Arquitetura de Computadores / Programação Assembly

O Guia Prático RISC-V é uma concisa introdução e referência para programadores de sistemas embarcados, estudantes e aos curiosos sobre uma arquitetura moderna, popular e aberta. O RISC-V abrange desde o microcontrolador de 32 bits de baixo custo até o mais rápido computador na nuvem de 64 bits.
Ao utilizar os recursos visuais ilustrados acima, o texto mostra como o RISC-V incorporou as boas ideias de arquiteturas passadas, evitandos os erros cometidos.

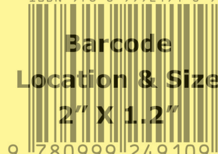
- Introduz o RISC-V em apenas 100 páginas, incluindo 75 figuras
- Cartão de Referência RISC-V de 2 páginas que resume todas as instruções
- Glossário de Instruções de 50 páginas que define todas as instruções detalhadamente
- 75 dicas de boas práticas de projeto de arquitetura utilizando os ícones acima
- 50 barras laterais com comentários interessantes e com o histórico do RISC-V
- 25 citações para transmitir o conhecimento de cientistas e engenheiros notáveis

Dez capítulos apresentam cada componente do conjunto de instruções modular RISC-V, muitas vezes, contrastando código compilado de C para RISC-V versus as arquiteturas ARM, Intel e MIPS, porém os leitores podem iniciar a programação após o Capítulo 2.

"I like RISC-V and this book as they are elegant—brief, to the point, and complete." — C. Gordon Bell

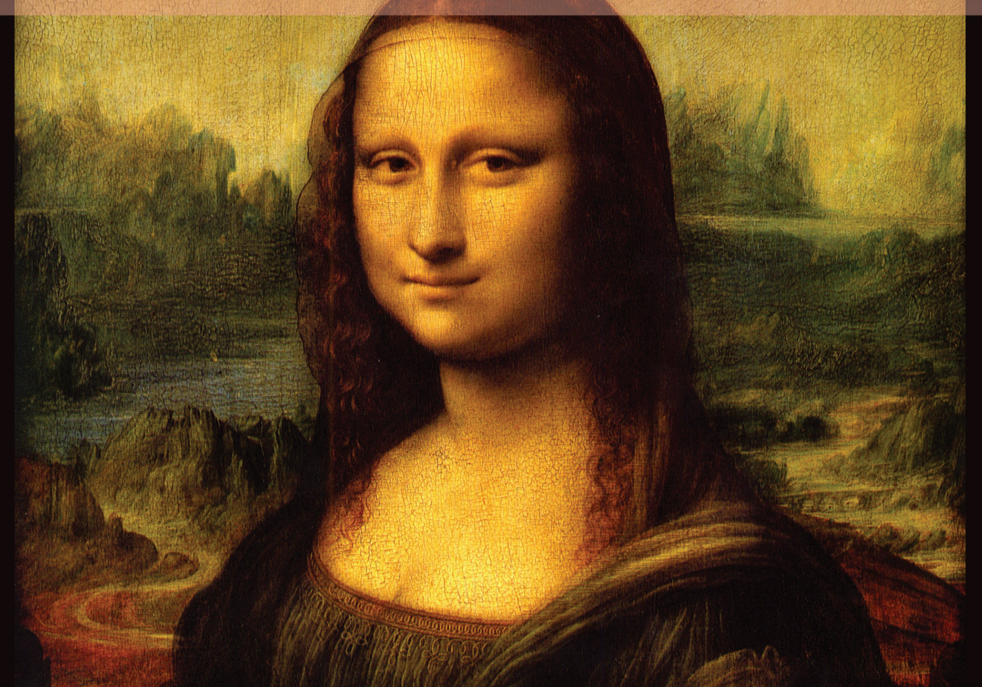
David Patterson (Google & UC Berkeley) e
Andrew Waterman (SiFive) são 2 dos 4 arquitetos RISC-V

Strawberry Canyon LLC
San Francisco, CA, USA



Guia Prático RISC-V

Patterson & Waterman



David Patterson
Andrew Waterman

GUIA PRÁTICO RISC–V

ATLAS DE UMA ARQUITETURA ABERTA

Depoimentos

Este livro oportuno descreve de forma concisa a ISA RISC-V simples, livre e aberta que está experimentando rápida absorção em muitas áreas diferentes da computação. O livro contém muitos insights sobre arquitetura de computadores em geral, e também descreve as escolhas particulares de projeto que fizemos na criação do RISC-V. Eu posso imaginar este livro se tornando um guia de referência bem utilizado por muitos usuários do RISC-V.

—Professor Krste Asanović, University of California, Berkeley, um dos quatro arquitetos do RISC-V

Eu gosto do RISC-V e deste livro, pois eles são elegantes, breves, direto ao ponto e completos. Os comentários do livro fornecem gratuitamente uma história, motivações e críticas de arquitetura.

—C. Gordon Bell, Microsoft e projetista das arquiteturas de conjunto de instruções Digital PDP-11 e VAX-11

Este livro prático resume fielmente todos os elementos essenciais da Arquitetura de Conjunto de Instruções RISC-V, um guia de referência perfeito para estudantes e desenvolvedores.

—Professor Randy Katz, University of California, Berkeley, um dos inventores do sistema de armazenamento RAID

O RISC-V é uma ótima opção para os alunos que desejam aprender sobre arquitetura de conjunto de instruções e programação assembly, os fundamentos básicos para o trabalho com linguagens de alto nível. Este livro escrito de forma clara oferece uma boa introdução ao RISC-V, ampliada com comentários sobre sua história evolutiva e comparações com outras arquiteturas familiares. Com base na experiência passada com outras arquiteturas, os projetistas do RISC-V conseguiram evitar recursos desnecessários, muitas vezes irregulares, produzindo uma pedagogia fácil. Embora simples, ainda é poderoso o suficiente para uso generalizado em aplicações reais. Há muito tempo, eu costumava ensinar um primeiro curso em programação assembly e se eu estivesse fazendo isso agora, eu ficaria feliz em utilizar este livro.

—John Mashey, um dos projetistas do MIPS

Este livro conta o que o RISC-V pode fazer e por que seus projetistas escolheram dotá-lo dessas habilidades. Ainda mais interessante, os autores dizem por que o RISC-V omite os elementos encontrados em máquinas anteriores. As razões são no mínimo tão interessantes quanto as qualidades e pontos fracos do RISC-V.

—Ivan Sutherland, *Turing Award laureate* conhecido como o pai da computação gráfica

O RISC-V mudará o mundo e este livro ajudará você a tornar-se parte dessa mudança.

—Professor Michael B. Taylor, University of Washington

Este livro será uma referência valiosa para quem trabalha com a ISA RISC-V. Os opcodes são apresentados em vários formatos úteis para referência rápida, facilitando a codificação assembly e interpretação. Além disso, as explicações e exemplos de como utilizar a ISA tornam o trabalho do programador ainda mais simples. As comparações com outras ISAs são interessantes e demonstram por que os criadores do RISC-V tomaram certas decisões de projeto.

—Megan Wachs, PhD, Engenheira da SiFive

CARTÃO DE REFERÊNCIA ABERTO ①

Instruções da Base de Números Inteiros: RV32I e RV64

Categoria	Nome	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`
Aritmética	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`	
Lógica	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`	
Comparação	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`	
Desvios	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`	
Salto & 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`	

Registrador de controle de Status (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`

Categoria	Nome	Fmt	RV32I Base	+RV64I
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`	`LWU rd,rs1,imm`
	Load Word	I	`LW 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`

Instruções Privilegiadas para RV

Categoria	Nome	Fmt	Mnemônica do RV
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`

Exemplos das 60 pseudo-instruções do RV

Nome	Fmt	Mnemônica
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`

Extensão de Instrução Compactada (16 bits): RV32C

Categoria	Nome	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`
Aritmética	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`
Desloc.	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`
Desvios	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Salto	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Salto & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
Sistema	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

Extensão Compactada Opcional: RV64C

All RV32C (except `C.JAL`*, 4 word loads, 4 word stores) 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`)

Formatos de instrução de 32 bits

	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

Formatos de instrução de 16 bits (RVC)

	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

Inteiro base RISC-V (RV32I / 64I), RV32 / 64C privilegiado e opcional. Registradores x1-x3 e o PC têm 32 bits de largura em RV32I e RV64I (x0 = 0).
.RV64I adiciona 12 instruções para os dados mais amplos. Cada instrução RVC de 16 bits é mapeada para uma instrução RISC-V de 32 bits.

CARTÃO DE REFERÊNCIA ABERTO

②

Extensão de Instrução Multiplicação-Divisão: RVM

Categoria	Nome	Fmt	RV32M (Multiplicação-Divisão)		+RV64M	
Multiplicação	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		
Divisão	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Restante	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Extensão de Instrucões Atômicas: RVA

Categoria	Nome	Fmt	RV32A (atômica)		+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
Soma	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Lógica	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

Duas extensões de instrução de ponto flutuante: RVF e RVD

Categoria	Nome	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
Conversão	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		
Aritmética	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-Soma	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		
Comparação	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		
Categorizar	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configurar	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		

Convenção de chamada

Registrador	Nome ABI	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

Registrador	Descrição
zero	Zero hardwired
ra	Endereço de ret.
sp	Ponteiro p. pilha
gp	Ponteiro global
tp	Thread pointer
t0-6,ft0-11	Temporários
s0-11,fs0-11	Registradores salvos
a0-7,fa0-7	Args. de função

Extensão Vetorial: RVV

Nome	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

Convenção de chamada RISC-V e cinco extensões opcionais: 8 RV32M; 11 RV32A; 34 instruções de ponto flutuante para dados de 32 e 64 bits (RV32F, RV32D); e 53 RV32V. Usando a notação regex, {} signica set, então FADD. {F | D} é tanto FADD.F quanto FADD.D. RV32 {F | D} adiciona registradores f0-f31, cuja largura corresponde à maior precisão, e comprimento v1. O RV64 adiciona instruções: o RVM obtém 4 , RVA 11, RVF 6, RVD 6 e RVV 0.

Guia prático RISC-V
Atlas de uma Arquitetura Aberta
Primeira edição, 1.0.0

David Patterson e Andrew Waterman

March 29, 2019

Copyright 2017 Strawberry Canyon LLC. All rights reserved.
No part of this book or its related materials may be reproduced in any form without the written consent of the copyright holder.

Book version: 1.0.0

A imagem de capa é uma foto da ***Mona Lisa***. Um retrato de Lisa Gherardini, pintado entre 1503 e 1506, por Leonardo da Vinci. O rei da França comprou-o de Leonardo por volta de 1530, e está exposto no Museu do Louvre em Paris desde 1797. A Mona Lisa é considerada a obra de arte mais conhecida do mundo. Mona Lisa representa a elegância, que acreditamos ser uma característica do RISC-V.

Este livro foi preparado com LaTeX. Os Makefiles, arquivos de estilo e a maioria dos scripts necessários estão disponíveis sob a licença BSD em `github.com/armandofox/latex2ebook`.
Arthur Klepchukov projetou as artes de capa e contracapa para todas as versões.

Publisher's Cataloging-in-Publication

Names: Patterson, David A. | Waterman, Andrew, 1986-
Title: The RISC-V reader: an open architecture atlas / David Patterson and Andrew Waterman.
Description: First edition. | [Berkeley, California] : Strawberry Canyon LLC, 2017. |
Includes bibliographical references and index.
Identifiers: ISBN 978-0-9992491-1-6
Subjects: LCSH: Computer architecture. | RISC microprocessors. |
Assembly languages (Electronic computers)
Classification: LCC QA76.9.A73 P38 2017 | DDC 004.22- -dc23

Dedicatórias

David Patterson dedica este livro a seus pais:

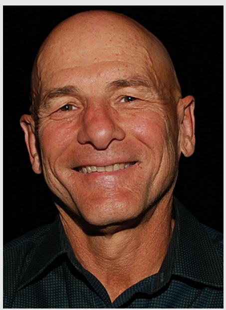
—Para meu pai, David, de quem eu herdei criatividade, atletismo e coragem para lutar pelo que é certo; e

—Para minha mãe Lucie, de quem herdei inteligência, otimismo e meu temperamento.

Obrigado por serem ótimos modelos, os quais me ensinaram o que significa ser um bom cônjuge, pai e avô.

Andrew Waterman dedica este livro a seus pais, John e Elizabeth, os quais o apoiam muito, mesmo a milhares de quilômetros de distância.

Sobre os Autores



David Patterson aposentou-se depois de 40 anos como professor de Ciência da Computação na UC Berkeley em 2016 e, em seguida, juntou-se ao Google Brain como *distinguished engineer*. Ele também atua como vice-presidente do Conselho de Administração da Fundação RISC-V. No passado, ele foi nomeado diretor da Divisão de Ciência da Computação de Berkeley e eleito membro da Computing Research Association e Presidente da Association for Computing Machinery. Na década de 1980, ele liderou quatro gerações de projetos RISC (Reduced Instruction Set Computer), que inspiraram o mais recente RISC de Berkeley a ser chamado de "RISC Five". Juntamente com Andrew Waterman, David foi um dos quatro arquitetos do RISC-V. Além de RISC, seus projetos de pesquisa mais conhecidos são *Redundant Arrays of Inexpensive Disks* (RAID) e *Network of Workstations* (NOW). Esta pesquisa resultou a muitos papers, 7 livros, e mais de 35 honrarias, incluindo a eleição para a National Academy of Engineering, a National Academy of Sciences e ao Silicon Valley Engineering Hall of Fame, além de ser nomeado um membro do Computer History Museum, ACM, IEEE e ambas as organizações AAAS. Seus prêmios na área de ensino incluem o distinto prêmio de ensino (UC Berkeley), o prémio educador destaque de Karlstrom (ACM), a Medalha de Educação Mulligan (IEEE) e o Prêmio de Ensino de Graduação (IEEE). Também ganhou o Textbook Excellence Awards ("Texty") da Text and Academic Authors Association por seus livros sobre arquitetura de computadores e engenharia de software. Ele recebeu todos os seus diplomas acadêmicos da UCLA, que lhe concedeu o prêmio Outstanding Engineering Academic Alumni Award. David cresceu no sul da Califórnia, e por diversão joga futebol e pedala com seus filhos e caminha na praia com sua esposa. Originalmente namorados de escola, eles celebraram seu 50º aniversário de casamento alguns dias após a publicação da edição Beta.

Andrew Waterman atua como chief engineer e co-fundador da SiFive. A SiFive foi fundada pelos criadores da arquitetura RISC-V para fornecer chips personalizados de baixo custo baseados em RISC-V. Andrew recebeu seu PhD em Ciência da Computação pela UC Berkeley, onde, cansado dos caprichos das arquiteturas dos conjuntos de instruções existentes, co-projetou a ISA e o primeiro microprocessador RISC-V. Ele é um dos principais contribuidores para o software open-source Rocket para criar chips RISC-V, a linguagem de construção de hardware Chisel, e o port RISC-V para o kernel do sistema operacional Linux e para o compilador e biblioteca C. Ele também tem um mestrado pela UC Berkeley, trabalho que foi a base para a extensão RVC do RISC-V, e um grau de bacharel em engenharia pela Duke University.

Sobre os Tradutores

O RISC-V fornece um novo panorama em questão de ISAs: muito mais clareza e simplicidade na abordagem se comparado com suas alternativas, como X86 e ARM. Além disso, é open-source. Em 2018, estas características atraíram bastante os estudantes da disciplina de Sistemas Microprocessados dos Cursos de Engenharia de Computação e Automação da Universidade Federal do Rio Grande. Nessa turma, a ISA base RISC-V (RV32I) e a extensão vetorial (RV32V) serviram como plataformas no ensino de tópicos como paralelismo através de pipelines e arquiteturas vetoriais. Ao final, percebeu-se o quão útil seria para as atividades uma versão deste livro em português. Assim começou, no final de 2018, o projeto que hoje resultou na presente tradução.

Luiz Gustavo Xavier (à direita na foto) é bolsista de iniciação científica CNPq. Tem interesse no desenvolvimento de aplicações distribuídas e paralelas, tolerância a falhas e computação gráfica. **Nathan Formentin** (ao centro na foto) tem interesse em ciência de dados e suas aplicações em diversos campos, como economia, esportes e biologia. **Marcelo Pias** (à esquerda na foto) tem interesse em *deep learning* para aplicações embarcadas, trabalhando na disseminação do RISC-V no Brasil. Marcelo é egresso da primeira turma de Engenharia de Computação da FURG, obteve PhD pela University College London (UCL) e fez pós-doutorado na Universidade de Cambridge. Trabalhou em laboratórios de pesquisa de empresas como AT&T/British Telecom Labs e Intel. Atualmente, é Professor Adjunto na Universidade Federal do Rio Grande (FURG). A FURG faz parte da Fundação RISC-V [1], cuja sede está em Berkeley, California (EUA).

[1] https://riscv.org/membership/4837/furg/

Guia Rápido

	Cartão de Referência RISC-V	**i**
	Lista de Figuras	**ix**
	Prefácio	**xiv**
1	**Por que RISC-V?**	**2**
2	**RV32I: ISA RISC-V Base para Números Inteiros**	**16**
3	**Linguagem Assembly do RISC-V**	**34**
4	**RV32M: Multiplicação e Divisão**	**48**
5	**RV32F e RV32D: Ponto Flutuante de Precisão Simples e Dupla**	**52**
6	**RV32A: Instruções Atômicas**	**64**
7	**RV32C: Instruções Compactadas**	**68**
8	**RV32V: Vetores**	**76**
9	**RV64: Instruções de Endereço de 64 bits**	**90**
10	**Arquitetura Privilegiada RV32/64**	**104**
11	**Futuras Extensões Opcionais do RISC-V**	**124**
Apêndice A	**Listagem de Instruções RISC-V**	**128**
Apêndice B	**Transliteração do RISC-V**	**178**
	Índice	**184**

Sumário

Lista de Figuras **xiii**

Prefácio **xiv**

1 Por que RISC-V? **2**
1.1 Introdução . . . 2
1.2 ISAs Modulares vs. Incrementais . . . 3
1.3 ISA Design 101 . . . 5
1.4 Uma Visão Geral deste Livro . . . 11
1.5 Considerações Finais . . . 13
1.6 Para Saber Mais . . . 14

2 RV32I: ISA RISC-V Base para Números Inteiros **16**
2.1 Introdução . . . 16
2.2 Formatos de instrução RV32I . . . 16
2.3 Registradores RV32I . . . 20
2.4 Computação de Inteiros em RV32I . . . 20
2.5 Loads e Stores em RV32I . . . 23
2.6 Desvio Condicional em RV32I . . . 24
2.7 Salto Incondicional em RV32I . . . 25
2.8 RV32I: Informações diversas . . . 25
2.9 Comparando RV32I, ARM-32, MIPS-32 e x86-32 . . . 26
2.10 Considerações Finais . . . 27
2.11 Para Saber Mais . . . 29

3 Linguagem Assembly do RISC-V **34**
3.1 Introdução . . . 34
3.2 Convenção de Chamada . . . 34
3.3 Assembly . . . 37
3.4 Linker . . . 42
3.5 Linkagem Estática vs. Linkagem Dinâmica . . . 45
3.6 Loader . . . 45
3.7 Considerações Finais . . . 46
3.8 Para Saber Mais . . . 46

4 RV32M: Multiplicação e Divisão **48**
4.1 Introdução 48
4.2 Considerações Finais 50
4.3 Para Saber Mais 50

5 RV32F e RV32D: Ponto Flutuante de Precisão Simples e Dupla **52**
5.1 Introdução 52
5.2 Registradores de Ponto Flutuante 52
5.3 Floating-Point Loads, Stores, and Arithmetic 57
5.4 Conversão e Movimentação de Ponto Flutuante 58
5.5 Instruções de Ponto Flutuante Diversas 58
5.6 Comparando RV32FD, ARM-32, MIPS-32 e x86-32 usando DAXPY 59
5.7 Considerações Finais 60
5.8 Para Saber Mais 60

6 RV32A: Instruções Atômicas **64**
6.1 Introdução 64
6.2 Considerações Finais 67
6.3 Para Saber Mais 67

7 RV32C: Instruções Compactadas **68**
7.1 Introdução 68
7.2 Comparando RV32GC, Thumb-2, microMIPS e x86-32 70
7.3 Considerações finais 71
7.4 Para Saber Mais 71

8 RV32V: Vetores **76**
8.1 Introdução 76
8.2 Instruções de Computação Vetorial 77
8.3 Registradores Vetoriais e Tipagem Dinâmica 78
8.4 Operações Vetoriais Load e Store 79
8.5 Paralelismo Durante a Execução Vetorial 80
8.6 Execução Condicional de Operações Vetoriais 81
8.7 Instruções de Vetores Diversas 81
8.8 Exemplo de Vetores: DAXPY em RV32V 82
8.9 Comparando RV32V, MIPS-32 MSA SIMD e x86-32 AVX SIMD 84
8.10 Considerações Finais 85
8.11 Para Saber Mais 87

9 RV64: Instruções de Endereço de 64 bits **90**
9.1 Introdução 90
9.2 Comparação com outras ISAs de 64 bits usando Ordenação por Inserção . . . 94
9.3 Tamanho do Programa 96
9.4 Considerações Finais 96
9.5 Para Saber Mais 98

10 Arquitetura Privilegiada RV32/64 104
10.1 Introdução 104
10.2 Modo de Máquina para Sistemas Embarcados Simples 105
10.3 Manipulação de Exceção em Modo de Máquina 107
10.4 Modo de Usuário e Isolamento de Processo em Sistemas Embarcados 112
10.5 Modo de Supervisor para Sistemas Operacionais Modernos 113
10.6 Memória Virtual Baseada em Páginas 115
10.7 CSRs de Identificação e Desempenho 119
10.8 Considerações Finais 120
10.9 Para Saber Mais 122

11 Futuras Extensões Opcionais do RISC-V 124
11.1 "B" Extensão Padrão para Manipulação de Bits 124
11.2 "E" Extensão Padrão para Embarcados 124
11.3 "H" Extensão de Arquitetura Privilegiada para Hypervisor Support 124
11.4 "J" Extensão Padrão para Linguagens Traduzidas Dinamicamente 124
11.5 "L" Extensão Padrão para Ponto Flutuante Decimal 124
11.6 "N" Extensão Padrão para Interrupções a Nível de Usuário 125
11.7 "P" Extensão Padrão para Instruções Packed-SIMD 125
11.8 "Q" Extensão Padrão para Ponto Flutuante de Precisão Quádrupla 125
11.9 Considerações Finais 125

A Listagem de Instruções RISC-V 128

B Transliteração do RISC-V 178
B.1 Introdução 178
B.2 Comparando RV32I, ARM-32 e x86-32 usando o algoritmo de soma de árvore 180
B.3 Conclusão 181

Índice 184

Lista de Figuras

1.1 Os membros corporativos da Fundação RISC-V 3
1.2 Crescimento do conjunto de instruções x86 ao longo de sua vida. 4
1.3 Descrição da instrução x86-32 *ASCII Adjust after Addition* (`aaa`). 4
1.4 Um wafer de 8 polegadas de diâmetro de *dies*(recortes) RISC-V projetado pela SiFive. 6
1.5 Tamanhos relativos de programa para RV32G, ARM-32, x86-32, RV32C e Thumb-2. 10
1.6 Número de páginas e palavras dos manuais ISA 13

2.1 Diagrama das instruções do RV32I. 17
2.2 Formatos de instrução RV32I 17
2.3 O mapa de opcode RV32I possui layout de instrução, opcodes, tipo de formato e nomes. 18
2.4 Os registradores do RV32I. 21
2.5 Ordenação por inserção em C. 26
2.6 Número de instruções e tamanho do código para Ordenação por Inserção para esses ISAs. 26
2.7 Lições que os arquitetos RISC-V aprenderam com os erros de ISAs do passado. 28
2.8 Código RV32I do algoritmo de Ordenação por Inserção na Figura 2.5. 30
2.9 Código ARM-32 do algoritmo de Ordenação por Inserção na Figura 2.5. . . . 31
2.10 Código MIPS-32 do algoritmo de Ordenação por Inserção MIPS-32 na Figura 2.5. 32
2.11 Código x86-32 do algoritmo de Ordenação por Inserção na Figura 2.5. 33

3.1 Passos de tradução de um código fonte C para um programa em execução. . . 35
3.2 Mnemônicos do *Assembler* para registradores inteiros e de ponto flutuante do RISC-V. 36
3.3 32 pseudoinstruções RISC-V que utilizam o x0, o registrador zero. 38
3.4 28 pseudoinstruções RISC-V que são independentes de x0, o registrador zero. 39
3.5 Programa Hello World escrito em C (`hello.c`). 40
3.6 Programa Hello World na linguagem assembly do RISC-V (`hello.s`). 40
3.7 Programa Hello World na linguagem de máquina do RISC-V (`hello.o`). . . 41
3.8 Programa Hello World na linguagem de máquina do RISC-V após linkagem. 41
3.9 Diretivas assembler comuns do RISC-V. 43

3.10 Alocação de memória para programa e dados do RV32I. 44

4.1 Diagrama das instruções RV32M. 48
4.2 O mapa de opcode da RV32M possui layout de instruções, opcodes, tipo de formatos e nomes. 49
4.3 Código RV32M para dividir por uma constante multiplicando. 49

5.1 Diagrama das instruções RV32F e RV32D. 53
5.2 Mapa de opcode RV32F contém layout de instrução, opcodes, tipo de formato e nomes. 54
5.3 Mapa de opcode RV32D contém layout de instrução, opcodes, tipo de formato e nomes. 55
5.4 Os registradores de ponto flutuante de RV32F e RV32D. 56
5.5 Controle de ponto flutuante e registrador de status. 57
5.6 Instruções de conversão RV32F e RV32D. 58
5.7 O programa DAXPY de ponto flutuante em C. 59
5.8 Número de instruções e tamanho do código de DAXPY para quatro ISAs. . . 59
5.9 Código RV32D para DAXPY da Figura 5.7. 61
5.10 Código ARM-32 para DAXPY da Figura 5.7. 61
5.11 Código MIPS-32 para DAXPY da Figura 5.7. 62
5.12 Código x86-32 para DAXPY da Figura 5.7. 62

6.1 Diagrama das instruções do RV32A. 65
6.2 O mapa de opcode RV32A tem layout de instrução, opcodes, tipo de formato e nomes. 65
6.3 Dois exemplos de sincronização. 67

7.1 Diagrama das instruções do RV32C. 69
7.2 Instruções e tamanho do código para Ordenação por Inserção e DAXPY para ISAs compactados. 70
7.3 Código RV32C para Ordenação por Inserção. 72
7.4 Código RV32DC para DAXPY. 73
7.5 O mapa de opcode RV32C (bits$[1:0]$ = 01) lista layout, opcodes, formatos e nomes. 73
7.6 O mapa de opcode RV32C (bits$[1:0]$ = 00) lista layout, opcodes, format e nomes. 74
7.7 O mapa do opcode RV32C (bits$[1:0]$ = 10) lista layout, opcodes, format e nomes. 74
7.8 Formatos de instrução compactadas RVC de 16 bits. 75

8.1 Diagrama das instruções do RV32V. 77
8.2 Codificações RV32V de tipos de registradores vetoriais. 79
8.3 Código RV32V para DAXPY na Figura 5.7. 83
8.4 Número de instruções e tamanho do código de DAXPY para ISAs vetoriais. . 85
8.5 Código MIPS-32 MSA para DAXPY na Figura 5.7. 88
8.6 Código x86-32 AVX2 para DAXPY na Figura 5.7. 89

9.1 Diagrama das instruções do RV64I. 91

9.2 Diagramas das instruções RV64M e RV64A. 91
9.3 Diagrama das instruções RV64F e RV64D. 92
9.4 Diagrama das instruções do RV64C. 92
9.5 Mapa de opcode RV64 das instruções base e extensões opcionais. 93
9.6 Número de instruções e tamanho do código para a classificação de inserção para quatro ISAs. 96
9.7 Tamanhos de programa relativos para RV64G, ARM-64 e x86-64 em comparação com o RV64GC. 97
9.8 Código RV64I para Ordenação por Inserção na Figura 2.5. 99
9.9 Código ARM-64 para Ordenação por Inserção na Figura 2.5. 100
9.10 Código MIPS-64 para Ordenação por Inserção na figura 2.5. 101
9.11 Código x86-64 para Ordenação por Inserção na Figure 2.5. 102

10.1 Diagrama das instruções privilegiadas do RISC-V. 104
10.2 Layout de instruções privilegiadas RISC-V, opcodes, tipo de formato e nome. 105
10.3 Exceções RISC-V e causas de interrupção. 106
10.4 O CSR `mstatus`. 108
10.5 CSRs de Interrupção de Máquina. 108
10.6 Máquina e supervisor causam as CSRs (`mcause` e `scause`). 108
10.7 CSRs de endereço base do vetor de interrupção trap de máquina e supervisor (`mtvec` e `stvec`). 108
10.8 CSRs associados a exceções e interrupções. 109
10.9 Níveis de privilégios do RISC-V e suas codificações. 109
10.10 Código RISC-V para um tratador de interrupção de temporizador simples. . . 111
10.11 Um endereço PMP e registrador de configuração. 112
10.12 O layout das configurações PMP nos CSRs de `pmpcfg`. 113
10.13 Os CSRs delegados. 114
10.14 CSRs supervisores de interrupção 114
10.15 A CSR `sstatus` 115
10.16 Uma entrada da tabela de páginas do RV32 Sv32 (PTE,, ou Page-Table Entry). 116
10.17 Uma entrada de tabela de páginas RV64 Sv39 (PTE, ou Page-Table Entry). . 117
10.18 A CSR `satp`. 117
10.19 A codificação do campo MODE no CSR `satp`. 118
10.20 Diagrama do processo de tradução de endereços do Sv32. 119
10.21 O CSR de ISA de máquina `misa` relata o ISA suportado. 120
10.22 A CSR `mvendorid` fornece o ID do fabricante JEDEC do núcleo. 121
10.23 Os CSRs de identificação de máquina (`marchid` , `mimpid`, `mhartid`) 121
10.24 Os CSRs de tempo de máquina(`mtime` and `mtimecmp`) medem o tempo . . . 121
10.25 Os registradores de ativação reversa `mcounteren` e `scounteren` 121
10.26 Os CSRs de monitoramento de performance do hardware 121
10.27 O algoritmo completo para tradução de endereços virtuais para físicos. 122

B.1 Instruções de acesso à memória RV32I transliteradas para ARM-32 e x86-32. 178
B.2 Instruções aritméticas RV32I transliteradas em ARM-32 e x86-32. 179
B.3 Instruções de fluxo de controle RV32I transliteradas em ARM-32 e x86-32. . 180
B.4 Uma rotina em C que soma os valores em uma árvore binária, usando uma travessia em ordem. 182

B.5 Código RV32I para percorrer da árvore em ordem. 182
B.6 Código ARM-32 para passagem de árvore em ordem. 183
B.7 Código x86-32 para percorrer a árvore em ordem. 183

Prefácio

Bem Vindo!

O RISC-V tem sido um fenômeno, crescendo rapidamente em popularidade desde sua introdução em 2011. Pensamos que um guia de programação compacto ajudaria os recém-chegados a entender por que RISC-V é um conjunto de instruções atraente, e também verem como ele difere-se das ISAs do passado. Livros para outras ISAs nos inspiraram, embora esperássemos que a simplicidade do RISC-V significaria escrever muito menos do que as mais de 500 páginas de bons livros, como *See MIPS Run*. Em um terço da duração total, pelo menos por essa medida, conseguimos. Na realidade, os dez capítulos que apresentam cada componente do conjunto modular de instruções RISC-V ocupam apenas 100 páginas - apesar da média de quase uma figura por página (75 no total) - o que facilita a leitura. Depois de explicar os princípios do projeto de conjuntos de instruções, mostramos como os arquitetos RISC-V aprenderam com os conjuntos de instruções dos últimos 40 anos para tomar emprestadas suas boas ideias e evitar seus erros. As ISAs são julgadas tanto pelo que é omitido quanto pelo que está incluso. Em seguida, introduzimos cada componente dessa arquitetura modular em uma sequência de capítulos. Cada capítulo tem um programa na linguagem assembly RISC-V que demonstra o uso das instruções apresentadas no capítulo, o que torna mais fácil para o programador assembly aprender código RISC-V. Também mostramos programas equivalentes em ARM, MIPS e x86 que destacam os benefícios de simplicidade e custo-energia-desempenho do RISC-V.

Para tornar a leitura mais divertida, incluímos quase 50 barras laterais nas margens da página com o que acreditamos serem comentários interessantes sobre o texto. Também incluímos cerca de 75 imagens nas margens para enfatizar características de um bom projeto ISA. (Nossas margens são bem utilizadas!) Finalmente, para o leitor dedicado, acrescentamos cerca de 25 elaborações ao longo do texto. Você pode se aprofundar nessas seções opcionais se estiver interessado em um tópico específico. Essas seções não são obrigatórias para entender o outro material do livro, portanto sinta-se à vontade para ignorá-las se elas não capturarem o seu interesse. Para os aficionados por arquitetura de computadores, citamos 25 artigos e livros que podem ampliar seus horizontes. Aprendemos muito lendo-os para escrever este livro!

Por que tantas Citações?

Achamos que as citações também tornam o livro mais divertido de ler, por isso, distribuímos 25 delas em todo o texto. Essas também são um mecanismo eficiente para transmitir a sabedoria dos mais antigos aos novatos, e ajudar a estabelecer padrões culturais para um bom projeto de ISA. Queremos que os leitores também captem um pouco da história da área, e é por isso que apresentamos citações de cientistas e engenheiros da computação famosos em todo o texto.

Introdução e Referência

Desejamos que este livro compacto funcione como uma introdução e uma referência ao RISC-V para estudantes e programadores de sistemas embarcados interessados em escrever código RISC-V. Este livro pressupõe que os leitores tenham visto previamente pelo menos um conjunto de instruções. Caso contrário, talvez você queira consultar em nosso livro de arquitetura de introdução com base no RISC-V: *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*.

As referências compactas neste livro incluem:

- **Cartão de Referência** – Esta descrição condensada de uma página (frente e verso) do RISC-V cobre RV32GCV e RV64GCV, que inclui a base e todas as extensões definidas: RVI, RVM, RVA, RVF, RVD, RVC e até RVV, embora ainda esteja sob desenvolvimento.

- **Diagramas de Instrução** – Essas descrições gráficas de meia página de cada extensão de instruções, que são as primeiras figuras dos capítulos, listam os nomes completos de todas as instruções RISC-V em um formato que permite ver facilmente as variações de cada instrução. Veja as Figuras 2.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1, 9.2, 9.3, e 9.4.

- **Mapas de opcode** – Essas tabelas mostram o layout da instrução, os opcodes, o tipo de formato e o mnemônico de instrução para cada extensão de instrução em uma fração de uma página. Veja as Figuras 2.3, 3.3, 3.4, 4.2, 5.2, 5.3, 6.2, 7.6, 7.5, 7.7, 9.5, e 10.1. (Os diagramas de instruções e os mapas opcode inspiraram o uso da palavra atlas no subtítulo do livro.)

- **Glossário de instruções** – O Apêndice A é uma descrição completa de cada instrução e pseudoinstrução RISC-V.[2] Inclui tudo: o nome da operação e operandos, uma descrição em português, uma definição de linguagem de transferência de registrador, qual extensão RISC-V ela está contida, o nome completo da instrução, o formato da instrução, um diagrama da instrução mostrando os opcodes e referências a versões compactas da instrução. Surpreendentemente, tudo isso cabe em menos de 50 páginas.

- **Tradutor de Instrução** –O Apêndice B ajuda programadores experientes em linguagem assembly, fornecendo tabelas que mostram as instruções ARM-32 ou x86-32 que são equivalentes às instruções RV32I. Esse também lista as saídas do compilador C para um programa simples de travessia de árvore para essas três arquiteturas e descreve

[2] O comitê que define RV32V não concluiu seu trabalho a tempo para a edição Beta, então omitimos essas instruções do Apêndice A. O Capítulo 8 é nossa melhor suposição sobre o que RV32V será, embora seja provável que mude um pouco.

as surpreendentemente pequenas diferenças entre elas. Ele conclui com dicas sobre como traduzir o código das arquiteturas mais antigas para o RISC-V, o que é mais fácil do que se imagina.

- **Índice** – Ele ajuda a encontrar a página que descreve a explicação, a definição ou o diagrama das instruções, seja pelo nome completo ou pelo mnemônico. Está organizado como um dicionário.

Errata e Conteúdo Suplementar

Pretendemos coletar a Errata juntos e liberar atualizações algumas vezes por ano. O site do livro mostra a última versão e uma breve descrição das mudanças desde a versão anterior. As erratas anteriores podem ser revisadas, e novas disponibilizadas, no website do livro (www.riscvbook.com). Pedimos desculpas antecipadamente pelos problemas que você encontra nesta edição e aguardamos seus comentários sobre como melhorar esse material.

A História desse Livro

No VI Workshop do RISC-V realizado de 8 a 11 de maio de 2017, em Xangai, vimos a necessidade de tal livro. Nós começamos algumas semanas depois. Dada a experiência muito maior de Patterson em escrever livros, o plano era que ele escrevesse a maioria dos capítulos. Nós dois colaboramos na organização e fomos os primeiros revisores dos capítulos uns dos outros. Patterson escreveu os Capítulos 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, o cartão de referência, e este Prefácio, enquanto Waterman escreveu 10, o Apêndice A—a maior seção do livro— o Apêndice B, e codificou todos os programas no livro. Waterman também manteve o pipeline LaTeX da Armando Fox que nos permitiu produzir o livro.

Oferecemos uma edição Beta do livro didático para 800 alunos da UC Berkeley no semestre de outono de 2017. Os leitores encontraram apenas alguns erros de digitação e erros no LaTeX, que nós corrigimos para a primeira edição. Também aprimoramos os ícones de margem para torná-los mais fáceis de lembrar e revisamos algumas figuras que não pareciam tão boas na página impressa quanto esperávamos.

Mais significativamente, a primeira edição expandiu o Capítulo 10 para incluir mais de 60 registradores de Controle e Status e adicionou o Apêndice B para ajudar os programadores interessados em converter programas de linguagem assembly das ISAs mais antigos em RISC -V.

A primeira edição foi publicada a tempo de estar disponível no Sétimo Workshop RISC no Vale do Silício, de 28 a 30 de novembro de 2017.

RISC-V foi um subproduto de um Projeto de pesquisa de Berkeley[1] que estava desenvolvendo tecnologia para facilitar a construção de hardware e software paralelos.

Agradecimentos

Gostaríamos de agradecer a Armando Fox pelo uso de seu pipeline LaTeX e conselhos sobre como navegar no mundo da auto-publicação.

Nossos mais sinceros agradecimentos vão para as pessoas que leram os primeiros rascunhos desse livro e ofereceram sugestões úteis: Krste Asanovi c, Nikhil Athreya, C. Gordon

Bell, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor e Megan Wachs.

Finalmente, agradecemos aos muitos alunos da UC Berkeley por sua ajuda de depuração e seu contínuo interesse neste material!

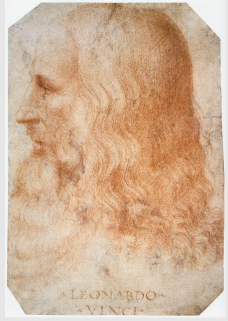
David Patterson e Andrew Waterman
16 de Novembro, 2017
Berkeley, California

Notes

[1] `http://parlab.eecs.berkeley.edu`

1 Por que RISC-V?

Leonardo da Vinci (1452-1519) foi um arquiteto, engenheiro, escultor renascentista e pintor da Mona Lisa.



Simplicity is the ultimate sophistication.

—Leonardo da Vinci

1.1 Introdução

O objetivo do RISC-V (“RISC five”) é tornar-se uma arquitetura de conjunto de instruções universal (*Instruction Set Architecture—ISA*). Para isso, ele deve satisfazer alguns requisitos:

- Atender a todos os tamanhos de processadores, desde o minúsculo controlador embarcado até o computador de alto desempenho mais rápido.
- Funcionar bem com uma grande variedade de softwares e linguagens de programação populares.
- Acomodar todas as tecnologias de implementação: FPGAs (Field-Programmable Gate Arrays), ASICs (Application-Specific Integrated Circuits), chips customizados e até mesmo futuras tecnologias de dispositivos.
- Ser eficiente para todos os tipos de microarquitetura: controle microcodificado ou hardwired; pipelines em ordem, desacoplados ou desordenados; emissão de instrução única ou superescalar; e assim por diante.
- Apoiar ampla especialização para atuar como uma base para aceleradores customizados, que aumentam em importância à medida que a Lei de Moore se desvanece.
- Ser estável, ou seja, a ISA base não deve mudar. Mais importante, a ISA não pode ser descontinuada, como aconteceu no passado com as ISAs proprietárias, como a AMD Am29000, a Digital Alpha, a Digital VAX, o Hewlett Packard PA-RISC, Intel i860, Intel i960, Motorola 88000, e a Zilog Z8000.

O RISC-V é incomum não apenas porque é uma ISA recente—nascida nesta década quando a maioria das alternativas data dos anos 1970 ou 1980—mas também porque é uma ISA *aberta*. Ao contrário de praticamente todas as arquiteturas anteriores, seu futuro é livre do destino ou dos caprichos de qualquer empresa, que condenou muitas ISAs no passado.

>$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		

Figura 1.1: Os membros corporativos da Fundação RISC-V de acordo com o sexto Workshop RISC-V em maio de 2017, classificado por vendas anuais. As empresas da coluna da esquerda excedem as vendas anuais de $US 50B, as empresas da coluna intermediária vendem menos de $US 50B, mas mais de $US 5B, e as vendas da coluna da direita são menores que $US 5B, mas de ordem maior que $US 0.5B. A fundação inclui outras 25 empresas menores, 5 empresas iniciantes (Antmicro Ltd, Blockstream, Esperanto Technologies, Greenwaves Technologies e SiFive), 4 organizações sem fins lucrativos (CSEM, Draper Laboratory, ICT e lowRISC) e 6 universidades (ETH Zurich, IIT Madras, National University of Defense Technology, Princeton e UC Berkeley). A maioria das 60 organizações tem sua sede fora dos EUA. Para saber mais, acesse www.riscv.org.

Adicionamos barras laterais às margens para oferecer comentários relevantes. Por exemplo, o RISC-V foi originalmente desenvolvido para uso interno em pesquisas e cursos da UC Berkeley. Tornou-se aberto porque as pessoas de fora começaram a usá-lo por conta própria. Os arquitetos do RISC-V aprenderam sobre o interesse externo quando começaram a receber reclamações sobre as mudanças da ISA em seu curso, que estava na web. Somente depois que os arquitetos entenderam a necessidade, tentaram torná-lo um padrão aberto de ISA.

Em vez disso, pertence a uma fundação aberta e sem fins lucrativos. O objetivo da RISC-V Foundation é manter a estabilidade do RISC-V, evoluí-lo lenta e cuidadosamente, apenas por razões técnicas, e tentar torná-lo tão popular para o hardware quanto o Linux é para sistemas operacionais. Como sinal de sua vitalidade, a Figura 1.1 lista os maiores membros corporativos da Fundação RISC-V.

1.2 ISAs Modulares vs. Incrementais

A Intel estava apostando seu futuro em um microprocessador de alta qualidade, mas ainda faltavam anos para isso. Para combater a Zilog, a Intel desenvolveu um processador stop-gap e o chamou de 8086. O objetivo era ter vida curta e não ter sucessores, mas não foi assim que as coisas aconteceram. O processador de ponta acabou chegando ao mercado e, quando saiu, ficou muito lento. Então, a arquitetura do 8086 continuou viva - evoluiu para um processador de 32 bits e, eventualmente, para um processador de 64 bits. Os nomes continuavam mudando (80186, 80286, i386, i486, Pentium), mas o conjunto de instruções permaneceu intacto.

—Stephen P. Morse, arquiteto do 8086 [Morse 2017]

A abordagem convencional para arquitetura de computadores é o desenvolvimento de ISAs *incrementais*, onde novos processadores devem implementar não apenas novas extensões ISA, mas também todas as extensões do passado. O objetivo é manter a *compatibilidade binária retroativa* para que versões binárias de programas de décadas possam ser executadas corretamente no processador mais recente. Esse requisito, quando combinado com o apelo comercial de anunciar novas instruções em uma nova geração de processadores, levou a ISAs que crescem substancialmente em tamanho com a idade. Por exemplo, a Figura 1.2 mostra o crescimento no número de instruções para uma ISA dominante hoje: o 80x86. Ela remonta a 1978, mas adicionou aproximadamente *três instruções por mês* ao longo de sua vida útil.

Essa convenção significa que toda implementação do x86-32 (o nome que usamos para a versão de endereço de 32 bits do x86) deve implementar os erros das extensões anteriores, mesmo quando elas não fazem mais sentido. Por exemplo, a Figura 1.3 descreve a instrução *ASCII Adjust after Addition* (`aaa`) do x86, que há muito sobrevive com sua inutilidade.

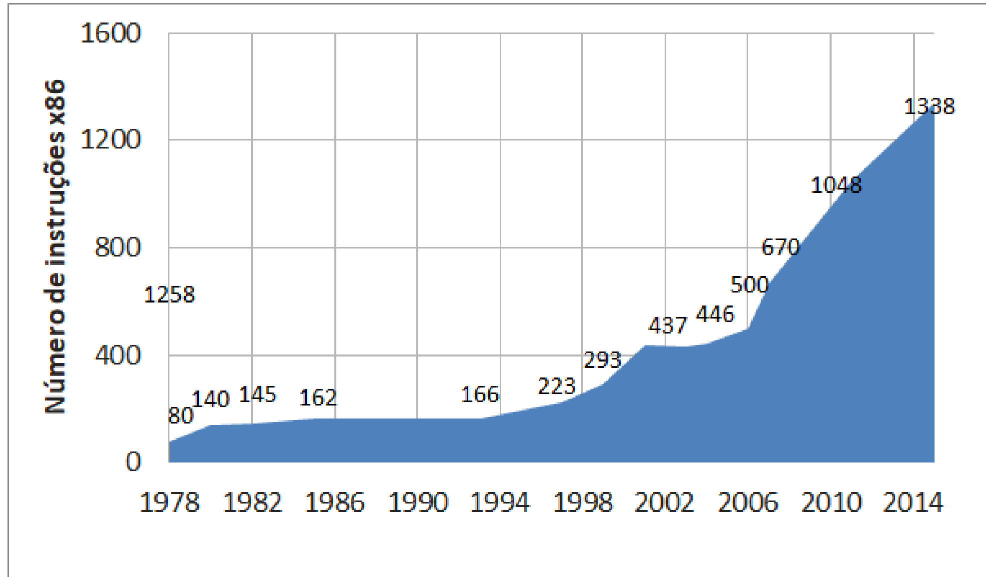


Figura 1.2: Crescimento do conjunto de instruções x86 ao longo de sua vida. O x86 iniciou com 80 instruções em 1978. Aumentou as instruções em uma fração de 16X para 1338 em 2015 e continua crescendo. Surpreendentemente, este gráfico é conservador. Um blog da Intel estima a contagem em 3600 instruções em 2015 [Rodgers and Uhlig 2017], o que aumentaria a taxa do x86 para uma nova instrução *a cada quatro dias* entre 1978 e 2015. Em nosso cálculo contabilizamos somente as instruções de linguagem assembly, e eles possivelmente acrescentaram a contagem de instruções de linguagem de máquina. Como o Capítulo 8 explica, uma grande parte do crescimento é porque a ISA x86 baseia-se em instruções SIMD para o paralelismo em nível de dados.

```
O registrador AL é a origem e o destino padrão.
Se os 4 bits mais baixos do registrador AL forem> 9,
   ou o transportador de flag auxiliar AF = 1,
 Então
    Adicione 6 aos 4 bits mais baixos de AL e descarte o overflow
    Incrementa o byte mais alto de AL
    Carry flag CF = 1
    transportador de flag auxiliar AF = 1
 Senão
    CF = AF = 0
4 bits mais altos de AL = 0
```

Figura 1.3: Descrição da instrução x86-32 *ASCII Adjust after Addition* (`aaa`). Esta executa a aritmética computacional em Binary Coded Decimal (BCD), representação númerica que já caiu na lixeira da história da tecnologia. O x86 também possui três instruções relacionadas para subtração (`aas`), multiplicação (`aam`) e divisão (`aad`). Como cada uma é uma instrução de um byte, elas ocupam coletivamente 1.6 %$(4/256)$ do precioso espaço de opcode.

