浮点数相减，并将舍入后的差写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000101	rs2	rs1	rm	rd	1010011

---

fsub.s rd, rs1, rs2 f[rd] = f[rs1] - f[rs2]

单精度浮点减*(Floating-point Subtract, Single-Precision)*. R-type, RV32F and RV64F.
把寄存器 f[*rs1*]和 f[*rs2*]中的单精度浮点数相减，并将舍入后的差写入 f[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000100	rs2	rs1	rm	rd	1010011

---

fsw rs2, offset(rs1) M[x[rs1] + sext(offset)] = f[rs2][31:0]

单精度浮点存储*(Floating-point Store Word)*. S-type, RV32F and RV64F.
将寄存器 f[*rs2*]中的单精度浮点数存入内存地址 x[*rs1*] + *sign-extend*(*offset*)中。
压缩形式：**c.fswsp** rs2, offset; **c.fsw** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100111

---

j offset pc += sext(offset)

跳转 *(Jump)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把 *pc* 设置为当前值加上符号位扩展的 *offset*，等同于 **jal** x0, offset.

---

jal rd, offset x[rd] = pc+4; pc += sext(offset)

跳转并链接 *(Jump and Link)*. J-type, RV32I and RV64I.
把下一条指令的地址(*pc+4*)，然后把 *pc* 设置为当前值加上符号位扩展的 *offset*。*rd* 默认为 x1。
压缩形式：**c.j** offset; **c.jal** offset

31 12	11 7	6 0
offset[20|10:1|11|19:12]	rd	1101111

---

jalr rd, offset(rs1)

t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t

跳转并寄存器链接 *(Jump and Link Register)*. I-type, RV32I and RV64I.

把 *pc* 设置为 x[*rs1*]+*sign-extend*(*offset*)，把计算出的地址的最低有效位设为 0，并将原 *pc+4* 的值写入 f[*rd*]。*rd* 默认为 x1。

压缩形式：**c.jr** rs1；**c.jalr** rs1

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	000	rd	1100111

---

jr rs1

pc = x[rs1]

寄存器跳转 *(Jump Register)*. 伪指令(Pseudoinstruction), RV32I and RV64I.

把 *pc* 设置为 x[*rs1*]，等同于 **jalr** x0, 0(rs1)。

---

la rd, symbol

x[rd] = &symbol

地址加载 *(Load Address)*. 伪指令(Pseudoinstruction), RV32I and RV64I.

将 *symbol* 的地址加载到 x[*rd*]中。当编译位置无关的代码时，它会被扩展为对全局偏移量表(Global Offset Table)的加载。对于 RV32I，等同于执行 **auipc** rd, offsetHi，然后是 **lw** rd, offsetLo(rd)；对于 RV64I，则等同于 **auipc** rd，offsetHi 和 **ld** rd, offsetLo(rd)。如果 offset 过大，开始的算加载地址的指令会变成两条，先是 **auipc** rd, offsetHi 然后是 **addi** rd, rd, offsetLo。

---

lb rd, offset(rs1)

x[rd] = sext(M [x[rs1] +sext(offset)]) [7:0])

字节加载 *(Load Byte)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经符号位扩展后写入 x[*rd*]。

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	000	rd	0000011

---

lbu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][7:0]

无符号字节加载 *(Load Byte, Unsigned)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取一个字节，经零扩展后写入 x[*rd*]。

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	100	rd	0000011

---

ld rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][63:0]

双字加载 *(Load Doubleword)*. I-type, RV64I Only

从地址 x[*rs1*] + *sign-extend*(*offset*)读取八个字节，写入 x[*rd*]。

压缩形式：**c.ldsp** rd, offset; **c.ld** rd, offset(rs1)

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[11:0]	rs1	011	rd	0000011

---

lh rd, offset(rs1)

x[rd] = sext(M [x[rs1] +sext(offset)])[15:0])

半字加载 *(Load Halfword)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经符号位扩展后写入 x[*rd*]。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[11:0]	rs1	001	rd	0000011

---

lhu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][15:0]

无符号半字加载 *(Load Halfword, Unsigned)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取两个字节，经零扩展后写入 x[*rd*]。

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[11:0]	rs1	101	rd	0000011

---

li rd, immediate

x[rd] = immediate

立即数加载 *(Load Immediate)*. 伪指令(Pseudoinstruction), RV32I and RV64I.