Como uma analogia, suponha que um restaurante sirva apenas uma refeição a preço fixo, que inicia por um pequeno jantar de hambúrguer e milkshake. Com o tempo, acrescenta-se batatas fritas e depois um *sundae* de sorvete, seguido de salada, torta, vinho, massa vegetariana, bife, cerveja, *ad infinitum* até se tornar um grande banquete. Pode fazer pouco sentido no total, mas os clientes podem encontrar o que já comeram em uma refeição passada naquele restaurante. A má notícia é que os clientes devem pagar o aumento do custo da expansão do banquete para cada jantar.

Além de ser recente e aberto, o RISC-V é incomum, pois, ao contrário de quase todas as ISAs anteriores, é *modular*. No núcleo há um ISA básico, chamado *RV32I*, que executa uma pilha completa de software. O RV32I está congelado e nunca será alterado, o que dá aos criadores de compiladores, desenvolvedores de sistemas operacionais e programadores de linguagem assembly um destino estável. A modularidade vem de extensões padrão opcionais que o hardware pode incluir ou não, dependendo das necessidades da aplicação. Essa modularidade permite implementações muito pequenas e de baixo consumo de energia, que podem ser críticas para aplicativos embarcados. Informando ao compilador RISC-V quais extensões estão incluídas, ele pode gerar o melhor código para esse hardware. A convenção é anexar as letras de extensão ao nome para indicar quais estão incluídas. Por exemplo, o RV32IMFD adiciona as extensões de multiplicação de ponto flutuante (RV32M), de precisão simples (RV32F) e de ponto flutuante de precisão dupla (RV32D) às instruções de base obrigatórias (RV32I).

Se o software usar uma instrução RISC-V omitida de uma extensão opcional, é gerado um alerta e executada a função desejada no software como parte de uma biblioteca padrão.

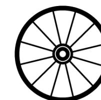
Voltando à nossa analogia, o RISC-V oferece um menu em vez de um buffet; o chef precisa cozinhar apenas o que os clientes querem—não um banquete para cada refeição—e os clientes pagam apenas pelo que pedem. O RISC-V não precisa adicionar instruções simplesmente para o marketing sizzle. A RISC-V Foundation decide quando deve-se adicionar uma nova opção ao menu, e eles o farão apenas por razões técnicas sólidas após uma ampla discussão aberta por um comitê de especialistas em hardware e software. Mesmo quando novas opções aparecem no menu, elas permanecem opcionais e não são um novo requisito para todas as implementações futuras, como acontece com ISAs incrementais.

1.3 ISA Design 101

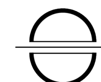
Antes de introduzir a ISA RISC-V, será útil entender os princípios e as escolhas que um arquiteto de computador deve fazer ao projetar uma ISA. Abaixo está uma lista das sete medidas, junto com os ícones que colocaremos nas margens da página para destacar os momentos em que RISC-V as aborda nos capítulos seguintes. (A contracapa do livro impresso tem uma legenda para os ícones.)

- custo (ícone da moeda de dólar)
- simplicidade (roda)
- desempenho (velocímetro)
- isolamento de arquitetura da implementação (metades separadas de um círculo)
- espaço para crescimento (acordeão)
- tamanho do programa (setas opostas)
- facilidade de programação / compilação / "linkagem" ("tão facil quanto o ABC").

Custo


Simplicidade

Desempenho


Isolamento de Arq. da Impl.

Espaço para Crescimento

Tamanho do Programa

Facilidade de Programação

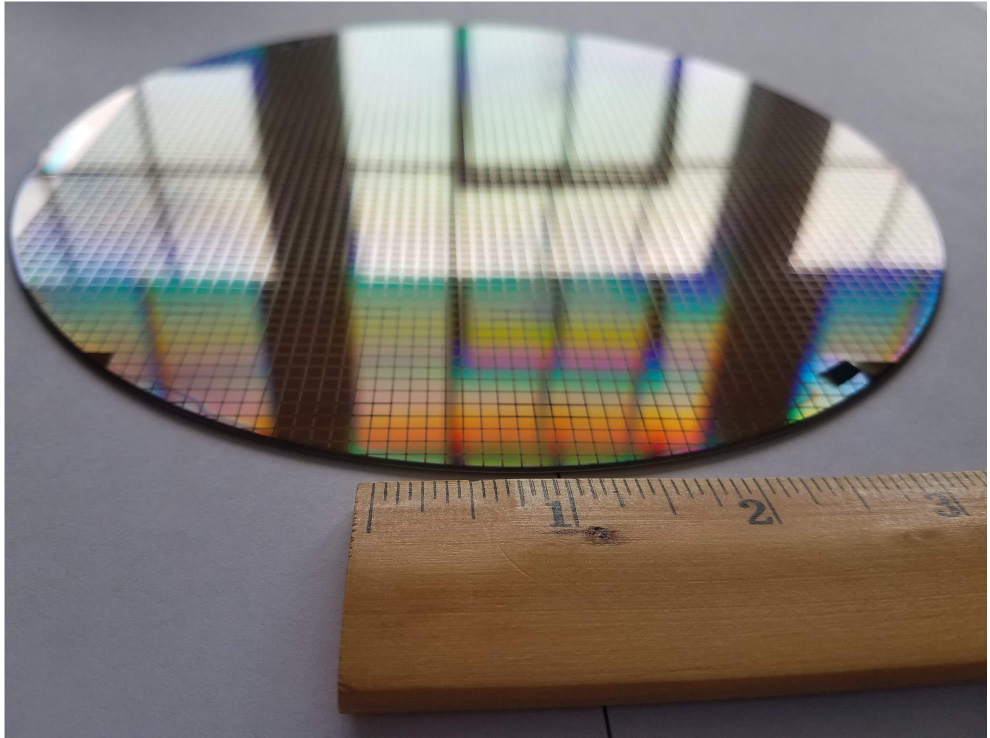


Figura 1.4: Um wafer de 8 polegadas de diâmetro de *dies*(recortes) RISC-V projetado pela SiFive. Possui dois tipos de *dies* RISC-V usando uma linha de processamento maior e mais antiga. Um molde FE310 é 2.65 mm$\times$2.72 mm e um *die* de teste SiFive que é 2.89 mm$\times$2.72 mm. O wafer contém 1846 do primeiro e 1866 do último, totalizando 3712 chips.

Para ilustrar o que queremos dizer, nesta seção mostraremos algumas escolhas feitas em ISAs mais antigas que parecem imprudentes ao analisarmos em retrospectiva, e em contrapartida, onde o RISC-V tomou decisões muito melhores.

Custo. Os processadores são implementados como circuitos integrados, comumente chamados de *chips* ou *dies*(recortes). Eles são chamados de *dies* porque iniciam como um pedaço de um único wafer redondo, que é *recortado* em muitas peças individuais. A Figura 1.4 mostra um wafer de processadores RISC-V. O custo é muito sensível à área dos *dies*:

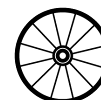
$$custo \approx f(die\ area^2)$$

Obviamente, quanto menor o *die*, mais *dies* são fabricados por wafer e a maior parte do custo do chip é a própria wafer processada. Menos óbvio é que quanto menor o *die*, maior o rendimento da produção, a fração de chips fabricados que funcionam. A razão é que a fabricação de silício resulta em pequenas falhas espalhadas sobre o wafer, então quanto menor a área, menor a fração que será defeituosa.

Processadores high-end podem ganhar desempenho combinando instruções simples, sem sobrecarregar todas as implementações de baixo custo com uma ISA maior e mais complicada. Essa técnica é chamada *macro-fusion*, pois ela combina "macro" instruções.

Um arquiteto deseja manter a ISA simples para reduzir o tamanho dos processadores que o implementam. Como veremos nos próximos capítulos, a ISA RISC-V é muito mais simples que o ISA ARM-32. Como um exemplo concreto do impacto da simplicidade, compararamos um processador RISC-V Rocket a um processador ARM-32 Cortex-A5 na mesma tecnologia (TSMC40GPLUS) usando as caches de mesmo tamanho (16 KiB). O *die* RISC-V é 0.27 mm^2 versus 0.53 mm^2 para o ARM-32. Em torno de duas vezes a área, o ARM-32 Cortex-A5 custa aproximadamente 4X (2^2) mais que o RISC-V Rocket. Mesmo um *die* menor em 10% reduz o custo por um fator de 1,2 ($1,1^2$).

Simplicidade. Dada a sensibilidade do custo à complexidade, os arquitetos desejam uma ISA simples para reduzir a área de impressão. A simplicidade também reduz os tempos de projeto e verificação de chips, o que pode representar muito do custo de desenvolvimento do chip. Esses custos devem ser adicionados ao custo final do produto, dependendo do número de chips vendidos. A simplicidade também reduz o custo da documentação e a dificuldade de fazer com que os clientes entendam como usar o ISA.


Simplicidade

Abaixo encontra-se um exemplo da complexidade do ISA ARM-32:

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

A instrução significa *LoaD Multiple, Increment-Address, on EQual*. Ela executa 5 cargas de dados e escreve em 6 registradores, mas executa somente se a condição EQ for satisfeita. Além disso, ela escreve um dos resultados no PC, por isso a instrução também executa um desvio condicional. Quanta utilidade!

Um processador simples pode ser útil para aplicativos embarcados já que é mais fácil prever o tempo de execução. Os programadores de linguagem assembly de micro-controladores geralmente querem manter o controle de tempo exato, então eles confiam no código que leva um número previsível de ciclos de clock que eles podem contar manualmente.

Ironicamente, as instruções simples são muito mais prováveis de serem usadas do que as complexas. Por exemplo, o x86-32 inclui uma instrução `enter`, que deveria ser a primeira instrução executada ao entrar em um procedimento para criar um quadro de pilha (*stack frame*)(veja o Capítulo 3). A maioria dos compiladores usa apenas estas duas instruções simples do x86-32:

```
push ebp      # Coloca o ponteiro de quadros na pilha
mov  ebp, esp # Copia o ponteiro da pilha para o ponteiro do quadro
```

Desempenho. Com exceção dos pequenos chips para aplicações embarcadas, os arquitetos normalmente se preocupam com desempenho e custo. O desempenho pode ser consider-

Desempenho

ado em três termos:

$$\frac{instrucoes}{programa} \times \frac{media\ ciclos\ clock}{instrucao} \times \frac{tempo}{ciclo\ clock} = \frac{tempo}{programa}$$

O último termo é o inverso da taxa de clock, então uma taxa de clock de 1 GHz significa que o tempo por ciclo de clock é de 1 ns ($1/10^9$).

Mesmo que uma ISA simples possa executar mais instruções por programa do que uma ISA complexa, ela pode compensar isso com um ciclo de clock mais rápido ou uma média menor de ciclos de clock por instrução (CPI).

Por exemplo, para o benchmark CoreMark [Gal-On and Levy 2012] (100.000 iterações), o desempenho no ARM-32 Cortex-A9 é

O número médio de ciclos de clock pode ser menor que 1 porque o A9 e o BOOM [Celio et al. 2015] são chamados de processadores *superscalar*, que executam mais de uma instrução por ciclo de clock.

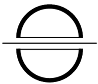
$$\frac{32.27\ B\ instrucoes}{programa} \times \frac{0.79\ ciclos\ clock}{instrucao} \times \frac{0.71\ ns}{ciclos\ clock} = \frac{18.15\ seg}{programa}$$

Para a implementação BOOM do RISC-V, a equação é

$$\frac{29.51\ B\ instrucoes}{programa} \times \frac{0.72\ ciclos\ clock}{instrucao} \times \frac{0.67\ ns}{ciclos\ clock} = \frac{14.26\ seg}{programa}$$

O processador ARM não executou menos instruções do que o RISC-V nesse caso. Como veremos, as instruções simples também são as instruções mais populares, de modo que a simplicidade da ISA pode superar em todas as métricas. Para este programa, o processador RISC-V ganha quase 10% em cada um dos três fatores, o que resulta em uma vantagem de desempenho de quase 30%. Se uma ISA mais simples também resultar em um chip menor, sua fração custo/benefício será excelente.

Isolamento de Arquitetura da Implementação. A distinção original entre *arquitetura* e *implementação*, que remete à década de 1960, é que arquitetura é o que um programador de linguagem assembly precisa saber para escrever um programa correto, mas sem se preocupar com o desempenho desse programa. A tentação de um arquiteto é incluir instruções em um ISA que auxiliem o desempenho ou o custo de uma implementação em um determinado momento, mas sobrecarregue implementações diferentes ou futuras.



Isolamento de Arq. da Impl.

Para a ISA MIPS-32, o exemplo lamentável foi o *delayed branch*. Desvios condicionais causam problemas na execução em pipeline porque o processador deseja ter a próxima instrução a ser executada no pipeline, mas não consegue decidir se deseja a próxima instrução sequencial (se o desvio não for tomado) ou a que está no endereço de destino do desvio (se for tomado). Para seu primeiro microprocessador com um pipeline de 5 estágios, essa indecisão poderia ter causado uma parada no ciclo do pipeline. O MIPS-32 resolveu esse problema redefinindo o desvio para ocorrer na instrução *após* a próxima. Assim, a instrução seguinte é *sempre* executada. A tarefa do programador ou do compilador era colocar algo útil no intervalo de atraso (*delay slot*).

Processadores com pipelines hoje antecipam os resultados dos desvios usando preditores de hardware, que podem exceder a precisão de 90% e trabalhar com qualquer tamanho de pipeline. Eles precisam apenas de um mecanismo para liberar e reiniciar o pipeline quando falham na predição.

Infelizmente, essa "solução" não ajudou mais tarde os processadores MIPS-32 com muito mais estágios de pipeline (com muitas outras instruções buscadas antes do resultado do desvio ser computado), e ainda tornou a vida mais difícil para os programadores do MIPS-32, desenvolvedores de compiladores, e projetistas de processadores, pois ISAs incrementais exigem compatibilidade retroativa (veja Seção 1.2). Além disso, torna o código MIPS-32 muito mais difícil de entender (veja a Figura 2.10 na Página 32).

Enquanto arquitetos não devem colocar recursos que *ajudam* apenas uma implementação em um determinado momento, esses também não devem colocar features que *complicam* algumas implementações. Por exemplo, o ARM-32 e alguns outros ISAs têm uma instrução

Load Multiple, conforme mencionado na página anterior. Essas instruções podem melhorar o desempenho de projetos com pipeline de emissão de instrução simples, mas prejudicam os pipelines de instruções múltiplas. O motivo é que a implementação direta impede o planejamento das cargas individuais de um Load Multiple em paralelo com outras instruções, reduzindo o "throughput" de instrução de tais processadores.

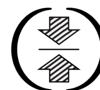
Espaço para Crescimento

Espaço para Crescimento. Com o fim da Lei de Moore, o único caminho a seguir para grandes melhorias no custo-desempenho é adicionar instruções personalizadas para domínios específicos, como *deep learning*, realidade aumentada, otimização combinatória, gráficos e assim por diante. Isso significa que é importante hoje que uma ISA reserve espaço de opcode para futuras melhorias.

Nas décadas de 1970 e 1980, quando a Lei de Moore estava em pleno vigor, não se pensava em economizar espaço de opcode para aceleradores futuros. Em vez disso, os arquitetos valorizavam o endereço maior e os campos imediatos para reduzir o número de instruções executadas por programa, o primeiro termo na equação de desempenho na página anterior.

A instrução ARM-32 `ldmiaeq` mencionada acima é ainda mais complicada, porque quando realiza um desvio também pode alterar os modos de conjunto de instruções entre o ARM-32 e Thumb/Thumb-2.

Um exemplo do impacto da escassez de espaço de opcode foi quando os arquitetos do ARM-32 mais tarde tentaram reduzir o tamanho do código adicionando instruções de 16 bits à ISA de 32 bits anteriormente uniforme. Simplesmente não havia mais espaço. Assim, a única solução foi criar uma nova ISA primeiro com instruções de 16 bits (Thumb) e depois uma nova ISA com instruções de 16 bits e 32 bits (Thumb-2) usando um bit de modo para alternar entre ISAs ARM. Para alterar os modos, o programador ou compilador desvia para um endereço de byte com um 1 no bit menos significativo, o que funcionou porque as instruções de 16 e 32 bits possuem 0 nesse bit.


Tamanho do Programa

Tamanho do Programa. Quanto menor o programa, menor a área necessária em um chip para a memória do programa, o que pode representar um custo significativo para dispositivos embarcados. Na verdade, esse problema inspirou arquitetos ARM a adicionar retroativamente instruções mais curtas nas ISAs Thumb e Thumb-2. Programas menores também levam a menos *cache miss* de instruções, o que economiza energia, já que os acessos DRAM fora do chip usam muito mais energia do que os acessos SRAM no chip, e por isso também apresentam um melhor desempenho. Tamanho pequeno de código é um dos principais objetivos dos arquitetos de ISAs.

Um exemplo de instrução x86-32 de 15 bytes é `lock add dword ptr ds:[esi+ecx*4 +0x12345678], 0xefcdab89`. Monta em (em hexadecimal): 67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef. Os últimos 8 bytes são 2 endereços e os primeiros 7 bytes representam a *atomic memory operation*, a operação add, dados de 32 bits, o registrador de segmento de dados, os dois registradores de endereço e o modo de endereçamento indexado. Um exemplo de instrução de 1 byte é `inc eax` que é convertido em em 40.

A ISA x86-32 tem instruções tão curtas quanto 1 e até 15 bytes. Seria de se esperar que as instruções de comprimento de variável em byte do x86 certamente levassem a programas menores do que as ISAs limitadas a instruções de comprimento de 32 bits, como o ARM-32 e o RISC-V. Logicamente, as instruções de tamanho variável de 8 bits também devem ser menores que as ISAs que oferecem apenas instruções de 16 bits e 32 bits, como Thumb-2 e RISC-V usando a extensão RV32C (consulte o Capítulo 7). A Figura 1.5 mostra que, enquanto o código ARM-32 e RISC-V é 6% a 9% maior que o código para x86-32 quando todas as instruções são 32 bits, surpreendentemente x86-32 é 26% maior que as versões compactadas (RV32C e Thumb-2) que oferecem instruções de 16 bits e 32 bits.

Enquanto uma nova ISA usando instruções variáveis de 8 bits provavelmente levaria a um código menor que o RV32C e o Thumb-2, os arquitetos da primeira versão do x86 nos anos 70 tinham preocupações diferentes. Além disso, dado o requisito de retrocompatibilidade de um ISA incremental (Seção 1.2), as centenas de novas instruções x86-32 são mais longas do que se poderia esperar, uma vez que elas carregam o ônus de um prefixo de um ou dois bytes para comprimi-los no espaço livre, de opcode restrio e disponível do x86 original.

Facilidade de Programação

Facilidade de Programação, Compilação, e "Linkagem". Como os dados em um registrador são muito mais rápidos de serem acessados do que os dados na memória, é im-

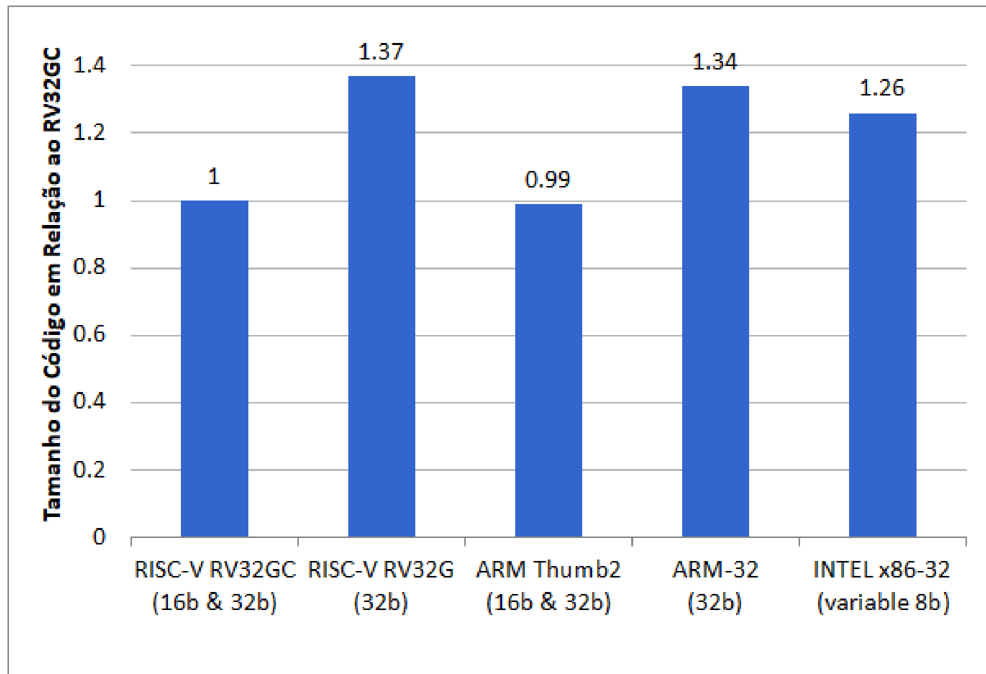


Figura 1.5: Tamanhos relativos de programa para RV32G, ARM-32, x86-32, RV32C e Thumb-2. As duas últimas ISAs destinam-se ao tamanho de código pequeno. Os programas são os benchmarks SPEC CPU2006 usando compiladores GCC. A pequena vantagem de tamanho do Thumb-2 sobre o RV32C deve-se à economia em tamanho de código de Load and Store Multiple na entrada do procedimento. O RV32C os exclui para manter o mapeamento um-para-um para as instruções do RV32G, que omite Load and Store Multiple para reduzir a complexidade da implementação em processadores high-end (veja abaixo). O Capítulo 7 descreve o RV32C. O RV32G indica uma combinação popular de extensões RISC-V (RV32M, RV32F, RV32D e RV32A), adequadamente denominada RV32IMAFD. [Waterman 2016]

portante que os compiladores realizem um bom trabalho na alocação de registradores. Essa tarefa é muito mais fácil quando há muitos registradores em vez de poucos. Sob essa ótica, o ARM-32 tem 16 registradores e o x86-32 tem apenas 8. A maioria das ISAs disponíveis, incluindo o RISC-V, tem um número relativamente generoso de 32 registradores inteiros. Mais registradores certamente facilitam a vida de compiladores e programadores de linguagem assembly.

Outro problema para compiladores e programadores assembly é descobrir a velocidade de uma sequência de código. Como veremos, as instruções RISC-V normalmente são de no máximo um ciclo de clock por instrução (ignorando *cache miss*), enquanto como vimos anteriormente, tanto o ARM-32 quanto o x86-32 possuem instruções que levam muitos ciclos de clock mesmo quando tudo se encaixa na cache. Além disso, ao contrário do ARM-32 e do RISC-V, as instruções aritméticas x86-32 podem ter operandos na memória, em vez de exigir que todos os operandos estejam nos registradores. Instruções complexas e operandos na memória dificultam que os projetistas de processadores forneçam previsibilidade de desempenho.

É útil para uma ISA ter suporte a código independente de posição (*position independent code — PIC*), porque este permite "linkagem" dinâmica (veja a Seção 3.5), já que o código da biblioteca compartilhada pode residir em endereços diferentes em programas diferentes. Os desvios relativos ao PC e o endereçamento de dados são uma vantagem para o PIC. Enquanto quase todas as ISAs fornecem desvios relativos ao PC, x86-32 e MIPS-32 omitem todo endereçamento de dados relativo ao PC.

■ ***Elaboração: ARM-32, MIPS-32, e x86-32***

Elaborações são seções opcionais nas quais os leitores podem aprofunda-se mais caso estiverem interessados em um tópico, mas você não precisa lê-las para entender o restante do livro. Por exemplo, nossos nomes de ISA não são os oficiais. A ISA ARM de 32 bits tem muitas versões, sendo a primeira em 1986 e a mais recente chamada ARMv7 em 2005. ARM-32 geralmente se refere à ISA ARMv7. O MIPS também tinha muitas versões de 32 bits, mas estamos nos referindo ao original, chamado MIPS I ("MIPS32" é uma ISA posterior e diferente do que chamamos de MIPS-32). A primeira arquitetura de endereços de 16 bits da Intel foi o 8086 em 1978, que a ISA 80386 expandiu para endereços de 32 bits em 1985. Nossa notação x86-32 geralmente se refere ao IA-32, a versão com endereço de 32 bits da ISA x86. Dada a extensa quantidade de variantes dessas ISAs, acreditamos que nossa terminologia fora do padrão menos confusa.

1.4 Uma Visão Geral deste Livro

Este livro pressupõe que você já tenha visto outros conjuntos de instruções antes do RISC-V. Caso não tenha, considere dar uma olhada em nosso livro introdutório de arquitetura introdutoria com base no RISC-V [Patterson and Hennessy 2017].

O Capítulo 2 introduz o RV32I, a base "congelada" de instruções com inteiros que são o coração do RISC-V. O Capítulo 3 descreve a linguagem assembly RISC-V restante além daquela introduzida no Capítulo 2, incluindo convenções de chamada e alguns truques inteligentes para "linkagem". A linguagem assembly inclui todas as instruções adequadas do RISC-V, além de algumas instruções úteis que estão fora do RISC-V. Essas *pseudoinstruções*, que são variações inteligentes de instruções reais, facilitam a escrita de programas em assembly sem a necessidade de complicar a ISA.

Os próximos três capítulos descrevem as extensões padrão RISC-V que, quando adicionadas ao RV32I, coletivamente chamamos RV32G (G é para geral):

- Capítulo 4: Multiplicação e Divisão (RV32M)
- Capítulo 5: Ponto Flutuante (RV32F and RV32D)
- Capítulo 6: Instruções Atômicas (RV32A)

O cartão de referência também é chamado de *green card* por causa da sombra da cor de fundo do resumo de uma página das ISAs da década de 1960. Mantivemos o plano de fundo branco para legibilidade em vez de verde para precisão histórica.

O "cartão de referência" do RISC-V nas páginas 3 e 4 é um resumo útil com *todas* instruções RISC-V apresentadas neste livro: RV32G, RV64G e RV32/64V.

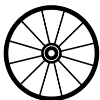
O Capítulo 7 descreve a extensão opcional compactada RV32C, um excelente exemplo da elegância do RISC-V. Ao restringir as instruções de 16 bits para serem versões curtas das instruções existentes de 32 bits do RV32G, elas são quase gratuitas. O assembler pode escolher o tamanho da instrução, permitindo que o programador assembly e o compilador sejam indiferentes ao RV32C. O decodificador de hardware para traduzir instruções RV32C de 16 bits em instruções RV32G de 32 bits precisa de apenas 400 portas, o que representa apenas uma pequena porcentagem da implementação mais simples do RISC-V.

O Capítulo 8 introduz o RV32V, a extensão vetorial. As instruções vetoriais são outro exemplo de elegância ISA, em comparação com as numerosas instruções de força bruta *Single Instruction Multiple Data* (*SIMD*) do ARM-32, MIPS-32 e x86-32. De fato, centenas de instruções adicionadas ao x86-32 na Figura 1.2 eram SIMD, e centenas mais estão surgindo. O RV32V é ainda mais simples que a maioria das ISAs vetoriais, pois associa o tipo de dados e o seu comprimento aos registradores vetoriais, em vez de incorporá-los nos opcodes. O RV32V pode ser o motivo mais convincente para mudar de uma ISA convencional baseada em SIMD para RISC-V.

O Capítulo 9 mostra a versão de endereços de 64 bits do RISC-V, RV64G. Como explica o capítulo, os arquitetos RISC-V precisaram apenas ampliar os registradores e adicionar algumas versões `word`, `doubleword` ou `long` das instruções RV32G para estender o endereço de 32 para 64 bits.

O Capítulo 10 descreve as instruções do sistema, mostrando como o RISC-V lida com paginação e os modos de privilégio Machine, User e Supervisor.

O último capítulo fornece uma descrição rápida das extensões restantes que estão atualmente sendo consideradas pela RISC-V Foundation.


Simplicidade

Em seguida vem a maior seção do livro, o Apêndice A, um resumo do conjunto de instruções em ordem alfabética. Esse define a ISA RISC-V completa com todas as extensões mencionadas acima e todas as pseudoinstruções em cerca de 50 páginas, um testemunho da simplicidade do RISC-V.

O Apêndice B mostra operações comuns da linguagem assembly e a quais instruções elas correspondem no RV32I, ARM-32 e x86-32. Essas três figuras são seguidas por um pequeno programa em C e a saída do compilador para três ISAs. Este apêndice tem dois propósitos: Para os leitores já familiarizados com as ISAs ARM-32 ou x86-32, isso é outra maneira interessante de aprender RISC-V. E para ajudar os programadores que estão convertendo programas assembly escritos nessas ISAs mais antigas para o RISC-V.

Nós finalizamos o livro com um índice.

ISA	Páginas	Palavras	Horas de leitura	Semanas de leitura
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

Figura 1.6: Número de páginas e palavras dos manuais ISA [Waterman and Asanović 2017a], [Waterman and Asanović 2017b], [Intel Corporation 2016], [ARM Ltd. 2014]. Horas e semanas para concluir pressupõem leitura a 200 palavras por minuto durante 40 horas por semana. Baseado em parte na Figura 1 de [Baumann 2017].

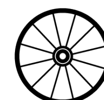
1.5 Considerações Finais

> *It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations ... The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.*
>
> —[von Neumann et al. 1947, 1947]

Uma versão anterior do relatório bem escrito de John von Neumann foi tão influente que esse estilo de computador é comumente chamado de arquitetura de *von Neumann*, embora este relatório tenha sido baseado no trabalho de outros. Foi escrito três anos antes do primeiro computador de programa armazenado em memória estar operacional!

O RISC-V é uma ISA recente, iniciada do zero, minimalista e aberta, informada por erros de ISA anteriores. O objetivo dos arquitetos RISC-V é que este seja eficaz para todos os dispositivos de computação, do menor ao mais rápido. Seguindo o conselho de von Neumann de 70 anos atrás, esta ISA enfatiza a simplicidade para manter os custos baixos e ter muitos registradores e velocidade de instrução transparente para ajudar compiladores e programadores de linguagem assembly a mapear problemas realmente importantes para um código rápido e apropriado.

Um indicativo de complexidade é o tamanho da documentação. A Figura 1.6 mostra o tamanho dos manuais do conjunto de instruções para RISC-V, ARM-32 e x86-32 medidos em páginas e palavras. Se você ler manuais de ISAs como um trabalho de período integral—8 horas por dia durante 5 dias por semana—levaria meio mês para ler uma única passagem pelo manual do ARM-32 e um mês completo para o x86- 32 Neste nível de complexidade, talvez nenhuma pessoa compreenda inteiramente o ARM-32 ou o x86-32. Usando esta métrica de senso comum, RISC-V é $\frac{1}{12}$ da complexidade do ARM-32 e $\frac{1}{10}$ a $\frac{1}{30}$ a complexidade do x86-32. De fato, o resumo do ISA RISC-V incluindo todas as extensões é de apenas duas páginas (veja o Cartão de Referência).



Simplicidade

Esta ISA mínima e aberta foi revelada em 2011 e agora é apoiado por uma fundação que irá evoluí-la adicionando extensões opcionais estritamente baseadas em justificativas técnicas após um debate prolongado. A abertura permite implementações gratuitas e compartilhadas do RISC-V, o que reduz os custos e as chances de segredos maliciosos indesejados serem escondidos em um processador.

No entanto, o hardware sozinho não faz um sistema. Os custos de desenvolvimento de software provavelmente superam os custos de desenvolvimento de hardware, portanto, embora o hardware estável seja importante, o software estável é mais importante. Ele requer sistemas operacionais, carregadores de inicialização, software de referência e ferramentas de software populares. A base oferece estabilidade para o ISA geral, e a base congelada significa que o núcleo RV32I que é o alvo da pilha de software nunca será alterado. Por sua ampla adoção e abertura, o RISC-V pode desafiar o domínio das ISAs proprietárias.

Elegância

Elegância é uma palavra que é raramente aplicada a ISAs, mas depois de ler este livro, você pode concordar conosco que se aplica ao RISC-V. Vamos destacar características que acreditamos que indicam elegância com um ícone da Mona Lisa nas margens.

1.6 Para Saber Mais

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. Asanović, 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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017b. URL `https://riscv.org/specifications/`.

2 RV32I: ISA RISC-V Base para Números Inteiros

Frances Elizabeth "Fran" Allen (1932-) foi agraciado com o Turing Award principalmente pelo seu trabalho em otimização de compiladores. O Turing Award é o maior prêmio na área de Ciência da Computação.

... the only way to realistically realize the performance goals and make them accessible to the user was to design the compiler and the computer at the same time. In this way features would not be put in the hardware which the software could not use ...

—Frances Elizabeth "Fran" Allen, 1981

2.1 Introdução

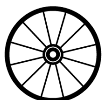
A Figura 2.1 é uma representação gráfica de uma página do conjunto de instruções base RV32I. Você pode ver o conjunto completo de instruções RV32I concatenando as letras sublinhadas da esquerda para a direita de cada diagrama. A notação definida utilizando {} lista as possíveis variações da instrução, utilizando letras sublinhadas ou o caractere de sublinhado _, significando que não há letras para essa variação. Por exemplo:

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

representa estas quatro instruções RV32I: `slt`, `slti`, `sltu`, `sltiu`.

O objetivo desses diagramas, que serão as primeiras figuras dos capítulos seguintes, é dar uma rápida e perspicaz visão das instruções de um capítulo.

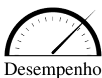
2.2 Formatos de instrução RV32I



Simplicidade

A Figura 2.2 mostra os seis formatos de instrução de base: Tipo-R para operações de registradores; Tipo-I para valores imediatos short e loads; Tipo-S para stores; Tipo-B para desvios condicionais; Tipo-U para valores imediatos longos; e tipo-J para saltos incondicionais. A Figura 2.3 lista os opcodes das instruções RV32I na Figura 2.1 utilizando os formatos da Figura 2.2.

Custo



Desempenho

Até mesmo os formatos de instrução demonstram vários exemplos em que a RISC-V ISA mais simples melhora o desempenho de custo. Primeiro, existem apenas seis formatos e todas as instruções são de 32 bits, o que simplifica a decodificação de instruções. O ARM-32 e particularmente o x86-32 possuem vários formatos, o que torna a decodificação cara em implementações de baixo custo e um desafio de desempenho para projetos de processadores de médio e grande porte. Segundo, as instruções RISC-V oferecem três operandos

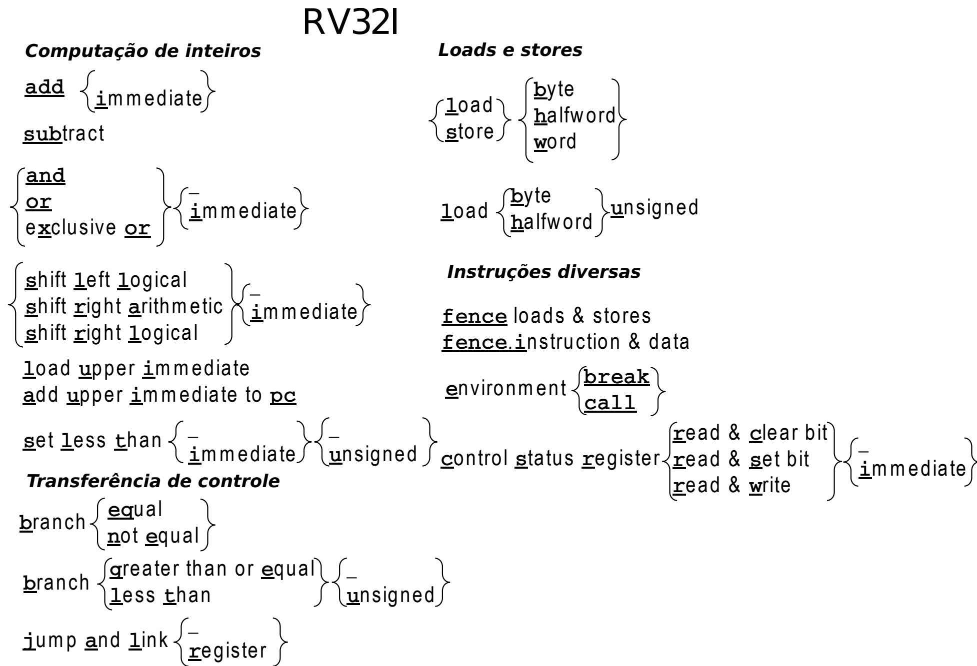


Figura 2.1: Diagrama das instruções do RV32I. As letras sublinhadas são concatenadas da esquerda para a direita para formar instruções RV32I. A notação de colchetes {} significa que cada item vertical no conjunto é uma variação diferente da instrução. O sublinhado _ dentro de um conjunto significa que uma opção é simplesmente o nome da instrução sem uma letra deste conjunto. Por exemplo, a notação próxima ao canto superior esquerdo representa as seis instruções a seguir: `and`, `or`, `xor`, `andi`, `ori`, `xori`.

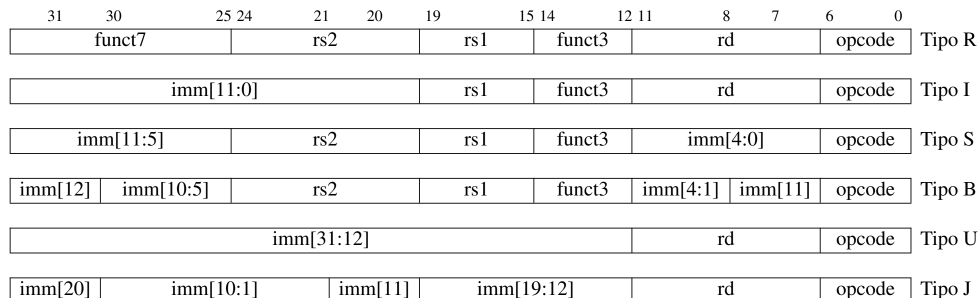


Figura 2.2: Formatos de instrução RV32I Nós rotulamos cada subcampo imediato com a posição do bit (imm [x]) no valor sendo produzido em imediato, ao invés da posição do bit no campo imediato da instrução como normalmente é feito. O capítulo 10 explica como as instruções de registrador de status de controle utilizam o formato Tipo-I de maneira um pouco diferente. (A Figura 2.2 de Waterman and Asanović 2017 é a base desta figura).

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	R sltu
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 csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figura 2.3: O mapa de opcode RV32I possui layout de instrução, opcodes, tipo de formato e nomes. (A Tabela 19.2 de [Waterman and Asanović 2017] é a base desta figura.)

de registradores, em vez de ter um campo compartilhado para origem e destino, como acontece com x86-32. Quando uma operação possui naturalmente três operandos distintos, mas o ISA fornece apenas uma instrução de dois operandos, o compilador ou o programador de linguagem assembly deve utilizar uma instrução de movimentação extra para preservar o operando de destino. Em terceiro lugar, no RISC-V os especificadores dos registradores a serem lidos e escritos estão sempre no mesmo local em todas as instruções, o que significa que os acessos ao registrador podem começar antes de decodificar a instrução. Muitas outras ISAs reutilizam um campo como uma fonte em algumas instruções e como um destino em outras (por exemplo, ARM-32 e MIPS-32), o que força a adição de hardware extra a ser colocado em um caminho potencialmente crítico para selecionar o campo. Em quarto lugar, os campos imediatos nesses formatos são sempre com sinal, e o bit de sinal é sempre o bit mais significativo da instrução. Esta decisão significa que a extensão de sinal do imediato, que também pode estar em um caminho de cronometragem crítico, pode prosseguir antes de decodificar a instrução.

O sinal estendido imediato até mesmo ajuda instruções lógicas. Por exemplo, x & 0xfffffff0 utiliza apenas a instrução Tandi no RISC-V, mas requer duas instruções no MIPS-32 (`addiu` para carregar a constante e, em seguida, `and`), uma vez que MIPS estende zeros imediatos lógicos. ARM-32 necessitava ainda adicionar uma instrução adicional, `bic`, que executa o & valor imediato rx para compensar os imediatos com extensão de zeros.

■ ***Elaboração: Formatos Tipo-B e Tipo-J?***

Como mencionado abaixo, o campo imediato é rotacionado em 1 bit para instruções de desvio, uma variação do formato S que nós reclassificamos o formato B. O campo imediato também é rotacionado para instruções de salto, uma variação do formato J remarcado em formato U. Portanto, existem, na verdade, quatro formatos básicos, mas podemos conservativamente considerar o RISC-V como tendo seis formatos.

Para ajudar os programadores, um padrão de bits preenchido por zeros é uma instrução ilegal no RV32I. Assim, saltos errôneos em regiões de memória zeradas serão imediatamente interceptados, ajudando a depuração. Da mesma forma, esse padrão de bits é uma instrução ilegal, que interceptará outros erros comuns, como dispositivos de memória não volátil não programados, barramentos de memória desconectados ou chips de memória quebrados.

Facilidade de Programação

Para deixar um espaço amplo para extensões do ISA, a base RV32I ISA utiliza menos de 1/8 do espaço de codificação na palavra de instrução de 32 bits. Os arquitetos também escolheram cuidadosamente os opcodes do RV32I para que as instruções com operações comuns do caminho de dados compartilhem o máximo possível dos mesmos valores de bit de opcode, o que simplifica a lógica de controle. Finalmente, como veremos, os endereços de desvio e salto nos formatos B e J devem ser deslocados para a esquerda 1 bit, de modo a multiplicar os endereços por 2, dando assim desvio e saltos de maior alcance. O RISC-V rotaciona os bits nos operandos imediatos a partir de um posicionamento natural para reduzir o fan-out do sinal de instrução e o custo de multiplexação imediato por quase um fator de dois, o que simplifica novamente a lógica do caminho de dados em implementações de baixo custo.

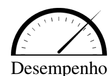
Espaço para Crescimento

Custo

Em que difere? Terminaremos cada seção neste e nos próximos capítulos com a descrição de como o RISC-V difere de outras ISAs. O contraste é frequentemente o que falta no RISC-V. Arquitetos demonstram bom gosto pelas características que omitem, bem como pelo que incluem.

Todas as implementações RISC-V utilizam os mesmos opcodes para as extensões opcionais, como RV32M, RV32F e assim por diante. Extensões não padrão exclusivas do processador são restritas a um espaço de opcode reservado no RISC-V.

O campo imediato de 12 bits ARM-32 não é simplesmente uma constante, mas uma entrada para uma função que produz uma constante: 8 bits são estendidos em zero até a largura total e depois rotacionados diretamente pelo valor nos 4 bits restantes multiplicados por 2. A esperança era a codificação de constantes mais úteis em 12 bits, o que reduziria o número de instruções executadas. O ARM-32 também dedica quatro preciosos bits na maioria dos formatos de instrução à execução condicional. Apesar de ser pouco utilizada, a


Desempenho

execução condicional aumenta a complexidade de *processadores fora de ordem*.

■ ***Elaboração: Processadores fora de ordem***

são processadores de alta velocidade que utilizam pipeline e que executam instruções oportunisticamente, em vez de na ordem do programa lock-step. Uma característica crítica de tais processadores é a *renomeação de registrador*, que mapeia os nomes dos registradores do programa em um número maior de registradores físicos internos. O problema com a execução condicional é que o novo registrador físico deve ser escrito independentemente da resposta, portanto, o valor antigo do destino registrador deve ser lido como um terceiro operando, para ser copiado para o novo registrador de destino, no caso de a condição não se sustentar. O operando extra aumenta o custo do arquivo de registrador, do renomeador de registrador e também do hardware de execução fora de ordem.

Facilidade de Programação

2.3 Registradores RV32I

Pipelining É utilizado por quase todos os processadores atuais, menos os processadores mais baratos, para obter bom desempenho. Como uma linha de montagem industrial, os pipelines têm um rendimento mais alto ao sobrepor a execução de muitas instruções de uma só vez. Para conseguir isso, os processadores predizem os resultados dos desvios, o que fazem com mais de 90% de acurácia. Quando eles interpretam errado, eles re-executam as instruções. Os primeiros microprocessadores tinham um pipeline de 5 estágios, o que significava que 5 instruções sobrepunham a execução. Os mais recentes têm mais de 10 etapas de pipeline. O ARM v8 sucessor do ARM-32 descartou o PC como um registrador de propósito geral, admitindo efetivamente que foi um erro.

A Figura 2.4 lista os registradores RV32I e seus nomes, conforme determinado pela interface binária do aplicativo RISC-V (ABI). Utilizaremos os nomes ABI em nossos exemplos de código para facilitar a leitura. Para a alegria dos programadores de linguagem assembly e escritores de compiladores, o RV32I possui 31 registradores mais x0, que sempre tem o valor 0. O ARM-32 tem apenas 16 registradores enquanto o x86-32 tem apenas 8 registradores!

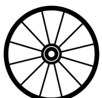
Em que difere? Dedicar um registrador a zero é um fator surpreendentemente grande na simplificação do ISA RISC-V. A Figura 3.3, na página 38, no capítulo 3 fornece muitos exemplos de operações que são instruções nativas no ARM-32 e x86-32, que não possuem um registrador zero, mas podem ser sintetizadas a partir das instruções do RV32I simplesmente utilizando o registrador zero como um operando.

O PC é um dos 16 registradores do ARM-32, o que significa que qualquer instrução que altere um registrador também pode ser como efeito colateral uma instrução de desvio. O PC como um registrador complica a previsão de desvio de hardware, cuja acurácia é vital para um bom desempenho de pipeline, já que toda instrução pode ser um desvio em vez de 10–20% de instruções executadas em programas para ISAs típicas. Isso significa também um registrador de propósito geral a menos.

2.4 Computação de Inteiros RV32I

O Apêndice A fornece detalhes de todas as instruções do RISC-V, incluindo formatos e opcodes. Nesta seção e em seções semelhantes dos capítulos a seguir, fornecemos uma visão geral do ISA que deve ser suficiente para programadores de linguagem assembly bem informados, bem como destacamos os recursos que demonstram as sete métricas do ISA a partir do capítulo 1.

As instruções aritméticas simples (`add`, `sub`), instruções lógicas (`and`, `or`, `xor`), e instruções de deslocamento (`sll`, `srl`, `sra`) na Figura 2.1 são exatamente o que se espera em qualquer ISA. Elas lêem dois valores de 32 bits dos registradores e gravam um resultado de 32 bits no registrador de destino. O RV32I também oferece versões imediatas dessas instruções. Ao contrário do ARM-32, os imediatos são sempre estendidos em sinal para que possam ser negativos quando necessário, razão pela qual não há necessidade de uma versão imediata de `sub`.


Simplicidade

31 — 0	
`x0 / zero`	Zero hardwired
`x1 / ra`	Endereço de retorno
`x2 / sp`	Ponteiro de pilha
`x3 / gp`	Ponteiro global
`x4 / tp`	Ponteiro de thread
`x5 / t0`	Temporário
`x6 / t1`	Temporário
`x7 / t2`	Temporário
`x8 / s0 / fp`	Registrador salvo, ponteiro de quadro
`x9 / s1`	Registrador salvo
`x10 / a0`	Argumento de função, valor de retorno
`x11 / a1`	Argumento de função, valor de retorno
`x12 / a2`	Argumento de função
`x13 / a3`	Argumento de função
`x14 / a4`	Argumento de função
`x15 / a5`	Argumento de função
`x16 / a6`	Argumento de função
`x17 / a7`	Argumento de função
`x18 / s2`	Registrador salvo
`x19 / s3`	Registrador salvo
`x20 / s4`	Registrador salvo
`x21 / s5`	Registrador salvo
`x22 / s6`	Registrador salvo
`x23 / s7`	Registrador salvo
`x24 / s8`	Registrador salvo
`x25 / s9`	Registrador salvo
`x26 / s10`	Registrador salvo
`x27 / s11`	Registrador salvo
`x28 / t3`	Temporário
`x29 / t4`	Temporário
`x30 / t5`	Temporário
`x31 / t6`	Temporário
32	

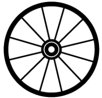
31 — 0
`pc`
32

Figura 2.4: Os registradores do RV32I. O Capítulo 3 explica a convenção de chamada RISC-V, a lógica por trás dos vários ponteiros (sp, gp, tp, fp), registradores salvos (s0-s11) e temporários (t0-t6). (A Figura 2.1 e a Tabela 20.1 of [Waterman and Asanović 2017] é a base dessa figura.)

Programas podem gerar um valor booleano a partir do resultado de uma comparação. Para acomodar tais casos, o RV32I oferece uma instrução *set less than*, que define o registrador de destino como 1 se o primeiro operando for menor que o segundo, ou 0 caso contrário. Como esperado, há uma versão com sinal, (`slt`), e uma versão sem sinal, (`sltu`), para inteiros com e sem sinal, bem como versões imediatas para ambos (`slti, sltiu`). Como veremos, enquanto os desvios RV32I podem verificar todas as relações entre dois registradores, algumas expressões condicionais envolvem relacionamentos entre muitos pares de registradores. O compilador ou programador de linguagem assembly poderia utilizar `slt` e as instruções lógicas `and`, `or`, `xor` para resolver expressões condicionais mais elaboradas.

Facilidade de Programação

As duas instruções de cálculo de inteiros restantes na Figura 2.1 ajuda com montagem e ligação. *Load upper immediate, ou carga de valor imediato superior* (`lui`) carrega uma constante de 20 bits nos 20 bits mais significativos de um registrador. Ele pode ser seguido por uma instrução imediata padrão para criar uma constante de 32 bits a partir de apenas duas instruções RV32I de 32 bits. *Add upper immediate to PC, ou Soma de imediatos superiores ao PC* (`auipc`) suporta seqüências de duas instruções para acessar deslocamentos arbitrários do PC para transferências de fluxo de controle e acessos de dados. A combinação de um `auipc` e os 12 bits imediatos em um `jalr` (ver abaixo) pode transferir o controle para qualquer endereço relativo ao PC de 32 bits, enquanto `auipc` mais o deslocamento imediato de 12 bits em instruções regulares de load ou store pode acessar qualquer endereço de dados relativo a 32 bits do PC.


Simplicidade

A ISA sucessora do ARM v8 para ARM-32 abandonou a operação de deslocamento opcional das instruções da ALU, novamente sugerindo que foi um erro para o ARM-32.

Em que difere? Primeiro, não há operações de computação de inteiros de byte ou halfword. As operações são sempre a largura total do registrador. Os acessos à memória consomem muito mais energia do que as operações aritméticas, portanto os acessos estreitos aos dados podem economizar energia significativa, mas operações estreitas não. ARM-32 tem o recurso incomum de ter uma opção para deslocar um dos operandos na maioria das operações de lógica aritmética, o que complica o caminho de dados e raramente é necessário [Hohl and Hinds 2016]; O RV32I tem instruções de deslocamento separadas. O RV32I também não inclui multiplicar e dividir; eles compreendem a extensão RV32M opcional (veja o capítulo 4). Ao contrário do ARM-32 e x86-32, a pilha de software RISC-V completa pode ser executada sem eles, o que pode reduzir os chips incorporados. Embora não seja um problema de hardware, o *assembler* MIPS-32 pode substituir um multiplicar por um deslocamento de sequência e adição para tentar melhorar o desempenho, o que pode confundir o programador vendo instruções executadas não encontradas no programa de linguagem assembly. O RV32I também omite instruções de rotação e detecção de overflow de aritmética de inteiros. Ambos podem ser calculados em algumas instruções RV32I (veja a seção 2.6).

Custo

■ ***Elaboração: Instruções de "rotação de bit"***

Tais como rotacionar estão sob consideração pela Fundação RISC-V como parte de uma extensão de instrução opcional chamada RV32B (veja o capítulo 11).

■ ***Elaboração:*** `xor` ***permite um truque de mágica.***

Você pode trocar dois valores sem utilizar um registrador intermediário! Este código troca os valores de x1 e x2. Deixamos a prova para o leitor. Dica: o operando "ou exclusivo" é comutativo ($a \oplus b = b \oplus a$), associativo ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), é o seu próprio inverso ($a \oplus a = 0$), e tem uma identidade ($a \oplus 0 = a$).

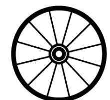
```
xor  x1,x1,x2 # x1' == x1^x2, x2' == x2
xor  x2,x1,x2 # x1' == x1^x2, x2' == x1'^x2 == x1^x2^x2 == x1
xor  x1,x1,x2 # x1'' == x1'^x2' == x1^x2^x1 == x1^x1^x2 == x2, x2' == x1
```

Por mais fascinante que seja, o amplo conjunto de registradores do RISC-V geralmente permite que os compiladores encontrem um registrador de rascunho, por isso raramente utiliza o XOR-swap.

2.5 Loads e Stores em RV32I

Além de fornecer loads e stores de palavras de 32 bits (`lw`, `sw`), A Figura 2.1 mostra que o RV32I tem loads para bytes e halfwords com e sem sinal (`lb`, `lbu`, `lh`, `lhu`) e armazena para bytes e halfwords (`sb`, `sh`). Bytes e halfwords com sinal são estendidos em sinal para 32 bits e gravados nos registradores de destino. Esta ampliação de dados estreitos permite que instruções subseqüentes de cálculo de números inteiros operem corretamente em todos os 32 bits, mesmo se os tipos de dados naturais forem mais estreitos. Bytes sem assinatura e halfwords, úteis para textos e números inteiros sem sinal, são estendidos de zero a 32 bits antes de serem gravados no registrador de destino.

O único modo de endereçamento para loads e stores é adicionar um sinal de 12 bits imediato a um registrador, chamado modo de endereçamento de deslocamento em x86-32. [Irvine 2014].


Simplicidade

Em que difere? O RV32I omitiu os sofisticados modos de endereçamento do ARM-32 e x86-32. Infelizmente, todos os modos de endereçamento ARM-32 não estão disponíveis para todos os tipos de dados, mas o endereçamento RV32I não discrimina nenhum tipo de dado. O RISC-V pode imitar alguns modos de endereçamento x86. Por exemplo, "setar" o campo imediato como 0 tem o mesmo efeito que o modo de endereçamento indireto de registrador. Ao contrário do x86-32, O RISC-V não possui instruções especiais de pilha. Utilizando um dos 31 registradores como o ponteiro da pilha (veja a Figura 2.4), o modo de endereçamento padrão obtém a maioria dos benefícios das instruções `push` e `pop` sem a complexidade adicional do ISA. Ao contrário do MIPS-32, RISC-V rejeitou o *delayed load*. Semelhante em espírito aos desvios atrasados, o MIPS-32 redefiniu o load para que os dados fiquem indisponíveis até duas instruções posteriores, quando apareceria em um pipeline de cinco estágios. Qualquer benefício que tenha evaporado para os pipelines mais longos que vieram depois.

Facilidade de Programação

Enquanto o ARM-32 e o MIPS-32 requerem que os dados sejam alinhados naturalmente aos limites de tamanho de dados na memória, o RISC-V não. Às vezes, acessos desalinhados são necessários ao portar código legado. Uma opção é não permitir acessos desalinhados no ISA base e fornecer suporte a instruções separadas para acessos desalinhados, como Load Word Left e Load Word Right do MIPS-32.

Custo

Esta opção complicaria o acesso ao registrador, no entanto, já que `lwl` e `lwr` requerem escrever partes de registradores em vez de simplesmente registradores completos. Exigir, em

vez disso, que os loads e stores regulares suportem acessos desalinhados simplifica o design geral.

■ ***Elaboração: Endianness***

O RISC-V escolheu a ordenação de bytes *little-endian* porque é dominante comercialmente: todos os sistemas x86-32 e Apple iOS, Google Android OS e Microsoft Windows para ARM são todos little-endian. Como a ordem endian é importante apenas ao acessar dados idênticos, tanto como uma palavra quanto como bytes, o endianness afeta poucos programadores.

2.6 Desvio Condicional em RV32I

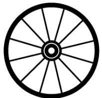
RV32I pode comparar dois registradores e desviar no resultado se eles forem iguais (`beq`), diferentes (`bne`), maior ou igual (`bge`), ou menor (`blt`). Os dois últimos casos são comparações com sinal, mas o RV32I também oferece versões sem sinal: `bgeu` e `bltu`. Os dois relacionamentos restantes (maior que e menor que ou igual) podem ser verificados simplesmente pela reversão dos operandos, uma vez que $x < y$ significa que $y > x$ e $x \geq y$ implica $y \leq x$.

`bltu` **permite que os limites de matriz com sinal sejam verificados com uma única instrução**, uma vez que qualquer índice negativo irá comparar mais do que qualquer limite não-negativo!

Já que as instruções RISC-V deve ser um múltiplo de dois bytes de comprimento—veja o capítulo 7 para aprender sobre as instruções opcionais de 2 bytes—o modo de endereçamento de desvio multiplica o imediato de 12 bits por 2, amplia o sinal e adiciona-o ao PC. Endereçamento relativo a PC ajuda com código independente de posição e, portanto, reduz o trabalho do linker e do loader (capítulo 3).

Facilidade de Programação

Em que difere? Como observado acima, O RISC-V excluiu o infame desvio atrasado do MIPS-32, Oracle SPARC e outros. Também evitou os códigos de condição de ARM-32 e x86-32 para desvios condicionais. Eles adicionam um estado extra que é implicitamente definido pela maioria das instruções, o que desnecessariamente complica o cálculo de dependência para execução fora de ordem. Finalmente, omitiu as instruções de loop do x86-32: `loop`, `loope`, `loopz`, `loopne`, `loopnz`.


Simplicidade

■ ***Elaboração: Adição Multiword sem códigos de condição***

é feito da seguinte forma no RV32I utilizando `sltu` para calcular o carry-out:

```
add  a0,a2,a4 # add lower 32 bits: a0 = a2 + a4
sltu a2,a0,a2 # a2' = 1 if (a2+a4) < a2, a2' = 0 otherwise
add  a5,a3,a5 # add upper 32 bits: a5 = a3 + a5
add  a1,a2,a5 # add carry-out from lower 32 bits
```

■ ***Elaboração: Lendo o PC***

O PC atual pode ser obtido "setando" o campo imediato U de `auipc` como 0. Para o x86-32, para ler o PC você precisa chamar uma função (que empurra o PC para a pilha); o receptor então lê o PC empurrado da pilha e, finalmente, retorna o PC (estourando a pilha). Então, ler o PC atual levou 1 store, 2 loads e 2 jumps!

■ ***Elaboração: Verificação de software de overflow***

A maioria dos programas, mas não todos, ignora o overflow de aritmética de inteiro. Portanto, o RISC-V depende da verificação de overflow por software. A adição sem sinal requer apenas uma instrução de desvio adicional após a adição: `addu t0, t1, t2; bltu t0, t1, overflow`.

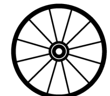
Para adição com sinal, se o sinal de um operando for conhecido, a verificação de overflow requer apenas um único desvio após a adição: `addi t0, t1, +imm; blt t0, t1, overflow`.

Isso cobre o caso comum de adição com um operando imediato. Para a adição geral com sinal, três instruções adicionais após a adição são necessárias, observando que a soma deve ser menor que um dos operandos se e somente se o outro operando for negativo.

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

2.7 Salto Incondicional em RV32I

A instrução única *jump and link* (`jal`) na Figura 2.1 serve funções duplas. Para suportar chamadas de procedimento, ele salva o endereço da próxima instrução PC + 4 no registrador de destino, normalmente o registrador de endereço de retorno `ra` (veja a Figura 2.4). Para suportar saltos incondicionais, utilizando o registrador zero (x0) em vez de `ra` como o registrador de destino, pois x0 não pode ser alterado. Como desvios, o `jal` multiplica seu endereço de desvio de 20 bits por 2, o sinal o estende e, em seguida, adiciona o resultado ao PC para formar o endereço de salto.



Simplicidade

A versão de registrador de jump e link (`jalr`) é similarmente multiuso. Ele pode chamar um procedimento para um endereço calculado dinamicamente ou simplesmente executar um retorno de procedimento selecionando o `ra` como o registrador de origem e o registrador zero (x0) novamente como o registrador de destino. As declarações switch ou case, que calculam um endereço de salto, também podem utilizar `jalr` com o registrador zero como registrador de destino.

Em que difere? O RV32I evitava instruções enredadas de chamada de procedimento, como as instruções `enter` e `leave` do x86-32, ou *register windows* como encontradas no Intel Itanium, Oracle SPARC e Cadence Tensilica.

Janelas de registradores chamada de função acelerada tendo muito mais registradores que 32. Uma nova função receberia um novo conjunto ou *window* de 32 registradores em uma chamada. Para passar argumentos, as janelas se sobrepunham, significando que alguns registradores estavam em duas janelas adjacentes.

2.8 RV32I: Informações diversas

As instruções de Registradores de Status de Controle (`csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi`) na Figura 2.1 fornece acesso fácil aos registradores que ajudam a medir o desempenho do programa. Esses contadores de 64 bits, que podem ser lidos 32 bits por vez, medem a hora do relógio de parede, os ciclos de clock executados e o número de instruções retiradas.

A instrução `ecall` faz solicitações ao ambiente de execução de suporte, como chamadas do sistema. Depuradores utilizam a instrução `ebreak` para transferir o controle para um

```
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;
  }
}
```

Figura 2.5: Ordenação por inserção em C. Embora simples, a Ordenação por Inserção apresenta muitas vantagens em relação aos algoritmos de classificação complicados: ele é eficiente na memória e rápido para pequenos conjuntos de dados, ao mesmo tempo em que é adaptável, estável e on-line. Os compiladores GCC produziram o código para as quatro figuras a seguir. Nós definimos os sinalizadores de otimização para reduzir o tamanho do código, pois isso produziu o código mais fácil de entender.

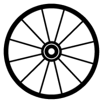
ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC
Instruções	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

Figura 2.6: Número de instruções e tamanho do código para Ordenação por Inserção para esses ISAs. Chapter 7 descreve ARM Thumb-2, microMIPS e RV32C.

ambiente de depuração.

As sequências de instruções `fence` acessam dispositivos de E/S e acessos à memória, conforme visualizadas por outros threads e por dispositivos externos ou coprocessadores. A instrução `fence.i` sincroniza a instrução e os fluxos de dados. O RISC-V não garante que os armazenamentos na memória da instrução sejam visíveis para buscas de instrução no mesmo processador até que uma instrução `fence.i` seja executada.

O Capítulo 10 abrange as instruções do sistema RISC-V.



Simplicidade

Em que difere? O RISC-V utiliza E/S de memória mapeada em vez das instruções do x86-32 `in`, `ins`, `insb`, `insw` e `out`, `outs`, `outsb`, `outsw`. Ele suporta strings utilizando bytes carregados e armazena em vez das 16 instruções especiais de string do x86-32 `rep`, `movs`, `coms`, `scas`, `lods`, ...

2.9 Comparando RV32I, ARM-32, MIPS-32 e x86-32 utilizando Ordenação por Inserção

Introduzimos o conjunto de instruções básicas RISC-V e comentamos suas escolhas em comparação com o ARM-32, o MIPS-32 e o x86-32. Agora faremos uma comparação cara-a-cara. A Figura 2.5 mostra o algoritmo de Ordenação por Inserção, implementado em C, que será nosso benchmark. A Figura 2.6 é uma tabela que resume o número de instruções e o número de bytes na Ordenação por Inserção para os ISAs.

As Figuras 2.8 e 2.11 mostram o código compilado para RV32I, ARM-32, MIPS-32 e x86-32. Apesar da ênfase na simplicidade, a versão RISC-V utiliza as mesmas instruções ou menos, e os tamanhos dos códigos das arquiteturas são bem próximos. Neste exemplo, os desvios comparar-e-executar do RISC-V salvam tantas instruções quanto os modos de

endereço mais sofisticados e as instruções push e pop do ARM-32 e x86-32 nas Figuras 2.9 e 2.11.

Nós movemos os exemplos de código para depois do final do texto do capítulo para manter o fluxo da escrita neste e nos próximos capítulos.

2.10 Considerações Finais

Those who cannot remember the past are condemned to repeat it.

—George Santayana, 1905

A Figura 2.7 utiliza as sete métricas de projeto de ISA do Capítulo 1 para organizar as lições das ISAs anteriores listadas nas seções anteriores e mostra os resultados positivos para RV32I. *Não* estamos insinuando que o RISC-V é o primeiro ISA a ter esses resultados. De fato, o RV32I herda o seguinte do RISC-I, seu tataravô [Patterson 2017]:

A genealogia de todas as instruções RISC-V são narradas em [Chen and Patterson 2016].

- Espaço de endereço endereçável por byte de 32 bits
- Todas as instruções são de 32 bits
- 31 registradores, todos de 32 bits, com registrador 0 conectado a zero
- Todas as operações acontecem entre registradores (nenhuma entre registrador-memória)
- A word Load/store mais o byte e halfword do load/store com sinal e sem sinal
- Opção imediata para todas as instruções aritméticas, lógicas e de deslocamento
- Imediatos sempre extendem sinal.
- Um modo de endereçamento de dados (registrador + imediato) e desvio relativo ao PC
- Sem instruções de multiplicação e divisão
- Uma instrução para carregar um imediato na parte superior do registrador, de modo que uma constante de 32 bits leve apenas duas instruções

O efeito Lindy [Lin 2017] observa que a expectativa de vida futura de uma tecnologia ou ideia é proporcional à sua idade. Se ela resistiu ao teste do tempo, então quanto mais tempo ele sobreviveu no passado, mais tempo ele provavelmente sobreviverá no futuro. Se essa hipótese for válida, a arquitetura RISC pode ser uma boa ideia por muito tempo.

O RISC-V se beneficia do início de um quarto a um terço de século mais tarde, o que permitiu que seus arquitetos seguissem o conselho de Santayana para pegar emprestadas as boas idéias, mas não para repetir os erros do passado - incluindo os do RISC-I no ISA RISC-V atual. Além disso, a Fundação RISC-V aumentará o ISA lentamente através de extensões opcionais para evitar o incrementalismo desenfreado que tem atormentado ISAs bem-sucedidas do passado.

Elegância

■ ***Elaboração: Seria o RV32I único?***

Os primeiros microprocessadores tinham chips separados para aritmética de ponto flutuante, então essas instruções eram opcionais. A Lei de Moore logo trouxe tudo em chip e a modularidade desapareceu nos ISAs. A subconjugação do ISA completo em processadores mais simples e o trapping para o software para emulá-los remontam a décadas, com o IBM 360 modelo 44 e o Digital Equipment microVAX como exemplos. O RV32I é diferente porque a pilha completa de software precisa apenas das instruções básicas, portanto, um processador RV32I não precisa ser interceptado repetidamente para instruções omitidas no RV32G. Provavelmente, o ISA mais próximo do RISC-V nesse aspecto é o Tenstalla Xtensa, que é voltado para aplicações embarcadas. Seu ISA base de 80 ou menos instruções é planejado para ser estendido por usuários com instruções personalizadas que aceleram seus aplicativos. O RV32I possui uma base ISA mais simples, possui uma versão de endereço de 64 bits e oferece extensões que visam supercomputadores e também microcontroladores.

	Erros do passado ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	Lições aprendidas RV32I (2011)
Custo	Multiplicação de inteiros mandatória	Multiplicação e divisão de inteiros mandatória	Operações de 8-bit e 16-bit. Multiplicação e divisão de inteiros mandatóriaInteger	Sem operações de 8-bit e 16-bit operations. Multiplicação e divisão de inteiros opcional (RV32M)
Simplicidade	Sem registrador zero. Execução de instrução condicional. Modos de endereço de dados complexos. Instruções de pilha (push/pop). Opção de deslocamento para instruções aritméticas e lógicas	Imediatos zero e de sinal estendido. Algumas instruções aritméticas podem causar armadilhas de overflow	Sem registrador zero. Instruções de chamada/retorno de procedimento complexo (Enter/Leave). Instruções de pilha (push/pop). Modos de endereço de dados complexos. Instruções de laço	Registrador x0 dedicado ao valor 0. Imediatos somente com sinal estendido. Modo de endereçamento de dados. Nenhuma execução condicional. Nenhuma instrução complexa de desvio/retorno. Nenhuma trap para overflow aritmético. Instruções de deslocamento separadas
Desempenho	Códigos de condição para desvios. Os reg. de origem e destino variam em formato de instrução. Load múltiplo. Imediatos computados. PC de propósito geral	Os registradores de origem e destino variam em formato de instrução.	Códigos de condição para desvios. No máximo 2 registradores por instrução	Instruções de compara e desvia. 3 registradores por instrução. Sem load múltiplo. Registradores de origem e destino fixados em formato de instrução. Imediatos constantes. PC não é um reg. de propósito geral
Isola a arquitetura da implementação	Expõe o comprimento do pipeline ao gravar o PC como um registrador de propósito geral	Desvio atrasado. Load atrasado. Registradores HI e LO utilizados apenas para multiplicação e divis	Registradores sem finalidade geral (AX, CX, DX, DI, SI tem usos exclusivos)	Sem desvio atrasado. Sem load atrasado. Registradores de propósito geral
Espaço para crescimento	Espaço disponível do opcode limitado	Espaço disponível do opcode limitado		GEspaço de opcode disponível generoso
Tamanho do programa	Apenas instruções de 32 bits (+Thumb-2 como ISA separada)	Apenas instruções de 32 bits (+microMIPS como ISA separada)	Instruções variáveis de byte, mas com más escolhas	Instruções de 32 bits + extensão RV32C de 16 bits
Facilidade de programação / compilação / linkagem	Apenas 15 registradores. Dados alinhados na memória. Modos de endereço de dados irregulares. Contadores de desempenho inconsistentes	Dados alinhados na memória. Contadores de desempenho inconsistentes	Apenas 8 registradores. Nenhum endereçamento de dados relativos ao PC. IContadores de desempenho inconsistentes	31 registradores. Os dados podem ficar desalinhados. Endereçamento de dados relativos ao PC. Modo de endereço de dados simétrico. PContadores de desempenho definidos na arquitetura

Figura 2.7: Lições que os arquitetos RISC-V aprenderam com os erros de ISAs do passado. Muitas vezes a lição era simplesmente evitar as "otimizações" do ISA do passado. As lições e erros são classificados pelas sete métricas ISA do Capítulo 1. Muitos recursos listados sob custo, simplicidade e desempenho podem ser trocados entre si, já que é uma questão de gosto, mas eles são importantes, não importa onde eles apareçam.

2.11 Para Saber Mais

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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

```
# 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 é ponteiro para a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Laço externo:
   8: 00b76463  bltu  a4,a1,10   # se i < n, salta para Continua Laço Externo
Sair Laço Externo:
   c: 00008067  jalr  x0,x1,0    # retorno da função
Continua Laço Externo:
  10: 0006a803  lw    a6,0(a3)   # x = a[i]
  14: 00068613  addi  a2,a3,0    # a2 é ponteiro para a[j]
  18: 00070793  addi  a5,a4,0    # j = i
Laço interno:
  1c: ffc62883  lw    a7,-4(a2)  # a7 = a[j-1]
  20: 01185a63  bge   a6,a7,34   # se a[j-1] <= a[i], salta para Sair Laço Interno
  24: 01162023  sw    a7,0(a2)   # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ffc60613  addi  a2,a2,-4   # decrementa a2 para apontar para a[j]
  30: fe0796e3  bne   a5,x0,1c   # if j != 0, salta para Laço Interno
Sair Laço Interno:
  34: 00279793  slli  a5,a5,0x2  # multiplica a5 por 4
  38: 00f507b3  add   a5,a0,a5   # a5 agora é endereço de byte de a[j]
  3c: 0107a023  sw    a6,0(a5)   # a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00468693  addi  a3,a3,4    # incrementa a3 para apontar para a[i]
  48: fc1ff06f  jal   x0,8       # salta para Laço Externo
```

Figura 2.8: Código RV32I do algoritmo de Ordenação por Inserção na Figura 2.5. O endereço em hexadecimal é à esquerda, o código de máquina em hexadecimal é o próximo e, em seguida, a instrução da linguagem assembly seguida por um comentário. O RV32I aloca dois registradores para apontar para `a[j]` e `a[j-1]`. Tem-se muitos registradores, alguns dos quais a ABI reserva para chamadas de procedimento. Ao contrário dos outros ISAs, ele ignora salvar e restaurar esses registradores na memória. Embora o tamanho do código seja maior que x86-32, o uso das instruções RV32C opcionais (consulte o capítulo 7) fecha a lacuna de tamanho. Observe que as instruções de comparação e desvio evitam as três instruções de comparação exigidas pelo ARM-32 e x86-32.

```
# 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              # compara i com. n (desnecessário?)
  8: e1a0c000 mov  ip, r0              # ip = a[0]
  c: 212fff1e bxcs lr                  # não deixa o endereço de retorno mudar ISAs
 10: e92d4030 push {r4, r5, lr}        # salva r4, r5, endereços de retorno
Laço Externo:
 14: e5bc4004 ldr  r4, [ip, #4]!       # x = a[i] ; incrementa ip
 18: e1a02003 mov  r2, r3              # j = i
 1c: e1a0e00c mov  lr, ip              # lr = a[0] (utilizando lr como registrador de rascunho)
Laço Interno:
 20: e51e5004 ldr  r5, [lr, #-4]       # r5 = a[j-1]
 24: e1550004 cmp  r5, r4              # compara a[j-1] com x
 28: da000002 ble  38                  # if a[j-1]<=a[i], salta para Sair Laço Interno
 2c: e2522001 subs r2, r2, #1          # j--
 30: e40e5004 str  r5, [lr], #-4       # a[j] = a[j-1]
 34: 1afffff9 bne  20                  # se j != 0, salta para Laço Interno
Sair Laço Interno:
 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                  # se i < n, salta para Laço Externo
 48: e8bd8030 pop  {r4, r5, pc}        # restaura r4, r5, e retorna endereço
```

Figura 2.9: Código ARM-32 do algoritmo de Ordenação por Inserção na Figura 2.5. O endereço em hexadecimal é à esquerda, o código de máquina em hexadecimal é o próximo e, em seguida, a instrução da linguagem assembly seguida por um comentário. Com poucos registradores, o ARM-32 salva dois deles na pilha para posterior reutilização junto com o endereço de retorno. Ele utiliza um modo de endereçamento que dimensiona `i` e `j` como endereços de bytes. Dado que um desvio tem o potencial de alterar os ISAs entre o ARM-32 e o Thumb-2, o bxcs primeiro define o bit menos significativo do endereço de retorno para 0 antes de salvá-lo. Os códigos de condição salvam uma instrução de comparação para verificar `j` após decrementá-la, mas ainda há três comparações em outro lugar.

```
# 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 é ponteiro para a[i]
   4: 24030001 li    v1,1      # i = 1
Laço Externo:
   8: 0065102b sltu  v0,v1,a1  # "seta" quando i < n
   c: 14400003 bnez  v0,1c     # se i<n, salta para Continua Laço Externo
  10: 00c03825 move  a3,a2     # a3 é ponteiro para a[j] (slot preenchido)
  14: 03e00008 jr    ra        # rretorno da função
  18: 00000000 nop             # slot de atraso de desvio não preenchido
Continua Laço Externo:
  1c: 8cc80000 lw    t0,0(a2)  # x = a[i]
  20: 00601025 move  v0,v1     # j = i
Laço Interno:
  24: 8ce9fffc lw    t1,-4(a3) # t1 = a[j-1]
  28: 00000000 nop             # slot de atraso de load não preenchido
  2c: 0109502a slt   t2,t0,t1  # "seta" a[i] < a[j-1]
  30: 11400005 beqz  t2,48     # se a[j-1]<=a[i], salta para Sair Laço Interno
  34: 00000000 nop             # slot de atraso de desvio não preenchido
  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, salta para Laço Interno
  44: 24e7fffc addiu a3,a3,-4  # decr. a3 para apontar para a[j] (slot preenchido)
Sair Laço Interno:
  48: 00021080 sll   v0,v0,0x2 #
  4c: 00821021 addu  v0,a0,v0  # v0 agora endereço de byte de a[j]
  50: ac480000 sw    t0,0(v0)  # a[j] = x
  54: 24630001 addiu v1,v1,1   # i++
  58: 1000ffeb b     8         # salta para laço externo
  5c: 24c60004 addiu a2,a2,4   # incr. a2 para apontar para a[i] (slot preenchido)
```

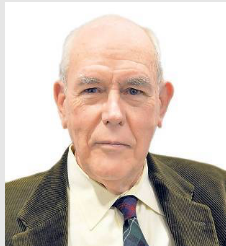
Figura 2.10: Código MIPS-32 do algoritmo de Ordenação por Inserção MIPS-32 na Figura 2.5. O endereço em hexadecimal é à esquerda, o código de máquina em hexadecimal é o próximo e, em seguida, a instrução da linguagem assembly seguida por um comentário. O código MIPS-32 possui três instruções `nop`, o que aumenta seu tamanho. Dois são devidos ao desvio atrasado e um é devido a um load atrasado. O compilador não conseguiu encontrar instruções úteis para colocar nesses intervalos de atraso. Os desvios atrasados também tornam o código mais difícil de entender, já que a instrução a seguir também é executada quando um desvio ou salto é executado. Por exemplo, a última instrução (`addiu`) no endereço 5c faz parte do loop, mesmo que ele esteja seguindo a instrução de desvio.

```
# x86-32 (20 instruções, 45 bytes)
# eax é j, ecx é x, edx é i
# ponteiro para a[0] está na memória no endereço esp+0xc, n está na memória em esp+0x10
  0: 56             push esi                # salva esi na pilha (esi necessário abaixo)
  1: 53             push ebx                # salva ebx na pilha (ebx necessário abaixo)
  2: ba 01 00 00 00 mov  edx,0x1            # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]      # ecx é ponteiro para a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]     # compara i com n
  f: 73 19          jae  2a <Exit Loop>     # se i >= n, salta para Sair Laço Externo
 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            # compara a[j-1] com x
 1c: 7e 06          jle  24 <Exit Loop>     # se a[j-1]<=a[i], salta para Sair Laço Interno
 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>    # se j != 0, salta para Laço Interno
Sair Laço Interno:
 24: 89 1c 81       mov  [ecx+eax*4],ebx    # a[j] = x
 27: 42             inc  edx                # i++
 28: eb e1          jmp  b <Outer Loop>     # salta para Laço Externo
Exit Outer Loop:
 2a: 5b             pop  ebx                # restaura o valor antigo de ebx da pilha
 2b: 5e             pop  esi                # restaura o valor antigo de esi da pilha
 2c: c3             ret                     # retorno da função
```

Figura 2.11: Código x86-32 do algoritmo de Ordenação por Inserção na Figura 2.5. O endereço em hexadecimal é à esquerda, o código de máquina em hexadecimal é o próximo e, em seguida, a instrução da linguagem assembly seguida por um comentário. Na falta de registradores, o x86-32 salva dois deles na pilha. Além disso, duas das variáveis alocadas para registradores em RV32I são mantidas na memória (`n` e o ponteiro para `a[0]`). Ele utiliza o modo de endereçamento Scaled Indexed para um bom efeito para acessar `a[i]` e `a[j]`. Sete das 20 instruções x86-32 têm um byte de comprimento, o que dá ao x86-32 um bom tamanho de código para este programa simples. Existem duas versões populares da linguagem assembly x86: Intel / Microsoft e AT & T / Linux. Utilizamos a sintaxe da Intel, em parte porque ela corresponde à ordem dos operandos RISC-V, ARM-32 e MIPS-32 com o destino à esquerda e a origem à direita, enquanto os operandos são vice-versa. AT & T (e os registradores prefixam um % antes de seus nomes). Esta questão aparentemente trivial é quase uma questão religiosa para alguns programadores. Pedagogia conduz a nossa escolha, não a ortodoxia.

3 Linguagem Assembly do RISC-V

Ivan Sutherland (1938-) é reconhecido como o pai da computação gráfica pela invenção do Sketchpad—o precursor da interface gráfica de usuário nos computadores atuais, criado em 1962—que mais tarde levou-o a receber um Turing Award.



It's very satisfying to take a problem we thought difficult and find a simple solution. The best solutions are always simple.

—Ivan Sutherland

3.1 Introdução

A Figura 3.1 ilustra as quatro etapas clássicas de tradução de um programa escrito na linguagem C para um programa em linguagem de máquina pronto para ser executado pelo computador. Este capítulo aborda as últimas três etapas, mas iniciamos com o papel que o *assembler* desempenha na convenção de chamada do RISC-V.

3.2 Convenção de Chamada

Existem seis estágios que são executados em uma chamada de função [Patterson and Hennessy 2017]:

1. Colocar os argumentos onde a função possa acessá-los.
2. Saltar para a função (utilizando a instrução `jal` do RV32I's).
3. Adquirir recursos de armazenamento local que a função necessita, salvando registradores conforme necessário.
4. Executar a tarefa desejada da função.
5. Colocar o valor do resultado da função onde o programa de chamada pode acessá-lo, restaurar qualquer registrador, e liberar quaisquer recursos de armazenamento local.
6. Como uma função pode ser chamada de vários pontos em um programa, retornar o controle para seu respectivo ponto de origem (utilizando `ret`).

Para obter um bom desempenho, tente manter as variáveis nos registradores em vez da memória, mas por outro lado, evite também acessar a memória frequentemente para salvar e restaurar esses valores.

O RISC-V, felizmente, tem registradores suficientes para oferecer o melhor dos dois mundos: manter os operandos nos registradores e reduzir a necessidade de salvá-los e restaurá-los.

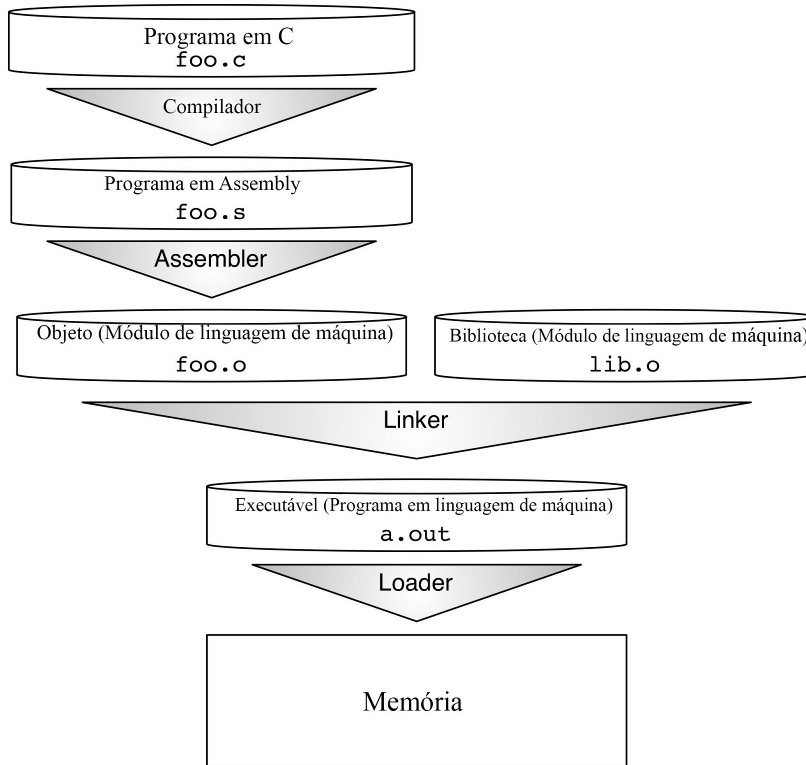


Figura 3.1: Passos de tradução de um código fonte C para um programa em execução. Estas são as etapas lógicas, embora algumas sejam combinadas para acelerar a tradução. Usamos a convenção de nomes de sufixos de arquivos Unix para cada tipo de arquivo. Os sufixos equivalentes no MS-DOS são `.C`, `.ASM`, `.OBJ`, `.LIB`, e `.EXE`.

Registrador	Nome ABI	Descrição	Preservado em toda a chamada?
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Endereço de retorno	Não
`x2`	`sp`	Ponteiro de pilha	Sim
`x3`	`gp`	Ponteiro global	—
`x4`	`tp`	Ponteiro de Thread	—
`x5`	`t0`	Registrador de link temporário/alternativo	Não
`x6–7`	`t1–2`	Temporários	Não
`x8`	`s0/fp`	Registrador salvo/Ponteiro de quadro	Sim
`x9`	`s1`	Registrador salvo	Sim
`x10–11`	`a0–1`	Argumentos de função / valores de retorno	Não
`x12–17`	`a2–7`	Argumentos de função	Não
`x18–27`	`s2–11`	Registradores salvos	Sim
`x28–31`	`t3–6`	Temporários	Não
`f0–7`	`ft0–7`	Temporários FP	Não
`f8–9`	`fs0–1`	Registradores salvos FP	Sim
`f10–11`	`fa0–1`	Argumentos e valores de retorno FP	Não
`f12–17`	`fa2–7`	Argumentos FP	Não
`f18–27`	`fs2–11`	Registradores salvos FP	Sim
`f28–31`	`ft8–11`	Temporários FP	Não

Figura 3.2: Mnemônicos do *Assembler* para registradores inteiros e de ponto flutuante do RISC-V. O RISC-V tem um número suficiente de registradores para que a ABI possa alocar registradores que procedimentos e métodos podem utilizar livremente sem a necessidade de salvar e restaurar seus valores nos casos em que outros procedimentos não são chamados. Os registradores preservados após uma chamada de procedimento também são denominados *caller saved*, e *callee saved* aqueles que não são. O Capítulo 5 descreve os registradores `f` de ponto flutuante. (Tabela 20.1 de [Waterman and Asanović 2017] é a base desta figura.)

A ideia é ter alguns registradores que não têm a garantia de serem preservados durante de uma chamada de função, chamados de *registradores temporários*(temporary registers), e alguns que são, os *registradores salvos*(saved registers). Funções que não realizam chamadas a outras funções são chamadas de funções *leaf*. Quando uma função leaf possui apenas alguns argumentos e variáveis locais, podemos manter tudo nos registradores sem "derramar" nada para a memória principal. Se essas condições persistirem, o programa não necessitará salvar os valores dos registradores na memória, e uma fração significativa de chamadas de função encontram-se nessa categoria.

Outros registradores dentro de uma chamada de função devem ser considerados da mesma classe que os registradores salvos, que são preservados através de uma chamada de função, ou na mesma classe que os registradores temporários, que não são preservados. Uma função alterará o(s) registrador(es) contendo o(s) valor(es) de retorno, como os mesmos sendo registradores temporários. Não há necessidade de preservar o endereço de retorno ou os argumentos da função, portanto, esses registradores são como temporários. O chamador pode confiar no último registradores, responsável pelo *stack pointer*, para permanecer inalterado após uma chamada de função. A Figura 3.2 lista os nomes de registradores do *RISC-V application binary interface*(ABI) e a convenção sobre os mesmos de serem ou não preservados nas chamadas de função.

A partir das convenções da ABI, é possível observar o código RV32I padrão para entrada

e saída de funções. A porção inicial *prologue* da função é dada da seguinte forma:

```
entry_label:
  addi sp,sp,-framesize    # Aloca espaço para stack frame
                           # ajustando stack pointer (registrador sp)
  sw   ra,framesize-4(sp)  # Salva endereço de retorno (registrador ra)
  # salva outros registradores na pilha, caso necessário
  ... # corpo da função
```

Se houverem muitos argumentos e variáveis de função para serem alocados nos registradores, o *prologue* da função aloca espaço na pilha para seu *frame*. Depois que a tarefa da função estiver completa, a posição final (*epilogue*) desfaz o *stack frame* e retorna ao ponto de origem da chamada de função.

```
# restaura estado dos registradores da pilha, caso necessário
lw   ra,framesize-4(sp)  # Restaura registrador ra
addi sp,sp, framesize    # Desaloca espaço do stack frame
ret                      # Retorna ao ponto de chamada
```

Veremos um exemplo que segue a ABI em breve, mas primeiro é necessário explicar as tarefas restantes do *assembly*, que encontram-se além da transformação dos nomes de registradores ABI em números de registradores.

■ ***Elaboração: Os registradores salvos e temporários não são contíguos***

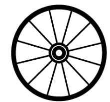
para fornecer suporte ao RV32E, uma versão embarcada do RISC-V que possui apenas 16 registradores (consulte o Capítulo 11). São utilizados números de registrador `x0` a `x15`, portanto, alguns registradores salvos e temporários estão neste intervalo, enquanto os restantes estão nos últimos 16. O RV32E é menor, mas ainda não possui suporte ao compilador, já que ele não corresponde ao RV32I.

3.3 Assembly

A entrada para esta etapa no Unix é um arquivo com o sufixo `.s`, como `foo.s`; para MS-DOS é da forma `.ASM`.

O trabalho da etapa *assembler* da Figura 3.1 não é simplesmente produzir código objeto a partir das instruções que são compreendidas pelo processador, mas estendê-las para incluir operações úteis para o programador *assembly* ou para o desenvolvedor do compilador. Esta categoria, baseada em configurações inteligentes de instruções regulares, é chamada de *pseudoinstruções*. As Figuras 3.3 e 3.4 listam as pseudoinstruções RISC-V, com as da primeira figura considerando o registrador x0 o valor zero, enquanto as da segunda lista não consideram esse contexto. Por exemplo, `ret` mencionado acima é na verdade uma pseudoinstrução em que o *assembler* substitui por `jalr x0, x1, 0` (veja a Figura 3.3). A maioria das pseudoinstruções RISC-V dependem de x0. Como você pode ver, reservar um dos 32 registradores para assumir o valor estático de zero simplifica bastante o conjunto de instruções do RISC-V, ao fornecer muitas operações populares —como *jump*, *return*, e *branch on equal to zero*— como pseudoinstruções.

A Figura 3.5 mostra o clássico programa "Hello world" escrito em C. A saída em *assembly* produzida pelo compilador é apresentada na Figura 3.6, usando a convenção de chamada da Figura 3.2 e as pseudoinstruções das Figuras 3.3 e 3.4.



Simplicidade

Tipicamente o programa "Hello world" é o primeiro programa executado em um processador recém projetado. Arquitetos de sistema tradicionalmente consideram a execução do sistema operacional bem o suficiente para imprimir "Hello world" como um forte sinal de que seu novo chip funciona. Eles enviam essa saída por e-mail para seus gerentes e colegas, e então comemoram.

Pseudo-Instrução	Instrução (ões) Base	Significado
`nop`	`addi x0, x0, 0`	Operação No
`neg rd, rs`	`sub rd, x0, rs`	Complemento de dois
`negw rd, rs`	`subw rd, x0, rs`	Palavra em complemento de dois
`snez rd, rs`	`sltu rd, x0, rs`	"Seta" se $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	"Seta" se $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	"Seta" se $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Desvia se $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Desvia se $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Desvia se $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Desvia se $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Desvia se $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Desvia se $>$ zero
`j offset`	`jal x0, offset`	Pula
`jr rs`	`jalr x0, rs, 0`	Registrador de pulo
`ret`	`jalr x0, x1, 0`	Retorna da sub-rotina
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Chamada de cauda sub-rotina far-away
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Lê o contador de instruções retiradas
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Lê o contador de ciclos
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Lê o relógio em tempo real
`csrr rd, csr`	`csrrs rd, csr, x0`	Lê o CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Escreve o CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	"Seta" bits em CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Limpa bits em CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Escreve CSR, imediato
`csrsi csr, imm`	`csrrsi x0, csr, imm`	"Seta" bits em CSR, imediato
`csrci csr, imm`	`csrrci x0, csr, imm`	Limpa bits em CSR, imediato
`frcsr rd`	`csrrs rd, fcsr, x0`	Lê o controle de FP/registrador de status
`fscsr rs`	`csrrw x0, fcsr, rs`	Escreve controle de FP/registrador de status
`frrm rd`	`csrrs rd, frm, x0`	Lê o modo de arredondamento do FP
`fsrm rs`	`csrrw x0, frm, rs`	Escreve o modo de arredondamento do
`frflags rd`	`csrrs rd, fflags, x0`	Lê flags de exceção de FP
`fsflags rs`	`csrrw x0, fflags, rs`	Escreve flags de exceção de FP

Figura 3.3: 32 pseudoinstruções RISC-V que utilizam o x0, o registrador zero. O Apêndice A inclui as pseudoinstruções RISC-V, bem como as instruções reais. Aquelas que lêem os contadores de 64 bits podem ler 32 bits superiores em RV32I utilizando a versão "h" das instruções, e os 32 bits inferiores usando a versão padrão. (Tabelas 20.2 e 20.3 of [Waterman and Asanović 2017] são a base desta figura.).