使用尽可能少的指令将常量加载到 x[*rd*]中。在 RV32I 中，它等同于执行 **lui** 和/或 **addi**；对于 RV64I，会扩展为这种指令序列 **lui, addi, slli, addi, slli, addi ,slli, addi**。

---

la rd, symbol

x[rd] = &symbol

本地地址加载 *(Load Local Address)*. 伪指令(Pseudoinstruction), RV32I and RV64I.

将 *symbol* 的地址加载到 x[*rd*]中。等同于执行 **auipc** rd, offsetHi，然后是 **addi** rd, rd, offsetLo。

---

lr.d rd, (rs1)

x[rd] = LoadReserved64(M[x[rs1]])

加载保留双字*(Load-Reserved Doubleword)*. R-type, RV64A.

从内存中地址为 x[*rs1*]中加载八个字节，写入 x[*rd*]，并对这个内存双字注册保留。

31 – 27	26	25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
00010	aq	rl	00000	rs1	011	rd	0101111

---

lr.w rd, (rs1)

x[rd] = LoadReserved32(M[x[rs1]])

加载保留字*(Load-Reserved Word)*. R-type, RV32A and RV64A.

从内存中地址为 x[*rs1*]中加载四个字节，符号位扩展后写入 x[*rd*]，并对这个内存字注册保留。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00010	aq	rl	00000	rs1	010	rd	0101111

---

lw rd, offset(rs1)

x[rd] = sext(M [x[rs1] +sext(offset)]) [31:0])

字加载 *(Load Word)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取四个字节，写入 x[*rd*]。对于 RV64I，结果要进行符号位扩展。

压缩形式：**c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	010	rd	0000011

---

lwu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][31:0]

无符号字加载 *(Load Word, Unsigned)*. I-type, RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取四个字节，零扩展后写入 x[*rd*]。

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	110	rd	0000011

---

lui rd, immediate

x[rd] = sext(immediate[31:12] << 12)

高位立即数加载 *(Load Upper Immediate)*. U-type, RV32I and RV64I.

将符号位扩展的 20 位立即数 *immediate* 左移 12 位，并将低 12 位置零，写入 x[*rd*]中。

压缩形式：**c.lui** rd, imm

31 12	11 7	6 0
immediate[31:12]	rd	0110111

---

mret

ExceptionReturn(Machine)

机器模式异常返回*(Machine-mode Exception Return)*. R-type, RV32I and RV64I 特权架构

从机器模式异常处理程序返回。将 *pc* 设置为 CSRs[mepc]，将特权级设置成 CSRs[mstatus].MPP, CSRs[mstatus].MIE 置成 CSRs[mstatus].MPIE，并且将 CSRs[mstatus].MPIE 为 1;并且，如果支持用户模式，则将 CSR [mstatus].MPP 设置为 0。

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

---

mul rd, rs1, rs2

$x[rd] = x[rs1] \times x[rs2]$

乘*(Multiply)*. R-type, RV32M and RV64M.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积写入 x[*rd*]。忽略算术溢出。

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000001	rs2	rs1	000	rd	0110011

---

mulh rd, rs1, rs2

$x[rd] = (x[rs1]\ {}_s\times_s x[rs2]) \gg_s XLEN$

高位乘*(Multiply High)*. R-type, RV32M and RV64M.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，都视为 2 的补码，将乘积的高位写入 x[*rd*]。

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000001	rs2	rs1	001	rd	0110011

---

mulhsu rd, rs1, rs2

$x[rd] = (x[rs1]\ {}_s\times_u x[rs2]) \gg_s XLEN$

高位有符号-无符号乘*(Multiply High Signed-Unsigned)*. R-type, RV32M and RV64M.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，x[*rs1*]为 2 的补码，x[*rs2*]为无符号数，将乘积的高位写入 x[*rd*]。

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000001	rs2	rs1	010	rd	0110011

---

mulhu rd, rs1, rs2

$x[rd] = (x[rs1]\ {}_u\times_u x[rs2]) \gg_u XLEN$

高位无符号乘*(Multiply High Unsigned)*. R-type, RV32M and RV64M.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，x[*rs1*]、x[*rs2*]均为无符号数，将乘积的高位写入 x[*rd*]。

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000001	rs2	rs1	011	rd	0110011

---

mulw rd, rs1, rs2

$x[rd] = sext((x[rs1] \times x[rs2])[31:0])$

乘字*(Multiply Word)*. R-type, RV64M only.
把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积截为 32 位，进行有符号扩展后写入 x[*rd*]。忽略算术溢出。

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000001	rs2	rs1	000	rd	0111011

---

mv rd, rs1

$x[rd] = x[rs1]$

移动*(Move)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把寄存器 x[*rs1*]复制到 x[*rd*]中。实际被扩展为 **addi** rd, rs1, 0

---

neg rd, rs2

$x[rd] = -x[rs2]$

取反*(Negate)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把寄存器 x[*rs2*]的二进制补码写入 x[*rd*]。实际被扩展为 **sub** rd, x0, rs2。

---

negw rd, rs2

$x[rd] = sext((-x[rs2])[31:0])$

取非字*(Negate Word)*. 伪指令(Pseudoinstruction), RV64I only.
计算寄存器 x[*rs2*]对于 2 的补码，结果截为 32 位，进行符号扩展后写入 x[*rd*]。实际被扩展为 **subw** rd, x0, rs2。

---

nop

Nothing

无操作*(No operation)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
将 *pc* 推进到下一条指令。实际被扩展为 **addi** x0, x0, 0。

---

not rd, rs1

$x[rd] = \sim x[rs1]$

取反*(NOT)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把寄存器 x[*rs1*]对于 1 的补码（即按位取反的值）写入 x[*rd*]。实际被扩展为 **xori** rd, rs1, -1。

---

or rd, rs1, rs2

$x[rd] = x[rs1] \mid x[rs2]$

取或*(OR)*. R-type, RV32I and RV64I.
把寄存器 x[*rs1*]和寄存器 x[*rs2*]按位取或，结果写入 x[*rd*]。
压缩形式：**c.or** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	110	rd	0110011

---

ori rd, rs1, immediate

$x[rd] = x[rs1] \mid sext(immediate)$

立即数取或*(OR Immediate)*. R-type, RV32I and RV64I.
把寄存器 x[*rs1*]和有符号扩展的立即数 *immediate* 按位取或，结果写入 x[*rd*]。
压缩形式：**c.or** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
Immediate[11:0]	rs2	rs1	110	rd	0010011

---

rdcycle rd

$x[rd] = CSRS[cycle]$

读周期计数器*(Read Cycle Counter)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把周期数写入 x[*rd*]。实际被扩展为 **csrrs** rd, cycle, x0。

---

rdcycleh rd

x[rd] = CSRs[cycleh]

读周期计数器高位*(Read Cycle Counte High)*. 伪指令(Pseudoinstruction), RV32I only.
把周期数右移 32 位后写入 x[*rd*]。实际被扩展为 **csrrs** rd, cycleh, x0。

---

rdinstret rd

x[rd] = CSRs[instret]

读已完成指令计数器*(Read Instruction-Retired Counter)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把已完成指令数写入 x[*rd*]。实际被扩展为 **csrrs** rd, instret, x0。

---

rdinstreth rd

x[rd] = CSRs[instreth]

读已完成指令计数器高位*(Read Instruction-Retired Counter High)*. 伪指令(Pseudoinstruction), RV32I only.
把已完成指令数右移 32 位后写入 x[*rd*]。实际被扩展为 **csrrs** rd, instreth, x0。