<table>
<tr><th>Pseudo-Instrução</th><th>Instrução (ões) Base</th><th>Significado</th></tr>
<tr><td>lla rd, symbol</td><td>auipc rd, symbol[31:12]
addi rd, rd, symbol[11:0]</td><td>Carrega endereço local</td></tr>
<tr><td>la rd, symbol</td><td>PIC: auipc rd, GOT[symbol][31:12]
l{w|d} rd, rd, GOT[symbol][11:0]
Non-PIC: Same as lla rd, symbol</td><td>Carrega endereço</td></tr>
<tr><td>l{b|h|w|d} rd, symbol</td><td>auipc rd, symbol[31:12]
l{b|h|w|d} rd, symbol[11:0](rd)</td><td>Load global</td></tr>
<tr><td>s{b|h|w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
s{b|h|w|d} rd, symbol[11:0](rt)</td><td>Store global</td></tr>
<tr><td>fl{w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
fl{w|d} rd, symbol[11:0](rt)</td><td>Load ponto-flutuante global</td></tr>
<tr><td>fs{w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
fs{w|d} rd, symbol[11:0](rt)</td><td>Store ponto-flutuante global</td></tr>
<tr><td>li rd, immediate</td><td>Miríades de sequências</td><td>Load valor imediato</td></tr>
<tr><td>mv rd, rs</td><td>addi rd, rs, 0</td><td>Copia registrador</td></tr>
<tr><td>not rd, rs</td><td>xori rd, rs, -1</td><td>Complemento de um</td></tr>
<tr><td>sext.w rd, rs</td><td>addiw rd, rs, 0</td><td>Estende o sinal da palavra</td></tr>
<tr><td>seqz rd, rs</td><td>sltiu rd, rs, 1</td><td>"Seta" se $=$ zero</td></tr>
<tr><td>fmv.s rd, rs</td><td>fsgnj.s rd, rs, rs</td><td>Copia registrador de precisão simples</td></tr>
<tr><td>fabs.s rd, rs</td><td>fsgnjx.s rd, rs, rs</td><td>Valor absoluto de precisão simples</td></tr>
<tr><td>fneg.s rd, rs</td><td>fsgnjn.s rd, rs, rs</td><td>Negação de precisão única</td></tr>
<tr><td>fmv.d rd, rs</td><td>fsgnj.d rd, rs, rs</td><td>Copiaa registrador de precisão dupla</td></tr>
<tr><td>fabs.d rd, rs</td><td>fsgnjx.d rd, rs, rs</td><td>Valor absoluto de precisão dupla</td></tr>
<tr><td>fneg.d rd, rs</td><td>fsgnjn.d rd, rs, rs</td><td>Negação de precisão dupla</td></tr>
<tr><td>bgt rs, rt, offset</td><td>blt rt, rs, offset</td><td>Desvia se $>$</td></tr>
<tr><td>ble rs, rt, offset</td><td>bge rt, rs, offset</td><td>Desvia se $\leq$</td></tr>
<tr><td>bgtu rs, rt, offset</td><td>bltu rt, rs, offset</td><td>Desvia se $>$, sem sinal</td></tr>
<tr><td>bleu rs, rt, offset</td><td>bgeu rt, rs, offset</td><td>Desvia se $\leq$, sem sinal</td></tr>
<tr><td>jal offset</td><td>jal x1, offset</td><td>Pula e linka</td></tr>
<tr><td>jalr rs</td><td>jalr x1, rs, 0</td><td>Jump e linka o registrador</td></tr>
<tr><td>call offset</td><td>auipc x1, offset[31:12]
jalr x1, x1, offset[11:0]</td><td>Chame a sub-rotina far-away</td></tr>
<tr><td>fence</td><td>fence iorw, iorw</td><td>Cerca em toda a memória e I/O</td></tr>
<tr><td>fscsr rd, rs</td><td>csrrw rd, fcsr, rs</td><td>Troca controle de FP/registrador de status</td></tr>
<tr><td>fsrm rd, rs</td><td>csrrw rd, frm, rs</td><td>Troca o modo de arredondamento do FP</td></tr>
<tr><td>fsflags rd, rs</td><td>csrrw rd, fflags, rs</td><td>Troca sinalizadores de exceção de FP</td></tr>
</table>

Figura 3.4: 28 pseudoinstruções RISC-V que são independentes de x0, o registrador zero. Para `la`, GOT significa *Global Offset Table*, a tabela que contém o endereço em tempo de execução de símbolos nas bibliotecas "linkadas" dinamicamente. O Apêndice A descreve estas pseudoinstruções RISC-V, bem como as instruções reais. (Tabelas 20.2 e 20.3 of [Waterman and Asanović 2017] são a base para esta figura.)

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

Figura 3.5: Programa Hello World escrito em C (`hello.c`).

```
  .text                     # Diretiva: insere a seção de texto
  .align 2                  # Diretiva: alinha o código a 2 ^ 2 bytes
  .globl main               # Diretiva: declara o símbolo global principal
main:                       # rótulo para início da função main:
  addi sp,sp,-16            # aloca quadro de pilha
  sw   ra,12(sp)            # salva o endereço de retorno
  lui  a0,%hi(string1)      # endereço de computação de
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # endereço de computação de
  addi a1,a1,%lo(string2)   #   string2
  call printf               # chama a função printf
  lw   ra,12(sp)            # restaura o endereço de retorno
  addi sp,sp,16             # desaloca o quadro de pilha
  li   a0,0                 # dá load no valor de retorno 0
  ret                       # retorna
  .section .rodata          # Diretiva: insira a seção de dados somente leitura
  .balign 4                 # Diretiva: alinha a seção de dados a 4 bytes
string1:                    # rótulo para primeira string
  .string "Hello, %s!\n"    # Diretiva: string terminada com nulo
string2:                    # rótulo para segunda string
  .string "world"           # Diretiva: string terminada com nulo
```

Figura 3.6: Programa Hello World na linguagem assembly do RISC-V (`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
```

Figura 3.7: Programa Hello World na linguagem de máquina do RISC-V (`hello.o`). As seis instruções que são posteriormente corrigidas pelo *linker* (locais 8 a 1c) têm zero em seus campos de endereço. A tabela de símbolos registra os rótulos e endereços de todas as instruções que necessitam ser editadas pelo *linker*.

```
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
```

Figura 3.8: Programa Hello World na linguagem de máquina do RISC-V após linkagem. Em sistemas Unix, o arquivo seria nomeado `a.out`.

Os comandos que iniciam com um ponto são as *diretivas assembler*. Esses são comandos específicos para o *assembler*, e não código para ser traduzido por ele. Estas diretivas informam ao *assembler* onde colocar código e dados, especificam constantes de código e dados para uso no programa, e assim por diante. A Figura 3.9 apresenta as diretivas *assembler* do RISC-V. Para o exemplo da Figura 3.6, as diretivas são:

- `.text`—Enter text section.
- `.align 2`—Align following code to 2^2 bytes.
- `.globl main`—Declare global symbol "main".
- `.section .rodata`—Enter read-only data section.
- `.balign 4`—Align data section to 4 bytes.
- `.string ''Hello, %s!\n''`—Create this null-terminated string.
- `.string ''world''`—Create this null-terminated string.

O *assembler* produz o arquivo de objeto (Figura 3.7) seguindo o padrão *Executable and Linkable Format*(ELF) [TIS Committee 1995].

3.4 Linker

Em vez de compilar todo o código-fonte toda vez que um arquivo é alterado, o *Linker*, ou ligador, permite que arquivos individuais sejam compilados e montados separadamente. Ele "costura" o novo código objeto junto aos módulos de linguagem de máquina existentes, como bibliotecas. Seu nome é derivado de uma de suas tarefas, que é editar todos os links das instruções de jump and link no arquivo objeto. Na verdade, o *linker* é a abreviação para editor de links, que foi historicamente o nome dessa etapa na Figura 3.1. Em sistemas Unix, a entrada para o linker são arquivos com o sufixo `.o` (E.x., `foo.o, libc.o`), e sua saída é um arquivo `a.out`. Para MS-DOS, as entradas são arquivos com o sufixo `.OBJ` ou `.LIB` e a saída é um arquivo `.EXE`.

A Figura 3.10 apresenta os endereços das regiões de memória alocadas para código e dados em um típico programa RISC-V. O linker deve ajustar o programa e os endereços de dados das instruções em todos os arquivos de objeto para corresponder aos endereços nessa figura. É menos trabalho para o linker se os arquivos de entrada conterem *código independente de posição* (*PIC*). O que significa que todas os desvios a instruções e referências a dados dentro do arquivo estarão corretas onde quer que o código seja executado. Como mencionado no Capítulo 2, o desvio relativo ao PC do RV32I torna o PIC muito mais fácil de ser garantido.

abc
Facilidade de Programação

Além das instruções, cada arquivo de objeto contém uma tabela de símbolos que inclui todos os rótulos no programa que devem receber endereços como parte do processo de vinculação realizado pelo linker. Essa lista inclui rótulos para dados, bem como para código. A Figura 3.6 tem dois rótulos de dados a serem "setados" (`string1` e `string2`) e dois rótulos de código a serem designados (`main` e `printf`). Um endereço de 32 bits é difícil de encaixar em uma instrução de 32 bits, portanto o linker deve ajustar duas instruções por rótulo no código RV32I, como mostra a Figura 3.6: `lui` e `addi` para endereços de dados, e `auipc` e `jalr` para endereços de código. A Figura 3.8 é a versão final `a.out`, correspondendo ao arquivo objeto na Figura 3.7.

Directive	Description
`.text`	Itens subsequentes são armazenados na seção de texto (código de máquina).
`.data`	Itens subsequentes são armazenados na seção de dados (variáveis globais).
`.bss`	Itens subsequentes são armazenados na seção bss (variáveis globais inicializadas em 0).
`.section .foo`	Itens subsequentes são armazenados na seção denominada `.foo`.
`.align n`	Alinha o próximo dado em um limite de 2^n-byte. Por exemplo, `.align 2` alinha o próximo valor em um limite de palavra.
`.balign n`	Alinha o próximo dado em um limite de n-byte. Por exemplo, `.balign 4` alinha o próximo valor em um limite de palavra.
`.globl sym`	Declara que o label sym é global e pode ser referenciado de outros arquivos.
`.string "str"`	Armazena a string str na memória e termina-a através de NULL.
`.byte b1,..., bn`	Armazena as quantidades n de 8 bits em bytes sucessivos de memória.
`.half w1,...,wn`	Armazena as n quantidades de 16 bits em meias palavras de memória sucessivas.
`.word w1,...,wn`	Armazena as n quantidades de 32 bits em palavras de memória sucessivas.
`.dword w1,...,wn`	Armazena as n quantidades de 64 bits em palavras duplas de memória sucessivas.
`.float f1,..., fn`	Armazena os n números de ponto flutuante de precisão única em palavras de memória sucessivas.
`.double d1,..., dn`	Armazene os n números de ponto flutuante de precisão dupla em palavras duplas de memória sucessivas.
`.option rvc`	Comprime as instruções subsequentes (veja o Capítulo 7).
`.option norvc`	Não comprime as instruções subsequentes.
`.option relax`	Permite relaxamentos do linker para instruções subsequentes.
`.option norelax`	Não permite relaxamentos do linker para instruções subsequentes.
`.option pic`	As instruções subsequentes são códigos independentes de posição.
`.option nopic`	Instruções subsequentes são códigos dependentes da posição.
`.option push`	Empurra a configuração atual de todos `.options` para uma pilha, de modo que um `.option pop` subsequente irá restaurar seu valor.
`.option pop`	Aplica um Pop na pilha de opções, restaurando todos `.options` a sua configuração no tempo do último `.option push`.

Figura 3.9: Diretivas assembler comuns do RISC-V.

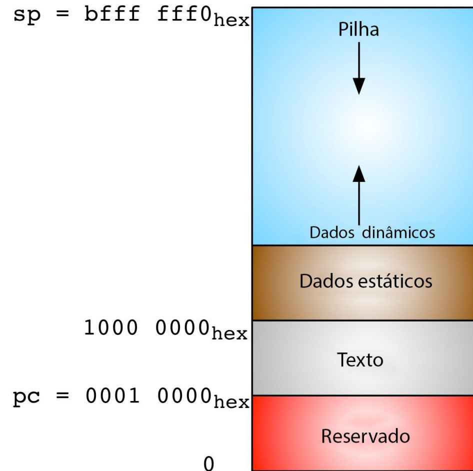


Figura 3.10: Alocação de memória para programa e dados do RV32I. Os endereços superiores estão no topo da figura, enquanto os inferiores estão na parte inferior. Nesta convenção de software do RISC-V, o ponteiro de pilha (`sp`) inicia em bfff fff0$_{hex}$ e cresce para baixo em direção aos dados estáticos. O *text* (código do programa) inicia em 0001 0000$_{hex}$ e inclui todas bibliotecas linkadas dinamicamente. Os dados estáticos iniciam imediatamente acima da região do text; Neste exemplo, consideramos que o endereço é 1000 0000$_{hex}$. Dados dinâmicos, alocados em C por `malloc()`, estão logo acima dos dados estáticos. Chamado de `heap`, este conjunto cresce em direção à pilha, e inclui todas bibliotecas linkadas dinamicamente.

Os compiladores RISC-V suportam várias ABIs, dependendo se as extensões F e D estão presentes. Para o RV32, as ABIs são denominadas ilp32, ilp32f e ilp32d. ilp32 significa que os tipos de dados de linguagem C int, long e pointer são todos de 32 bits; o sufixo opcional indica como os argumentos de ponto flutuante são passados. No ilp32, argumentos de ponto flutuante são passados em registradores de inteiros. No ilp32f, argumentos de ponto flutuante de precisão simples são passados em registradores de ponto flutuante. No ilp32d, argumentos de ponto flutuante de precisão dupla também são passados em registradores de ponto flutuante.

Naturalmente, para passar um argumento de ponto flutuante em um registrador de ponto flutuante, é necessário a extensão de ponto flutuante F ou D da ISA (consulte o Capítulo 5). Para compilar o código para o RV32I (GCC flag '-march=rv32i'), você deve usar o ABI ilp32 (GCC flag '-mabi=ilp32'). Por outro lado, ter instruções de ponto flutuante não significa que a convenção de chamada deva utilizá-las; Assim, por exemplo, o RV32IFD é compatível com todas as três ABIs: ilp32, ilp32f e ilp32d.

O linker verifica se a ABI do programa corresponde a todas as suas bibliotecas. Embora o compilador tenha suporte muitas combinações de extensões ISA e ABIs, apenas alguns conjuntos de bibliotecas podem ser instalados. Assim, um erro comum é vincular um programa sem ter as bibliotecas compatíveis instaladas. O linker não produzirá uma mensagem de diagnóstico útil nesse caso; ele tentará simplesmente vincular-se a uma biblioteca incompatível e, em seguida, informará a incompatibilidade. Esse erro geralmente ocorre apenas quando compila-se em um computador para um computador diferente (*cross compiling*).

■ ***Elaboração: Relaxamento do Linker***

A instrução de *jump and link* tem um campo de endereço relativo ao PC de 20 bits, portanto com uma única instrução é possível realizar um salto mais longo. Enquanto o compilador gera duas instruções para cada função externa, muitas vezes apenas uma instrução é necessária. Como essa otimização economiza tempo e espaço, o linker fará passagens sobre o código para substituir sempre que puder duas instruções por uma. Como um passe pode reduzir a distância entre uma chamada e a função, de modo que ela se encaixa em uma única instrução, o linker continua otimizando o código até que não haja mais alterações. Este processo é chamado de *Linker relaxation*, com o nome referindo-se a técnicas de relaxamento para resolver sistemas de equações. Além das chamadas de procedimento, o "linkador" RISC-V relaxa o endereçamento de dados para usar o ponteiro global quando o dado está dentro de ± 2 KiB de `gp`, removendo um `lui` ou `auipc`. Ele também relaxa o endereçamento de armazenamento local quando o dado está dentro de ± 2 KiB de `tp`.

3.5 Linkagem Estática vs. Linkagem Dinâmica

Arquitetos normalmente medem o desempenho do processador usando benchmarks *static linked* apesar da maioria dos programas reais serem *linkado dinamicamente*. A desculpa é que os usuários interessados em desempenho devem linkar estaticamente, mas isso é uma justificativa fraca. Faz mais sentido acelerar o desempenho de programas reais, e não de benchmarks.

A seção anterior descreve a *linkagem estática*, onde todo o código de biblioteca potencial é vinculado e, em seguida, carregado antes da execução. Essas bibliotecas podem ser relativamente grandes, portanto, vincular uma biblioteca popular a vários programas desperdiça memória. Além disso, as bibliotecas são legadas quando vinculadas—mesmo quando são atualizadas posteriormente para correção de bugs—forçando o código vinculado estaticamente a usar a versão legado com bugs.

Para evitar ambos problemas, a maioria dos sistemas dependem de *linkagem dinâmica*, onde uma função externa desejada é carregada e vinculada ao programa somente após ser chamada pela primeira vez; se nunca for chamada, nunca é carregada e vinculada. Cada chamada após a primeira usa um link de acesso rápido, portanto, a sobrecarga dinâmica acontece apenas uma vez. Quando um programa é iniciado, o mesmo vincula-se à versão atual das funções da biblioteca de que esse necessita, que é como ele obtém a versão mais recente. Além disso, se vários programas usam a mesma biblioteca vinculada dinamicamente, o código da biblioteca aparece apenas uma vez na memória.

O código que o compilador gera se assemelha àquele gerado para linkagem estática. Em vez de saltar para uma função real, ele salta para funções de *stub* (funções curtas de aproximadamente três instruções). A função stub carrega o endereço da função real de uma tabela na memória e, em seguida, salta para ela. No entanto, na primeira chamada a tabela não possui o endereço da função real, mas contém o endereço da rotina de linkagem dinâmica. Quando chamado, o linker dinâmico usa a tabela de símbolos para localizar a função real, copia-a na memória e atualiza a tabela para apontar para a função real. Cada chamada subsequente paga apenas a sobrecarga de três instruções da função stub.

3.6 Loader

Um programa como o apresentado na Figura 3.8 é um arquivo executável armazenado no computador. Quando esse deve ser executado, o trabalho do *Loader* é de carregá-lo na memória e pular para o endereço inicial. O “loader” hoje é o sistema operacional; dito de outra maneira, a carga de `a.out` é uma das muitas tarefas de um sistema operacional.

A carga é uma tarefa um pouco mais complicada para programas linkados dinamicamente. Em vez de simplesmente iniciar o programa, o sistema operacional deve iniciar o linkador dinâmico, que por sua vez inicia o programa desejado e, em seguida, lida com todas as chamadas externas, copia as funções na memória e edita o programa após cada chamada para apontá-lo para a função correta.

3.7 Considerações Finais

Keep it simple, stupid.

—Kelly Johnson, engenheiro aeronáutico que criou o "KISS Principle," 1960

Facilidade de Programação

Custo

Desempenho

Elegância

O assembler aprimora uma simples ISA RISC-V com 60 pseudo-instruções que tornam o código RISC-V mais fácil de ler e escrever sem aumentar os custos de hardware. Simplesmente dedicar um registrador RISC-V para assumir o valor zero habilita muitas operações úteis. As instruções *Load Upper Immediate* (`lui`) e *Add Upper Immediate to PC* (`auipc`) tornam mais fácil para o compilador e o linker ajustar endereços de dados e funções externas, e os desvios relativas ao PC facilitam o trabalho do linker com código independente de posição. Possuir vários registradores permite uma convenção de chamada que torna a chamada e retorno de função mais rápidas, reduzindo o número de derramamentos e restaurações de registradores.

O RISC-V oferece uma excelente seleção de mecanismos simples e impactantes, que reduzem custos, melhoram o desempenho e facilitam a programação.

3.8 Para Saber Mais

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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

4 RV32M: Multiplicação e Divisão

William de Occam (1287-1347) foi um teólogo inglês que promoveu o que hoje é chamado de "navalha de Occam", uma preferência pela simplicidade no método científico.

Entities should not be multiplied beyond necessity.

—William de Occam, *ca.*1320

4.1 Introdução

A extensão RV32M adiciona instruções de multiplicação e divisão de números inteiros para RV32I. A Figura 4.1 é uma representação gráfica do conjunto de instruções da extensão RV32M e Figura 4.2 lista seus respectivos *opcodes*.

O processo de divisão é simples. Lembre-se de que:

$$Quociente = (Dividendo - Restante) \div Divisor$$

ou então

$$Dividendo = Quociente \times Divisor + Restante$$

$$Restante = Dividendo - (Quociente \times Divisor)$$

A RV32M tem instruções de divisão para números inteiros com e sem sinal: *divide* (`div`) e *divide unsigned* (`divu`), que colocam o quociente no registrador de destino. De forma menos

`srl` **pode realizar divisões sem sinal por** 2^i**.** Por exemplo, se a2 = 16 (2^4), então `srli t2,a1,4` produz o mesmo valor que `divu t2,a1,a2`.

RV32M

multiply

multiply **h**igh { _ / **u**nsigned / **s**igned **u**nsigned }

{ **div**ide / **rem**ainder } { _ / **u**nsigned }

Figura 4.1: Diagrama das instruções RV32M.

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

Figura 4.2: O mapa de opcode da RV32M possui layout de instruções, opcodes, tipo de formatos e nomes. (Tabela 19.2 de [Waterman and Asanović 2017] é a base para esta figura.)

```
# Calcule a divisão sem sinal de a0 por 3 utilizando multiplicação.
   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)
```

Figura 4.3: Código RV32M para dividir por uma constante multiplicando. É necessária uma análise numérica cuidadosa para mostrar que esse algoritmo funciona para qualquer dividendo e, para alguns outros divisores, a etapa de correção é mais complicada. A prova de correção e o algoritmo para gerar os recíprocos e as etapas de correção estão em [Granlund and Montgomery 1994].

frequente, os programadores querem o resto em vez do quociente, então RV32M oferece as instruções *remainder* (`rem`) e *remainder unsigned* (`remu`), que gravam o resto em vez do quociente.

`sll` **pode realizar uma multiplicação com e sem sinal por 2^i.** Por exemplo, se a2 = 16 (2^4) então `slli t2,a1,4` produz o mesmo valor que `mul t2,a1,a2`.

A equação de multiplicação é simplesmente:

$$Produto = Multiplicando \times Multiplicador$$

Essa é mais complicada do que a da divisão porque o tamanho do produto é a soma dos tamanhos do multiplicador e do multiplicando; a multiplicação de dois números de 32 bits gera um produto de 64 bits. Para produzir um produto de 64 bits devidamente com ou sem sinal, o RISC-V possui quatro instruções de multiplicação. Para obter o produto inteiro de 32 bits — os 32 bits inferiores do produto completo — utilize `mul`. Para obter os 32 bits superiores do produto de 64 bits, utilize `mulh` se ambos os operandos estiverem na representação com sinal, `mulhu` caso contrário e `mulhsu` se um estiver com sinal, mas o outro não. Como isso complicaria o hardware para gravar o produto de 64 bits em dois registradores de 32 bits em uma única instrução, o RV32M exige duas instruções múltiplas para produzir um produto de 64 bits.

Para muitos microprocessadores, a divisão de números inteiros é uma operação relativamente lenta. Como mencionado acima, os deslocamentos à direita podem substituir a divisão de um número sem sinal por potências de 2. Acontece que a divisão por outras constantes também pode ser otimizada através da multiplicação pela aproximação recíproca, aplicando uma correção à metade superior do produto. Por exemplo, a Figura 4.3 mostra o código da divisão de um número sem sinal por 3.

Para quase todos processadores, multiplicações são mais lentas que deslocamentos ou adições e divisões são muito mais lentas que multiplicações.

Em que difere? O ARM-32 por muito tempo teve instruções de multiplicação, mas nenhuma de divisão. A inclusão da divisão não se tornou obrigatória até 2005, quase 20

anos após o primeiro processador ARM. O MIPS-32 usa registradores especiais (HI e LO) como os únicos registradores destino para instruções de multiplicação e divisão. Embora esse design tenha reduzido a complexidade das implementações iniciais do MIPS, é necessária uma instrução de movimentação extra para usar o resultado da multiplicação ou divisão, potencialmente reduzindo o desempenho. Os registradores HI e LO também aumentam o estado arquitetural, tornando-o um pouco mais lento para alternar entre diferentes tarefas.

■ ***Elaboração:*** `mulh` ***e*** `mulhu` ***podem verificar se há overflow na multiplicação.***

Não há overflow ao usar `mul` para multiplicação sem sinal se o resultado de `mulhu` for zero. Da mesma forma, não há overflow ao usar `mul` para multiplicação com sinal se todos os bits no resultado de `mulh` corresponderem ao bit de sinal do resultado de `mul`, ou seja, igual a 0 se positivo ou ffff ffff_{hex} se negativo.

■ ***Elaboração: Também é fácil verificar a divisão por zero.***

Basta adicionar um teste `beqz` do divisor antes da divisão. O RV32I não gera um trap em divisão por zero porque poucos programas requerem esse comportamento e os que o fazem podem facilmente verificar em software se há zero. É claro que as divisões por constantes nunca precisam de verificações.

■ ***Elaboração:*** `mulhsu` ***é útil para multiplicações com várias palavras.***

`mulhsu` gera a metade superior do produto quando o multiplicador possui sinal e o multiplicando não. É como um subpasso de multiplicação com sinal de múltiplas palavras ao multiplicar a palavra mais significativa do multiplicador (que contém o bit de sinal), com as palavras menos significativas do multiplicando (que são sem sinal). Esta instrução melhora o desempenho da multiplicação de várias palavras em cerca de 15%.

4.2 Considerações Finais

The cheapest, fastest and most reliable components of a computer system are those that aren't there.

—

Para oferecer o menor processador RISC-V para aplicativos embarcados, multiplicar e dividir fazem parte da primeira extensão padrão opcional do RISC-V. Desta forma, muitos processadores RISC-V incluirão o RV32M.

4.3 Para Saber Mais

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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

5 RV32F e RV32D: Ponto Flutuante de Precisão Simples e Dupla

Antoine de Saint-Exupéry (1900-1944) foi um escritor e aviador francês mais conhecido pelo livro *O Pequeno Príncipe*.

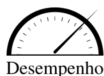
Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

—Antoine de Saint-Exupéry, *Terre des Hommes*, 1939

5.1 Introdução

Embora RV32F e RV32D sejam duas extensões distintas presentes no conjunto de instruções opcionais, elas são frequentemente incluídas juntas. Dadas as versões de precisão simples e dupla (32 e 64 bits) de quase todas as instruções de ponto flutuante, apresentamos as mesmas em um único capítulo por uma questão de simplificação. A Figura 5.1 é uma representação gráfica dos conjuntos de instruções de extensão RV32F e RV32D. A Figura 5.2 lista os opcodes do RV32F e a Figura 5.3 lista os opcodes do RV32D. Como virtualmente todos os outros ISAs modernos, o RISC-V obedece ao padrão de ponto flutuante IEEE 754-2008 [IEEE Standards Committee 2008].

5.2 Registradores de Ponto Flutuante



RV32F e o RV32D usam 32 registradores `f` separados em vez dos registradores `x`. A principal razão para os dois conjuntos é que os processadores podem melhorar o desempenho dobrando a capacidade de armazenamento e a largura de banda, tendo dois conjuntos de registradores sem aumentar o espaço para o especificador de registradores no formato de instrução RISC-V. O principal impacto no conjunto de instruções é ter novas instruções para carregar e armazenar os registradores `f` e para transferir dados entre os registradores `x` e `f`. A Figura 5.4 lista os registradores RV32D e RV32F e seus nomes, conforme determinado pela ABI RISC-V.

Se um processador tiver ambos RV32F e RV32D, os dados de precisão simples usarão apenas os 32 bits inferiores dos registradores `f`. Ao contrário de x0 em RV32I, o registrador `f0` não é *hardwired* para 0, mas é um registrador alterável como todos os outros registradores 31 `f`.

O padrão IEEE 754-2008 fornece várias maneiras de arredondar a aritmética de ponto flutuante, que são úteis para determinar limites de erro e escrever bibliotecas numéricas. O mais preciso e mais comum é o arredondamento para o par mais próximo (*Round Next Even*—RNE). O modo de arredondamento é "setado" no controle de ponto flutuante e no registrador

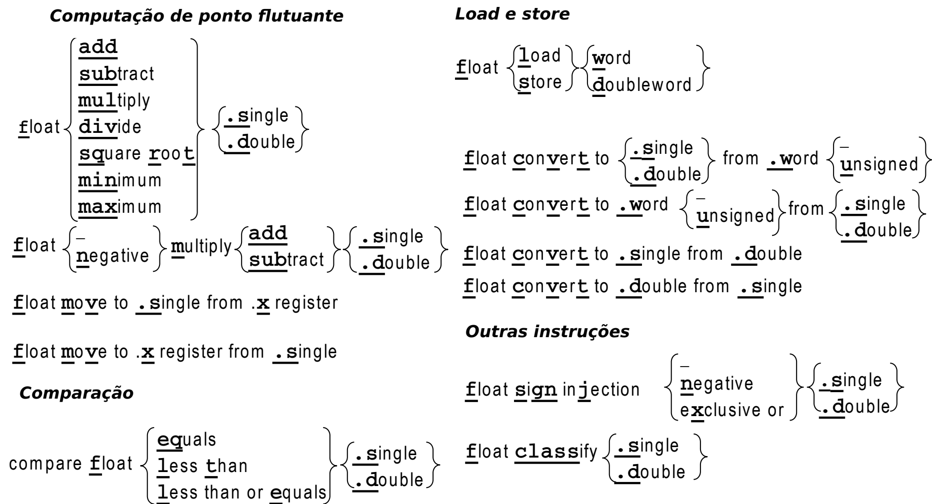


Figura 5.1: Diagrama das instruções RV32F e RV32D.

de status `fcsr`. A Figura 5.5 mostra `fcsr` e lista as opções de arredondamento, contendo também as flags acumulativas de exceção exigidas pelo padrão.

Em que difere? Tanto o ARM-32 quanto o MIPS-32 possuem 32 registradores de ponto flutuante de precisão simples, mas apenas 16 registradores de precisão dupla. Ambos mapeiam dois registradores de precisão simples nas metades de 32 bits esquerda e direita de um registrador de precisão dupla. A aritmética de ponto flutuante x86-32 não possui nenhum registrador, mas usa uma pilha. As entradas da pilha tinham 80 bits de largura para melhorar a precisão, por isso instruções *load* convertiam operandos de 32 ou 64 bits para 80 bits e vice-versa para instruções *store*. Uma versão subsequente do x86-32 incluiu 8 registradores tradicionais de ponto flutuante de 64 bits e instruções associadas. Ao contrário do RV32FD e do MIPS-32, o ARM-32 e o x86-32 ignoraram as instruções para mover dados diretamente entre os registradores de ponto flutuante e números inteiros. A única solução é armazenar um registrador de ponto flutuante na memória e carregá-lo da memória para um registrador inteiro e vice-versa.

Ter apenas 16 registradores de precisão dupla foi o maior erro do ISA no MIPS, de acordo com John Mashey, um de seus arquitetos.

■ ***Elaboração: O RV32FD permite que o modo de arredondamento seja "setado" por instrução.***

Chamado *static rounding*, ajuda o desempenho quando você só precisa alterar o modo de arredondamento para uma instrução. O padrão é usar o modo de arredondamento dinâmico em `fcsr`. O arredondamento estático é especificado como um último argumento opcional, pois `fadd.s ft0,ft1,ft2,rtz` irá arredondar para zero, independentemente de `fcsr`. A legenda da Figura 5.5 lista os nomes dos modos de arredondamento.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0000111	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	S fsw
rs3	00	rs2	rs1	rm	rd	1000011	R4 fmadd.s
rs3	00	rs2	rs1	rm	rd	1000111	R4 fmsub.s
rs3	00	rs2	rs1	rm	rd	1001011	R4 fnmsub.s
rs3	00	rs2	rs1	rm	rd	1001111	R4 fnmadd.s
0000000		rs2	rs1	rm	rd	1010011	R fadd.s
0000100		rs2	rs1	rm	rd	1010011	R fsub.s
0001000		rs2	rs1	rm	rd	1010011	R fmul.s
0001100		rs2	rs1	rm	rd	1010011	R fdiv.s
0101100		00000	rs1	rm	rd	1010011	R fsqrt.s
0010000		rs2	rs1	000	rd	1010011	R fsgnj.s
0010000		rs2	rs1	001	rd	1010011	R fsgnjn.s
0010000		rs2	rs1	010	rd	1010011	R fsgnjx.s
0010100		rs2	rs1	000	rd	1010011	R fmin.s
0010100		rs2	rs1	001	rd	1010011	R fmax.s
1100000		00000	rs1	rm	rd	1010011	R fcvt.w.s
1100000		00001	rs1	rm	rd	1010011	R fcvt.wu.s
1110000		00000	rs1	000	rd	1010011	R fmv.x.w
1010000		rs2	rs1	010	rd	1010011	R feq.s
1010000		rs2	rs1	001	rd	1010011	R flt.s
1010000		rs2	rs1	000	rd	1010011	R fle.s
1110000		00000	rs1	001	rd	1010011	R fclass.s
1101000		00000	rs1	rm	rd	1010011	R fcvt.s.w
1101000		00001	rs1	rm	rd	1010011	R fcvt.s.wu
1111000		00000	rs1	000	rd	1010011	R fmv.w.x

Figura 5.2: Mapa de opcode RV32F contém layout de instrução, opcodes, tipo de formato e nomes. A principal diferença nas codificações entre esta e a próxima figura é que o bit 12 é 0 para as duas primeiras instruções e o bit 25 é 0 para o resto das instruções, onde ambos os bits são 1 em RV32D. (Tabela 19.2 de [Waterman and Asanović 2017] é a base desta figura.)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsd
rs3	01	rs2	rs1	rm	rd	1000011	R4 fmadd.d
rs3	01	rs2	rs1	rm	rd	1000111	R4 fmsub.d
rs3	01	rs2	rs1	rm	rd	1001011	R4 fnmsub.d
rs3	01	rs2	rs1	rm	rd	1001111	R4 fnmadd.d
0000001		rs2	rs1	rm	rd	1010011	R fadd.d
0000101		rs2	rs1	rm	rd	1010011	R fsub.d
0001001		rs2	rs1	rm	rd	1010011	R fmul.d
0001101		rs2	rs1	rm	rd	1010011	R fdiv.d
0101101		00000	rs1	rm	rd	1010011	R fsqrt.d
0010001		rs2	rs1	000	rd	1010011	R fsgnj.d
0010001		rs2	rs1	001	rd	1010011	R fsgnjn.d
0010001		rs2	rs1	010	rd	1010011	R fsgnjx.d
0010101		rs2	rs1	000	rd	1010011	R fmin.d
0010101		rs2	rs1	001	rd	1010011	R fmax.d
0100000		00001	rs1	rm	rd	1010011	R fcvt.s.d
0100001		00000	rs1	rm	rd	1010011	R fcvt.d.s
1010001		rs2	rs1	010	rd	1010011	R feq.d
1010001		rs2	rs1	001	rd	1010011	R flt.d
1010001		rs2	rs1	000	rd	1010011	R fle.d
1110001		00000	rs1	001	rd	1010011	R fclass.d
1100001		00000	rs1	rm	rd	1010011	R fcvt.w.d
1100001		00001	rs1	rm	rd	1010011	R fcvt.wu.d
1101001		00000	rs1	rm	rd	1010011	R fcvt.d.w
1101001		00001	rs1	rm	rd	1010011	R fcvt.d.wu

Figura 5.3: Mapa de opcode RV32D contém layout de instrução, opcodes, tipo de formato e nomes. Existem algumas instruções nestas duas figuras que não diferem simplesmente pela largura dos dados. Esta figura possui exclusivamente `fcvt.s.d` e `fcvt.d.s`, enquanto a outra tem `fmv.x.w` e `fmv.w.x`. (Tabela 19.2 de [Waterman and Asanović 2017] é a base desta figura.)

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

Figura 5.4: Os registradores de ponto flutuante de RV32F e RV32D. Os registradores de precisão simples ocupam a metade mais à direita dos 32 registradores de precisão dupla. O Capítulo 3 explica a convenção de chamada RISC-V para os registradores de ponto flutuante, a lógica por trás dos registradores FP Argument (fa0-fa7), FP Saved (fs0-fs11) e FP Temporaries (ft0 -ft11). (Tabela 20.1 de [Waterman and Asanović 2017] é a base desta figura.)

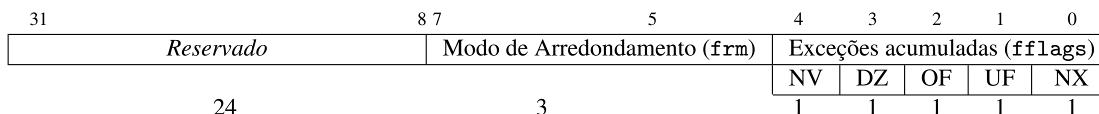


Figura 5.5: Controle de ponto flutuante e registrador de status. Esse contém os modos de arredondamento e as flags de exceção. Os modos de arredondamento são: arredondamento para os mais próximos, desempate par (rte, 000 em `frm`); arredondamento para zero (rtz, 001); arredondamento para baixo, em direção a $-\infty$ (rdn, 010); arredondamento para $+\infty$ (rup, 011); e arredondamento para o mais próximo, desempate por máxima magnitude (rmm, 100). Os cinco sinalizadores de exceção acumulados indicam as condições de exceção que surgiram em qualquer instrução aritmética de ponto flutuante desde que o campo foi redefinido pela última vez pelo software: NV é Operação Inválida; DZ é Divide by Zero; OF é Overflow; UF é Underflow; e o NX é Inexact. (Figura 8.2 de [Waterman and Asanović 2017] é a base desta figura.)

5.3 Loads, Stores e Aritmética de Ponto Flutuante

O RISC-V possui duas instruções de load (`flw`, `fld`) e duas instruções de store (`fsw`, `fsd`) para RV32F e RV32D. Ambos possuem o mesmo modo de endereçamento e formato de instrução como `lw` e `sw`.

Ao contrário da aritmética inteira, o tamanho do produto de uma multiplicação de ponto flutuante é o mesmo que seus operandos. Além disso, RV32F e RF32D omitem as instruções de resto de ponto flutuante.

Adicionando às operações aritméticas padrão (`fadd.s, fadd.d, fsub.s, fsub.d, fmul.s, fmul.d, fdiv.s, fdiv.d`), RV32F e RV32D incluem raiz quadrada (`fsqrt.s, fsqrt.d`). Eles também têm mínimo e máximo (`fmin.s, fmin.d, fmax.s, fmax.d`), que gravam os valores menores ou maiores do par de operandos de origem sem usar uma instrução de desvio.

Muitos algoritmos de ponto flutuante, como multiplicação de matriz, executam uma multiplicação imediatamente seguida por uma adição ou uma subtração. Assim, o RISC-V oferece instruções que multiplicam dois operandos e, em seguida, adicionam (`fmadd.s, fmadd.d`) ou subtraem (`fmsub.s, fmsub.d`) um terceiro operando a esse produto antes escrevendo a soma. Ele também possui versões que negam o produto antes de adicionar ou subtrair o terceiro operando: `fnmadd.s, fnmadd.d, fnmsub.s, fnmsub.d`. Estas instruções de adição múltipla *fundidas* são requeridas pelo padrão IEEE 754-2008 para sua precisão aumentada: elas são arredondadas apenas uma vez (após o acréscimo) ao invés de duas vezes (após a multiplicação, depois do acréscimo). Ignorar o arredondamento intermediário faz uma grande diferença quando o produto e a adição possuem magnitudes semelhantes, mas sinais opostos, o que faz com que a maioria dos bits da mantissa seja cancelada em subtração. Essas instruções precisam de um novo formato de instrução para especificar 4 registradores, chamados R4. Figuras 5.2 e 5.3 mostram o formato R4, que é uma variação do formato R.

Em vez de instruções de desvio de ponto flutuante, RV32F e RV32D fornecem instruções de comparação que definem um registrador inteiro como 1 ou 0 com base na comparação de dois registradores de ponto flutuante: `feq.s, feq.d, flt.s, flt. d, fle.s, fle.d`. Essas instruções permitem que uma instrução de desvio inteiro realize um desvio com base em uma condição de ponto flutuante. Por exemplo, este código desvia para `Exit` se `f1<f2`:

```
flt x5, f1, f2   # x5 = 1 se f1 < f2; caso contrário x5 = 0
bne x5, x0, Exit # se x5 != 0, jump para Exit
```

Para	De			
	32b com sinal inteiro (`w`)	32b sem sinal inteiro (`wu`)	32b flutuante ponteiro (`s`)	64b flutuante ponteiro (`d`)
inteiro com sinal 32b (`w`)	–	–	`fcvt.w.s`	`fcvt.w.d`
inteiro sem sinal de 32b (`wu`)	–	–	`fcvt.wu.s`	`fcvt.wu.d`
ponto flutuante de 32b (`s`)	`fcvt.s.w`	`fcvt.s.wu`	–	`fcvt.s.d`
ponto flutuante de 64b (`d`)	`fcvt.d.w`	`fcvt.d.wu`	`fcvt.d.s`	–

Figura 5.6: Instruções de conversão RV32F e RV32D. As colunas listam os tipos de dados de origem e as linhas mostram o tipo de dados de destino convertidos.

5.4 Conversão e Movimentação de Ponto Flutuante

O RV32F e o RV32D têm instruções que executam todas as combinações de conversões úteis entre inteiros com sinal de 32 bits, inteiros sem sinal de 32 bits, ponto flutuante de 32 bits e ponto flutuante de 64 bits. A Figura 5.6 exibe essas 10 instruções por tipo de dados de origem e tipo de dados de destino convertidos.

O RV32F também oferece instruções para mover dados para registradores `x` de registradores `f` (`fmv.x.w`) e vice-versa (`fmv.w.x`).

5.5 Instruções de Ponto Flutuante Diversas

O RV32F e o RV32D oferecem instruções incomuns que ajudam no uso de bibliotecas matemáticas, além de fornecer pseudoinstruções úteis. (O padrão de ponto flutuante IEEE 754 exige uma maneira de copiar e manipular sinais e classificar dados de ponto flutuante, o que inspirou essas instruções.)

A primeira é a instrução de *sign-injection*, que copia tudo, desde o primeiro registrador de origem, com exceção do bit de sinal. O valor do bit de sinal depende da instrução:

1. Float sign inject (`fsgnj.s, fsgnj.d`): o bit de sinal resultante é o bit de sinal de rs2.
2. Float sign inject negative (`fsgnjn.s, fsgnjn.d`): o bit de sinal resultante é o oposto do bit de sinal de rs2.
3. Float sign inject exclusive-or (`fsgnjx.s, fsgnjx.d`): o bit de sinal resultante é o XOR dos bits de sinal de rs1 e rs2.

abc
Facilidade de Programação

Além de ajudar na manipulação de sinais em bibliotecas matemáticas, as instruções *sign-injection* fornecem três pseudoinstruções populares de ponto flutuante (veja a Figura 3.4 na Página 39):

1. Copia para registradores de ponto flutuante:
   `fmv.s rd,rs` é na verdade `fsgnj.s rd,rs,rs` e
   `fmv.d rd,rs` é `fsgnj.d rd,rs,rs`.
2. Negação:
   `fneg.s rd,rs` se traduz em `fsgnjn.s rd,rs,rs` e
   `fneg.d rd,rs` para `fsgnjn.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];
  }
}
```

Figura 5.7: O programa DAXPY de ponto flutuante em C.

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

Figura 5.8: Número de instruções e tamanho do código de DAXPY para quatro ISAs. Lista o número de instruções por loop e total. Capítulo 7 descreve ARM Thumb-2, microMIPS e RV32C.

3. Valor absoluto (since $0 \oplus 0 = 0$ e $1 \oplus 1 = 0$):
   `fabs.s rd,rs` vira `fsgnjx.s rd,rs,rs` e
   `fabs.d rd,rs` vira `fsgnjx.d rd,rs,rs`.

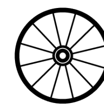
A segunda instrução incomum de ponto flutuante é *classify* (`fclass.s, fclass.d`). Instruções de classificação também são uma ótima ajuda para bibliotecas matemáticas. Essas testam um operando de origem para ver quais das 10 propriedades de ponto flutuante se aplicam (consulte a tabela abaixo) e, em seguida, aplica-se uma *máscara* nos 10 bits inferiores do registrador de número inteiro de destino com a resposta. Apenas um dos dez bits é "setado" como 1, com o restante "setado" como 0s.

x[rd] bit	Significado
0	f[*rs1*] é $-\infty$.
1	f[*rs1*] é um número normal negativo.
2	f[*rs1*] é um número subnormal negativo.
3	f[*rs1*] é -0.
4	f[*rs1*] é $+0$.
5	f[*rs1*] é um número subnormal positivo.
6	f[*rs1*] é um número normal positivo.
7	f[*rs1*] é $+\infty$.
8	f[*rs1*] é a sinalização NaN.
9	f[*rs1*] é um quiet NaN.

O nome DAXPY vem da própria fórmula: Double-precision A times X Plus Y. A versão de precisão simples é chamada de SAXPY.

5.6 Comparando RV32FD, ARM-32, MIPS-32 e x86-32 usando DAXPY

Agora faremos uma comparação frente-a-frente usando o DAXPY como nosso benchmark de ponto flutuante (Figura 5.7). Calcula $Y = a \times X + Y$ em precisão dupla, onde `X` e `Y` são vetores e `a` é um escalar. A Figura 5.8 resume o número de instruções e número de bytes em DAXPY de programas para os quatro ISAs. O código deles está nas Figuras 5.9 a 5.12.


Simplicidade

Desempenho

Como foi o caso do Insertion Sort no Capítulo 2, apesar de sua ênfase na simplicidade, a versão RISC-V novamente tem aproximadamente as mesmas instruções ou menos, e os

tamanhos dos códigos das arquiteturas são bem próximos. Neste exemplo, as linhas de comparação e execução do RISC-V salvam tantas instruções quanto os modos de endereço mais sofisticados e as instruções push e pop do ARM-32 e x86-32.

5.7 Considerações Finais

> *Less is more*
>
> —Robert Browning, 1855. A escola minimalista de arquitetura de sistemas adotou esse poema como um axioma nos anos 80.

O padrão de ponto flutuante IEEE 754-2008 [IEEE Standards Committee 2008] define os tipos de dados de ponto flutuante, a precisão do cálculo e as operações necessárias. Seu sucesso reduz muito a dificuldade de portar programas de ponto flutuante e também significa que os ISAs de ponto flutuante são provavelmente mais uniformes do que seus equivalentes em outros capítulos.

■ ***Elaboração: Aritmética de ponto flutuante de 16 bits, 128 bits e decimal***

O padrão revisado de ponto flutuante IEEE (IEEE 754-2008) descreve vários novos formatos além da precisão simples e dupla, que são chamamados *binary32* e *binary64*. A adição menos surpreendente é a precisão quádrupla, denominada *binary128*. O RISC-V tem uma extensão provisória planejada para ele, chamada RV32Q (consulte o Capítulo 11). O padrão também fornece mais dois tamanhos para o intercâmbio de dados binários, indicando que os programadores podem armazenar esses números na memória ou no armazenamento, mas não esperar poder computar esses tamanhos. Eles são de precisão intermediária (*binary16*) e precisão octupla (*binary256*). Apesar da intenção do padrão, as GPUs calculam com precisão e também as mantêm na memória. O plano para o RISC-V é incluir a semi-precisão nas instruções vetoriais (RV32V no Capítulo 8), com a ressalva de que processadores que suportam a meia-precisão do vetor também irão adicionar instruções escalares de meia precisão. A adição mais surpreendente ao padrão revisado é o ponto flutuante decimal, para o qual o RISC-V separou o RV32L (consulte o Capítulo 11). Os três formatos decimais autoexplicativos são chamados *decimal32*, *decimal64* e *decimal128*.

5.8 Para Saber Mais

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

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

```
# 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           # se n == 0, pula para Saída
  4: 00351513 slli    a0,a0,0x3       # a0 = n*8
  8: 00a60533 add     a0,a2,a0        # a0 = endereço de x[n] (último elemento)
Loop:
  c: 0005b787 fld     fa5,0(a1)       # fa5 = x[]
 10: 00063707 fld     fa4,0(a2)       # fa4 = y[]
 14: 00860613 addi    a2,a2,8         # a2++ (incrementa ponteiro para y)
 18: 00858593 addi    a1,a1,8         # a1++ (incrementa ponteiro para 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         # se i != n, salta para Laço
Saída:
 28: 00008067         ret             # retorno
```

Figura 5.9: Código RV32D para DAXPY da Figura 5.7. O endereço em hexadecimal se encontra à esquerda, o código em linguagem de máquina em representação ao lado e, em seguida, a instrução da linguagem de montagem seguida por um comentário. As instruções de comparação e desvio evitam as duas instruções de comparação no código de ARM-32 e x86-32.

```
# ARM-32 (6 insns in loop; 10 insns/40 bytes total; 28 bytes Thumb-2)
# r0 é n, d0 é a, r1 é ponteiro para x[0], r2 é ponteiro para y[0]
  0: e3500000 cmp      r0, #0             # compara n com 0
  4: 0a000006 beq      24 <daxpy+0x24>    # se n == 0, salta para Saída
  8: e0820180 add      r0, r2, r0, lsl #3 # r0 = endereço de x[n] (último elemento)
Laço:
  c: ecb16b02 vldmia   r1!,{d6}           # d6 = x[i], incrementa o ponteiro para 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 para y
 1c: e1520000 cmp      r2, r0             # i vs. n
 20: 1affffff9 bne     c <daxpy+0xc>      # if i != n, salta para Laço
Saída:
 24: e12fff1e bx       lr                 # retorno
```

Figura 5.10: Código ARM-32 para DAXPY da Figura 5.7. O modo de endereçamento de incremento automático do ARM-32 salva duas instruções em comparação com o RISC-V. Ao contrário do Insertion Sort, não há necessidade de empilhar e remover registradores para o DAXPY no ARM-32.

```
# 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>  # se n == 0, salta para Saída
  4: 000420c0 sll    a0,a0,0x3           # a0 = n*8 (slot de atraso de desvio preenchido)
  8: 00c42021 addu   a0,a2,a0            # a0 = endereço de x[n] (último elemento)
Laço:
  c: 24c60008 addiu  a2,a2,8             # a2++ (incrementa ponteiro para y)
 10: d4a00000 ldc1   $f0,0(a1)           # f0 = x[i]
 14: 24a50008 addiu  a1,a1,8             # a1++ (incrementa ponteiro para 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> # se i != n, pula para Laço
 24: f4c0fff8 sdc1   $f0,-8(a2)          # y[i] = a*x[i] + y[i] (slot de atraso preenchido)
Saída:
 28: 03e00008 jr     ra                  # retorno
 2c: 00000000 nop                        # (slot de atraso de desvio não preenchido)
```

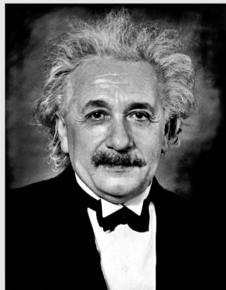
Figura 5.11: Código MIPS-32 para DAXPY da Figura 5.7. Dois dos três slots de atraso de são preenchidos com instruções úteis. A capacidade de verificar a igualdade entre dois registradores evita as duas instruções de comparação encontradas no ARM-32 e no x86-32. Ao contrário dos loads de inteiros, os loads de ponto flutuante não possuem um intervalo de atraso.

```
# x86-32 (6 insns no laço; 16 insns/50 bytes no total)
# eax é i, n está na memória em esp+0x8, a está na memória em esp+0xc
# ponteiro para x[0] está na memória em esp+0x14
# ponteiro para y[0] está na memória em esp+0x18
  0: 53                push        ebx                     # salva ebx
  1: 8b 4c 24 08       mov         ecx,[esp+0x8]           # ecx tem uma cópia de n
  5: c5 fb 10 4c 24 0c vmovsd      xmm1,[esp+0xc]          # xmm1 tem uma cópia de a
  b: 8b 5c 24 14       mov         ebx,[esp+0x14]          # ebx aponta para x[0]
  f: 8b 54 24 18       mov         edx,[esp+0x18]          # edx aponta para y[0]
 13: 85 c9             test        ecx,ecx                 # compara n com 0
 15: 74 19             je          30 <daxpy+0x30>         # se n==0, salta para Saída
 17: 31 c0             xor         eax,eax                 # i = 0 (já que x^x==0)
Laço:
 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                 # compara i com n
 2e: 75 e9             jne         19 <daxpy+0x19>         # se i!=n, pula para Laço
Saída:
 30: 5b                pop         ebx                     # restaura ebx
 31: c3                ret                                 # retorno
```

Figura 5.12: Código x86-32 para DAXPY da Figura 5.7. A falta de registradores x86-32 é evidente neste exemplo, com quatro variáveis alocadas na memória que estão em registradores no código de outras ISAs. Ele também demonstra expressões x86-32 para comparar o valor de um registrador com zero (`test ecx,ecx`) ou para definir um registrador como zero (`xor eax,eax`).

6 RV32A: Instruções Atômicas

Albert Einstein (1879-1955) foi o cientista mais famoso do século XX. Ele inventou a teoria da relatividade e defendeu a construção da bomba atômica para a Segunda Guerra Mundial.



Everything should be made as simple as possible, but no simpler.

—Albert Einstein, 1933

6.1 Introdução

Nossa suposição é que você já entende o suporte da ISA para multiprocessamento, então nosso trabalho é apenas explicar as instruções do RV32A e o que elas fazem. Se você acha que não tem experiência suficiente ou precisa revisar, estude sincronização (ciência da computação) na Wikipedia (https://en.wikipedia.org/wiki/Synchronization_(computer_science)) ou leia a Seção 2.1 do nosso livro relacionado à arquitetura RISC-V [Patterson and Hennessy 2017].

O RV32A possui dois tipos de operações atômicas para sincronização:

- Operações de memória atômica (AMO), e
- load reservado / store condicional.

A Figura 6.1 é uma representação gráfica do conjunto de instruções da extensão RV32A e a Figura 6.2 lista seus opcodes e formatos de instrução.

As instruções AMO (Atomic Memory Operation) executam atomicamente executam uma operação em um operando na memória e "setam" o registrador de destino para o valor original de memória original. Atômico significa que não pode haver interrupção entre a leitura e a escrita em memória, nem outros processadores podem modificar o valor da memória entre a leitura e escrita da memória da instrução AMO.

AMOs e LR/SC requerem endereços de memória naturalmente alinhados porque é dispendioso para o hardware garantir a atomicidade nos limites da linha de cache.

Load reservado e store condicional fornecem uma operação atômica entre duas instruções. O load reservado lê uma palavra da memória, grava-a no registrador de destino e registra uma reserva nessa palavra na memória. O store condicional armazena uma palavra no endereço em um registrador de origem *desde que exista uma reserva de carga nesse endereço de memória*. Ele grava zero no registrador de destino se a operação store tiver êxito, ou em caso contrário um código de erro diferente de zero. Uma pergunta óbvia seria: por que o RV32A tem duas maneiras de executar operações atômicas? A resposta é que existem dois casos de uso bem distintos.

Os desenvolvedores de linguagem de programação consideram que a arquitetura possa executar uma operação de compare-and-swap atômica: compara um valor de registrador a

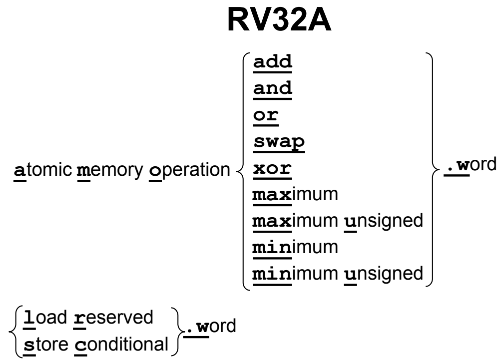
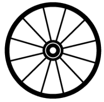


Figura 6.1: Diagrama das instruções do RV32A.

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

Figura 6.2: O mapa de opcode RV32A tem layout de instrução, opcodes, tipo de formato e nomes. (Table 19.2 of [Waterman and Asanović 2017] é a base dessa figura.

um valor na memória endereçado por outro registrador e, se forem iguais, troque um terceiro valor de registrador pelo valor na memória. Eles fazem essa consideração porque é uma primitiva de sincronização universal, ou seja, qualquer outra operação de sincronização de uma única palavra pode ser sintetizada a partir de comparar-e-trocar. [Herlihy 1991].



Simplicidade

Embora esse seja um poderoso argumento para adicionar tal instrução a um ISA, isso requer três registradores de origem em uma instrução. Infelizmente, passar de dois para três operandos de origem complicaria a interface do sistema com a memória, o caminho de dados para números inteiros, o controle do inteiro e o formato da instrução. (Os três operandos de origem das instruções multiply-add do RV32FD afetam o caminho de dado de ponto flutuante, não o caminho de dados para números inteiros.) Felizmente, load reservado e store condicional tem apenas dois registradores de origem e pode implementar a operação atômica de compare-and-swap. (veja a metade superior da Figura 6.3).

Desempenho

A justificativa para também ter instruções AMO é que elas se comportam melhor em grandes sistemas multiprocessadores do que as operações de load reservado e store condicional. Elas também podem ser usadas para implementar operações de redução com eficiência. AMOs são úteis também para comunicação com dispositivos de E/S, porque elas executam uma leitura e uma gravação em uma única transação de átomos de barramento. Essa atomicidade pode simplificar os drivers de dispositivo e melhorar o desempenho de E/S. A metade inferior da Figura 6.3 mostra como escrever uma seção crítica usando swap atômico.

■ ***Elaboração: Modelos de consistência de memória***

O RISC-V possui um modelo de consistência relaxada de memória, portanto, outras threads podem visualizar alguns acessos fora de ordem à memória. A Figura 6.2 mostra que todas as instruções RV32A possuem um *obter bit*(`aq` e um *liberar bit*(`rl`. Uma operação atômica com o aq bit "setado" garante que outras threads enxergarão o AMO em-ordem nos acesso *subsequentes* à memória. Se o bit `rl` estiver "setado", outras threads verão a operação atômica em ordem nos acessos de memória *anteriores*. Para saber mais, [Adve and Gharachorloo 1996] é um excelente tutorial sobre o assunto.

Em que difere? O MIPS-32 original não tinha mecanismo para sincronização, mas os arquitetos adicionaram instruções de load reservado / store condicional a uma especificação posterior à ISA MIPS.

```
# Palavra de memória M[a0] de comparação e troca (CAS) utilizando lr/sc.
# Valor antigo esperado em a1; novo valor desejado em a2.
   0: 100526af    lr.w  a3,(a0)     # Carrega o valor antigo
   4: 06b69e63    bne   a3,a1,80    # Valor antigo igual a 1?Old value equals a1?
   8: 18c526af    sc.w  a3,a2,(a0)  # Troca pelo novo valor se assim for
   c: fe069ae3    bnez  a3,0        # Tente novamente se o store falhou
        ... código que segue um CAS de sucesso CAS vai aqui  ...
  80:                               # CAS malsucedido.


# Seção crítica protegida por spinlock de teste e configuração usando uma AMO.
   0: 00100293    li           t0,1        # Inicializa o valor de bloqueio
   4: 0c55232f    amoswap.w.aq t1,t0,(a0)  # Tentativa de adquirir bloqueio
   8: fe031ee3    bnez         t1,4        # Tenta novamente se não obter êxito
        ... seção crítica vai aqui ...
  20: 0a05202f    amoswap.w.rl x0,x0,(a0)  # Libera o bloqueio.
```

Figura 6.3: Dois exemplos de sincronização. O primeiro usa load reservado/store condicional `lr.w,sc.w` para implementar comparar-e-trocar, e o segundo usa uma troca atômica `amoswap.w` para implementar um mutex.

6.2 Considerações Finais

O RV32A é opcional e um processador RISC-V é mais simples sem esse. No entanto, como Einstein disse, tudo deve ser tão simples *quanto possível*, mas não mais simples. Muitas situações exigem o RV32A.

6.3 Para Saber Mais

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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

7 RV32C: Instruções Compactadas

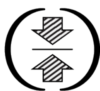
E. F. Schumacher (1911-1977) escreveu este livro de economia que defendia tecnologias de escala humana, descentralizadas e apropriadas. Traduzido em várias línguas, foi nomeado um dos 100 livros mais influentes desde a Segunda Guerra Mundial.

Small is Beautiful.

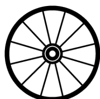
—E. F. Schumacher, 1973

7.1 Introdução

As ISAs anteriores expandiram significativamente o número e formatos de instruções para reduzir o tamanho do código: adicionando instruções curtas com dois operandos em vez de três, pequenos campos imediatos e assim por diante. O ARM e o MIPS inventaram duas vezes ISAs completas de forma a reduzir o código: ARM Thumb e Thumb-2, além de MIPS16 e microMIPS. Essas novas ISAs prejudicaram o processador e o compilador e aumentaram a carga cognitiva para o programador de linguagem assembly. O RV32C adota uma nova abordagem: *cada* instrução curta *deve* mapear para *uma* instrução RISC-V padrão de 32 bits. Além disso, somente o assembler e o linker estão cientes das instruções de 16 bits, e cabe a eles substituir uma instrução completa por sua equivalente. O programador do compilador e o programador de linguagem assembly podem ignorar as instruções do RV32C e seus formatos, exceto pelo fato de terem de acabar com programas menores que a maioria. A Figura 7.1 é uma representação gráfica do conjunto de instruções da extensão RV32C.



Tamanho do Programa



Simplicidade

Os arquitetos RISC-V escolheram as instruções na extensão RVC para obter boas compactação de código em uma variedade de programas, usando três observações para encaixá-los em 16 bits. Primeiro, dez registradores populares (`a0` - `a5`, `s0` - `s1`, `sp` e `ra`) são acessados muito mais que o resto. Segundo, muitas instruções sobrescrevem um de seus operandos de origem. Terceiro, operandos imediatos tendem a ser pequenos, e algumas instruções adequam-se melhor para imediatos. Assim, muitas instruções RV32C podem acessar apenas os registradores populares; algumas instruções sobrescrevem implicitamente um operando de origem; e quase todos valores imediatos são reduzidos em tamanho, com loads e stores usando somente offsets em múltiplos do tamanho do operando.

As Figuras 7.3 and 7.4 mostram o código RV32C para Ordenação por Inserção e DAXPY. Mostramos as instruções do RV32C para demonstrar explicitamente o impacto da compactação, mas normalmente essas instruções são invisíveis no programa em linguagem assembly. Os comentários mostram as instruções equivalentes de 32 bits às instruções RV32C (entre parênteses). O Apêndice A inclui a instrução RISC-V de 32 bits que corresponde a cada instrução RV32C de 16 bits. Por exemplo, no endereço 4 no algoritmo de Ordenação

RV32C

Computação de inteiros

c.add { —, immediate }
c.add immediate * 16 to stack pointer
c.add immediate * 4 to stack pointer nondestructive
c.subtract
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 e stores

c. { —, float } { load, store } word { —, using stack pointer }
c.float { load, store } doubleword { —, using stack pointer }

Transferência de controle

c.branch { equal, not equal } to zero
c.jump { —, and link }
c.jump { —, and link } register

Outras instruções

c.environment break

Figura 7.1: Diagrama das instruções do RV32C. Os campos imediatos das instruções de deslocamento e `c.addi4spn` **são estendidos como zero e para as outras instruções são estendidos como sinal.**

por Inserção na Figura 7.3, o assembler substituiu a seguinte instrução RV32I de 32 bits:

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

Por esta instrução RV32C de 16 bits:

```
c.li a4,1     # (expands to addi a4,x0,1) i = 1
```

A instrução imediata de load RV32C é mais restrita porque deve especificar apenas um registradores e um pequeno valor imediato. O código de máquina `c.li` tem apenas quatro dígitos hexadecimais na Figura 7.3, mostrando que a instrução `c.li` tem, de fato, 2 bytes de comprimento.

Outro exemplo é no endereço 10 na Figura 7.3, onde o assembler substituiu:

```
add a2,x0,a3  # a2 is pointer to a[j]
```

Por esta instrução RV32C de 16 bits:

```
c.mv a2,a3    # (expands to add a2,x0,a3) a2 is pointer to a[j]
```

A instrução de movimento do RV32C tem apenas 16 bits, porque especifica apenas dois registradores.

Embora o projetista do processador não possa ignorar as instruções do RV32C, um truque as torna de baixo custo para implementar: um decodificador converte todas as instruções de 16 bits em suas versões equivalentes de 32 bits *antes* da execução. As Figuras 7.6 para 7.8 listam os formatos de instrução RV32C e opcodes que o decodificador traduz. É equivalente a apenas 400 portas enquanto o menor processador de 32 bits - sem qualquer extensão RISC-V - é de 8000 portas. Se é 5% de um projeto tão pequeno, o decodificador quase desaparece dentro de um processador moderado que com caches é ordem 100.000 portões.

Custo

Em que difere? Não há instruções de byte ou halfword no RV32C porque outras instruções têm uma influência maior no tamanho do código. A pequena vantagem de tamanho

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

Figura 7.2: Instruções e tamanho do código para Ordenação por Inserção e DAXPY para ISAs compactados.

do Thumb-2 sobre o RV32C na Figura 1.5 na página 10 é devido à economia de tamanho de código de Load e Store Múltiplo na entrada e saída de procedimento. O RV32C os exclui para manter o mapeamento um-para-um com as instruções do RV32G, o que os omite para reduzir a complexidade da implementação para processadores high-end. Como o Thumb-2 é um ISA a parte do ARM-32, mas um processador pode alternar entre eles, o hardware deve ter dois decodificadores de instrução: um para o ARM-32 e outro para o Thumb-2. O RV32GC é uma ISA única, portanto, os processadores RISC-V precisam apenas de um único decodificador.

■ ***Elaboração: Por que os arquitetos poderiam não utilizar o RV32C?***

O decodificador de instruções pode ser um gargalo para processadores superescalares que tentam buscar várias instruções por ciclo de relógio. Outro exemplo é o *macrofusion*, pelo qual o decodificador de instruções combina instruções RISC-V para formar instruções mais elaboradas para execução (veja o Capítulo 1). Uma combinação de instruções RV32C de 16 bits e RV32I de 32 bits pode dificultar a conclusão da decodificação sofisticada dentro do ciclo de relógio de uma implementação de alto desempenho.

7.2 Comparando RV32GC, Thumb-2, microMIPS e x86-32

A Figura 7.2 sumariza o tamanho da Ordenação por Inserção e DAXPY para estas quatro ISAs.

Das 19 instruções RV32I originais em Ordenação por Inserção, 12 tornam-se RV32C, então o código reduz de $19 \times 4 = 76$ bytes para $12 \times 2 + 7 \times 4 = 52$ bytes, economizando $24/76 = 32$ %. DAXPY diminui de $11 \times 4 = 44$ bytes para $8 \times 2 + 3 \times 4 = 28$bytes, ou $16/44 = 36$ %.

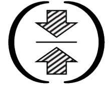
Os resultados para esses dois pequenos exemplos estão alinhados de forma surpreendente com a Figura 1.5 na página 10 do Capítulo 1, que mostra o código RV32G é cerca de 37 % maior que o código RV32GC, para um conjunto maior de programas muito maiores. Para alcançar esse nível de economia, mais da metade das instruções nos programas tinham que ser instruções RV32C.

■ ***Elaboração: O RV32C é realmente único?***

Instruções RV32I são indistinguíveis no RV32IC. O Thumb-2 é, na verdade, um ISA separado incluindo a maioria porém nem todas as instruçóes do ARMv7 com instruções de 16 bits, mas não de todo o ARMv7. Por exemplo, *Compare e Branch on Zero* está no Thumb-2, mas não encontra-se no ARMv7, e vice-versa para *Reverse Subtract with Carry*. Nem o microMIPS32 é um superconjunto do MIPS32. Por exemplo, o microMIPS multiplica por dois os deslocamentos para o desvio, porém o fator é quatro no MIPS32. O RISC-V *sempre* multiplica por dois.

7.3 Considerações finais

> *I would have written a shorter letter, but I did not have the time.*
>
> —Blaise Pascal, 1656.
>
> Blaise Pascal foi um matemático que construiu uma das primeiras calculadoras mecânicas, o que levou o vencedor do Prêmio Turing, Niklaus Wirth, a nomear uma linguagem de programação em sua homenagem.


Tamanho do Programa

O RV32C hoje oferece ao RISC-V um dos menores tamanhos de código. Você pode pensar neles como pseudoinstruções assistidas por hardware. No entanto, neste caso o assembler está escondendo do programador de linguagem assembly e do compilador, em vez de, como no Capítulo 3, expandir o conjunto de instruções reais com operações mais populares que tornam o código RISC-V mais fácil de ler e utilizá-lo. Ambas as abordagens facilitam a produtividade do programador.

Elegância

Consideramos o RV32C como um dos melhores exemplos do RISC-V para um mecanismo simples e poderoso que melhora seu custo-desempenho.

7.4 Para Saber Mais

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

```
# 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 é ponteiro para a[i]
  4: 4705     c.li   a4,1      # (expande para addi a4,x0,1) i = 1
Laço externo:
  6: 00b76363 bltu   a4,a1,c   # se i < n, salta para Continue o ciclo externo
  a: 8082     c.ret            # (expande para jalr x0,ra,0) retorno da função
Continue o laço externo:
  c: 0006a803 lw     a6,0(a3)  # x = a[i]
 10: 8636     c.mv   a2,a3     # (expande para add a2,x0,a3) a2 é ponteiro para a[j]
 12: 87ba     c.mv   a5,a4     # (expande para add a5,x0,a4) j = i
Laço interno:
 14: ffc62883 lw     a7,-4(a2) # a7 = a[j-1]
 18: 01185763 ble    a7,a6,26  # se a[j-1] <= a[i], salta para Saida Laço Interno
 1c: 01162023 sw     a7,0(a2)  # a[j] = a[j-1]
 20: 17fd     c.addi a5,-1     # (expande para addi a5,a5,-1) j--
 22: 1671     c.addi a2,-4     # (expande para addi a2,a2,-4)decr a2 para apontar para a[j]
 24: fbe5     c.bnez a5,14     # (expande para bne a5,x0,14) se j!=0, salta para Laço inter
Saída Laço interno:
 26: 078a     c.slli a5,0x2    # (expande para slli a5,a5,0x2) multiplica a5 por 4
 28: 97aa     c.add  a5,a0     # (expande para add a5,a5,a0)a5 = endereço de byte de a[j]
 2a: 0107a023 sw     a6,0(a5)  # a[j] = x
 2e: 0705     c.addi a4,1      # (expande para addi a4,a4,1) i++
 30: 0691     c.addi a3,4      # (expande para addi a3,a3,4) incr a3 para apontar para a[i]
 32: bfd1     c.j    6         # (expands to jal x0,6) salta para Laço Externo
```

Figura 7.3: Código RV32C para Ordenação por Inserção. As doze instruções de 16 bits tornam o código 32% menor. A largura de cada instrução é evidente pelo número de caracteres hexadecimais na segunda coluna. As instruções RV32C (começando com `c.`) São apresentadas explicitamente neste exemplo, mas normalmente os programadores de linguagem assembly e compiladores não podem vê-los.

```
# RV32DC (11 instruções, 28 bytes)
# a0 é n, a1 é ponteiro para x[0], a2 é ponteiro para y[0], fa0 é a
  0: cd09     c.beqz a0,1a        # (expande para beq a0,x0,1a) se n==0, salta para Saída
  2: 050e     c.slli a0,a0,0x3    # (expande para slli a0,a0,0x3) a0 = n*8
  4: 9532     c.add a0,a2         # (expande para add a0,a0,a2) a0 = endereço de x[n]
Laço:
  6: 2218     c.fld fa4,0(a2)     # (expande para fld fa4,0(a2) ) fa5 = x[]
  8: 219c     c.fld fa5,0(a1)     # (expande para fld fa5,0(a1) ) fa4 = y[]
  a: 0621     c.addi a2,8         # (expande para addi a2,a2,8) a2++ (incr. ptr para y)
  c: 05a1     c.addi a1,8         # (expande para addi a1,a1,8) a1++ (incr. ptr para 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         # se i != n, salta para Laço
Saída:
 1a: 8082     ret                 # (expande para jalr x0,ra,0) retorno da função
```

Figura 7.4: Código RV32DC para DAXPY. As oito instruções de 16 bits reduzem o código em 36%. A largura de cada instrução é evidente pelo número de caracteres hexadecimais na segunda coluna. As instruções RV32C (começando com `c.`) São apresentada s explicitamente neste exemplo, mas normalmente elas são invisíveis ao o programador de linguagem assembly e compiladores.

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

Figura 7.5: O mapa de opcode RV32C (bits$[1:0]$ = 01) lista layout, opcodes, formatos e nomes. `rd'`, `rs1'`, e `rs2'` referem-se aos 10 registradores mais populares `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (A tabela 12.5 of Waterman and Asanović 2017] é a base desta figura.)

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

Figura 7.6: O mapa de opcode RV32C (bits$[1:0]$ = 00) lista layout, opcodes, format e nomes. `rd’`, `rs1’`, e `rs2’` referem-se aos 10 registradores mais populares `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (A tabela 12.4 of Waterman and Asanović 2017] é a base desta figura.)

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

Figura 7.7: O mapa do opcode RV32C (bits$[1:0]$ = 10) lista layout, opcodes, format e nomes. (A tabela 12.6 of Waterman and Asanović 2017] é a base desta figura.)

Formato	Significado	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Registrador	funct4		rd/rs1		rs2		op
CI	Imediato	funct3	imm	rd/rs1		imm		op
CSS	Store relativo a pilha	funct3	imm			rs2		op
CIW	Amplo imediato	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Desvio	funct3	offset		rs1′	offset		op
CJ	Salto	funct3	jump target					op

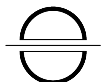
Figura 7.8: Formatos de instrução compactadas RVC de 16 bits. `rd’`, `rs1’`, and `rs2’` referem-se aos 10 registradores mais populares `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (A tabela 12.1 of Waterman and Asanović 2017] é a base desta figura.)

8 RV32V: Vetores

Seymour Cray (1925-1996) foi arquiteto do Cray-1 em 1976, o primeiro supercomputador de sucesso comercial que utilizava uma arquitetura vetorial. O Cray-1 era uma joia; foi o computador mais veloz do mundo, mesmo não utilizando instruções vetoriais.

Desempenho

As extensões multimídia da Intel (MMX) em 1997 tornaram o SIMD popular. Foram adotadas e expandidas por meio das Streaming SIMD Extensions (SSE) em 1999 e Advanced Vector Extensions (AVX) em 2010. A fama da MMX foi alimentada por uma campanha publicitária da Intel mostrando trabalhadores dançando disco em uma linha de semicondutores vestidos em roupas de proteção coloridas (https://www.youtube.com/watch?v=paU16B-bZEA).



Isolamento de Arq. da Impl.

I'm all for simplicity. If it's very complicated I can't understand it.

—Seymour Cray

8.1 Introdução

Este capítulo concentra-se no *paralelismo a nível de dados*, onde há muitos dados em que a aplicação pode calcular simultaneamente, arrays são um exemplo comum. Embora fundamentais para aplicações científicas, os programas de multimídia também utilizam matrizes. A primeira aplicação utiliza dados com ponto flutuante de precisão simples e dupla e esta última utiliza frequentemente dados inteiros de 8 e 16 bits.

A arquitetura mais conhecida para o paralelismo de nível de dados é *Single Instruction Multiple Data* (*SIMD*). O SIMD tornou-se popular ao particionar registradores de 64 bits em muitas partes de 8, 16 ou 32 bits e, em seguida, computá-los em paralelo. O opcode fornecia a largura dos dados e a operação. As transferências de dados são simplesmente loads e stores de um único registrador (wide) SIMD.

O primeiro passo para particionar os registradores de 64 bits existentes é tentador, porque é simples. Para tornar o SIMD mais rápido, os arquitetos subsequentemente ampliavam os registradores para computar mais partições simultaneamente. Como as ISAs SIMD pertencem à escola incremental de projeto, e o opcode especifica a largura dos dados, a expansão dos registradores SIMD também expande o conjunto de instruções SIMD. Cada etapa subsequente de duplicação na largura dos registradores SIMD e do número de instruções SIMD levou as ISAs para o caminho crescente da complexidade, que é sustentada pelos projetistas de processadores, desenvolvedores de compiladores e programadores de linguagem assembly.

Uma alternativa mais antiga e, em nossa opinião, mais elegante para explorar o paralelismo de nível de dados é a arquitetura vetorial. Este capítulo fornece nossa justificativa para utilizar vetores em vez de SIMD no RISC-V.

Computadores vetoriais buscam objetos da memória principal e os colocam em registradores vetoriais longos e sequenciais. Unidades de execução em pipeline realizam computações de maneira muito eficiente nesses registradores. Arquiteturas vetoriais dispersam os resultados dos registradores para a memória principal. O tamanho do registrador vetorial é determinado pela implementação, em vez de ser colocado no opcode, como acontece no SIMD. Como veremos, separar o comprimento do vetor e o número máximo de operações

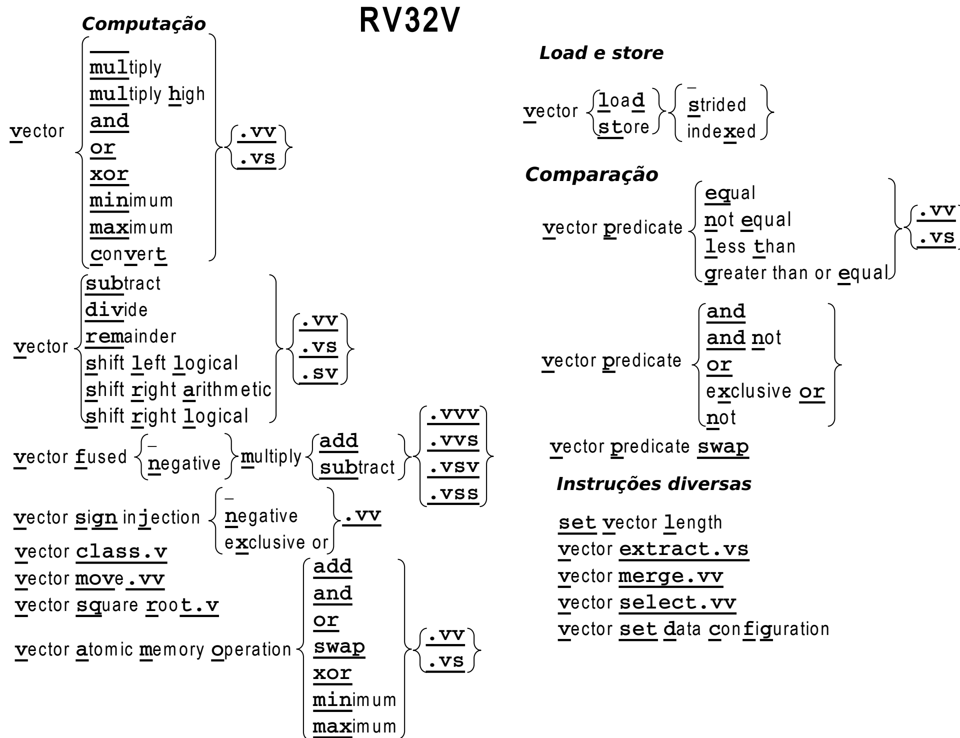


Figura 8.1: Diagrama das instruções do RV32V. Por conta da tipagem dinâmica de registradores, este diagrama de instruções também funciona sem alteração para RV64V no Capítulo 9.

por ciclo de clock da codificação da instrução é o ponto crucial da arquitetura vetorial: o microarquiteto pode projetar com flexibilidade o hardware paralelo de dados sem afetar o programador e o programador pode tirar proveito de vetores mais longos sem reescrever o código. Além disso, arquiteturas vetoriais têm muito menos instruções do que arquiteturas SIMD. Arquiteturas vetoriais também possuem uma tecnologia de compilador bem estabelecida, ao contrário do SIMD.

Arquiteturas vetoriais são mais raras do que as arquiteturas SIMD, portanto, poucos leitores conhecem as ISAs vetoriais. Assim, este capítulo terá um teor mais tutorial do que os anteriores. Se você quiser aprofundar-se sobre arquiteturas vetoriais, leia o Capítulo 4 e o Apêndice G de [Hennessy and Patterson 2011]. O RV32V também possui novos recursos que simplificam a ISA, o que requer mais explicações, mesmo se você já estiver familiarizado com arquiteturas vetoriais.

Facilidade de Programação

Simplicidade

8.2 Instruções de Computação Vetorial

A Figura 8.1 é uma representação gráfica do conjunto de instruções de extensão RV32V. A codificação RV32V não foi ainda finalizada, portanto, esta edição não inclui o diagrama usual do layout de instruções.

Praticamente todas as instruções de computação de inteiros e de ponto flutuante de capítulos anteriores têm uma versão vetorial: Figura 8.1 herda operações de RV32I, RV32M,

RV32F, RV32D e RV32A. Existem vários tipos de instruções vetoriais dependendo se os operandos origem são todos vetores (sufixo `.vv`) ou um operando origem é vetorial e um operando de origem escalar (sufixo `.vs`). Um sufixo escalar significa que um registrador `x` ou `f` é um operando junto com um registrador vetorial (`v`). Por exemplo, nosso programa DAXPY (Figura 5.7 na Página 59 no Capítulo 5) calcula $Y = a \times X + Y$, onde X e Y são vetores e a é um escalar. Para operações vetoriais escalares, o campo rs1 especifica o registrador escalar a ser acessado.

Operações assimétricas como subtração e divisão oferecem uma terceira variação de instruções vetoriais onde o primeiro operando é escalar e o segundo é um vetor (sufixo `.sv`). Operações como $Y = a - X$ as utilizam. Pelo fato de serem meras incrementações supérfluas para operações simétricas como adição e multiplicação, essas instruções não têm uma versão `.sv`. As instruções agrupadas como multiplicaçã-adição possuem três operandos, por isso elas têm a maior combinação de opções vetoriais e escalares: `.vvv`, `.vvs`, `.vsv` e `.vss`.

Os leitores podem perceber que a Figura 8.1 ignora o tipo de dados e a largura das operações vetoriais. A próxima seção explicará o porquê.

8.3 Registradores Vetoriais e Tipagem Dinâmica

O RV32V adiciona 32 registradores vetoriais, cujos nomes iniciam com a letra `v`, porém o número de *elementos* por registrador vetorial varia. Esse número depende da largura das operações e da quantidade de memória dedicada aos registradores vetoriais, que cabe ao projetista do processador. Por exemplo, se o processador alocou 4096 bytes para registradores vetoriais, isso é suficiente para que um dos 32 registradores vetoriais tenha 16 elementos de 64 bits, 32 elementos de 32 bits, 64 elementos de 16 bits ou 128 elementos de 8 bits.

Para manter o número de elementos flexíveis em uma ISA vetorial, um processador vetorial calcula o *maximum vector length* (`mvl`) que é utilizado pelos programas para executarem corretamente em processadores com quantidades diferentes de memória para registradores vetoriais. O registrador de comprimento de vetor (`vl`) define o número de elementos em um vetor para uma operação específica, o que ajuda programas quando uma dimensão de uma matriz não é um múltiplo de `mvl`. Demonstraremos `mvl`, `vl` e os oito registradores predicados (`vp` *i*) em mais detalhes nas seções seguintes.

O RV32V utiliza a abordagem inovadora de associar o tipo e o comprimento de dados aos *registradores vetoriais* em vez de associar aos opcodes de instrução. Um programa marca o vetor registra com seu tipo e largura de dados antes de executar as instruções de computação vetorial. Em português, tipagem dinâmica de registrador (*Dynamic register typing*) reduz o número de instruções vetoriais, um ato importante porque, de forma geral, há seis inteiros e três versões de ponto flutuante para cada instrução vetorial como mostra a Figura 8.1. Como veremos na Seção 8.9 quando confrontamos as numerosas instruções SIMD, uma arquitetura vetorial tipada dinamicamente reduz a carga cognitiva sobre o programador de linguagem assembly e diminui a dificuldade do gerador de código do compilador.

abc
Facilidade de Programação

Outra vantagem da tipagem dinâmica é que os programas podem desabilitar registradores vetoriais não utilizados. Esse recurso aloca toda a memória de vetor para os registradores vetoriais ativos. Por exemplo, suponha que apenas dois registradores vetoriais estejam habilitados, eles sejam do tipo floats de 64 bits e o processador tenha 1024 bytes de memória para registradores vetoriais. O processador iria dividir a memória pela metade, dando a cada registrador vetorial 512 bytes ou $512/8 = 64$ elementos e, portanto, configurando `mvl` para 64. Assim, `mvl` é dinâmico, mas seu valor é definido pelo processador e não pode ser alterado

Type	Ponto flutuante		Inteiro com sinal		Inteiro sem sinal	
Largura	Nome	`vetype`	Nome	`vetype`	Nome	`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

Figura 8.2: Codificações RV32V de tipos de registradores vetoriais. Os três bits mais à direita do campo mostram a largura dos dados e os dois bits mais à esquerda fornecem seu tipo. X64 e U64 estão disponíveis apenas para RV64V. F16 e F32 requerem a extensão RV32F e o F64 requer RV32F e RV32D. F16 é o formato de ponto flutuante de 16 bits IEEE 754-2008 (binary16). A configuração `vetype` para 00000 desativa os registradores vetoriais. (Tabela 17.4 de [Waterman and Asanović 2017] é a base desta figura.)

diretamente pelo software.

Os registradores de origem e destino determinam o tipo e tamanho da operação e o resultado, portanto, as conversões estão implícitas na tipagem dinâmica. Por exemplo, um processador pode multiplicar um vetor de números de ponto flutuante de precisão dupla por um escalar de precisão simples sem primeiro ter que converter os operandos na mesma precisão. Este benefício adicional reduz o número total de instruções vetoriais e o número de instruções executadas.

A instrução `vsetdcfg` define os tipos de registradores vetoriais. A Figura 8.2 mostra os tipos de registradores vetoriais disponíveis para RV32V e para RV64V (veja o Capítulo 9). O RV32V requer que as operações de vetor de ponto flutuante possuam também as versões escalares. Assim, você deve ter pelo menos RV32FV para utilizar o tipo F32 e RV32FDV para utilizar o tipo F64. O RV32V introduz um formato de ponto flutuante de 16 bits tipo F16. Se uma implementação tiver suporte a RV32V e RV32F, ela deverá suportar os formatos F16 e F32.

■ ***Elaboração: O RV32V pode mudar de contexto rapidamente.***

Uma razão pela qual as arquiteturas vetoriais eram menos populares do que as arquiteturas SIMD era a preocupação de que a inclusão de grandes registradores vetoriais aumentaria o tempo para salvar e restaurar um programa na ocorrência de uma interrupção, evento chamado de *troca de contexto*. A tipagem dinâmica de registradores ajuda nesse sentido. O programador deve informar ao processador quais registradores vetoriais estão sendo utilizados, o que significa que o processador precisa armazenar e restaurar apenas os registradores em uma troca de contexto. A convenção RV32V é desabilitar *todos* registradores vetoriais quando as instruções vetoriais não estão sendo utilizadas, o que significa que um processador pode ter o benefício de desempenho dos registradores vetoriais, mas pagar o preço da mudança de contexto extra vale somente se uma interrupção ocorrer enquanto as instruções vetoriais estão sendo executadas. Arquiteturas vetoriais projetadas anteriormente tiveram que arcar com o pior custo da mudança de contexto para salvar e restaurar todos os registradores vetoriais sempre que uma interrupção ocorria.

Preocupação com lenta troca de contexto levou a Intel a evitar a inclusão de registradores na extensão original do MMX SIMD. Ele simplesmente reutilizou os registradores de ponto flutuante existentes, o que significava que nenhum contexto extra era trocado, porém um programa não conseguia misturar instruções de ponto flutuante e multimídia.

8.4 Operações Vetoriais Load e Store

O caso mais fácil para operações vetoriais load e store vetores é lidar com matrizes de dimensão única que são armazenados sequencialmente na memória. O load preenche um registrador vetorial com dados de endereços sequenciais na memória, começando com o endereço na in-

strução `vld`. O tipo de dados associado ao registrador vetorial determina o tamanho dos elementos de dados, e o registrador de comprimento de vetor `vl` define o número de elementos a serem carregados. O store de vetores `vst` faz a operação inversa de `vld`. Por exemplo, se `a0` tiver 1024 e o tipo de `v0` for X32, então `vld v0, 0(a0)` gerará os endereços 1024, 1028, 1032, 1036, .. até atingir o limite definido por `vl`.

Cada load e store tem um imediato de offset sem sinal de 7 bits que é dimensionado pelo tipo de elemento no registrador de destino para loads e o registrador de origem para stores.

Para matrizes (vetores multidimensionais), alguns acessos não serão sequenciais. Se os elementos são armazenados na ordem principal da linha, os acessos sequenciais à coluna em uma matriz bidimensional desejam elementos de dados separados pelo tamanho da linha. As arquiteturas vetoriais suportam esses acessos com transferências de dados espaçados (*strided*): `vlds` e `vsts`. Enquanto poderia-se ter o mesmo efeito de `vld` e `vst` definindo o passo "stride" para o tamanho do elemento em `vlds` e `vsts`, `vld` e `vst` garante que todos os acessos serão sequenciais, o que facilita a entrega de alta largura de banda de memória. Outra razão é que fornecer `vld` e `vst` reduz o tamanho do código e as instruções executadas para o caso comum de "stride" de uma unidade. Essas instruções especificam dois registradores de origem, com um deles fornecendo o endereço inicial e outro especificando o "stride" em bytes.

Facilidade de Programação

Por exemplo, suponha que o endereço inicial em `a0` fosse o 1024 e o tamanho de uma linha em `a1` tenha 64 bytes. `vlds v0,a0,a1` enviaria esta sequência de endereços para a memória: 1024, 1088 ($1024 + 1 \times 64$), 1152 ($1024 + 2 \times 64$), 1216 ($1024 + 3 \times 64$), e assim por diante até que o registrador de comprimento de vetor `vl` indique a parada. Os dados retornados são gravados em elementos sequenciais do registrador vetorial de destino.

Facilidade de Programação

Até agora, consideramos que o programa está trabalhando com matrizes densas. Para suportar matrizes esparsas, as arquiteturas vetoriais oferecem as transferências de dados *indexados*: `vldx` e `vstx`. Um registrador de origem para essas instruções refere-se a um registrador vetorial e o outro a um registrador escalar. O registrador escalar tem o endereço inicial do array esparso, e cada elemento do registrador vetorial contém o índice em bytes dos elementos diferentes de zero do array esparso.

Suponha que o endereço inicial em `a0` fosse o endereço 1024 e o registrador de vetor `v1` tivesse esses índices de byte nos primeiros 4 elementos: 16, 48, 80, 160. `vldx v0,a0,v1` enviaria essa sequência de endereços para a memória: 1040 (1024 + 16), 1072 (1024 + 48), 1104 (1024 + 80), 1184 (1024 + 160). Carregado então os dados retornados em elementos sequenciais do registrador vetorial de destino.

O load indexado também é chamado de *gather* e stores indexados de *scatter*.

Utilizamos matrizes esparsas como nossa motivação para loads e stores indexados, mas há muitos outros algoritmos que acessam dados indiretamente por meio de tabelas de índices.

8.5 Paralelismo Durante a Execução Vetorial

Desempenho

Enquanto um processador vetorial simples pode operar em um elemento vetorial por vez, as operações sobre elemento são independentes por definição e, portanto, um processador poderia, teoricamente, computar todas elas simultaneamente. Os dados mais amplos do RV32G são 64 bits, e os processadores vetoriais atuais normalmente executam dois, quatro ou oito elementos de 64 bits por ciclo de clock. O hardware lida com os casos de sobreposição quando o comprimento do vetor não é um múltiplo da quantidade de elementos executados por ciclo de clock.

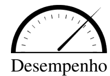
Como no SIMD, o número de operações sobre dados menores é a relação entre as larguras dos dados mais estreitos com os dados mais amplos. Assim, um processador vetorial que calcula 4 operações de 64 bits por ciclo de clock normalmente iniciaria 8 operações de 32

bits, 16 de 16 bits e 32 de 8 bits por ciclo de clock. No SIMD, a arquitetura da ISA determina o número máximo de operações paralelas sobre dados por ciclo de clock e o número de elementos por registrador. Em contraste, o projetista do processador RV32V utiliza os dois parâmetros sem ter que alterar o ISA ou o compilador, enquanto cada duplicação da largura do registrador SIMD dobra o número de instruções SIMD e requer mudanças nos compiladores SIMD. Esta flexibilidade oculta significa que o mesmo programa RV32V roda sem mudanças nos processadores vetoriais mais simples e agressivos.

Facilidade de Programação

8.6 Execução Condicional de Operações Vetoriais

Algumas computações vetoriais incluem declarações `if`. Em vez de depender de desvios condicionais, as arquiteturas vetoriais incluem uma máscara que suprime as operações em alguns elementos de uma operação vetorial. As instruções de predicado na Figura 8.1 realizam testes condicionais entre dois vetores ou um vetor e escalar e escrevem em cada elemento da máscara vetorial a 1 se a condição for válida ou um 0 caso contrário. (A máscara de vetor deve ter o mesmo número de elementos que o vetor registra.) Qualquer instrução de vetor subsequente pode então utilizar essa máscara, um "1" em um bit *i* significa que o elemento *i* é alterado por operações de vetor, e um "0" significa que o elemento *i* permanece inalterado.



O RV32V fornece 8 *vector predicate registers* (registradores de predicados vetoriais—`vp` *i*) para atuar como máscaras vetoriais. As instruções `vpand`, `vpandn`, `vpor`, `vpxor` e `vpnot` executam instruções lógicas para combiná-las para permitir o processamento eficiente de instruções condicionais aninhadas.

Um programa é chamado *vetorizável* se a maioria das operações for executada por instruções vetoriais. As instruções de gather, scatter e predicate aumentam o número de programas vetorizáveis.

As instruções RV32V especificam `vp0` ou `vp1` para ser a máscara que controla uma operação de vetor. Para executar uma operação normal em todos os elementos, um desses dois registradores de predicado deve ser definido para todos os outros. Para trocar rapidamente um dos outros seis registradores predicados para `vp0` ou `vp1`, o RV32V possui a instrução `vpswap`. Os registradores de predicado também são ativados dinamicamente, e desativá-los "reseta-os" em sua totalidade rapidamente.

Por exemplo, suponha que todos os elementos de numeração par do vetor registrador `v3` fossem inteiros negativos e todos os elementos ímpares fossem inteiros positivos. O resultado deste código:

```
vplt.vs     vp0,v3,x0  # seta bits da máscara quando elemento de v3 < 0
add.vv,vp0  v0,v1,v2   # muda elementos de v0 para v1+v2 quando verdadeiro
```

"setaria" todos os bits pares de `vp0` para 1, todos os bits ímpares para 0 e substituiria todos os elementos pares de `v0` pela soma dos elementos correspondentes de `v1` e `v2`. Os elementos ímpares de `v0` não seriam alterados.

8.7 Instruções de Vetores Diversas

Complementando a instrução que configura os tipos de dados de registradores vetoriais mencionados acima (`vsetdcfg`), `setvl` define o registrador de comprimento de vetor (`vl`) e o registrador de destino com o menor do operando de origem e o comprimento máximo do vetor (`mvl`). A razão para escolher o mínimo é decidir em loops se o código do vetor pode ser executado no comprimento máximo do vetor (`mvl`) ou ele deve ser executado em um valor

menor para cobrir os elementos restantes. Assim, para manipular elementos da cauda do vetor, `setvl` é executado em cada iteração de loop.

O RV32V também possui três instruções que manipulam elementos dentro de um registrador vetorial.

Vector Select (`vselect`) produz um novo vetor resultado reunindo elementos de um vetor origem de dados nas localizações de elementos especificadas pelo segundo vetor origem de índice:

```
# vindices contém valores de 0..mvl-1 que selecionam elementos de vsrc
vselect vdest, vsrc, vindices
```

Portanto, se os primeiros quatro elementos de `v2` contiverem 8, 0, 4, 2, então `vselect v0,v1,v2` substituirá o elemento zero de `v0` pelo oitavo elemento de `v1`, o primeiro elemento de `v0` com o elemen to zero de `v1`, o segundo elemento de `v0` com o quarto elemento de `v1` e o terceiro elemento de `v0` com o segundo elemento de `v1`.

Vector Merge (`vmerge`) assemelha-se a seleção de vetor, mas utiliza um registrador de predicado vetorial para escolher qual das fontes utilizar. Ele produz um novo vetor resultado reunindo elementos de um dos dois registradores de origem, dependendo do valor de predicado. O novo elemento vem de vsrc1 se o elemento do registrador predicado vetorial for 0 ou de vsrc2 se for 1:

```
# vp0 bit i determina se o novo elemento i para vdest
# vem de vsrc1 (se bit i == 0) ou vsrc2 (se bit i == 1)
vmerge,vp0 vdest, vsrc1, vsrc2
```

Assim, se os primeiros quatro elementos de `vp0` contiverem 1, 0, 0, 1, os primeiros quatro elementos de `v1` contêm 1, 2, 3, 4 e os quatro primeiros elementos de `v2` contém 10, 20, 30, 40, então `vmerge,vp0 v0,v1,v2` fará com que os primeiros quatro elementos de `v0` sejam 10, 2, 3, 40.