---

rdtime rd

x[rd] = CSRs[time]

读取时间*(Read Time)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
把当前时间写入 x[*rd*]，时间频率与平台相关。实际被扩展为 **csrrs** rd, time, x0。

---

rdtimeh rd

x[rd] = CSRs[timeh]

读取时间高位*(Read Time High)*. 伪指令(Pseudoinstruction), RV32I only.
把当前时间右移 32 位后写入 x[*rd*]，时间频率与平台相关。实际被扩展为 **csrrs** rd, timeh, x0。

---

rem rd, rs1, rs2

x[rd] = x[rs1] $\%_s$ x[rs2]

求余数*(Remainder)*. R-type, RV32M and RV64M.
x[*rs1*]除以 x[*rs2*]，向 0 舍入，都视为 2 的补码，余数写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0110011

---

remu rd, rs1, rs2

x[rd] = x[rs1] $\%_u$ x[rs2]

求无符号数的余数*(Remainder, Unsigned)*. R-type, RV32M and RV64M.
x[*rs1*]除以 x[*rs2*]，向 0 舍入，都视为无符号数，余数写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	111	rd	0110011

---

remuw rd, rs1, rs2

$x[rd] = sext(x[rs1][31:0] \%_u x[rs2][31:0])$

求无符号数的余数字(Remainder Word, Unsigned). R-type, RV64M only.

x[*rs1*]的低 32 位除以 x[*rs2*]的低 32 位，向 0 舍入，都视为无符号数，将余数的有符号扩展写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	111	rd	0111011

---

remw rd, rs1, rs2

$x[rd] = sext(x[rs1][31:0] \%_s x[rs2][31:0])$

求余数字(Remainder Word). R-type, RV64M only.

x[*rs1*]的低 32 位除以 x[*rs2*]的低 32 位，向 0 舍入，都视为 2 的补码，将余数的有符号扩展写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0111011

---

ret

$pc = x[1]$

返回(Return). 伪指令(Pseudoinstruction), RV32I and RV64I.

从子过程返回。实际被扩展为 **jalr** x0, 0(x1)。

---

sb rs2, offset(rs1)

$M[x[rs1] + sext(offset)] = x[rs2][7:0]$

存字节(Store Byte). S-type, RV32I and RV64I.

将 x[*rs2*]的低位字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011

---

sc.d rd, rs2, (rs1)

$x[rd] = StoreConditonal64(M[x[rs1], x[rs2])$

条件存入双字(Store-Conditional Doubleword). R-type, RV64A only.

如果内存地址 x[*rs1*]上存在加载保留，将 x[*rs2*]寄存器中的 8 字节数存入该地址。如果存入成功，向寄存器 x[*rd*]中存入 0，否则存入一个非 0 的错误码。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	011	rd	0101111

---

sc.w rd, rs2, (rs1)

x[rd] = StoreConditonal32(M[x[rs1],x[rs2])

条件存入字*(Store-Conditional Word)*. R-type, RV32A and RV64A.
内存地址 x[*rs1*]上存在加载保留，将 x[*rs2*]寄存器中的 4 字节数存入该地址。如果存入成功，向寄存器 x[*rd*]中存入 0，否则存入一个非 0 的错误码。

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	010	rd	0101111

---

sd rs2, offset(rs1)

M[x[rs1] + sext(offset)]= x[rs2][63:0]

存双字*(Store Doubleword)*. S-type, RV64I only.
将 x[*rs2*]中的 8 字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。
压缩形式：**c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100011

---

seqz rd, rs1

x[rd] = (x[rs1] == 0)

等于 0 则置位*(Set if Equal to Zero)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
如果 x[*rs1*]等于 0，向 x[*rd*]写入 1，否则写入 0。实际被扩展为 **sltiu** rd, rs1, 1。

---

sext.w rd, rs1

x[rd] = sext(x[rs1][31:0])