A instrução *Vector Extract* utiliza elementos a partir do meio de um vetor e coloca-os no início de um segundo registrador vetorial:

```
# start é um registrador escalar contendo o offset de vsrc
vextract vdest, vsrc, start
```

Por exemplo, se o tamanho do vetor `vl` for 64 e `a0` conter o valor 32, então `vextract v0,v1,a0` copiará os últimos 32 elementos de `v1` nos primeiros 32 elementos de `v0`.

A instrução `vextract` auxilia reduções seguindo uma abordagem de divisão recursiva para qualquer operador associativo binário. Por exemplo, para somar todos os elementos de um registrador vetorial, use *vector extract* para copiar a última metade de um vetor na primeira metade de outro registrador vetorial e reduzia pela metade o comprimento do vetor. Em seguida, adicione esses dois registradores vetoriais e repita a recursividade com redução pela metade com sua soma até que o comprimento do vetor seja igual a 1. O resultado no elemento zero será a soma de todos os elementos originais no registrador vetorial.

O V em RISC-V também vem de vetor. Os arquitetos do RISC-V tinham ampla experiência positiva com arquiteturas vetoriais, e estavam frustrados com o fato de o SIMD dominar os microprocessadores. Portanto, o V é para o quinto projeto RISC de Berkeley *e* porque sua ISA destacaria vetores.

8.8 Exemplo de Vetores: DAXPY em RV32V

A Figura 8.3 mostra a linguagem assembly RV32V para DAXPY (Figura 5.7 na Página 59 no Capítulo 5), a qual iremos explicar um passo por vez.

```
# a0 é n, a1 é ponteiro para x[0], a2 é ponteiro para y[0], fa0 é a
  0:  li t0, 2<<25
  4:  vsetdcfg t0            # habilita 2 registradores de ponto flutuante de 64b
laço:
  8:  setvl  t0, a0          # vl = t0 = min(mvl, n)
  c:  vld    v0, a1          # carrega o vetor x
 10:  slli   t1, t0, 3       # t1 = vl * 8 (em bytes)
 14:  vld    v1, a2          # carrega o vetor y
 18:  add    a1, a1, t1      # incrementa o ponteiro C para x por 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          # guarda Y
 28:  add    a2, a2, t1      # incrementa o ponteiro C para y por vl*8
 2c:  bnez   a0, loop        # repete se n != 0
 30:  ret                    # retorno
```

Figura 8.3: Código RV32V para DAXPY na Figura 5.7. A linguagem da máquina está ausente porque os opcodes do RV32V ainda precisam ser definidos.

O RV32V DAXPY inicia ativando os registradores vetoriais necessários para esta função. O programa requer apenas dois registradores vetoriais para conter partes de x e y, que são números de ponto flutuante de precisão dupla com 8 bytes de largura cada. A primeira instrução cria uma constante e a segunda escreve no registrador de status de controle que configura registradores vetoriais (vcfgd) para obter dois registradores do tipo F64 (veja a Figura 8.2). Por definição, o hardware aloca os registradores configurados em ordem numérica, resultando em v0 e v1.

Supomos que nosso processador RV32V tenha 1024 bytes de memória dedicados a registradores vetoriais. O hardware aloca a memória uniformemente entre os dois registradores vetoriais, que contêm números de ponto flutuante de precisão dupla (8 bytes). Cada registrador vetorial tem $512/8 = 64$ elementos, então o processador "seta" o comprimento máximo do vetor (mvl) em 64 para esta função.

A primeira instrução no loop define o comprimento do vetor para as próximas instruções vetoriais. A instrução setvl escreve o menor valor entre mvl e n em vl e t0. O "insight" é que, se o número de iterações do loop for maior que n, o código não poderá processar mais que 64 valores por vez, portanto, "seta" vl como mvl. Se n for menor que mvl, então não podemos ler ou escrever além do fim de x e y, portanto devemos computar apenas nos últimos n elementos nesta iteração final do loop. setvl também é copiado para t0 para ajudar no computação posterior do loop na localização 10.

Arquiteturas vetoriais sem setvl tem um código extra de *strip-mining* para definir vl para os últimos n elementos do loop e para verificar se n é inicialmente zero.

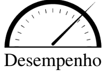
A instrução vld no endereço c é um load vetorial do endereço de x no registrador escalar a1. A operação transfere vl elementos de x da memória para v0. A seguinte instrução de deslocamento slli multiplica o comprimento do vetor pela largura dos dados em bytes(8) para uso posterior no incremento de ponteiros para x e y.

A instrução no endereço 14 (vld) carrega elementos vl de y da memória em v1 e a próxima instrução (add) incrementa o ponteiro para x.

A instrução no endereço 1c é o jackpot. vfmadd multiplica vl elementos de x (v0) pelo escalar a (f0) e adiciona cada produto a vl elementos de y (v1) e armazena as somas vl de volta em y (v1).

Tudo o que resta é armazenar os resultados na memória e alguma sobrecarga de loop. A

instrução no endereço 20 (sub) decrementa n (a0) por vl para registrar o número de operações concluídas nesta iteração do loop. A seguinte instrução (vst) armazena os resultados vl em y na memória. A instrução no endereço 28 (add) incrementa o ponteiro para y e a seguinte instrução repete o loop se n (a0) não for zero. Se n for zero, a instrução final ret retornará ao ponto de chamada.



O poder da arquitetura vetorial é que cada iteração desse loop de 10 instruções executa $3 \times 64 = 192$ acessos de memória e $2 \times 64 = 128$ multiplicações e adições de ponto flutuante (supondo que n seja pelo menos 64). Que calcula a média de 19 acessos de memória e 13 operações por instrução. Como veremos na próxima seção, essas proporções para o SIMD são muito piores.

8.9 Comparando RV32V, MIPS-32 MSA SIMD e x86-32 AVX SIMD

ARM-32 tem uma extensão SIMD chamada NEON mas não têm suporte para instruções de ponto flutuante de precisão dupla, por isso não facilita muito para o DAXPY.

Tal código de book keeping é considerado parte do `strip mining` em arquiteturas de vetores. Como a legenda da Figura 8.5 explica, o registrador de comprimento de vetor `vl` renderiza esse código de book keeping SIMD discutível para RV32V. Arquiteturas vetoriais tradicionais precisam de código extra para lidar com o caso de canto de n = 0. RV32V apenas faz instruções vetoriais agirem como “nops” quando n = 0.

Agora veremos como o SIMD e o RV32V executam o DAXPY. Se você prestar atenção, poderá ver o SIMD como uma arquitetura vetorial restrita com registradores vetoriais curtos—oito “elementos” de 8 bits—mas sem nenhum registrador de comprimento de vetor e nenhuma transferência de dados por etapas ou indexadas.

MIPS SIMD. A Figura 8.5 na Página 88 mostra a versão MIPS SIMD Architecture (MSA) do DAXPY. Cada instrução MSA SIMD pode operar em dois números de ponto flutuante, pois os registradores MSA têm 128 bits de largura.

Ao contrário do RV32V, como não há um registrador de comprimento vetorial, o MSA exige instruções adicionais de book keeping para verificar os valores do problema de n. Quando n é ímpar, existe um código extra para calcular um único multiplicador de ponto flutuante, já que o MSA deve operar em pares de operandos. Esse código é encontrado nos locais 3c a 4c na Figura 8.5. No caso improvável, mas possível, quando n é zero, o desvio na localização 10 ignorará o loop de computação principal.

Se não desviar em torno do loop, a instrução no local 18 (splati.d) coloca cópias de a nas duas metades do registrador SIMD w2. Para adicionar dados escalares no SIMD, precisamos replicá-lo para serem tão amplos quanto a largura do registrador SIMD.

Dentro do loop, a instrução ld.d no local 1c carrega dois elementos de y no registrador SIMD w0 e, em seguida, incrementa o ponteiro para y. Em seguida, ele faz o carregamento de dois elementos de x no registrador SIMD w1. A instrução a seguir no local 28 incrementa o ponteiro para x. A instrução de multiplicação-adição no local 2c é a próxima.

O desvio (atrasado) no final do loop testa para ver se o ponteiro para y foi incrementado além do último elemento par de y. Se não, o loop se repete. O armazenamento SIMD no slot de atraso no endereço 34 grava o resultado em dois elementos de y.

Depois que o loop principal termina, o código verifica se n é ímpar. Em caso afirmativo, ele executa o último acréscimo de multiplicação utilizando instruções escalares do Capítulo 5. A instrução final retorna ao ponto de chamada.

O loop com 7 instruções do código MIPS DAXPY do MSA faz 6 acessos à memória de precisão dupla e 4 multiplicações e adições de ponto flutuante. A média é de cerca de 1 acesso à memória e 0,5 operações por instrução.

x86 SIMD. A Intel passou por muitas gerações de extensões SIMD, que podemos observar no código da Figura 8.6 na Página 89. A expansão SSE para o SIMD de 128 bits levou aos registradores xmm e instruções que podem usá-los, e a expansão para o SIMD de 256 bits como parte do AVX criou os registradores ymm e suas instruções.

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instruções (estáticas)	22	29	13
Bytes (estática)	88	92	52
Instruções por Laço principal	7	6	10
Resultados por Laço principal	2	4	64
Instruções (dinâmico, n=1000)	3511	1517	163

Figura 8.4: Número de instruções e tamanho do código de DAXPY para ISAs vetoriais. Lista o número total de instruções (estáticas), o tamanho do código, o número de instruções e os resultados por loop e o número de instruções executadas (`n` = 1000). O microMIPS com MSA reduz o tamanho do código para 64 bytes e o RV32FDCV reduz para 40 bytes.

O primeiro grupo de instruções nos endereços 0 a 25 carrega as variáveis da memória, faz quatro cópias de `a` em um registrador `ymm` de 256 bits e testa para garantir que `n` seja pelo menos 4 antes de entrar no loop principal. Ele utiliza duas instruções SSE e uma instrução AVX. (A legenda da Figura 8.6 explica em mais detalhes.)

O loop principal é o coração do cálculo DAXPY. A instrução AVX `vmovapd` no endereço 27 carrega 4 elementos de `x` em `ymm0`. A instrução AVX `vfmadd213pd` no endereço 2c multiplica 4 cópias de `a` (`ymm2`) vezes 4 elementos de `x` (`ymm0`), adiciona 4 elementos de `y` (na memória no endereço ecx + edx * 8) e coloca as 4 somas em `ymm0`. A seguinte instrução AVX no endereço 32, `vmovapd`, armazena os 4 resultados em `y`. As próximas três instruções incrementam os contadores e repetem o loop, se necessário.

Como foi o caso do MIPS MSA, o "código de borda" entre os endereços 3e e 57 lida com os casos em que `n` não é um múltiplo de 4. Ele se baseia em três instruções SSE.

As 6 instruções do loop principal no código x86-32 AVX2 DAXPY fazem 12 acessos de memória de precisão dupla e 8 multiplicações e adições de ponto flutuante. Com uma média de 2 acessos à memória e cerca de 1 operação por instrução.

■ ***Elaboração: O Illiac IV foi o primeiro a mostrar a dificuldade de compilar para o SIMD.***

Com 64 unidades paralelas de ponto flutuante de 64 bits (FPUs), o Illiac IV foi planejado para ter mais de 1 milhão de portas lógicas antes de Moore publicar sua lei. Seu arquiteto originalmente previu 1.000 milhões de operações de ponto flutuante por segundo (MFLOPS), porém o desempenho real foi de 15 MFLOPS na melhor das hipóteses. Os custos aumentaram de $8 milhões em 1966 para $31 milhões em 1972, apesar da construção de apenas 64 dos 256 FPUs planejados. O projeto iniciou em 1965, mas levou até 1976 para executar sua primeira aplicação real, o ano em que o Cray-1 foi anunciado. Talvez o mais infame supercomputador, o ILLIAC IV entrou para a lista dos 10 maiores desastres de engenharia [Falk 1976].

8.10 Considerações Finais

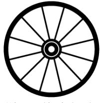
If the code is vectorizable, the best architecture is vector.

—Jim Smith, keynote speech, International Symposium on Computer Architecture, 1994

A Figura 8.4 resume o número de instruções e o número de bytes em DAXPY de programas para RV32IFDV, MIPS-32 MSA e x86-32 AVX2. O código de computação SIMD é diminuído pelo código de book keeping. Dois terços a três quartos do código para MIPS-32 MSA

e x86-32 AVX2 é a sobrecarga do SIMD, seja para preparar os dados para o loop SIMD principal ou para manipular os elementos marginais quando `n` não é um múltiplo do número de números de ponto flutuante em um registrador SIMD.

O código RV32V na Figura 8.3 não precisa desse código de book keeping, que diminui pela metade o o número de instruções. Ao contrário do SIMD, o RV32V possui um registrador de comprimento vetorial, o que faz com que as instruções vetoriais funcionem com qualquer valor de `n`. Você pode pensar que o RV32V teria um problema quando `n` é 0. Isso não acontece porque as instruções vetoriais RV32V deixam tudo inalterado quando `vl` $= 0$.


Simplicidade

Desempenho

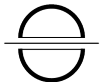
No entanto, a diferença mais significativa entre SIMD e processamento vetorial não é o tamanho do código estático. O SIMD executa de 10 a 20 vezes mais instruções do que o RV32V, porque cada loop do SIMD opera em 2 ou 4 elementos, ao invés de 64 no caso do vetor. As buscas extras de instruções e decodificações de instrução significam um maior uso de energia para executar a mesma tarefa.

Comparando os resultados na Figura 8.4 com as versões escalares de DAXPY na Figura 5.8 na Página 32 no Capítulo 5, vemos que o SIMD praticamente dobra o tamanho do código em instruções e bytes, mas o loop principal é do mesmo tamanho. A redução no número dinâmico de instruções executadas é um fator de 2 ou 4, dependendo da largura dos registradores SIMD. No entanto, o tamanho do código vetorial RV32V aumenta por um fator de 1,2 (com o loop principal 1,4X), mas a contagem de instruções dinâmicas é um fator de 43X menor!

Embora a contagem de instruções dinâmicas seja uma grande diferença, em nossa opinião, essa é a segunda disparidade mais significativa entre o SIMD e a arquitetura vetorial. A falta de um registrador de comprimento de vetor explode o número de instruções, bem como o código de book keeping. ISAs como MIPS-32 e x86-32 que seguem a doutrina incrementalista devem duplicar todas as antigas instruções SIMD definidas para registradores SIMD mais curtos sempre que dobrarem a largura SIMD. Certamente, centenas de instruções MIPS-32 e x86-32 foram criadas ao longo de muitas gerações de ISAs SIMD e outras centenas ainda virão no futuro. A carga cognitiva sobre o programador assembly deve ser esmagadora com essa abordagem de força bruta para a evolução da ISA. Como alguém pode lembrar o que a instrução `vfmadd213pd` significa e quando usá-la?

Facilidade de Programação

Em comparação, o código RV32V não é afetado pelo tamanho da memória dos registradores vetoriais. Não apenas o RV32V permanece inalterado se o tamanho da memória vetorial se expande, como você não precisa recompilar o código. Como o processador fornece o valor do comprimento máximo de vetor `mvl`, o código na Figura 8.3 não é alterado se um processador aumenta a memória vetorial de 1024 bytes para, digamos, 4096 bytes, ou a diminui para 256 bytes.


Isolamento de Arq. da Impl.

Ao contrário do SIMD, onde o ISA determina o hardware necessário—e alterar o ISA significa modificar o compilador—o RV32V ISA permite que os projetistas de processador escolham os recursos para o paralelismo de dados em sua aplicação sem afetar o programador ou o compilador. Pode-se argumentar que o SIMD viola o princípio de projeto do ISA do do Capítulo 1 de isolar a arquitetura da implementação.

Elegância

Acreditamos que a grande diferença em custo-energia-desempenho, complexidade e facilidade de programação entre a abordagem vetorial modular do RV32V e as arquiteturas incrementalistas SIMD de ARM-32, MIPS-32 e x86-32 pode ser o argumento mais persuasivo para RISC-V.

8.11 Para Saber Mais

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. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

, `vmovsd`) manipulam quando mod (`n`, 4) $\neq 0$. Elas executam a computação DAXPY apenas um elemento por vez, com o loop repetindo até que a função tenha executado exatamente as `n` operações de multiplicação-adição. Mais uma vez, tal código é desnecessário para o RV32V porque o registrador de comprimento de vetor `vl` e a instrução `setvl` fazem o loop funcionar para qualquer valor de `n`.

```
# a0 é n, a2 é ponteiro para x[0], a3 é ponteiro para y[0], $w13 é a
00000000 <daxpy>:
  0: 2405fffe  li       a1,-2
  4: 00852824  and      a1,a0,a1         # a1 = floor(n/2)*2 (mascara o bit 0)
  8: 000540c0  sll      t0,a1,0x3        # t0 = endereço de byte de a1
  c: 00e81821  addu     v1,a3,t0         # v1 = &y[a1]
 10: 10e30009  beq      a3,v1,38         # se y==&y[a1] goto Fringe (t0==0 so n is 0 | 1)
 14: 00c01025  move     v0,a2            # (slot de atraso) v0 = &x[0]
 18: 78786899  splati.d $w2,$w13[0]      # w2 = preenche o registrador SIMD com cópias de a
Laço:
 1c: 78003823  ld.d     $w0,0(a3)         # w0 = 2 elementos de y
 20: 24e70010  addiu    a3,a3,16          # Incrementa o ponteiro C para y por dois números
 24: 78001063  ld.d     $w1,0(v0)         # w1 = 2 elementos de x
 28: 24420010  addiu    v0,v0,16          # incrementa o ponteiro C para x por dois números
 2c: 7922081b  fmadd.d  $w0,$w1,$w2       # w0 = w0 + w1 * w2
 30: 1467fffa  bne      v1,a3,1c          # se (fim de y != ponteiro para  y) vai para Laço
 34: 7bfe3827  st.d     $w0,-16(a3)       # (slot de atraso) guarda 2 elts de y
Margem:
 38: 10a40005  beq      a1,a0,50          # se (n é par) vai para Pronto
 3c: 00c83021  addu     a2,a2,t0          # (slot de atraso) 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 se n é ímpar)
 4c: f46d0000  sdc1     $f13,0(v1)        # y[n-1] = f13 (guarda resultado ímpar)
Pronto:
 50: 03e00008  jr      ra                 # retorno
 54: 00000000  nop                        # (slot de atraso)
```

Figura 8.5: Código MIPS-32 MSA para DAXPY na Figura 5.7. A sobrecarga de book keeping do SIMD é evidente quando se compara este código com o código RV32V na Figura 8.3. A primeira parte do código MIPS MSA (endereços 0 a 18) duplica a variável escalar `a` em um registrador SIMD e verifica para garantir que `n` seja pelo menos 2 antes de entrar no loop principal. A terceira parte do código MIPS do MSA (endereços 38 a 4c) manipula o caso quando `n` não é um múltiplo de 2. Esse código de book keeping é desnecessário no RV32V porque o registrador de comprimento de vetor `vl` e a instrução `setvl` permitem que o loop funcione para todos os valores de `n`, seja esse ímpar ou par.

```
# eax é i, n é esi, a é xmm1, ponteiro para x[0] é ebx, ponteiro para 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              # se n < 4 vai para Margem
   25: 31 d2             xor    edx,edx         # edx = 0
 Laço:
   27: c5 fd 28 04 d3    vmovapd ymm0,[ebx+edx*8] # carrega quatro elementos de 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 # guarda em quatro elementos de y
   37: 83 c2 04          add    edx,0x4
   3a: 39 c2             cmp    edx,eax           # compara com n
   3c: 72 e9             jb     27                # repete o laço se < n
 Margem:
   3e: 39 c6             cmp    esi,eax           # algum elemento de margem?
   40: 76 17             jbe    59                # se (n mod 4) == 0 vai para Pronto
 LaçoMargem:
   42: c5 fb 10 04 c3    vmovsd xmm0,[ebx+eax*8] # carrega elemento de 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 # guarda no elemento de y
   52: 83 c0 01          add    eax,0x1           # incrementa o contador da Margem
   55: 39 c6             cmp    esi,eax           # compara os contadores do Laço e Margem
   57: 75 e9             jne    42 <daxpy+0x42>   # repete LaçoMargem se != 0
 Done:
   59: 5b                pop    ebx               # Epílogo da função
   5a: 5e                pop    esi
   5b: c3                ret
```

Figura 8.6: Código x86-32 AVX2 para DAXPY na Figura 5.7. A instrução SSE `vmovsd` **no endereço a carrega** `a` **na metade do registrador** `xmm1` **de 128 bits. A instrução SSE** `vmovddup` **no endereço 14 duplica** `a` **em ambas as metades de** `xmm1` **para a computação posterior de SIMD. A instrução AVX** `vinsertf128` **no endereço 1d faz quatro cópias de** `a` **em** `ymm2` **iniciando nas duas cópias de** `a` **em** `xmm1`**. As três instruções AVX nos endereços 42 a 4d** (`vmovsd`, `vfmadd213s`

9 RV64: Instruções de Endereço de 64 bits

C. Gordon Bell (1934-) foi um dos principais arquitetos de duas das arquiteturas de minicomputadores mais populares da época: o PDP-11 da Digital Equipment Corporation (endereçamento de 16 bits), anunciado em 1970, e seu sucessor sete anos depois, a Digital Equipment Corporation VAX-11 (Virtual Address eXtension), com endereçamento de 32 bits.

> *There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.*
>
> —C. Gordon Bell, 1976

9.1 Introdução

As Figuras 9.1 até 9.4 mostram representações gráficas das versões RV64G das instruções RV32G. Elas ilustram o pequeno aumento no número de instruções para alternar para um ISA de 64 bits no RISC-V. Assim, `sub` no RV64I subtrai dois números de 64 bits em vez de dois números de 32 bits como no RV32I. O RV64 é uma ISA próximo, mas na verdade diferente do RV32; ela adiciona algumas instruções e as instruções básicas fazem coisas ligeiramente diferentes.

Por exemplo, o algoritmo de Ordenaçaõ por Inserção para RV64I na Figura 9.8 está bem parecido com o código do RV32I na Figura 2.8 na página 30 no Capítulo 2. É o mesmo número de instruções e de bytes. As únicas mudanças são que as instruções load e store word se tornam load e store doublewords, e o incremento de endereço alterna de 4 (words de 4 bytes) para 8 (doublewords de 8 bytes). A Figura 9.5 lista os opcodes das instruções RV64GC que estão presentes nas Figuras 9.1 até 9.4.

Apesar do RV64I ter endereços de 64 bits e um tamanho de dados padrão de 64 bits, palavras de 32 bits são tipos de dados válidos em programas. Portanto, o RV64I precisa suportar palavras, assim como o RV32I precisa suportar bytes e halfwords. Mais especificamente, como os registradores têm agora 64 bits de largura, o RV64I adiciona versões de palavras de adição e subtração: `addw, addiw, subw`. Eles truncam seus resultados para 32 bits e gravam o resultado com sinal estendido no registrador de destino. O RV64I também inclui versões em word das instruções de deslocamento para obter o resultado de deslocamento de 32 bits em vez de um resultado de deslocamento de 64 bits: `sllw, slliw, srlw, srliw, sraw, sraiw`. Para fazer transferências de dados de 64 bits, ele tem load e store doubleword: `ld, sd`. Finalmente, assim como há versões não assinadas de load byte e load halfword no RV32I, o RV64I deve ter uma versão não assinada de load word: `lwu`.

Por razões semelhantes, o RV64M precisa adicionar versões em word de multiplicar, dividir e de restante: `mulw, divw, divuw, remw, remuw`. Para permitir que o programador sincronize em ambos tipos, words e double words, o RV64A adiciona versões de double words de todas as 11 instruções.

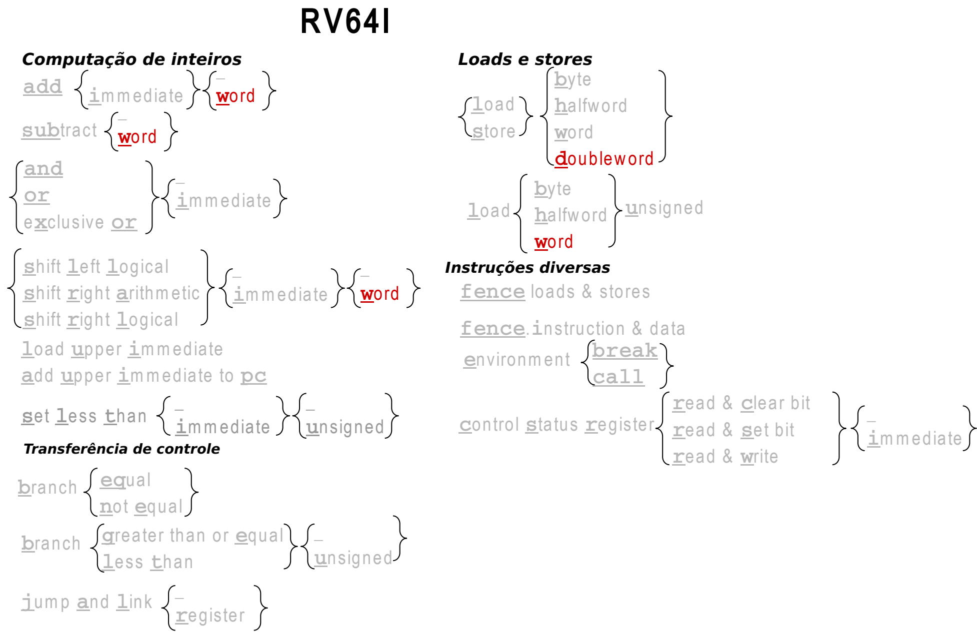


Figura 9.1: Diagrama das instruções do RV64I. As letras sublinhadas são concatenadas da esquerda para a direita para formar instruções RV64I. A parte escurecida compreende as antigas instruções RV32I, estendidas para operar em registradores de 64 bits; a parte escura (vermelha) compreende as novas instruções para o RV64I.

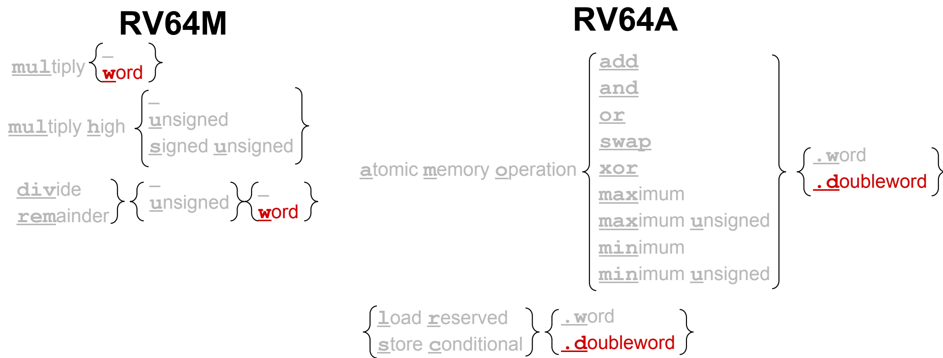


Figura 9.2: Diagramas das instruções RV64M e RV64A.

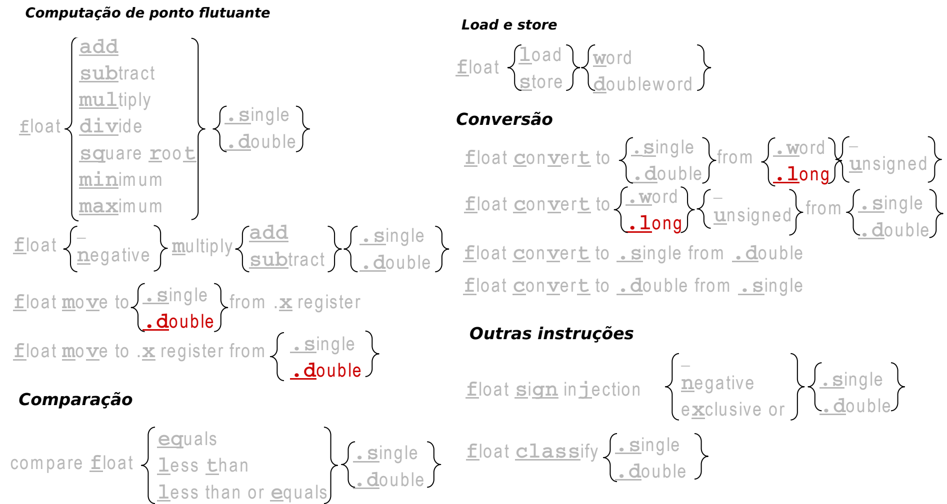


Figura 9.3: Diagrama das instruções RV64F e RV64D.

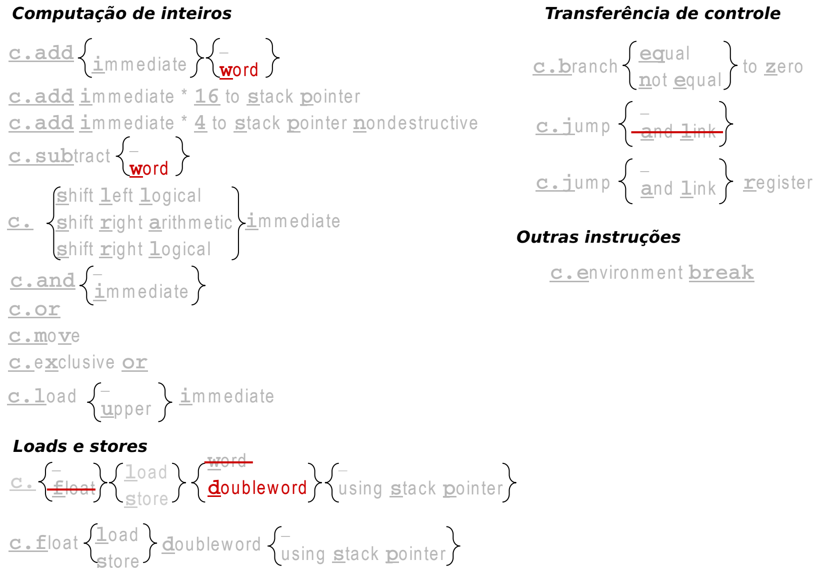


Figura 9.4: Diagrama das instruções do RV64C.

31 — 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
000000	shamt	rs1	001	rd	0010011	I slli
000000	shamt	rs1	101	rd	0010011	I srli
010000	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

Extensão Padrão RV64M (além do 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

Extensão Padrão RV64A (além do 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

Extensão Padrão RV64F (além do 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

Extensão Padrão RV64D (além do 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

Figura 9.5: Mapa de opcode RV64 das instruções base e extensões opcionais. Mostra layout de instruções, opcodes, tipo de formato e nome. (A Tabela 19.2 de [Waterman and Asanović 2017] é a base desta figura.)

RV64F e RV64D adicionam doublewords de inteiros às instruções de conversão, chamando-as de *longs* para evitar confusão com dados de ponto flutuante de precisão dupla: `fcvt.ls, fcvt.ld, fcvt.lu.s, fcvt.lu .d, fcvt.sl, fcvt.s.lu, fcvt.dl, fcvt.d.lu`. Como os registradores inteiros `x` têm agora 64 bits de largura, eles agora podem conter dados de ponto flutuante de precisão dupla, então o RV64D adiciona duas instruções de movimentação de ponto flutuante: `fmv.xw` e `fmv.wx` .

A única exceção à relação de superconjunto entre o RV64 e o RV32 são as instruções compactadas. O RV64C substituiu algumas instruções do RV32C, já que outras instruções reduziram o código para endereços de 64 bits. Além disso, o RV64C descarta o salto e o link compactados (`c.jal`) e os loads de inteiros e de ponto flutuante e as instruções de store words (`c.lw`, `c.sw`, `c.lwsp`, `c.swsp`, `c.flw`, `c.fsw`, `c.flwsp`, and `c.fswsp`). Em seu lugar, são adicionadas as instruções de adição e subtração de words mais populares (`c.addw, c.addiw, c.subw`) e instruções de load e store de double words (`c.ld, c.sd, c.ldsp, c.sdsp`).

■ ***Elaboração: Os ABIs do RV64 são lp64, lp64f, and lp64d.***

lp64 significa que os tipos de dados da linguagem C "long" e "pointer" são de 64 bits; "int" ainda é 32 bits. Os sufixos f e d indicam como os argumentos de ponto flutuante são passados, ocorrendo o mesmo no RV32 (veja o Capítulo 3).

■ ***Elaboração: Não há diagrama de instruções para RV64V***

porque corresponde exatamente ao RV32V devido à tipagem dinâmica dos registradores. A única mudança é que os tipos dinâmicos de registradores X64 e X64U na Figura 8.2 na página 79 estão disponíveis no RV64V, mas não no RV32V.

9.2 Comparação com outras ISAs de 64 bits usando Ordenação por Inserção

Como Gordon Bell disse na abertura deste capítulo, a falha fatal de arquitetura é ficar sem bits de endereço. Conforme os programas forçaram os limites de um espaço de endereçamento de 32 bits, os arquitetos começaram a fazer versões de endereços de 64 bits de seus ISAs [Mashey 2009].

O mais antigo foi o MIPS em 1991. Ele estendeu todos os registradores e contadores de programa de 32 para 64 bits e adicionou novas versões de 64 bits das instruções do MIPS-32. Todas as instruções de linguagem assembly do MIPS-64 começam com a letra "d", como `daddu` ou `dsll` (veja a Figura 9.10). Os programadores podem misturar as instruções MIPS-32 e MIPS-64 no mesmo programa. O MIPS-64 derrubou o slot de atraso de load do MIPS-32 (o pipeline para em uma dependência de leitura após gravação).

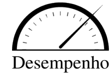
Facilidade de Programação

Uma década depois, era hora de um sucessor para x86-32. Quando os arquitetos aumentaram o tamanho do endereçamento, aproveitaram a oportunidade para fazer mais algumas melhorias no x86-64:

- Aumentou o número de registradores inteiros de 8 para 16 (`r8–r15`);
- Aumentou o número de registradores SIMD de 8 para 16 (`xmm8–xmm15`); and

- Adicionado endereçamento de dados relativos ao PC para melhor suporte ao código independente de posição.

Essas melhorias suavizaram algumas arestas do x86-32.



Você pode ver os benefícios comparando a versão x86-32 do Ordenação por Inserção na Figura 2.11 na Página 33 no Capítulo 2 para a versão x86-64 na Figura 9.11. A ISA mais nova mantém todas as variáveis nos registradores em vez de ter várias na memória, o que reduz a contagem de instruções de 20 para 15 instruções. O tamanho do código é realmente maior em um byte com a ISA mais nova, apesar de ter menos instruções: 46 e 45. A razão é que, para inserir os novos opcodes para permitir mais registradores, o x86-64 adicionou um byte de prefixo para identificar as novas instruções. O comprimento médio da instrução aumenta mais em x86-64 do que em x86-32.

A ARM enfrentou o mesmo problema de endereço mais uma década depois. Em vez de evoluir a antiga ISA para ter endereços de 64 bits, como o x86-64, eles usaram a oportunidade para inventar uma nova ISA. Com o novo começo, eles mudaram muitos dos traços estranhos do ARM-32 para torná-la uma ISA moderna.

- Aumenta o número de registradores inteiros de 15 para 31;
- Remoção do PC do conjunto de registradores;
- Forneça um registrador que seja conectado a zero para a maioria das instruções (`r31`);
- Ao contrário do ARM-32, todos os modos de endereçamento de dados ARM-64 funcionam com todos os tamanhos e tipos de dados;
- o ARM-64 parou de utilizar as várias instruções de load e store do ARM-32; e
- omitiu a opção de execução condicional de instruções ARM-32.

Ele ainda compartilha alguns pontos fracos do ARM-32: códigos de condição para campos de registradores de desvio, origem e destino são movidos no formato de instrução, instruções de movimentação condicional, modos de endereçamento complexos, contadores de desempenho inconsistentes e instruções de apenas 32 bits. O ARM-64 não pode mudar para a ISA Thumb-2, já que o Thumb-2 só funciona com endereços de 32 bits.

Ao contrário do RISC-V, o ARM decidiu adotar uma abordagem maximalista para o design do ISA. Embora certamente seja um ISA melhor que o ARM-32, ele também é maior. Por exemplo, ele tem mais de 1000 instruções e o manual do ARM-64 tem 3185 páginas [ARM 2015]. Além disso, ainda está crescendo. Houve três expansões de ARM-64 desde o seu anúncio há alguns anos.

A Intel não inventou o ISA x86-64. Ao mudar para endereços de 64 bits, a Intel inventou um novo ISA chamado Itanium que era incompatível com o x86-32. Como seu concorrente para processadores x86-32 estava preso ao Itanium, a AMD inventou uma versão de 64 bits do x86-32 chamada AMD64. O Itanium eventualmente falhou, então a Intel foi forçada a adotar o ISA AMD64 como o sucessor de 64 bits do x86-32, que chamamos de x86-64 [Kerner and Padgett 2007].

O código ARM-64 para Ordenação por Inserção na Figura 9.9 parece mais próximo do código RV64I ou do código x86-64 do que do código ARM-32. Por exemplo, com 31 registradores, não há necessidade de salvar e restaurar registradores da pilha. E como o PC não é mais um dos registradores, o ARM-64 usa uma instrução de retorno separada.

A Figura 9.6 se trata de uma tabela que resume o número de instruções e o número de bytes na Ordenação por Inserção para os ISAs. As Figuras 9.8 até 9.11 mostram o código compilado para RV64I, ARM-64, MIPS-64 e x86-64. Frases parênteses nos comentários desses quatro programas identificam as diferenças entre as versões RV32I no Capítulo 2 e estas versões RV64I.

O MIPS-64 precisa de mais instruções, principalmente por causa das instruções `nop` dos slots não preenchidos de desvios atrasados. O RV64I precisa de menos por causa das instruções de comparação e desvio e por não possuir desvios com atraso. Enquanto o ARM-64

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

Figura 9.6: Número de instruções e tamanho do código para a classificação de inserção para quatro ISAs. O ARM Thumb-2 e o microMIPS são ISAs de endereço de 32 bits, portanto, não estão disponíveis para o ARM-64 e o MIPS-64.

e o x86-64 precisam de duas instruções de comparação que são desnecessárias para o RV64I, seus modos de endereçamento, os quais escalam,evitam as instruções aritméticas de endereço necessárias no RV64I, dando-lhes o menor número de instruções. Entretanto, RV64I+RV64C possui um tamanho de código muito menor, conforme explica a próxima seção.

■ ***Elaboração: ARM-64, MIPS-64 e x86-64 não são os nomes oficiais.***

Os nomes oficiais são: ARMv8 é o que chamamos de ARM-64, MIPS-IV é MIPS-64 e AMD64 é x86-64 (veja a barra lateral na página anterior para ler a história do x86-64).

9.3 Tamanho do Programa

A Figura 9.7 compara tamanhos médios de código para RV64, ARM-64 e x86-64. Compare esta figura com a Figura 1.5 na página 10 no Capítulo 1. Primeiro, o código RV32GC é quase idêntico em tamanho ao RV64GC; é apenas 1% menor. Essa proximidade também é verdadeira para RV32I e RV64I. Enquanto o código ARM-64 é 8% menor que o código ARM-32, não existe uma versão de endereço de 64 bits do Thumb-2, portanto, todas as instruções permanecem com 32 bits. Logo, o código ARM-64 é 25% *maior* que o código ARM Thumb-2. O código para x86-64 é 7% maior que o código x86-32 devido à adição de opcodes de prefixo a instruções x86-64 para acomodar novas operações e o conjunto expandido de registradores. O RV64GC ganha, visto que o código ARM-64 é 23% maior que o código RV64GC e x86-64 é 34% maior que o RV64GC. Essa diferença é grande o suficiente para melhorar o desempenho devido a taxas mais baixas de falta de cache de instrução ou para reduzir custos, permitindo um cache de instruções menor que ainda fornece taxas de falhas satisfatórias.

9.4 Considerações Finais

Um dos problemas de ser pioneiro é que você sempre comete erros, e eu nunca, nunca quero ser um pioneiro. É sempre melhor chegar em segundo, quando você pode ver os erros cometidos pelos pioneiros.

—Seymour Cray, arquiteto do primeiro supercomputador, 1976

MIPS já está em seu terceiro dono. A Imagination Technologies, que comprou o MIPS ISA em 2012 por 100 milhões, vendeu sua divisão MIPS para a Tallwood Venture Capital em 2017 por 65 milhões.

Ficar sem bits para endereçamento é o calcanhar de Aquiles da arquitetura de computadores. Muitas arquiteturas morreram de uma ferida ali. O ARM-32 e o Thumb-2 permanecem como arquiteturas de 32 bits, portanto, não ajudam em grandes programas. Algumas ISAs como o MIPS-64 e o x86-64 sobreviveram à transição, mas o x86-64 não é um bom exemplo de design da ISA e o futuro do MIPS-64 não está claro no momento em que este texto foi escrito. O ARM-64 é uma nova ISA grande, e o tempo dirá quão bem sucedido será.

abc
Facilidade de Programação

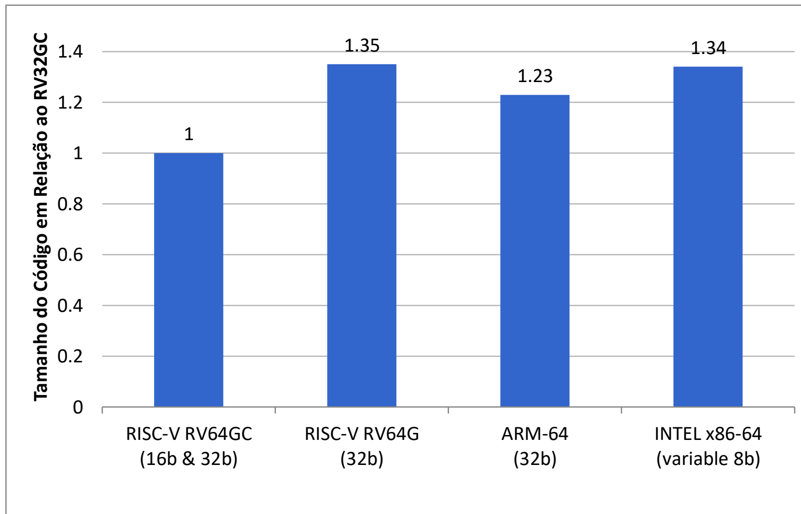


Figura 9.7: Tamanhos de programa relativos para RV64G, ARM-64 e x86-64 em comparação com o RV64GC. Esta comparação mede programas muito maiores do que na Figura 9.6. Este gráfico é o equivalente, com endereço de 64 bits, ao gráfico de ISAs de 32 bits na Figura 1.5 na Página 10 no Capítulo 1. O tamanho do código RV32C quase corresponde ao RV64C; é 1% menor. Não existe a opção Thumb-2 para o ARM-64, portanto, o núcleo de outros ISAs de 64 bits excede significativamente o tamanho do código RV64GC. Os programas medidos foram os benchmarks SPEC CPU2006 usando os compiladores GCC [Waterman 2016].

Espaço para Crescimento

Tamanho do Programa

Elegância

O RISC-V beneficiou-se de projetar as arquiteturas de 32 bits e de 64 bits juntas, enquanto as ISAs mais antigas tiveram que arquitetá-las sequencialmente. Sem surpresa, a transição entre 32 bits e 64 bits é mais fácil para programadores RISC-V e escritores de compiladores; o RV64I ISA possui praticamente todas as instruções do RV32I. De fato, é por isso que podemos listar ambos RV32GCV e RV64GCV em apenas duas páginas do cartão de referência. Mais importante, o design simultâneo significava que a arquitetura de 64 bits não precisava ser espremida em um espaço apertado de opcode de 32 bits. O RV64I tem muito espaço para extensões de instrução opcionais, particularmente RV64C, o que o torna líder no tamanho do código.

Vemos a arquitetura de 64 bits como mais uma evidência da solidez do RISC-V, reconhecidamente mais fácil de ser alcançado se você começar 20 anos depois, para que possa pegar emprestado as boas ideias dos pioneiros e aprender com seus erros.

■ ***Elaboração: RV128***

O RV128 começou como uma piada interna com os arquitetos RISC-V, simplesmente para mostrar que um endereço de 128 bits ISA era possível. No entanto, os computadores de escala warehouse podem em breve ter mais de 2^{64} bytes de armazenamento de semicondutores (DRAM e memória Flash), que os programadores podem querer acessar como um endereço de memória. Há também propostas para usar um endereço de 128 bits para melhorar a segurança [Woodruff et al. 2014]. O manual do RISC-V especifica um ISA completo de 128 bits chamado RV128G [Waterman and Asanović 2017]. As instruções adicionais são basicamente as mesmas necessárias para ir do RV32 para o RV64, no qual as Figuras 9.1 à 9.4 ilustram. Todos os registradores também crescem para 128 bits, e as novas instruções RV128 especificam versões de 128 bits de algumas instruções (usando Q no nome para `quadword`) ou versões de 64 bits de outros (usando D no nome para `doubleword`).

9.5 Para Saber Mais

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. Asanović, 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 instruções, 76 bytes, ou 52 bytes com RV64C)
# a1 é n, a3 aponta para a[0], a4 é i, a5 é j, a6 é x
   0: 00850693  addi  a3,a0,8    # (8 vs 4) a3 é ponteiro para a[i]
   4: 00100713  li    a4,1       # i = 1
Laço Externo:
   8: 00b76463  bltu  a4,a1,10   # se i < n, salta para Continuar Laço Externo
Sair Laço Externo:
   c: 00008067  ret              # retorno da função
Continuar Laço Externo:
  10: 0006b803  ld    a6,0(a3)   # (ld vs lw) x = a[i]
  14: 00068613  mv    a2,a3      # a2 é ponteiro para a[j]
  18: 00070793  mv    a5,a4      # j = i
Laço Interno:
  1c: ff863883  ld    a7,-8(a2)  # (ld vs lw, 8 vs 4) a7 = a[j-1]
  20: 01185a63  ble   a7,a6,34   # se a[j-1] <= a[i], salta para Sair Laço Interno
  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) decrementa a2 para apontar para a[j]
  30: fe0796e3  bnez  a5,1c      # if j != 0, salta para Laço Interno
Sair Laço Interno:
  34: 00379793  slli  a5,a5,0x3  # (8 vs 4) multiplica a5 por 8
  38: 00f507b3  add   a5,a0,a5   # a5 é agora endereço de byte de 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    # incrementa a3 para apontar para a[i]
  48: fc1ff06f  j     8          # salta para Laço Externo
```

Figura 9.8: Código RV64I para Ordenação por Inserção na Figura 2.5. O programa de linguagem assembly RV64I é muito semelhante à linguagem assembly RV32I na Figura 2.8 na página 30 no Capítulo 2. Listamos as diferenças entre parênteses nos comentários. O tamanho dos dados é agora de 8 bytes em vez de 4, então três instruções alteram a constante de 4 para 8. Essa largura extra também estende duas words de load (`lw`) para carregar as doublewords (`ld`) e duas armazene palavras (`sw`) para armazenar doublewords (`sd`).

```
# ARM-64 (16 instructions, 64 bytes)
# x0 aponta para a[0], x1 é n, x2 é j, x3 é i, x4 é x
  0: d2800023  mov  x3, #0x1            # i = 1
Laço Externo:
  4: eb01007f  cmp  x3, x1              # compara i com n
  8: 54000043  b.cc 10                  # se i < n, salta para Continua Laço Externo
Sair Laço Externo:
  c: d65f03c0  ret                      # retorno da função
Continua Laço Externo:
 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
Laço Interno:
 18: 8b020c05  add  x5, x0, x2, lsl #3  # x5 é ponteiro para a[j]
 1c: f85f80a5  ldur x5, [x5, #-8]       # x5 = a[j]
 20: eb0400bf  cmp  x5, x4              # compara a[j-1] com x
 24: 5400008d  b.le 34                  # if a[j-1]<=a[i], salta para Sair Laço Interno
 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                  # se j != 0, salta para Laço Interno
Sair Laço Interno:
 34: f8227804  str  x4, [x0, x2, lsl #3] # a[j] = x
 38: 91000463  add  x3, x3, #0x1        # i++
 3c: 17fffff2  b    4                   # salta para Laço Externo
```

Figura 9.9: Código ARM-64 para Ordenação por Inserção na Figura 2.5. O programa de linguagem assembly ARM-64 é diferente da linguagem assembly do ARM-32 na Figura 2.11 na Página 33 no Capítulo 2 pois é um novo conjunto de instruções. Os registradores começam com x em vez de a. Os modos de endereçamento de dados podem deslocar um registrador em 3 bits para dimensionar o índice para um endereço de byte. Com 31 registradores, não há necessidade de salvar e restaurar registradores da pilha. Como o PC não é um dos registradores, ele usa uma instrução de retorno separada. De fato, o código parece mais próximo do código RV64I ou do código do x86-64 do que do código ARM-32.

```
# MIPS-64 (24 instruções, 96 bytes)
# a1 é n, a3 é ponteiro para a[0], v0 é j, v1 é i, t0 é x
  0: 64860008 daddiu a2,a0,8   # (daddiu vs addiu, 8 vs 4) a2 é ponteiro para a[i]
  4: 24030001 li     v1,1      # i = 1
Laço Externo:
  8: 0065102b sltu   v0,v1,a1  # "seta" quando i < n
  c: 14400003 bnez   v0,1c     # se i < n, salta para Continuar Laço Externo
 10: 00c03825 move   a3,a2     # a3 é ponteiro para a[j] (slot preenchido)
 14: 03e00008 jr     ra        # retorno da função
 18: 00000000 nop              # slot de atraso de desvio não preenchido
Continuar Laço Externo:
 1c: dcc80000 ld     a4,0(a2)  # (ld vs lw) x = a[i]
 20: 00601025 move   v0,v1     # j = i
Laço Interno:
 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  # (sem slot de atraso de store) "seta" a[i] < a[j-1]
 2c: 11400005 beqz   a6,44     # se a[j-1] <= a[i], pula para Sair Laço Interno
 30: 00000000 nop              # slot de atraso de desvio não preenchido
 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     # se j != 0, salta para Laço Interno (próximo slot preenchido)
 40: 64e7fff8 daddiu a3,a3,-8  # (daddiu vs addiu, 8 vs 4) decr a3 ponteiro para a[j]
Sair do Laço Interno:
 44: 000210f8 dsll   v0,v0,0x3 # (dsll vs sll)
 48: 0082102d daddu  v0,a0,v0  # (daddu vs addu) v0 é agora enderçeo de byte de 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         # salta para Laço Externo (próximo slot de atraso preenchido)
 58: 64c60008 daddiu a2,a2,8   # (daddiu vs addiu, 8 vs 4) incr a2 ponteiro para a[i]
 5c: 00000000 nop              # Desnecessário(?)
```

Figura 9.10: Código MIPS-64 para Ordenação por Inserção na figura 2.5. O programa de linguagem assembly MIPS-64 possui várias diferenças em relação à linguagem assembly MIPS-32 na Figura 2.10 na Página 32 no Capítulo 2. Novamente, como RV64I, a largura extra também estende duas words de load (`lw`) para carregar words em dobro (`ld`) e duas words de armazenamento (`sw`) para armazenar double words (`sd`) . Por fim, o MIPS-64 não possui o slot de atraso de load do MIPS-32; o pipeline trava em uma dependência de leitura após gravação.

```
# x86-64 (15 instruções, 46 bytes)
# rax é j, rcx é x, rdx é i, rsi é n, rdi é ponteiro para a[0]
  0: ba 01 00 00 00 mov edx,0x1
Laço Externo:
  5: 48 39 f2       cmp rdx,rsi            # compara i com n
  8: 73 23          jae 2d <Exit Loop>     # if i >= n, salta para Sair Laço Externo
  a: 48 8b 0c d7    mov rcx,[rdi+rdx*8]    # x = a[i]
  e: 48 89 d0       mov rax,rdx            # j = i
Laço Interno:
 11: 4c 8b 44 c7 f8 mov r8,[rdi+rax*8-0x8] # r8 = a[j-1]
 16: 49 39 c8       cmp r8,rcx             # compara a[j-1] com x
 19: 7e 09          jle 24 <Exit Loop>     # se a[j-1]<=a[i], salta para Sair Laço Interno
 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>    # se j != 0, salta para Laço Interno
Sair Laço Interno:
 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>     # salta para Laço Externo
Sair Laço Externo:
 2d: c3             ret                    # retorno da função
```

Figura 9.11: Código x86-64 para Ordenação por Inserção na Figure 2.5. O programa de linguagem assembly x86-64 é bem diferente da linguagem assembly do x86-32 na Figura 2.11 na Página 33 no Capítulo 2. Primeiro, ao contrário do RV64I, os registradores mais amplos têm nomes diferentes `rax, rcx, rdx, rsi, rdi, r8`. Segundo, porque x86-64 adicionou mais 8 registradores, agora há o suficiente para manter todas as variáveis nos registradores em vez de na memória. Terceiro, as instruções x86-64 são mais longas do que as de x86-32, já que muitas precisam preceder 8 bits ou 16 bits para se ajustarem às novas instruções no espaço opcode. Por exemplo, incrementar ou decrementar um registrador (`inc, dec`) utiliza 1 byte em x86-32, mas 3 bytes em x86-64. Portanto, embora muitas instruções a menos, o tamanho de código x86-64 da Ordenação por Inserção é quase idêntico a x86-32: 45 bytes vs. 46 bytes.

10 Arquitetura Privilegiada RV32/64

Edsger W. Dijkstra (1930–2002) recebeu o Prêmio Turing de 1972 por suas contribuições fundamentais para o desenvolvimento de linguagens de programação.

Simplicity is prerequisite for reliability.

—Edsger W. Dijkstra

10.1 Introdução

Até o momento, o livro se concentrou no suporte RISC-V para computação de propósito geral: todas as instruções que introduzimos estão disponíveis no *modo de usuário*, onde o código do aplicativo geralmente é executado. Este capítulo apresenta dois novos Modos de *privilégio*: *modo de máquina*, que executa o código mais confiável e o *modo supervisor*, que provê suporte para sistemas operacionais como Linux, FreeBSD e Windows. Ambos os novos modos são mais privilegiados do que o modo de usuário, daí o título do capítulo. Os modos mais privilegiados geralmente têm acesso a todos os recursos de modos menos privilegiados, e adicionam funcionalidades adicionais não disponíveis nos modos menos privilegiados, como a capacidade de lidar com interrupções e executar E/S. Os processadores costumam gastar a maior parte do tempo de execução em seu modo menos privilegiado; interrupções e excessões transferem o controle para o modo mais privilegiado.

Os programas embarcados e os sistemas operacionais usam os recursos desses novos modos para responder a eventos externos, como a chegada de pacotes de rede; suporte a multitarefa e proteção entre tarefas; e abstração e virtualização dos recursos de hardware. Dada a amplitude desses tópicos, um guia para programadores completo seria um livro adicional inteiro; em vez disso, este capítulo tem como objetivo mostrar os mais importantes recursos

Instruções RV32/64 Privilegiadas

$\left\{\begin{array}{l}\underline{\text{m}}\text{achine-mode}\\ \underline{\text{s}}\text{upervisor-mode}\end{array}\right\}$ trap $\underline{\texttt{ret}}$urn

$\underline{\text{s}}$upervisor-mode $\underline{\texttt{fence}}$.$\underline{\text{v}}$irtual $\underline{\text{m}}$emory $\underline{\text{a}}$ddress

$\underline{\text{w}}$ait $\underline{\text{f}}$or $\underline{\text{i}}$nterrupt

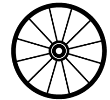
Figura 10.1: Diagrama das instruções privilegiadas do RISC-V.

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

Figura 10.2: Layout de instruções privilegiadas RISC-V, opcodes, tipo de formato e nome. (A tabela 6.1 do [Waterman and Asanović 2017] é a base desta figura.)

relacionados do RISC-V. Os programadores desinteressados no tempo de execução e sistemas operacionais em sistemas embarcados podem pular ou dar uma olhada rápida neste capítulo.

A Figura 10.1 é uma representação gráfica das instruções privilegiadas do RISC-V, e a Figura 10.2 lista essas instruções de operação. Como você pode ver, a arquitetura privilegiada adiciona pouquíssimas instruções; em vez disso, vários novos registradores de controle e status (CSRs) expõem a funcionalidade adicional.



Simplicidade

Este capítulo descreve as arquiteturas com privilégios RV32 e RV64 juntas. Alguns conceitos diferem apenas no tamanho de um registrador de inteiros, portanto, para manter descrições concisas, introduzimos o termo XLEN para se referir à largura de um registrador de inteiros em bits. XLEN é 32 para RV32 ou 64 para RV64.

***Hart* É uma abreviação de *hardware thread*.** Usamos o termo para distingui-las dos threads de software, que a maioria dos programadores estão familiarizados. Threads de software são multiplexadas por tempo de harts. A maioria dos núcleos de processador suporta apenas uma hart.

10.2 Modo de Máquina para Sistemas Embarcados Simples

O modo de máquina, abreviado como M-mode, é o modo mais privilegiado que uma *hart*(thread de hardware) RISC-V pode executar. As harts em execução no M-mode têm acesso total à memória, E/S e recursos de sistema de baixo nível necessários para boot e configurar o sistema. Como tal, é o único modo de privilégio que todos os processadores padrão RISC-V implementam; de fato, os microcontroladores RISC-V simples suportam *apenas* o M-mode. Tais sistemas são o foco desta seção.

A característica mais importante do modo de máquina é a capacidade de interceptar e lidar com *exceções*: eventos incomuns em tempo de execução. RISC-V classifica exceções em duas categorias. *Exceções síncronas* surgem como um resultado da execução de instruções, como ao acessar um endereço de memória inválido ou executando uma instrução com um código de operação inválido. *Interrupções* são eventos externos que são assíncronos com o fluxo de instruções, como um clique no botão do mouse. Exceções no RISC-V são *precisas*: todas instruções antes da exceção executam completamente, e nenhuma das instruções subseqüentes parecem ter começado a execução. A Figura 10.3 lista as causas da exceção padrão.

Exceções de endereço de instrução desalinhadas não podem ocorrer com a extensão C porque nunca é possível pular para um endereço estranho: desvios e JAL imediatos são sempre pares, e JALR esconde o bit menos significativo do seu endereço efetivo. Sem a extensão C, este exceção ocorre ao saltar para um endereço que é igual a 2 mod 4.

Cinco tipos de exceções síncronas podem ocorrer durante a execução no M-mode:

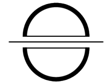
- *Exceções de falha de acesso* surgem quando um endereço de memória física não suporta o tipo de acesso — por exemplo, tentando armazenar em uma ROM.
- *Exceções de ponto de interrupção* surgem da execução de um instrução `ebreak`, ou quando um endereço ou datum corresponde a um gatilho de depuração.
- *Exceções de chamada de ambiente* surgem da execução de um `ecall` instrução.
- *Exceções de instrução ilegal* resultam da decodificação de um código de operação inválido.

Interrupt / $\overline{\text{Exceção}}$ mcause[XLEN-1]	Código de Exceção mcause[XLEN-2:0]	Descrição
1	1	Interrupção de software de supervisor
1	3	Interrupção de software da máquina
1	5	Interrupção de temporizador de supervisor
1	7	Interrupção do temporizador da máquina
1	9	Interrupção externa do supervisor
1	11	Interrupção externa da máquina
0	0	Endereço de instrução desalinhado
0	1	Falha de acesso à instrução
0	2	Instrução ilegal
0	3	Ponto de parada
0	4	Endereço de load desalinhado
0	5	Falha de acesso de Load
0	6	Endereço do store desalinhado
0	7	Erro de acesso ao Store
0	8	Chamada de ambiente do U-mode
0	9	Chamada de ambiente do S-mode
0	11	Chamada de ambiente do M-mode
0	12	Falha na página de instruções
0	13	Falha na página de Load
0	15	Falha na página de Store

Figura 10.3: Exceções RISC-V e causas de interrupção. O bit mais significativo de `mcause` está "setado" como 1 para interrupções ou 0 para exceções síncronas, e os bits menos significativos identificam a interrupção ou exceção. Interrupções de supervisor e exceções de falha de página só são possíveis se o modo supervisor é implementado (veja a Seção 10.5). (A tabela 3.6 de [Waterman and Asanović 2017] é a base desta figura.)

- *Exceções de endereço desalinhadas* ocorre quando o endereço efetivo não é divisível pelo tamanho do acesso—por exemplo, `amoadd.w` com um endereço de `0x12`.

Se você se lembra da afirmação do Capítulo 2 de que loads e stores desalinhados são permitidos, você pode estar se perguntando por que eles são listados na Figura 10.3. Existem duas razões. Primeiro, as operações atômicas de memória no Capítulo 6 requerem endereços naturalmente alinhados. Segundo, alguns implementadores optam por omitir o suporte de hardware para loads e stores regulares desalinhados, porque é um recurso difícil de implementar e é usado com pouca frequência. Processadores sem este hardware dependem de um tratador de exceção para interceptar e emular loads e stores desalinhados em software, usando uma sequência de loads e stores menores e alinhados. O código da aplicação não é o mais esperto: a memória desalinhada acessa o trabalho como esperado, embora lentamente, enquanto o hardware permanece simples. Alternativamente, processadores mais eficientes podem implementar loads e stores desalinhados em hardware. Esta flexibilidade de implementação deve-se a decisão do RISC-V de permitir loads e stores desalinhados usando os opcodes regulares load e store, seguindo a diretriz do Capítulo 1 para isolar a arquitetura da implementação.


Isolamento de Arq. da Impl.

Existem três fontes padrão de interrupções: software, timer e eventos externos. As interrupções de software são acionadas ao armazenar em um registrador com mapeamento de memória e geralmente são usados por uma hart para interromper outra hart, um mecanismo que outras arquiteturas se referem como *interrupção do interprocessador*. As interrupções temporizadas são geradas quando o comparador de tempo de uma hart, um registrador com mapeamento de memória chamado `mtimecmp`, corresponde ou excede o contador em tempo real `mtime`. As interrupções externas são geradas por um controlador de interrupção no nível da plataforma, a qual a maioria dos dispositivos externos estão conectados. Como diferentes plataformas de hardware possuem diferentes mapeamentos de memória e exigem características divergentes de seus controladores de interrupção, os mecanismos para aumentar e eliminar essas interrupções diferem de plataforma para plataforma. O que *é* constante em todos os sistemas RISC-V é como as exceções são tratadas e as interrupções são mascaradas, o tópico da próxima seção.

10.3 Manipulação de Exceção em Modo de Máquina

Oito registradores de status e controle (CSRs) são parte integrante do manejo da exceção do modo de máquina:

- `mstatus`, *Machine Status, ou Status da Máquina*, detém a permissão de interrupção global, juntamente com uma infinidade de outros estados, como mostra a Figura 10.4.
- `mip`, *Machine Interruption Pending, ou Interrupção da máquina pendente*, lista as interrupções atualmente pendentes (Figura 10.5).
- `mie`, *Machine Interrupt Enable, ou Habilitação de Interrupção da Máquina*, lista que interrupções o processador pode tomar e quais deve ignorar (Figura 10.5).
- `mcause`, *Machine Exception Cause, ou Causa de Exceção de Máquina* , indica qual exceção ocorreu (Figura 10.6).
- `mtvec`, *Machine Trap Vector, ou Vetor de trap de máquina*, detém o endereço para qual o processador salta quando ocorre uma exceção (Figura 10.7).

XLEN-1	XLEN-2 ... 23	22	21	20	19	18	17
SD	*Reserved*	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	*Res.*	SPP	MPIE	*Res.*	SPIE	*Res.*	MIE	*Res.*	SIE	*Res.*
2	2	2	2	1	1	1	1	1	1	1	1	1

Figura 10.4: O CSR `mstatus`. Os únicos campos presentes em processadores simples com apenas o modo Máquina e sem as extensões F e V são os de ativação de interrupção global, MIE e MPIE, que após uma exceção mantém o valor antigo de MIE. XLEN é 32 para RV32 ou 64 para RV64. A Figura 3.7 de [Waterman and Asanović 2017] é a base desta figura; veja a Seção 3.1 desse documento para uma descrição dos outros campos.

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	MEIP	*Res.*	SEIP	*Res.*	MTIP	*Res.*	STIP	*Res.*	MSIP	*Res.*	SSIP	*Res.*
Reserved	MEIE	*Res.*	SEIE	*Res.*	MTIE	*Res.*	STIE	*Res.*	MSIE	*Res.*	SSIE	*Res.*
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figura 10.5: CSRs de Interrupção de Máquina. Eles são registradores de leitura/gravação XLEN-bit que mantêm pendente as interrupções (`mip`) e os bits de ativação da interrupção (`mie`). Somente os bits correspondentes a interrupções de software de privilégio inferior (SSIP), interrupções de temporizador (STIP) e interrupções externas (SEIP) em `mip` são graváveis por meio desse endereço de CSR; os bits restantes são somente leitura.

XLEN-1	XLEN-2 ... 0
Interrupção	Código de Exceção
1	XLEN-1

Figura 10.6: Máquina e supervisor causam as CSRs (`mcause` e `scause`). Quando uma interrupção trap é tomada, o CSR é escrito com um código indicando o evento que causou a interrupção trap. O bit de interrupção é "setado" se a interrupção trap foi causada por uma interrupção. O campo Código de Exceção contém um código que identifica a última exceção. A Figura 10.3 mapeia os valores do código para a razão das interrupções trap.

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

Figura 10.7: CSRs de endereço base do vetor de interrupção trap de máquina e supervisor (`mtvec` e `stvec`). São registradores de leitura/gravação XLEN-bit que mantêm a configuração do vetor de interrupção trap, consistindo de um endereço base vetorial (BASE) e um modo vetorial (MODE). O valor no campo BASE deve sempre ser alinhado em um limite de 4 bytes. MODE = 0 significa que todas as exceções configuram o PC como BASE. MODE = 1 "seta" o PC como $(BASE + (4 \times cause))$ em interrupções assíncronas.

XLEN-1 0
Registrador de valor de trap `[m/s]tval`
Registrador PC de exceção `[m/s]epc`
Registrador rascunho para tratadores de interrupção trap `[m/s]scratch`
XLEN

Figura 10.8: CSRs associados a exceções e interrupções. Os registradores de valores de interrupção trap (`mtval` e `stval`) contêm informações adicionais úteis sobre interrupção trap, como o endereço com falha ou uma instrução ilegal. Os PCs de exceção (`mepc` e `sepc`) apontam para a instrução com falha. Os registradores de rascunho (`mscratch` e `sscratch`) dão aos tratadores de interrupções trap um registrador livre para usar.

Codificação	Nome	Abreviação
`00`	Usuário	U
`01`	Supervisor	S
`11`	Máquina	M

Figura 10.9: Níveis de privilégios do RISC-V e suas codificações.

- `mtval`, *Machine Trap Value, ou Valor trap de máquina*, contém informações adicionais sobre exceção trap: o endereço com falha para exceções de endereço, a própria instrução para exceções de instrução ilegal e zero para outras exceções (Figura 10.8).

- `mepc`, *Machine Exception PC, ou PC de exceção de máquina*, aponta para a instrução onde a exceção ocorreu (Figura 10.8).
- `mscratch`, *Machine Scratch, ou Esquema de Máquina*, detém uma palavra de dados para armazenamento temporário para tratadores de traps (Figura 10.8).

O RISC-V também suporta interrupções vetoriais, onde no processador salta para um endereço específico de interrupção, em vez de um único ponto de entrada. Este endereçamento elimina a necessidade de ler e decodificar `mcause`, acelerando a manipulação da interrupção. "setar" `mtvec[0]` como 1 ativa este recurso; a interrupção causa *x* e, em seguida, "seta" o PC para (`mtvec`+4*x*), em vez do habitual `mtvec`.

Ao executar no M-mode, as interrupções são tomadas somente se o bit de habilitação de interrupção, `mstatus`.MIE, está "setado". Além disso, cada interrupção tem seu próprio bit de ativação no `mie` CSR. As posições dos bits em `mie` correspondem aos códigos de interrupção na Figura 10.3: por exemplo, `mie` [7] corresponde ao temporizador de interrupção M-mode. O CSR `mip` tem o mesmo layout e indica quais interrupções estão atualmente pendentes. Colocando todos os três CSRs juntos, uma interrupção do temporizador de máquina pode ser tomada se `mstatus`.MIE=1, `mie`[7]=1, e `mip`[7]=1.

Quando um hart recebe uma exceção, o hardware sofre atomicamente várias transições de estado:

- O PC da instrução excepcional é preservado em `mepc`, e o PC está "setado" como `mtvec`. (Para exceções síncronas, `mepc` aponta para a instrução que causou a exceção; para interrupções, aponta onde a execução deve ser retomada após a interrupção da manipulação.)

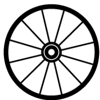
- `mcause` está "setado" para a causa da exceção, conforme codificado na Figura 10.3, e `mtval` está "setado" para o endereço com falha ou outra palavra de informação específica da exceção.

- As interrupções são desabilitadas ao "setar" MIE = 0 no CSR `mstatus`, e o O valor anterior do MIE é preservado no MPIE.

- O modo de privilégio de pré-exceção é preservado no campo MPP do `mstatus` e o modo de privilégio é alterado para M. A Figura 10.9 mostra a codificação do campo MPP. (Se o processador implementar apenas o M-mode, essa etapa será efetivamente ignorada.)

Para evitar sobrescrever o conteúdo dos registradores de inteiros, o prólogo de um tratador de interrupção geralmente começa trocando um registrador de inteiros (digamos, a0) com o CSR de `mscratch`. Normalmente, o software terá dimensionado `mscratch` para poder conter um ponteiro para um espaço adicional no esquema da memória, que o tratador usa para salvar tantos registradores de inteiros quanto seu corpo precisar. Depois que o corpo é executado, o epílogo de um tratador de interrupções restaura os registradores salvos na memória, novamente troca a0 com `mscratch`, restaurando ambos os registradores para seus valores de pré-exceção. Por fim, o tratador retorna com `mret`, uma instrução exclusiva para o modo-M. `mret` configura o PC como `mepc`, restaura a configuração de interrupção anterior copiando o campo MPIE `mstatus` para MIE e "seta" o modo de privilégio para o valor no campo MPP de `mstatus`, essencialmente invertendo as ações descrito no parágrafo anterior.

A Figura 10.10 mostra o código em linguagem assembly do RISC-V para um tratador de interrupção do temporizador seguindo este padrão. O código simplesmente incrementa o comparador temporal e retorna à tarefa anterior, enquanto uma interrupção temporizada mais realista pode chamar um escalonador para alternar entre as tarefas. Não é preemptiva, por isso mantém interrupções desativadas em todo o tratador. Com essas ressalvas à parte, é um exemplo completo de um tratador de interrupções RISC-V em uma única página!


Simplicidade

As vezes é desejável ter uma interrupção de prioridade mais alta durante o processamento de uma exceção de prioridade mais baixa. Infelizmente, há apenas uma cópia dos CSRs de `mepc`, `mcause`, `mtval`, e `mstatus`; tomar uma segunda interrupção destruiria os valores antigos nesses registradores, causando perda de dados sem nenhuma ajuda externa via software. Um tratador de interrupções preemptivo pode salvar esses registradores em uma pilha na memória antes de ativar interrupções, em seguida, antes de sair, desabilita as interrupções e restaura os registradores da pilha.

Facilidade de Programação

Além da instrução `mret` que introduzimos acima, o M-mode fornece apenas uma outra instrução: `wfi` (*Wait For Interrupt, ou Aguarde a interrupção*). `wfi` informa ao processador que não há trabalho útil a fazer, por isso deve iniciar um modo de baixo consumo até que qualquer interrupção fique pendente, por exemplo, (`mie` & `mip`)$\neq$0. Os processadores RISC-V implementam essa instrução de várias maneiras, incluindo interromper o relógio até que uma interrupção se torne pendente; alguns simplesmente executam como um `nop`. Portanto, `wfi` é normalmente usado dentro de um loop.

■ ***Elaboração:*** `wfi` ***funciona independente das interrupções estarem globalmente ativadas.***

Se `wfi` é executado quando as interrupções são ativadas globalmente (`mstatus`.MIE = 1), e então uma interrupção ativada se torna pendente, o processador salta para o tratador de exceções. Se, por outro lado, `wfi` é executado quando as interrupções estão globalmente desativadas e, em seguida, uma interrupção ativada torna-se pendente, o processador continua executando o código após o `wfi`. Esse código normalmente examina o CSR de `mip` para decidir o próximo passo. Esta estratégia pode reduzir a latência de interrupção em comparação com o salto para o tratador de exceção, porque não há necessidade de salvar e restaurar registradores de inteiros.

```
# salva registradores
csrrw a0, mscratch, a0  # salva a0; "seta" a0 = &temp storage
sw a1, 0(a0)            # salva a1
sw a2, 4(a0)            # salva a2
sw a3, 8(a0)            # salva a3
sw a4, 12(a0)           # salva a4

# decodifica a causa da interrupção
csrr a1, mcause         # lê a causa da exceção
bgez a1, exception      # desvia se não for uma interrupção
andi a1, a1, 0x3f       # isola a causa de interrupção
li a2, 7                # a2 = causa de interrupção do temporizador
bne a1, a2, otherInt    # desvia se não for uma interrupção do timer

# trata a interrupção do temporizador incrementando o comparador de tempo
la a1, mtimecmp         # a1 = &time comparator
lw a2, 0(a1)            # carrega os 32 bits mais em baixo do comparador
lw a3, 4(a1)            # carrega os 32 mais acima do comparador
addi a4, a2, 1000       # incrementa os bits mais baixos em 1000 ciclos
sltu a2, a4, a2         # gera o carry-out
add a3, a3, a2          # incrementa os bits mais acima
sw a3, 4(a1)            # guarda os 32 bits acima
sw a4, 0(a1)            # guarda os 32 bits mais abaixo