有符号字扩展*(Sign-extend Word)*. 伪指令(Pseudoinstruction), RV64I only.
读入 x[*rs1*]的低 32 位，有符号扩展，结果写入 x[*rd*]。实际被扩展为 **addiw** rd, rs1, 0。

---

sfence.vma rs1, rs2

Fence(Store, AddressTranslation)

虚拟内存屏障*(Fence Virtual Memory)*. R-type, RV32I and RV64I 特权指令。
根据后续的虚拟地址翻译对之前的页表存入进行排序。当 *rs2*=0 时，所有地址空间的翻译都会受到影响；否则，仅对 x[*rs2*]标识的地址空间的翻译进行排序。当 *rs1*=0 时，对所选地址空间中的所有虚拟地址的翻译进行排序；否则，仅对其中包含虚拟地址 x[*rs1*]的页面地址翻译进行排序。

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

---

sgtz rd, rs2

x[rd] = (x[rs1] $>_s$ 0)

大于 0 则置位*(Set if Greater Than Zero)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
如果 x[*rs2*]大于 0，向 x[*rd*]写入 1，否则写入 0。实际被扩展为 **slt** rd, x0, rs2。

---

sh rs2, offset(rs1) M[x[rs1] + sext(offset)]= x[rs2][15: 0]

存半字(Store Halfword). S-type, RV32I and RV64I.

将 x[*rs2*]的低位 2 个字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

31 25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011

---

sw rs2, offset(rs1) M[x[rs1] + sext(offset)]= x[rs2][31: 0]

存字(Store Word). S-type, RV32I and RV64I.

将 x[*rs2*]的低位 4 个字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

压缩形式：**c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)

31 25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011

---

sll rd, rs1, rs2 x[rd] = x[rs1] ≪ x[rs2]

逻辑左移(Shift Left Logical). R-type, RV32I and RV64I.

把寄存器 x[*rs1*]左移 x[*rs2*]位，空出的位置填入 0，结果写入 x[*rd*]。x[*rs2*]的低 5 位（如果是 RV64I 则是低 6 位）代表移动位数，其高位则被忽略。

31 25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	001	rd	0110011

---

slli rd, rs1, shamt x[rd] = x[rs1] ≪ shamt

立即数逻辑左移(Shift Left Logical Immediate). I-type, RV32I and RV64I.

把寄存器x[*rs1*]左移*shamt*位，空出的位置填入0，结果写入x[*rd*]。对于RV32I，仅当*shamt*[5]=0时，指令才是有效的。

压缩形式：**c.slli** rd, shamt

31 26 25	20 19	15 14	12 11	7 6	0
000000	shamt	rs1	001	rd	0010011

---

slliw rd, rs1, shamt x[rd] = sext((x[rs1] ≪ shamt)[31: 0])

立即数逻辑左移字(Shift Left Logical Word Immediate). I-type, RV64I only.

把寄存器 x[*rs1*]左移 *shamt* 位，空出的位置填入 0，结果截为 32 位，进行有符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时，指令才是有效的。

31 26 25	20 19	15 14	12 11	7 6	0
000000	shamt	rs1	001	rd	0011011

---

sllw rd, rs1, rs2 $x[rd] = sext((x[rs1] \ll x[rs2][4:0])[31:0])$

逻辑左移字*(Shift Left Logical Word)*. R-type, RV64I only.

把寄存器 x[*rs1*]的低 32 位左移 x[*rs2*]位，空出的位置填入 0，结果进行有符号扩展后写入 x[*rd*]。x[*rs2*]的低 5 位代表移动位数，其高位则被忽略。

31 25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	001	rd	0111011

---

slt rd, rs1, rs2 $x[rd] = (x[rs1] <_s x[rs2])$

小于则置位*(Set if Less Than)*. R-type, RV32I and RV64I.

比较 x[*rs1*]和 x[*rs2*]中的数，如果 x[*rs1*]更小，向 x[*rd*]写入 1，否则写入 0。

31 25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	010	rd	0110011

---

slti rd, rs1, immediate $x[rd] = (x[rs1] <_s sext(immediate))$

小于立即数则置位*(Set if Less Than Immediate)*. I-type, RV32I and RV64I.

比较 x[*rs1*]和有符号扩展的 *immediate*，如果 x[*rs1*]更小，向 x[*rd*]写入 1，否则写入 0。

31 20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	010	rd	0010011

---

sltiu rd, rs1, immediate $x[rd] = (x[rs1] <_u sext(immediate))$

无符号小于立即数则置位*(Set if Less Than Immediate, Unsigned)*. I-type, RV32I and RV64I.

比较 x[*rs1*]和有符号扩展的 *immediate*，比较时视为无符号数。如果 x[*rs1*]更小，向 x[*rd*]写入 1，否则写入 0。

31 20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	011	rd	0010011

---

sltu rd, rs1, rs2 $x[rd] = (x[rs1] <_u x[rs2])$

无符号小于则置位*(Set if Less Than, Unsigned)*. R-type, RV32I and RV64I.

比较 x[*rs1*]和 x[*rs2*]，比较时视为无符号数。如果 x[*rs1*]更小，向 x[*rd*]写入 1，否则写入 0。

31 25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	011	rd	0110011

---

sltz rd, rs1

$x[rd] = (x[rs1] <_s 0)$

小于 0 则置位*(Set if Less Than to Zero)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
如果 x[*rs1*]小于 0，向 x[*rd*]写入 1，否则写入 0。实际扩展为 **slt** rd, rs1, x0。

---

snez rd, rs2

$x[rd] = (x[rs2] \neq 0)$

不等于 0 则置位*(Set if Not Equal to Zero)*. 伪指令(Pseudoinstruction), RV32I and RV64I.
如果 x[*rs1*]不等于 0，向 x[*rd*]写入 1，否则写入 0。实际扩展为 **sltu** rd, x0, rs2。

---

sra rd, rs1, rs2

$x[rd] = (x[rs1] \gg_s x[rs2])$

算术右移*(Shift Right Arithmetic)*. R-type, RV32I and RV64I.
把寄存器 x[*rs1*]右移 x[*rs2*]位，空位用 x[*rs1*]的最高位填充，结果写入 x[*rd*]。x[*rs2*]的低 5 位（如果是 RV64I 则是低 6 位）为移动位数，高位则被忽略。

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0110011

---

srai rd, rs1, shamt

$x[rd] = (x[rs1] \gg_s shamt)$

立即数算术右移*(Shift Right Arithmetic Immediate)*. I-type, RV32I and RV64I.
把寄存器 x[*rs1*]右移 *shamt* 位，空位用 x[*rs1*]的最高位填充，结果写入 x[*rd*]。对于 RV32I，仅当 *shamt*[5]=0 时指令有效。
压缩形式：**c.srai** rd, shamt

31 26	25 20	19 15	14 12	11 7	6
010000	shamt	rs1	101	rd	0010011

---

sraiw rd, rs1, shamt

$x[rd] = sext(x[rs1][31:0] \gg_s shamt)$

立即数算术右移字*(Shift Right Arithmetic Word Immediate)*. I-type, RV64I only.
把寄存器 x[*rs1*]的低 32 位右移 *shamt* 位，空位用 x[*rs1*][31]填充，结果进行有符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时指令有效。
压缩形式：**c.srai** rd, shamt

31 26	25 20	19 15	14 12	11 7	6
010000	shamt	rs1	101	rd	0011011

---

sraw rd, rs1, rs2

$x[rd] = sext(x[rs1][31:0] \gg_s x[rs2][4:0])$

算术右移字(Shift Right Arithmetic Word). R-type, RV64I only.

把寄存器 x[*rs1*]的低 32 位右移 x[*rs2*]位，空位用 x[*rs1*][31]填充，结果进行有符号扩展后写入 x[*rd*]。x[*rs2*]的低 5 位为移动位数，高位则被忽略。

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0111011

sret

ExceptionReturn(Supervisor)

管理员模式例外返回(Supervisor-mode Exception Return). R-type, RV32I and RV64I 特权指令。从管理员模式的例外处理程序中返回，设置 *pc* 为 CSRs[spec]，权限模式为 CSRs[sstatus].SPP，CSRs[sstatus].SIE 为 CSRs[sstatus].SPIE，CSRs[sstatus].SPIE 为 1，CSRs[sstatus].spp 为 0。

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00010	00000	000	00000	1110011

srl rd, rs1, rs2

$x[rd] = (x[rs1] \gg_u x[rs2])$

逻辑右移(Shift Right Logical). R-type, RV32I and RV64I.

把寄存器 x[*rs1*]右移 x[*rs2*]位，空出的位置填入 0，结果写入 x[*rd*]。x[*rs2*]的低 5 位（如果是 RV64I 则是低 6 位）代表移动位数，其高位则被忽略。

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0110011

srli rd, rs1, shamt

$x[rd] = (x[rs1] \gg_u shamt)$

立即数逻辑右移(Shift Right Logical Immediate). I-type, RV32I and RV64I.

把寄存器 x[*rs1*]右移 *shamt* 位，空出的位置填入 0，结果写入 x[*rd*]。对于 RV32I，仅当 *shamt*[5]=0 时，指令才是有效的。

压缩形式：**c.srli** rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0010011

srliw rd, rs1, shamt

$x[rd] = sext(x[rs1][31:0] \gg_u shamt)$

立即数逻辑右移字(Shift Right Logical Word Immediate). I-type, RV64I only.

把寄存器 x[*rs1*]右移 *shamt* 位，空出的位置填入 0，结果截为 32 位，进行有符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时，指令才是有效的。

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0011011

srlw rd, rs1, rs2

$x[rd] = sext(x[rs1][31:0] \gg_u x[rs2][4:0])$

逻辑右移字(Shift Right Logical Word). R-type, RV64I only.

把寄存器 x[*rs1*]的低 32 位右移 x[*rs2*]位，空出的位置填入 0，结果进行有符号扩展后写入 x[*rd*]。x[*rs2*]的低 5 位代表移动位数，其高位则被忽略。

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0111011

---

sub rd, rs1, rs2

$x[rd] = x[rs1] - x[rs2]$

减(Substract). R-type, RV32I and RV64I.

x[*rs1*]减去 x[*rs2*]，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.sub** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0110011

---

subw rd, rs1, rs2

$x[rd] = sext((x[rs1] - x[rs2])[31:0])$

减去字(Substract Word). R-type, RV64I only.

x[*rs1*]减去 x[*rs2*]，结果截为 32 位，有符号扩展后写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.subw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

---

tail symbol

pc = &symbol; clobber x[6]

尾调用(Tail call). 伪指令(Pseudoinstuction), RV32I and RV64I.

设置 *pc* 为 *symbol*，同时覆写 x[6]。实际扩展为 **auipc** x6, offsetHi 和 **jalr** x0, offsetLo(x6)。

---

wfi

while (noInterruptPending) idle

等待中断(Wait for Interrupt). R-type, RV32I and RV64I 特权指令。

如果没有待处理的中断，则使处理器处于空闲状态。

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

---

xor rd, rs1, rs2 x[rd] = x[rs1] ^ x[rs2]

异或*(Exclusive-OR)*. R-type, RV32I and RV64I.

x[*rs1*]和 x[*rs2*]按位异或，结果写入 x[*rd*]。

压缩形式：**c.xor** rd, rs2

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000000	rs2	rs1	100	rd	0110011

---

xori rd, rs1, immediate x[rd] = x[rs1] ^ sext(immediate)

立即数异或*(Exclusive-OR Immediate)*. I-type, RV32I and RV64I.

x[*rs1*]和有符号扩展的 *immediate* 按位异或，结果写入 x[*rd*]。

压缩形式：**c.xor** rd, rs2

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
immediate[11:0]	rs1	100	rd	0010011