# restaura registradores e retorna
lw a4, 12(a0)           # restaura a4
lw a3, 4(a0)            # restaura a3
lw a2, 4(a0)            # restaura a2
lw a1, 0(a0)            # restaura a1
csrrw a0, mscratch, a0  # restaura a0; mscratch = &temp storage
mret                    # retorna do tratador
```

Figura 10.10: Código RISC-V para um tratador de interrupção de temporizador simples. O código assume que as interrupções foram globalmente habilitadas pela configuração do `mstatus`.MIE; que as interrupções temporizadas foram habilitado pela configuração `mie`[7]; que o CSR `mtvec` foi "setado" para o endereço deste tratador; e que o CSR de `mscratch` foi "setado" para o endereço de um buffer que contém 16 bytes de armazenamento temporário para salvar registra. O prólogo salva cinco registradores, preservando `a0` em `mscratch` e `a1`–`a4` na memória. Em seguida, ele decodifica a causa da exceção examinando `mcause`: interrompendo se `mcause` < 0 ou exceção síncrona se `mcause`≥0. Se for uma interrupção, verifica-se caso os bits inferiores `mcause` são iguais a 7, indicando uma interrupção do temporizador de M-mode. Se é uma interrupção temporal, adiciona 1000 ciclos ao comparador de tempo, para que a próxima interrupção por tempo ocorra cerca de 1000 ciclos de temporizador no futuro. Finalmente, o epílogo restaura os registradores `a0`–`a4` e `mscratch`, então retorna de onde veio usando `mret`.

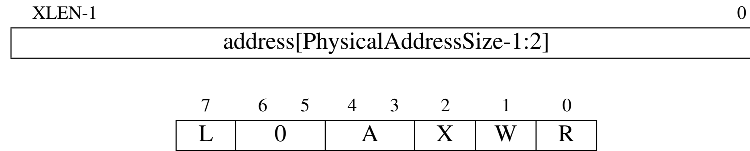


Figura 10.11: Um endereço PMP e registrador de configuração. O registrador de endereço é deslocado para a direita por 2, e se os endereços físicos são menores do que os XLEN-2 bits, os bits superiores são zeros. Os campos R, W e X concedem permissões de leitura, gravação e execução. O campo A "seta" o modo PMP e o campo L bloqueia o PMP e os registradores de endereço correspondentes.

10.4 Modo de Usuário e Isolamento de Processo em Sistemas Embarcados

Facilidade de Programação

Embora o modo Máquina seja suficiente para sistemas embarcados simples, é adequado apenas quando toda a base de código é confiável, uma vez que o modo M tem acesso irrestrito à plataforma de hardware. Normalmente, não é prático confiar em todo o código da aplicação, porque não é conhecido de antemão ou é muito grande para provarmos que é correto. Então, o RISC-V fornece mecanismos para proteger o sistema do código não confiável e para proteger processos não confiáveis uns dos outros.

Código não confiável deve ser proibido de executar instruções privilegiadas, como `mret`, e acessar CSRs privilegiados, como `mstatus`, já que estes permitiriam que o programa assuma o controle do sistema. Essa restrição é realizada com bastante facilidade: um modo de privilégio adicional, *User Mode, ou modo de usuário* (U-mode), nega acesso a esses recursos, gerando uma exceção de instrução ilegal ao tentar usar uma instrução de modo M ou CSR. Caso contrário, U-mode e o M-mode se comportam muito similarmente. O software do modo M pode entrar no modo U configurando `mstatus`.MPP para U (que, como mostra a Figura 10.9, é codificado como 0), e então executando uma instrução `mret`. Se ocorrer uma exceção no U-mode, o controle é retornado para o M-mode.

O código não confiável também deve ser restrito para acessar apenas sua própria memória. Os processadores que implementam os modos M e U têm um recurso chamado *Physical Memory Protection* (PMP), que permite que o modo M especifique quais endereços de memória O modo U pode acessar. O PMP consiste em vários registradores de endereços (geralmente oito a dezesseis) e registradores de configuração correspondentes, que concedem ou negam permissões de escrita, leitura e execução. Quando um processador no U-mode tenta buscar uma instrução, ou executar um load ou store, o endereço é comparado com todos os registradores de endereços do PMP. Se o endereço for maior ou igual a endereço i do PMP, mas menor que o endereço PMP i+1, então o registrador de configuração PMP $i + 1$ decide se esse acesso pode prosseguir; caso contrário isto gera uma exceção de acesso.

A Figura 10.11 mostra o layout de um endereço PMP e registrador de configuração. Ambos são CSRs, com os registradores de endereços chamados de `pmpaddr0` até `pmpaddr`N, onde N +1 é o número de PMPs implementados. Os registradores de endereço são deslocados para a direita dois bits porque os PMPs tem uma granularidade de quatro bytes. Os de configuração são densamente compactados nos CSRs para acelerar a troca de contexto, como mostra a Figura 10.12. A configuração de um PMP consiste em R, W, e X bits, que quando "setado" permite loads, stores e fetches respectivamente, e um campo de modo, A,

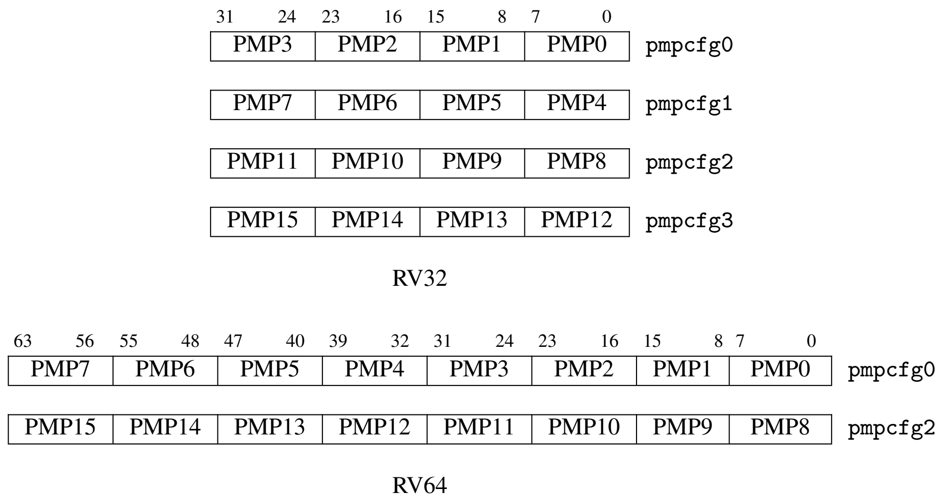


Figura 10.12: O layout das configurações PMP nos CSRs de `pmpcfg`. Para o RV32 (acima), os dezesseis registradores de configuração são agrupados em quatro CSRs. Para o RV64 (abaixo), eles são agrupados nos dois CSRs com numeração par.

que quando 0 desabilita este PMP e quando 1 o habilita. A configuração do PMP também suporta outros modos e podem ser bloqueados, recursos descritos em [Waterman and Asanović 2017].

10.5 Modo de Supervisor para Sistemas Operacionais Modernos

O esquema PMP descrito na seção anterior é atraente para sistemas embarcados porque fornece proteção de memória a um custo relativamente baixo, mas tal esquema tem várias desvantagens que limitam seu uso na computação de propósito geral. Já que o PMP suporta apenas um número fixo de regiões de memória, ele não é escalável para aplicações complexas. E como essas regiões devem ser contíguas na memória física, o sistema pode sofrer de fragmentação de memória. Por fim, o PMP não fornece um suporte eficiente a paginação para armazenamento secundário.

fragmentação ocorre quando a memória está disponível, mas não em pedaços contíguos suficientemente grandes para serem úteis.

Processadores RISC-V mais sofisticados lidam com esses problemas da mesma forma que quase todas as arquiteturas de propósito geral: usando memória virtual baseada em páginas. Esse recurso forma o núcleo do *modo supervisor* (S-mode), um modo de privilégio opcional projetado para suportar sistemas operacionais modernos semelhantes ao Unix, como Linux, FreeBSD e Windows. S-mode é mais privilegiado que o U-mode, mas menos privilegiado do que o M-mode. Como no U-mode, o software de S-mode não pode usar as instruções e CSRs do M-mode, e está sujeito às restrições do PMP. Esta seção abrange as interrupções e exceções do S-mode, e a próxima seção detalha o sistema de memória virtual também do S-mode.

Por que não delegar incondicionalmente as interrupções para o S-mode? Um dos motivos é a virtualização: se o M-mode quiser virtualizar um dispositivo para S-mode, suas interrupções devem ir para o M-mode, não para o S-mode.

Por padrão, todas as exceções, independentemente do modo de privilégio, transferem o controle para o tratador de exceções do M-mode. A maioria das exceções em um sistema Unix, no entanto, deve invocar o sistema operacional, que é executado no S-mode. O tratador de exceções do M-mode poderia redirecionar as exceções para o modo S, mas esse

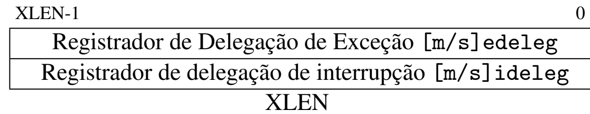


Figura 10.13: Os CSRs delegados. Exceção de máquina e supervisor e CSRs de delegação de interrupções (`medeleg`, `sedeleg`, `mideleg`, `sideleg`). Eles habilitam a delegação para um tratador de interrupções traps de privilégios mais baixos, com o índice da posição de bit habilitando a exceção ou interrupção correspondente no registrador [m/s]ip.

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
Reserved	SEIP	*Res.*	*Res.*	STIP	*Res.*	*Res.*	SSIP	*Res.*
Reserved	SEIE	*Res.*	*Res.*	STIE	*Res.*	*Res.*	SSIE	*Res.*
XLEN-10	1	1	2	1	1	2	1	1

Figura 10.14: CSRs supervisores de interrupção Eles são registradores XLEN-bit de leitura/gravação que mantêm interrupções pendentes (`sip`) e os bits de permissão de interrupção (`sie`) .

código extra retardaria o tratamento da maioria das exceções. Portanto, o RISC-V fornece um mecanismo de *delegação de exceção*, pelo qual interrupções e exceções síncronas podem ser delegadas para o modo S seletivamente, ignorando M-mode completamente.

O modo S não controla diretamente o temporizador e interrupções de software mas, em vez disso, usa a instrução `ecall` para solicitar ao modo M que configure temporizadores ou enviar interrupções de interprocessador em seu nome. Esta convenção de software faz parte da *Interface Binária do Supervisor*.

A CSR `mideleg` (*Delegação de interrupção de máquina*) controla quais interrupções são delegadas para o S-mode (Figura 10.13). Como `mip` e `mie`, cada bit em `mideleg` corresponde ao código de exceção do mesmo número na Figura 10.3. Por exemplo, `mideleg`[5] corresponde ao temporizador de interrupção do S-mode; se "setado", o temporizador de interrupção do S-mode transfere o controle para o tratador de exceção do S-mode, em vez do tratador de exceção do M-mode.

Qualquer interrupção delegada ao modo S pode ser mascarada pelo software de modo S. As CSRs `sie` (*Habilitação de Interrupção do Supervisor*) e `sip` (*Supervisor de Interrupção pendente*) são CSRs do S-Mode que são subconjuntos da CSR de `mie` e `mip`(Figura 10.14). Eles têm o mesmo layout que suas contrapartes do M-mode, mas apenas os bits correspondentes às interrupções que foram delegadas em `mideleg` são legíveis e graváveis através de `sie` e `sip`. Os bits correspondendo a interrupções que não foram delegadas são sempre zero.

O modo M também pode delegar exceções síncronas para o S-mode usando a CSR de `medeleg` (*Delegação de exceção de máquina*) (Figura 10.13). O mecanismo é análogo a delegação de interrupção, mas os bits em `medeleg` correspondem em vez dos códigos de exceção síncrona na Figura 10.3. Por exemplo, a configuração de `medeleg`[15] delegará store às falhas de página no S-mode.

Observe que as exceções nunca transferirão o controle para um modo menos privilegiado, independente das configurações de delegação. Uma exceção que ocorre no M-mode é sempre manipulado no M-mode. Uma exceção que ocorre no S-mode pode ser manipulada por M-mode ou S-mode, dependendo da configuração de delegação, mas nunca pelo U-mode.

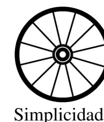
O modo S tem vários CSRs de tratamento de exceção, `scause`, `stvec`, `sepc`, `stval`, `sscratch`, e `sstatus`, que executam a mesma função que suas contrapartes do M-mode descritas na Seção 10.2 (As Figuras 10.7 à 10.8). A Figura 10.15 mostra o layout do registrador

XLEN-1	XLEN-2 ... 20	19	18	17
SD	*Reserved*	MXR	SUM	*Res.*
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]	*Res.*	SPP	*Res.*	SPIE	UPIE	*Res.*	SIE	UIE
2	2	4	1	2	1	1	2	1	1

Figura 10.15: A CSR `sstatus` `sstatus` é um subconjunto de `mstatus` (Figura 10.4), daí o layout similar. SIE e SPIE mantém a atual habilitação e também a habilitação de interrupções de pré-exceções, análogo ao MIE e MPIE em `mstatus`. XLEN é 32 para RV32 ou 64 para RV64. A Figura 4.2 de [Waterman and Asanović 2017] é a base nesta figura; veja Secção 4.1 desse documento para uma descrição dos outros campos.

`sstatus`. A instrução de retorno de exceção do supervisor, `sret`, se comporta da mesma forma que `mret`, mas ele age nos CSRs de tratamento de exceção do S-mode, em vez dos CSRs do M-mode.


Simplicidade

O ato de tirar uma exceção também é muito semelhante ao M-mode. Se um hart leva uma exceção e é delegado ao S-mode, o hardware atomicamente sofre várias transições de estado semelhantes, usando CSRs do S-mode em vez de CSRs do M-mode:

- O PC da instrução excepcional é preservado em `sepc`, e o PC está "setado" como `stvec`.
- `scause` é "setado" como a causa da exceção, conforme codificado na Figura 10.3 e `stval` estão configuradas para o endereço com falha ou outra palavra de informação específica da exceção.
- As interrupções são desabilitadas ao "setar" SIE = 0 no CSR `sstatus`, e o O valor anterior do SIE é preservado no SPIE.
- O modo de privilégio de pré-exceção é preservado no campo SPP de `sstatus`, e o modo de privilégio é alterado para S.

10.6 Memória Virtual Baseada em Páginas

O modo S fornece um sistema convencional de memória virtual que divide a memória em *páginas* de tamanho fixo para fins de conversão de endereço e memória proteção. Quando a paginação está habilitada, a maioria dos endereços (incluindo load e store em endereços efetivos e o PC) são *endereços virtuais* que devem ser traduzidos em *endereços físicos* para acessar a memória física. Os endereços virtuais são convertidos em endereços físicos por meio da travessia de uma árvore de alta raiz conhecida como *page table*. Um nó folha na page table indica se o endereço virtual mapeia para uma página física e, se sim, quais modos de privilégio e tipos de acesso têm permissão para acessar a página. Acessar uma página que não é mapeada ou conceder permissões insuficientes resulta em uma exceção de falha de página (*page fault exception*).

páginas de 4 KiB têm sido populares por cinco décadas começando com o modelo IBM 360 67. Atlas, o primeiro computador com paginação, tinha páginas de 3 KiB (com words de 6 bytes). Nós achamos notável que, depois de meio século de crescimento exponencial no desempenho do computador e capacidade da memória, o tamanho da página permanece praticamente inalterado.

Os esquemas de paginação RISC-V são chamados Sv*X*, onde *X* é o tamanho de um endereço virtual em bits. O esquema de paginação do RV32, Sv32, suporta um espaço

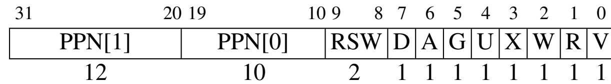


Figura 10.16: Uma entrada da tabela de páginas do RV32 Sv32 (PTE,, ou Page-Table Entry).

de endereço virtual 4 GiB, que é dividido em 2^{10} *megapáginas* de tamanho 4 MiB. Cada megapágina é subdividida em 2^{10} *páginas base* — a unidade fundamental de paginação — cada uma de 4 KiB. Assim, a tabela de páginas do Sv32 é uma árvore de dois níveis de base 2^{10}. Cada valor na tabela de páginas possui um valor de quatro bytes, logo a tabela de páginas por si só possui 4 KiB. Não é coincidência que uma tabela de páginas tenha exatamente o tamanho de uma página: esse design simplifica a alocação de memória do sistema operacional.

A Figura 10.16 mostra o layout de uma entrada de tabela de páginas do Sv32 (PTE), que tem os seguintes campos, explicados da direita para a esquerda:

- O bit V indica se o restante deste PTE é válido (V = 1). E caso V = 0, qualquer tradução de endereço virtual que atravesse este PTE resulta em uma falha de página.
- Os bits R, W e X indicam se a página têm permissões para leitura (read), escrita (store) e execução, respectivamente. Se todos os três bits forem 0, este PTE é um ponteiro para o próximo nível da tabela de páginas; caso contrário, é uma folha do árvore.
- O bit U indica se esta página é uma página do usuário. Se U = 0, o U-mode não pode acessar esta página, mas o S-mode pode. Se U = 1, o U-mode pode acessar esta página, mas o S-mode não.
- O bit G indica que esse mapeamento existe em todos os espaços de endereço virtual, informação que o hardware pode usar para melhorar o desempenho da tradução de endereços. Ele é geralmente usado apenas para páginas que pertencem ao sistema operacional.
- O bit A indica se a página foi acessada desde o último vez que o bit A foi apagado.
- O bit D indica se a página foi suja (isto é, escrita) desde a última vez que o bit D foi apagado.
- O campo RSW é reservado para o uso do sistema operacional; o hardware o ignora.
- O campo PPN contém um número de página físico, que faz parte de um endereço. Se esta PTE é uma folha, a PPN é parte da tradução física endereço. Caso contrário, a PPN fornece o endereço do próximo nível da tabela de páginas. (A Figura 10.16 divide a PPN em dois subcampos para simplificar a descrição do algoritmo de tradução de endereços.)

O sistema operacional depende dos bits A e D para decidir quais páginas para trocar para armazenamento secundário. Limpar periodicamente os bits A ajuda o sistema operacional a saber quais páginas foram usadas menos recentemente. O bit D indica uma página que é ainda mais custosa para trocar, porque ela deve ser escrita de volta para armazenamento secundário.

O RV64 suporta vários esquemas de paginação, mas descrevemos apenas o mais popular, o Sv39, que usa a mesma página base de 4 KiB que o Sv32. As entradas de tabela de páginas duplicam em tamanho para oito bytes, para que possam armazenar endereços físicos maiores. Para manter a invariante que uma tabela de páginas é exatamente o tamanho de uma página, a ordem de grandeza da árvore cai correspondentemente para 2^9, com três níveis de profundidade. O espaço de endereço de 512 GiB do Sv39 está dividido em 2^9 *gigapáginas*,

Os outros esquemas de paginação do RV64 simplesmente adicionam mais níveis a tabela de páginas. O Sv48 é quase idêntico ao Sv39, mas o seu endereço virtual espaço é 2^9 vezes maior e sua tabela de páginas é um nível mais profundo.

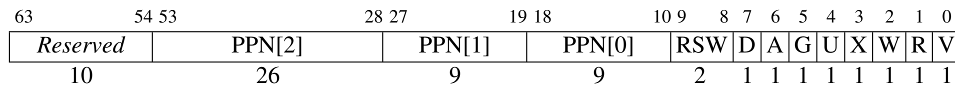


Figura 10.17: Uma entrada de tabela de páginas RV64 Sv39 (PTE, ou Page-Table Entry).

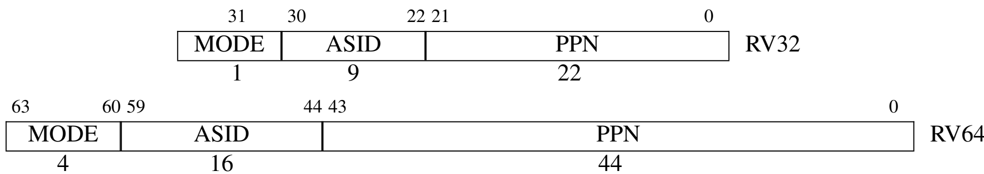


Figura 10.18: A CSR `satp`. As Figuras 4.11 e 4.12 de [Waterman and Asanović 2017] são as bases dessa figura.

cada 1 GiB. Cada gigapágina é subdividida em 2^9 megapáginas, que no Sv39 são um pouco menores que no Sv32: 2 MiB. Cada megapágina é subdividida em 2^9 4 KiB páginas base.

A Figura 10.17 smostra o layout de um Sv39 PTE. É idêntico a um Sv32 PTE, exceto que o campo PPN foi ampliado para 44 bits para suportar Endereços físicos de 56 bits, ou 2^{26} GiB de espaço de endereço físico.

■ ***Elaboração: Bits de endereço não utilizados***

Como os endereços virtuais do Sv39 são menores que um registrador inteiro RV64, você pode se perguntar o que acontece com os 25 bits restantes. O Sv39 dita que bits de endereço 63–39 sejam cópias do bit 38. Assim, os endereços virtuais válidos são `0000_0000_0000_0000`$_{hex}$–`0000_003f_ffff_ffff`$_{hex}$ and `ffff_ffc0_0000_0000`$_{hex}$–`ffff_ffff_ffff_ffff`$_{hex}$. A diferença entre esses dois intervalos é, obviamente, 2^{25} vezes maior que o tamanho dos dois intervalos combinados, aparentemente perdendo 99,9999997% dos valores que um registrador de 64 bits pode representar. Por que não aproveitar melhor esses 25 bits extras? A resposta é que, à medida que os programas crescem, são necessários mais de 512 GiB de espaço de endereço virtual, os arquitetos querem aumentar o espaço de endereço sem quebrar a compatibilidade com versões anteriores. Se permitíssemos que os programas armazenassem dados extras nos 25 bits superiores, seria impossível recuperar posteriormente esses bits para armazenar endereços maiores. Permitir o armazenamento de dados em bits de endereço não utilizados é um erro grave, que se repetiu muitas vezes na história da computação.

Um CSR em S-mode, `satp` (*Supervisor Address Translation and Protection, ou Tradução e Proteção de Endereços de Supervisor*), controla o sistema de paginação. Como mostra a Figura 10.18, `satp` tem três campos. O campo MODE habilita paginação e seleciona a profundidade da tabela de páginas; a Figura 10.19 mostra sua codificação. O campo ASID (*Address Space Identifier, ou Identificador de espaço de endereço*) é opcional e pode ser usado para reduzir o custo da troca de contexto. Finalmente, o campo PPN mantém o endereço físico da tabela de páginas raiz, dividido pela página de tamanho 4 KiB. Normalmente, o software do M-mode escreverá zero em `satp` antes de entrar S-mode pela primeira vez, desabilitando a paginação e, em seguida, o software escreverá novamente depois de configurar as tabelas de páginas.

RV32		
Valor	Nome	Descrição
0	Bare	Sem tradução ou proteção.
1	Sv32	Endereçamento virtual de 32 bits baseado em página.

RV64		
Valor	Nome	Descrição
0	Bare	Sem tradução ou proteção
8	Sv39	Endereçamento virtual de 39 bits baseado em página.
9	Sv48	Endereçamento virtual de 48 bits baseado em página.

Figura 10.19: A codificação do campo MODE no CSR `satp`. A tabela 4.3 de [Waterman and Asanović 2017] é a base dessa figura.

Quando a paginação está habilitada no registrador `satp`, os endereços virtuais de S-mode e o U-mode virtuais são traduzidos em endereços físicos ao atravessar a tabela de páginas, começando no nodo raiz. A Figura 10.20 descreve este processo:

1. O `satp`.PPN fornece o endereço base da tabela de páginas de primeiro nível e VA[31:22] fornece o índice de primeiro nível, então o processador lê o PTE localizado no endereço (`satp`.PPN×4096 + VA[31:22]×4).
2. Esse PTE contém o endereço base da tabela de páginas de segundo nível e VA[21:12] fornece o índice de segundo nível, para que o processador leia a folha PTE localizada em (PTE.PPN×4096 + VA[21:12]×4).
3. O campo PPN da folha PTE e o *deslocamento de página* (tos doze bits menos significativos do endereço virtual original) forma o resultado final: o endereço físico é (LeafPTE.PPN×4096 + VA[11:0]).

O processador então realiza o acesso à memória física. O processo de tradução é quase o mesmo para Sv39 como para Sv32, mas com maiores PTEs e mais um nível de indireção. A Figura 10.27, no final deste capítulo, fornece uma descrição completa do algoritmo para percorrer as tabelas de páginas, detalhando as condições excepcionais e o caso especial de traduções de superpáginas.

Custo

Isso é quase tudo o que há para o sistema de paginação do RISC-V, salvo por um pequeno detalhe. Se todas as buscas, loads e stores de instruções resultassem em vários acessos às tabelas de páginas, a paginação teria seu desempenho reduzido substancialmente! Todos os processadores modernos reduzem essa sobrecarga com um cache de conversão de endereço (geralmente chamado de *TLB*, significando Translation Lookaside Buffer). Para reduzir o custo deste cache, a maioria dos processadores não os mantêm coerentes automaticamente com a tabela de páginas. Se o sistema operacional modifica a tabela de páginas, o cache se torna obsoleto. O S-mode adiciona mais uma instrução para resolver este problema: `sfence.vma` informa ao processador que o software pode ter modificado as tabelas de páginas, para que o processador possa liberar a caches em conformidade. São necessários dois argumentos opcionais, que restringem o escopo do esvaziamento de cache: rs1 indica qual tradução de endereço virtual foi modificado na tabela de páginas, e rs2 fornece o identificador de espaço de endereço do processo cuja tabela de páginas foi modificada. Se x0 é dado para ambos, todo o cache de tradução é liberado.

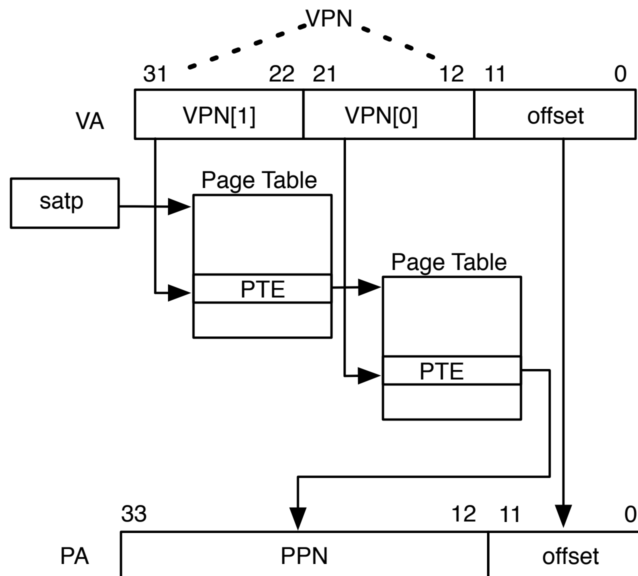


Figura 10.20: Diagrama do processo de tradução de endereços do Sv32.

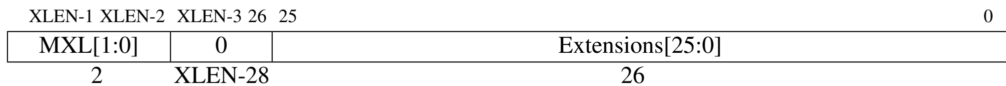
■ ***Elaboração: Coerência do cache de conversão de endereços em multiprocessadores***

`sfence.vma` afeta apenas o hardware de tradução de endereços para a hart que executou a instrução. Quando uma hart muda uma tabela de páginas que outra hart está usando, a primeira hart deve usar uma interrupção do interprocessador para informar à segunda hart que deve executar uma instrução `sfence.vma`. Este procedimento é muitas vezes referido como um *TLB shootdown*.

10.7 CSRs de Identificação e Desempenho

Os CSRs restantes identificam recursos do processador ou ajudam a medir o desempenho. Os CSRs de identidade são:

- A CSR de ISA de máquina `misa` dá a largura do endereço do processador (32, 64 ou 128 bits) e identifica quais extensões de instruções estão incluídas (Figura 10.21).
- A CSR de ID do fornecedor `mvendorid` fornece o ID do fabricante JEDEC do provedor do núcleo (Figura 10.22).
- A CSR de ID da arquitetura da máquina `marchid` dá a microarquitetura de base. Combinando `mvendorid` com `marchid`, identifica de forma exclusiva a microarquitetura implementada (Figura 10.23).
- A CSR de ID de implementação da máquina `mimpid` fornece a versão da *implementação* da microarquitetura de base em `marchid` (Figura 10.23).



XLEN-1 XLEN-2	XLEN-3 26	25 0
MXL[1:0]	0	Extensions[25:0]
2	XLEN-28	26

Figura 10.21: O CSR de ISA de máquina `misa` relata o ISA suportado. O campo MXL (XLEN de máquina) codifica a largura da ISA de base de inteiros nativa: 1 é 32 bits, 2 é 64 e 3 é 128. O campo Extensões codifica a presença das extensões padrão, com um único bit por letra do alfabeto (o bit 0 codifica a presença da extensão "A", o bit 1 codifica a presença da extensão "B", até o bit 25 que codifica "Z").

- A CSR de ID do Hart `mhartid` dá o ID inteiro do hart sendo executado no momento (Figura 10.23).

Aqui estão os CSRs de medição:

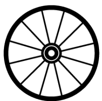
- A CSR de tempo de máquina `mtime` é um contador em tempo real de 64 bits (Figura 10.24).
- A CSR de comparação de tempo de máquina `mtimecmp` provoca uma interrupção quando `mtime` corresponde ou excede seu valor (Figura 10.24).
- A CSR de máquina de 32 bits e o supervisor de ativação (`mcounteren` e `scounteren`) controlam a disponibilidade dos CSRs do monitor de desempenho de hardware no próximo nível mais baixo de privilégios (Figura 10.25).
- As 31 CSRs de monitoramento de performance de hardware (`mcycle`, `minstret`, `mhpmcounter3`, ..., `mhpmcounter31`) contam ciclos de clock, instruções retiradas e, em seguida, até 29 eventos selecionados pelo programador usando as CSRs `mhpmevent3`, ..., `mhpmevent31` (Figura 10.26).

10.8 Considerações Finais

Estudos após estudos mostram que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais claras e produzidas com menos esforço. As diferenças entre o grande e o médio abordam uma ordem de grandeza

—Fred Brooks, Jr., 1986.

Brooks é ganhador do Prêmio Turing e arquiteto da família de computadores IBM System/360, que demonstrou a importância de diferenciar a arquitetura da implementação. As arquiteturas descendentes dessa arquitetura de 1964 ainda estão vendendo hoje.



Simplicidade

Facilidade de Programação

A modularidade das arquiteturas privilegiadas RISC-V atende às necessidades de uma variedade de sistemas. O modo de máquina minimalista suporta aplicações embarcadas baremetal a baixo custo. O modo de usuário adicional e a proteção de memória física juntos permitem multitarefa em ambientes embarcados mais sofisticados. Finalmente, o modo supervisor e a memória virtual baseada em páginas fornecem a flexibilidade necessária para hospedar sistemas operacionais modernos.

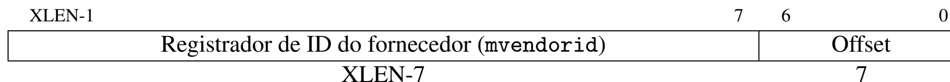


Figura 10.22: A CSR `mvendorid` fornece o ID do fabricante JEDEC do núcleo.

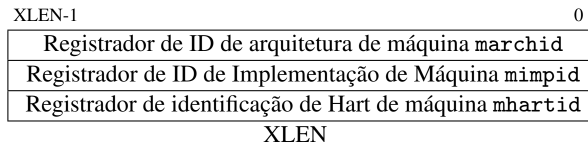


Figura 10.23: Os CSRs de identificação de máquina (`marchid` , `mimpid`, `mhartid`) identificam a microarquitetura e a implementação do processador e o número da thread hart atualmente em execução.

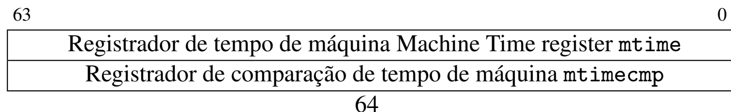


Figura 10.24: Os CSRs de tempo de máquina(`mtime` and `mtimecmp`) medem o tempo e causam uma interrupção quando `mtime` $\geq$ `mtimecmp`.

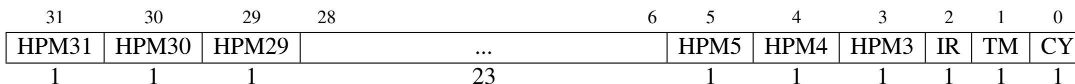


Figura 10.25: Os registradores de ativação reversa `mcounteren` e `scounteren` controlam a disponibilidade dos contadores de monitoramento de desempenho do hardware para o modo privilegiado mais baixo.

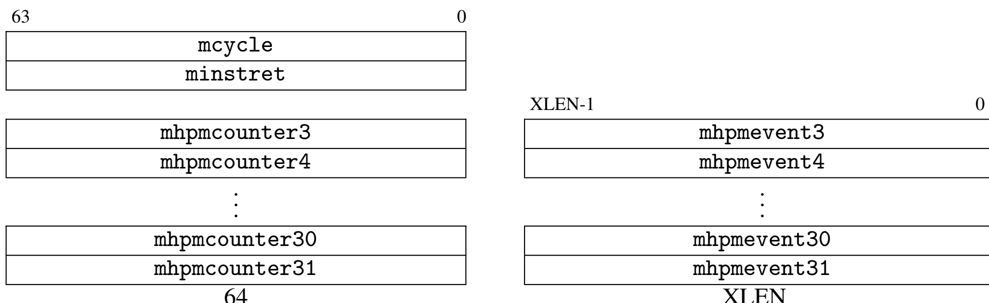


Figura 10.26: Os CSRs de monitoramento de performance do hardware (`mcycle` , `minstret`, `mhpmcounter3`, ..., `mhpmcounter31`) e os eventos que eles contam `mhpmevent3`, ..., `mhpmevent31`. Apenas para RV32, as leituras das CSRs `mcycle`, `minstret`, and `mhpmcountern` retornam os 32 bits baixos, enquanto leituras das CSRs `mcycleh`, `minstreth`, e `mhpmcounternh` retornam bits entre 63–32 do contador correspondente.

1. Sendo a igual a `satp`.ppn × PAGESIZE, e $i = \text{LEVELS} - 1$.
2. Sendo pte igual ao valor do PTE no endereço $a + va.vpn[i] \times$ PTESIZE.
3. Se $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, para e gera uma exceção de falha de página.
4. Caso contrário, o PTE é válido. Se $pte.r = 1$ or $pte.x = 1$, Vá para o passo 5. Caso contrário, este PTE é um ponteiro para o próximo nível da tabela de páginas. Deixa $i = i - 1$. Se $i < 0$, para e gera uma exceção de falha de página. Caso contrário, deixe $a = pte.ppn \times$ PAGESIZE e vá para o passo 2.
5. Uma folha PTE foi encontrada. Determina se o acesso à memória solicitado é permitido pelos bits $pte.r$, $pte.w$, $pte.x$, e $pte.u$, Considerando a modo de privilégio atual e o valor dos campos SUM e MXR do registrador `mstatus`. Caso contrário, para e gera uma exceção de falha de página.
6. Se $i > 0$ e $pa.ppn[i-1:0] \neq 0$, esta é uma super página desalinhada; para e gera uma exceção de falha de página.
7. Se $pte.a = 0$, ou se o acesso à memória for um store e $pte.d = 0$, então um dos dois:
   - Gera uma exceção de falha de página ou:
   - "Seta" $pte.a$ como 1 e, se o acesso à memória for um store, também "seta" $pte.d$ como 1.
8. A tradução é bem sucedida. O endereço físico traduzido é dado como segue:
   - *pa.pgoff* = *va.pgoff*.
   - Se $i > 0$, então esta é uma tradução de super página e $pa.ppn[i-1:0] = va.vpn[i-1:0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

Figura 10.27: O algoritmo completo para tradução de endereços virtuais para físicos. va é a entrada do endereço virtual e pa é a saída do endereço físico A constante PAGESIZE é 2^{12}. Para Sv32, LEVELS=2 e PTESIZE=4, enquanto para Sv39, LEVELS=3 e PTESIZE=8. A Seção 4.3.2 de [Waterman and Asanović 2017] é a base para este figura.

10.9 Para Saber Mais

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

11 Futuras Extensões Opcionais do RISC-V

Alan Perlis (1922–1990) foi o primeiro a receber o Prêmio Turing (1966), concedido por sua influência em linguagens de programação e compiladores. Em 1958 ele ajudou a projetar o ALGOL, que influenciou de certa forma toda linguagem de programação imperativa, incluindo C e Java.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

—Alan Perlis, 1982

A Fundação RISC-V desenvolverá pelo menos oito extensões opcionais.

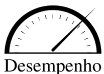
11.1 "B" Extensão Padrão para Manipulação de Bits

A extensão B oferece manipulação de bits, incluindo campos de inserção, extração e bit de teste; rotações; deslocamentos *funnel*; permutações de bits e bytes; contar zeros iniciais e finais; e contar os bits "setados".

11.2 "E" Extensão Padrão para Embarcados

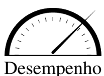
Para reduzir o custo de núcleos de baixo custo, esta extensão possui 16 registradores a menos. O RV32E é o motivo pelo qual os registradores salvos e temporários são divididos entre os registradores 0-15 e 16-31 (Figura 3.2).

11.3 "H" Extensão de Arquitetura Privilegiada para Hypervisor Support



A extensão H para a arquitetura privilegiada adiciona um novo modo *hypervisor* e um segundo nível de conversão de endereço baseado em página para melhorar a eficiência de execução de múltiplos sistemas operacionais na mesma máquina.

11.4 "J" Extensão Padrão para Linguagens Traduzidas Dinamicamente



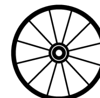
Muitas linguagens populares são geralmente implementadas através de tradução dinâmica, incluindo Java e Javascript. Essas linguagens podem se beneficiar do suporte ISA adicional para verificações dinâmicas e *garbage collection*. (J vem de *Just-In-Time compiler*.)

11.5 "L" Extensão Padrão para Ponto Flutuante Decimal

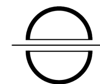
A extensão L provê suporte a aritmética de ponto flutuante decimal, conforme definido no padrão IEEE 754-2008. O problema com números binários é que eles não podem representar

algumas frações decimais comuns, como 0.1. A motivação para o RV32L é que a base de cálculo pode ser idêntica à base da entrada e da saída.

11.6 "N" Extensão Padrão para Interrupções a Nível de Usuário



A extensão N permite que interrupções e exceções que ocorrem em programas no nível do usuário transfiram o controle diretamente para um tratador de interrupções *traps* em nível do usuário sem chamar o ambiente de execução externo. Interrupções em nível de usuário são principalmente destinadas a suportar sistemas embarcados seguros com apenas modo M e modo U presentes (Capítulo 10). No entanto, eles também podem suportar o tratamento de traps em nível do usuário em sistemas que executam sistemas operacionais *Unix-Like*.



Quando usado em um ambiente Unix, o manuseio de sinal convencional provavelmente permaneceria, mas interrupções no nível do usuário poderiam ser usadas como blocos de construção para futuras extensões que gerem eventos no nível do usuário, como barreiras de coleta de lixo, overflow de inteiros e traps de ponto flutuante.

11.7 "P" Extensão Padrão para Instruções Packed-SIMD

A extensão P subdivide os registradores existentes na arquitetura para fornecer computação paralela de dados em tipos de dados menores. O padrão Packed-SIMD representa uma característica de projeto razoável ao reutilizar recursos existentes do caminho de dados. No entanto, se recursos adicionais significativos forem dedicados à execução paralela de dados, o Capítulo 8 mostra que projetos para arquiteturas vetoriais são uma melhor escolha, e arquitetos devem usar a extensão RVV.

11.8 "Q" Extensão Padrão para Ponto Flutuante de Precisão Quádrupla

A extensão Q adiciona instruções de ponto flutuante binário de precisão quádrupla de 128 bits compatíveis com o padrão aritmético IEEE 754-2008. Os registradores de ponto flutuante agora são estendidos para conter um valor de ponto flutuante de precisão simples, dupla ou quádrupla. A extensão de ponto flutuante binária de precisão quádrupla necessita do RV64IFD.

11.9 Considerações Finais

> *Simplify, simplify.*
>
> —Henry David Thoreau, eminente escritor do século XIX, 1854

Ter uma abordagem de comitê aberta e padronizada para expandir o RISC-V, significa que o feedback e o debate ocorrerão antes de que as instruções sejam finalizadas, e não depois, quando for tarde demais para mudar. No melhor caso, alguns integrantes da Fundação RISC-V implementarão a proposta antes que ela seja ratificada, o que se torna muito mais fácil com a ajuda de FPGAs. Propor extensões de instrução através dos comitês da fundação também acarretará em uma quantidade significativa de trabalho, o que deve manter a taxa de mudança lenta, ao contrário do que aconteceu com x86-32 (veja a Figura 1.2 na Página 4 do

Capítulo 1). Não esqueça de que tudo descrito neste capítulo será opcional, não importando quantas extensões forem adotadas.

Nossa esperança é que o RISC-V possa evoluir com demandas tecnológicas, mantendo sua reputação como uma ISA simples e eficiente. Se for bem-sucedido, o RISC-V será uma mudança significativa das ISAs incrementais do passado.

Elegância

A Listagem de Instruções RISC-V

Coco Chanel (1883-1971) Fundadora da marca de moda Chanel, sua busca pela simplicidade cara moldou a moda do século XX.

Simplicity is the keynote of all true elegance.

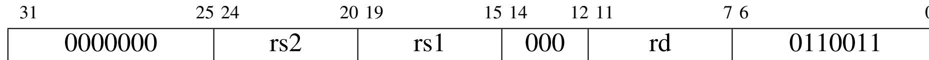
—Coco Chanel, 1923

Este apêndice lista todas as instruções para o RV32 / 64I, todas as extensões cobertas neste livro, exceto RVV (RVM, RVA, RVF, RVD e RVC), e todas as pseudo-instruções. Cada entrada tem o nome da instrução, operandos, uma definição de nível de transferência de registrador, tipo de formato de instrução, descrição em português, versões compactadas (se houver) e uma figura mostrando o layout real com os opcodes. Achamos que você tem tudo o que precisa para entender todas as instruções desses resumos compactos. No entanto, se você quiser ainda mais detalhes, consulte as especificações oficiais do RISC-V[Waterman and Asanović 2017].

Para ajudar os leitores a encontrar a instrução desejada neste apêndice, o cabeçalho das páginas da esquerda (par) contém a primeira instrução do topo daquela página e o cabeçalho à direita (ímpar) contém a última instrução da parte inferior da página. O formato é semelhante aos cabeçalhos dos dicionários, o que ajuda a procurar a página em que sua palavra está. Por exemplo, o cabeçalho da próxima página *par* mostra **AMOADD.W**, a primeira instrução na página e o cabeçalho da página ímpar seguinte mostra **AMOMINU.D**, a última instrução nessa página. Estas são as duas páginas onde você encontrará qualquer uma dessas 10 instruções: `amoadd.w`, `amoand.d`, `amoand.w`, `amomax.d`, `amomax.w`, `amomaxu.d`, `amomaxu.w`, `amomin.d`, `amomin.w` e `amominu.d`.

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

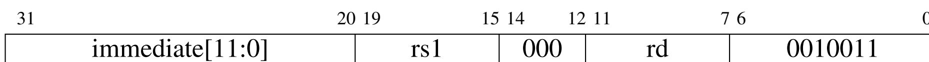
Add. R-type, RV32I and RV64I.
Adiciona o registrador x[*rs2*] ao registrador x[*rs1*] e grava o resultado em x[*rd*]. O overflow aritmético é ignorado.
Formas comprimidas: **c.add** rd, rs2; **c.mv** rd, rs2



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)`

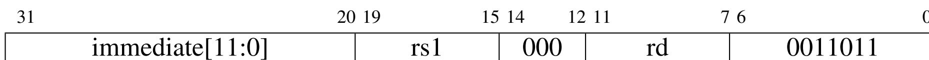
Add Immediate. I-type, RV32I and RV64I.
Adiciona o *valor imediato* de sinal estendido ao registrador x[*rs1*] e escreve o resultado em x[*rd*]. O overflow aritmético é ignorado.
Formas comprimidas: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm



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])`

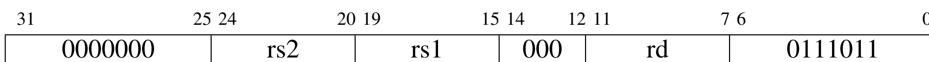
Add Word Immediate. I-type, RV64I only.
Adiciona o *valor imediato* com extensão de sinal para x[*rs1*], trunca o resultado para 32 bits e grava o resultado do sinal estendido em x[*rd*]. O overflow aritmético é ignorado.
Formas comprimidas: **c.addiw** rd, imm



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])`

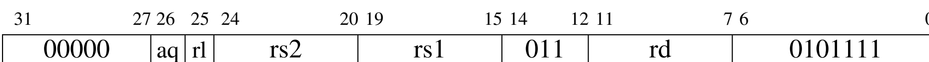
Add Word. R-type, RV64I only.
Adiciona o registrador x[*rs2*] ao registrador x[*rs1*], trunca o resultado para 32 bits e grava o resultado de sinal estendido em x[*rd*]. O overflow aritmético é ignorado.
emphForma comprimida: **c.addw** rd, rs2



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])`

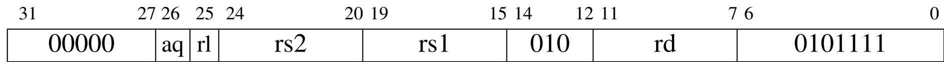
Atomic Memory Operation: Add Doubleword. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" a palavra dupla de memória para *t* + x[*rs2*]. "Seta" x[*rd*] para *t*.



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])`

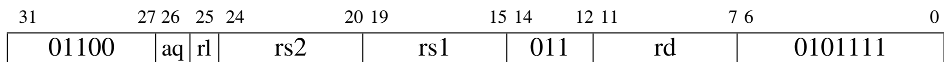
Atomic Memory Operation: Add Word. R-type, RV32A and RV64A.
Atomicamente, deixa *t* ser o valor da palavra de memória no endereço x[*rs1*], e então "seta" essa palavra de memória para *t* + x[*rs2*]. "Seta" x[*rd*] para a extensão de sinal de *t*.



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])`

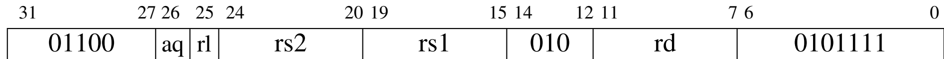
Atomic Memory Operation: AND Doubleword. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" a palavra dupla de memória para o bitwise AND de *t* e x[*rs2*]. "Seta" x[*rd*] para *t*.



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])`

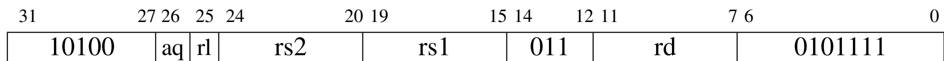
Atomic Memory Operation: AND Word. R-type, RV32A and RV64A.
Atomicamente, deixa *t* ser o valor da palavra de memória no endereço x[*rs1*], e então "seta" essa palavra de memória para o bit a bit AND de *t* e x[*rs2*]. "Seta" x[*rd*] para a extensão de sinal de *t*.



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])`

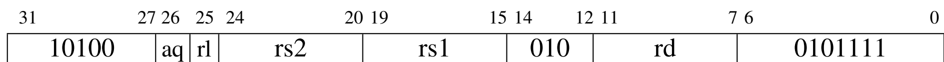
Atomic Memory Operation: Maximum Doubleword. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" essa palavra dupla de memória para o maior entre *t* e x[*rs2*], utilizando uma comparação de complemento de dois. "Seta" x[*rd*] para *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])`

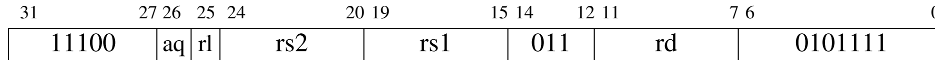
Atomic Memory Operation: Maximum Word. R-type, RV32A and RV64A.
Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], depois "seta" essa palavra de memória para o maior entre *t* e x[*rs2*], utilizando uma comparação de complemento de dois. "Seta" x[*rd*] para a extensão de sinal de *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])`

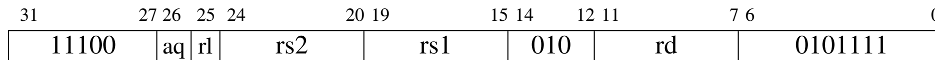
Atomic Memory Operation: Maximum Doubleword, Unsigned. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" essa palavra dupla para o maior entre *t* e x[*rs2*], utilizando uma comparação sem sinal. "Seta" x[*rd*] para *t*.



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])`

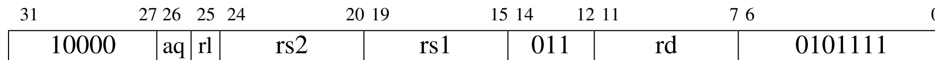
Atomic Memory Operation: Maximum Word, Unsigned. R-type, RV32A and RV64A.
Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], depois "seta" essa palavra de memória para o maior entre *t* e x [*rs2*], utilizando uma comparação sem sinal. "Seta" x[*rd*] para a extensão de sinal de *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])`

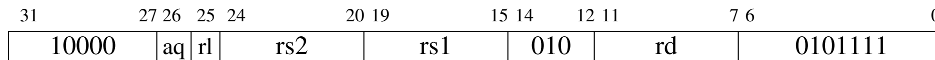
Atomic Memory Operation: Minimum Doubleword. R-type, RV64A only.
Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the smaller of *t* and x[*rs2*], using a two's complement comparison. Set x[*rd*] to *t*.



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])`

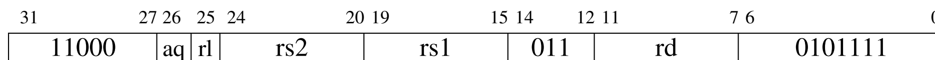
Atomic Memory Operation: Minimum Word. R-type, RV32A and RV64A.
Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], depois "seta" essa palavra de memória para o menor entre *t* e x[*rs2*], utilizando uma comparação de complemento de dois. "Seta" x[*rd*] para a extensão de sinal de *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])`

Atomic Memory Operation: Minimum Doubleword, Unsigned. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*] e, em seguida, "seta" essa palavra dupla de memória para o menor entre *t* e x[*rs2*], utilizando uma comparação sem sinal. "Seta" x[*rd*] para *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])`

Atomic Memory Operation: Minimum Word, Unsigned. R-type, RV32A and RV64A.

Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], depois "seta" essa palavra de memória para o menor entre *t* e x[*rs2*], utilizando uma comparação sem sinal. "Seta" x[*rd*] para a extensão de sinal de *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])`

Atomic Memory Operation: OR Doubleword. R-type, RV64A only.

Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" a palavra de memória dupla para o OR bitwise de *t* e x[*rs2*]. "Seta" x[*rd*] para *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])`

Atomic Memory Operation: OR Word. R-type, RV32A and RV64A.

Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], e então "seta" essa palavra de memória para o OR bitwise de *t* e x[*rs2*]. "Seta" x[*rd*] para a extensão de sinal de *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])`

Atomic Memory Operation: Swap Doubleword. R-type, RV64A only.

Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" essa palavra dupla para x[*rs2*]. "Seta" x[*rd*] para *t*.

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

---

amoswap.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])`

Atomic Memory Operation: Swap Word. R-type, RV32A and RV64A.

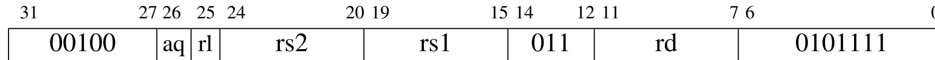
Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[emph rs1], e depois "seta" essa palavra de memória para x[*rs2*]. "Seta" x[*rd*] para a extensão de sinal de *t*.

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])`

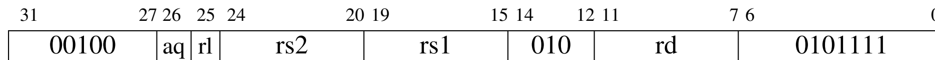
Atomic Memory Operation: XOR Doubleword. R-type, RV64A only.
Atomicamente, deixa que *t* seja o valor da palavra dupla da memória no endereço x[*rs1*], e então "seta" a palavra dupla de memória para o XOR bitwise de *t* e x[*rs2*]. "Seta" x[*rd*] para *t*.



---

amoxor.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] ^ x[rs2])`

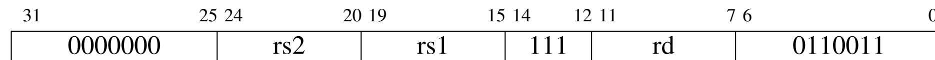
Atomic Memory Operation: XOR Word. R-type, RV32A and RV64A.
Atomicamente, deixa que *t* seja o valor da palavra de memória no endereço x[*rs1*], e então "seta" essa memória palavra para o XOR bitwise de *t* e x[*rs2*]. "Seta" x[*rd*] para a extensão de sinal de *t*.



---

and rd, rs1, rs2 `x[rd] = x[rs1] & x[rs2]`

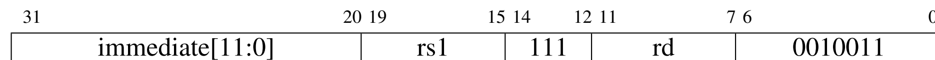
AND. R-type, RV32I and RV64I.
Calcula o AND bitwise dos registradores x[*rs1*] e x[*rs2*] e escreve o resultado em x[*rd*].
Forma comprimida: **c.and** rd, rs2



---

andi rd, rs1, immediate `x[rd] = x[rs1] & sext(immediate)`

AND Immediate. I-type, RV32I and RV64I.
Calcula o AND bitwise do *valor imediato* com sinal estendido e do registrador x[*rs1*] e escreve o resultado em x[*rd*].
Forma comprimida: **c.andi** rd, imm



---

auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. U-type, RV32I and RV64I.
Adiciona o *valor imediato* com sinal estendido de 20 bits, deslocada para a esquerda por 12 bits, ao *pc*, e escreve o resultado em x[*rd*].

---

beq rs1, rs2, offset

`if (rs1 == rs2) pc += sext(offset)`

Branch if Equal. B-type, RV32I and RV64I.
Se o registrador x[*rs1*] for igual ao registrador x[*rs2*], "seta" o *pc* para o *pc* atual mais o *offset* com sinal estendido.
Formas comprimidas: **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)`

Branch if Equal to Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **beq** rs1, x0, offset.

---

bge rs1, rs2, offset

if (rs1 $\geq_s$ rs2) pc += sext(offset)

Branch if Greater Than or Equal. B-type, RV32I and RV64I.
Se o registrador x[*rs1*] é pelo menos igual a x[*rs2*], tratando os valores como números em complemento de dois, "seta" o *pc* para o *pc* atual mais o *offset* com sinal estendido.

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 $\geq_u$ rs2) pc += sext(offset)

Branch if Greater Than or Equal, Unsigned. B-type, RV32I and RV64I.
If register x[*rs1*] is at least x[*rs2*], treating the values as unsigned numbers, set the *pc* to the current *pc* plus the sign-extended *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 $\geq_s$ 0) pc += sext(offset)

Branch if Greater Than or Equal to Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **bge** rs1, x0, offset.

---

bgt rs1, rs2, offset

if (rs1 $>_s$ rs2) pc += sext(offset)

Branch if Greater Than. Pseudoinstruction, RV32I and RV64I.
Expande para **blt** rs2, rs1, offset.

---

bgtu rs1, rs2, offset

if (rs1 $>_u$ rs2) pc += sext(offset)

Branch if Greater Than, Unsigned. Pseudoinstruction, RV32I and RV64I.
Expande para **bltu** rs2, rs1, offset.

---

bgtz rs2, offset
if (rs2 $>_s$ 0) pc += sext(offset)

Branch if Greater Than Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **blt** x0, rs2, offset.

---

ble rs1, rs2, offset
if (rs1 $\leq_s$ rs2) pc += sext(offset)

Branch if Less Than or Equal. Pseudoinstruction, RV32I and RV64I.
Expande para **bge** rs2, rs1, offset.

---

bleu rs1, rs2, offset
if (rs1 $\leq_u$ rs2) pc += sext(offset)

Branch if Less Than or Equal, Unsigned. Pseudoinstruction, RV32I and RV64I.
Expande para **bgeu** rs2, rs1, offset.

---

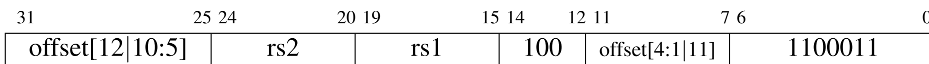
blez rs2, offset
if (rs2 $\leq_s$ 0) pc += sext(offset)

Branch if Less Than or Equal to Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **bge** x0, rs2, offset.

---

blt rs1, rs2, offset
if (rs1 $<_s$ rs2) pc += sext(offset)

Branch if Less Than. B-type, RV32I and RV64I.
Se o registrador x[*rs1*] é menor que o registrador x[*rs2*], tratando os valores como números em complemento de dois, "seta" o *pc* para o *pc* atual mais *offset* com sinal estendido.



31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[12|10:5]	rs2	rs1	100	offset[4:1|11]	1100011

---

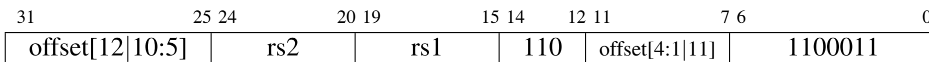
bltz rs1, offset
if (rs1 $<_s$ 0) pc += sext(offset)

Branch if Less Than Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **blt** rs1, x0, offset.

---

bltu rs1, rs2, offset
if (rs1 $<_u$ rs2) pc += sext(offset)

Branch if Less Than, Unsigned. B-type, RV32I and RV64I.
Se o registrador x[*rs1*] é menor que o registrador x[*rs2*], tratando os valores como números sem sinal, "seta" o *pc* para o *pc* atual mais o *offset* de sinal estendido.



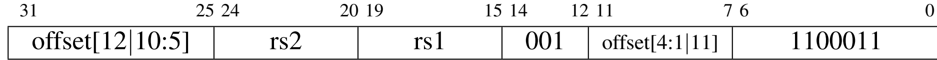
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[12|10:5]	rs2	rs1	110	offset[4:1|11]	1100011

---

bne rs1, rs2, offset

`if (rs1 ≠ rs2) pc += sext(offset)`

Branch if Not Equal. B-type, RV32I and RV64I.
Se o registrador x[*rs1*] não é igual ao registrador x[*rs2*], "seta" o *pc* para o *pc* atual mais o *offset* com sinal estendido.
Forma comprimida: **c.bnez** rs1, offset



---

bnez rs1, offset

`if (rs1 ≠ 0) pc += sext(offset)`

Branch if Not Equal to Zero. Pseudoinstruction, RV32I and RV64I.
Expande para **bne** rs1, x0, offset.

---

c.add rd, rs2

`x[rd] = x[rd] + x[rs2]`

Add. RV32IC e RV64IC.
Expande para **add** rd, rd, rs2. Inválida quando rd=x0 ou rs2=x0.

---

c.addi rd, imm

`x[rd] = x[rd] + sext(imm)`

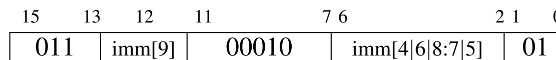
Add Immediate. RV32IC e RV64IC.
Expande para **addi** rd, rd, imm.

---

c.addi16sp imm

`x[2] = x[2] + sext(imm)`

Add Immediate, Scaled by 16, to Stack Pointer. RV32IC and RV64IC.
Expande para **addi** x2, x2, imm. Inválida quando imm=0.

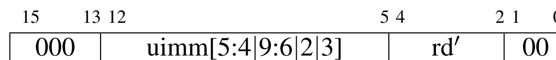


---

c.addi4spn rd′, uimm

`x[8+rd′] = x[2] + uimm`

Add Immediate, Scaled by 4, to Stack Pointer, Nondestructive. RV32IC e RV64IC.
Expande para **addi** rd, x2, uimm, onde rd=8+rd′. Inválida quando uimm=0.



---

c.addiw rd, imm
`x[rd] = sext((x[rd] + sext(imm))[31:0])`

Add Word Immediate. Somente RV64IC.
Expande para **addiw** rd, rd, imm. Inválida quando rd=x0.

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′]`

AND. RV32IC e RV64IC.
Expande para **and** rd, rd, rs2, onde rd=8+rd′ e 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])`

Add Word. Somente RV64IC.
Expande para **addw** rd, rd, rs2, onde rd=8+rd′ e 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)`

AND Immediate. RV32IC e RV64IC.
Expande para **andi** rd, rd, imm, onde 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)`

Branch if Equal to Zero. RV32IC e RV64IC.
Expande para **beq** rs1, x0, offset, onde 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)`

Branch if Not Equal to Zero. RV32IC e RV64IC.
Expande para **bne** rs1, x0, offset, onde 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)`

Environment Breakpoint. RV32IC e RV64IC.
Expande para **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]`

Floating-point Load Doubleword. RV32DC e RV64DC.
Expande para **fld** rd, uimm(rs1), onde rd=8+rd′ e 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]`

Floating-point Load Doubleword, Stack-Pointer Relative. RV32DC e RV64DC.
Expande para **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]`

Floating-point Load Word. Somente RV32FC.
Expande para **flw** rd, uimm(rs1), onde rd=8+rd′ e 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]`

Floating-point Load Word, Stack-Pointer Relative. Somente RV32FC.
Expande para **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′]`

Floating-point Store Doubleword. RV32DC e RV64DC.
Expande para **fsd** rs2, uimm(rs1), onde rs2=8+rs2′ e 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]`

Floating-point Store Doubleword, Stack-Pointer Relative. RV32DC e RV64DC.
Expande para **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′]`

Floating-point Store Word. Somente RV32FC.
Expande para **fsw** rs2, uimm(rs1), onde rs2=8+rs2′ e 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]`

Floating-point Store Word, Stack-Pointer Relative. Somente RV32FC.
Expande para **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)`

Jump. RV32IC e RV64IC.
Expande para **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)`

Jump and Link. Somente RV32IC.
Expande para **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`

Jump and Link Register. RV32IC e RV64IC.
Expande para **jalr** x1, 0(rs1). Inválida quando rs1=x0.

15–13	12	11–7	6–2	1–0
100	1	rs1	00000	10

---

c.jr rs1

`pc = x[rs1]`

Jump Register. RV32IC e RV64IC.
Expande para **jalr** x0, 0(rs1). Inválida quando rs1=x0.

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]`

Load Doubleword. Somente RV64IC.
Expande para **ld** rd, uimm(rs1), onde rd=8+rd′ e 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]`

Load Doubleword, Stack-Pointer Relative. Somente RV64IC.
Expande para **ld** rd, uimm(x2). Inválida quando rd=x0.

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)`

Load Immediate. RV32IC e RV64IC.
Expande para **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)`

Load Upper Immediate. RV32IC e RV64IC.
Expande para **lui** rd, imm. Inválida quando rd=x2 or imm=0.

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])`

Load Word. RV32IC e RV64IC.
Expande para **lw** rd, uimm(rs1), onde rd=8+rd′ e 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])`

Load Word, Stack-Pointer Relative. RV32IC e RV64IC.
Expande para **lw** rd, uimm(x2). Inválida quando rd=x0.

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]`

Move. RV32IC e RV64IC.
Expande para **add** rd, x0, rs2. Inválida quando rs2=x0.

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′]`

OR. RV32IC e RV64IC.
Expande para **or** rd, rd, rs2, onde rd=8+rd′ e 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′]`

Store Doubleword. Somente RV64IC.
Expande para **sd** rs2, uimm(rs1), onde rs2=8+rs2′ e 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]`

Store Doubleword, Stack-Pointer Relative. Somente RV64IC.
Expande para **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`

Shift Left Logical Immediate. RV32IC e RV64IC.
Expande para **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`

Shift Right Arithmetic Immediate. RV32IC e RV64IC.
Expande para **srai** rd, rd, uimm, onde 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`

Shift Right Logical Immediate. RV32IC e RV64IC.
Expande para **srli** rd, rd, uimm, onde 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′]`

Subtract. RV32IC e RV64IC.
Expande para **sub** rd, rd, rs2, onde rd=8+rd′ e 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])`

Subtract Word. Somente RV64IC.
Expande para **subw** rd, rd, rs2, onde rd=8+rd′ e 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′]`

Store Word. RV32IC e RV64IC.
Expande para **sw** rs2, uimm(rs1), onde rs2=8+rs2′ e 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]`

Store Word, Stack-Pointer Relative. RV32IC e RV64IC.
Expande para **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′]`

Exclusive-OR. RV32IC e RV64IC.
Expande para **xor** rd, rd, rs2, onde rd=8+rd′ e 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`

Call. Pseudoinstruction, RV32I e RV64I.
Escreve o endereço da próxima instrução (*pc*+8) a x[*rd*], e então "seta" o *pc* para *symbol*. Expande para **auipc** rd, offsetHi e depois **jalr** rd, offsetLo(rd). Se *rd* é omitido, x1 está implícito.

---

csrr rd, csr `x[rd] = CSRs[csr]`

Control and Status Register Read. Pseudoinstruction, RV32I e RV64I.
Copia o conteúdo do registrador de controle e status *csr* a x[*rd*]. Expande para **csrrs** rd, csr, x0.

---

csrc csr, rs1 `CSRs[csr] &= ∼x[rs1]`

Control and Status Register Clear. Pseudoinstruction, RV32I e RV64I.
Para cada bit "setado" em x[*rs1*], limpa o bit correspondente no registrador de controle e status *csr*. Expande para **csrrc** x0, csr, rs1.

---

csrci csr, zimm[4:0] `CSRs[csr] &= ∼zimm`

Control and Status Register Clear Immediate. Pseudoinstruction, RV32I e RV64I.
Para cada bit "setado" no valor imediato de cinco bits e zero bits de extensão, limpa o bit correspondente no registrador de controle e status *csr*. Expande para **csrrci** x0, csr, zimm.

---

csrrc rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = t &∼ x[rs1]; x[rd] = t`

Control and Status Register Read and Clear. I-type, RV32I and RV64I.
Sendo *t* o valor do registrador de controle e status *csr*, escreve o AND bitwise de *t* e o complemento de um do registrador x[*rs1*] até *csr*, e então escreve *t* para 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`

Control and Status Register Read and Clear Immediate. I-type, RV32I and RV64I.
Deixa que *t* seja o valor do registrador de controle e status *csr*. Escreve o AND bitwise de *t* e o complemento de um do imediato de cinco bits e com zero bits de extensão *zimm* para o *csr*, e então escreve de *t* a x[*rd*]. (Bits igual ou acima de 5 e acima no *csr* não são modificados.)

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`

Control and Status Register Read and Set. I-type, RV32I and RV64I.
Sendo *t* o valor do registrador de controle e status *csr*. Escreva o OR bit a bit de *t* e x [*rs1*] para o *csr*, em seguida, escreva *t* para 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`

Control and Status Register Read and Set Immediate. I-type, RV32I and RV64I.
Sendo *t* o valor do registrador de controle e status *csr*. Escreva o OR bit a bit de *t* e o comprimento estendido de cinco bits do imediato *zimm* para *csr*, então escreva *t* para x[*rd*]. (Os bits acima dos 5 primeiros no *csr* não são modificados.)

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`

Control and Status Register Read and Write. I-type, RV32I and RV64I.
Sendo *t* o valor do registrador de controle e status *csr*. Copie x[*rs1*] para *csr*, então escreva *t* para 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`

Control and Status Register Read and Write Immediate. I-type, RV32I and RV64I.
Copia o valor do registrador de controle e status *csr* para x[*rd*] em seguida, escreve os 5 bits do imediato com extensão de zero para *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]`

Control and Status Register Set. Pseudoinstruction, RV32I e RV64I.
Para cada bit "setado" em x [*rs1*], defina o bit correspondente no controle e registrador de status *csr*. Expande para **csrrs** x0, csr, rs1.

---

csrsi csr,zimm[4:0] `CSRs[csr] |= zimm`

Control and Status Register Set Immediate. Pseudoinstruction, RV32I e RV64I.
Para cada bit "setado" nos 5 bits imediato com extensão de zero, "seta" o bit correspondente no controle e no registrador de status *csr*. Expande para **csrrsi** x0, csr, zimm.

---

csrw csr,rs1 `CSRs[csr] = x[rs1]`

Control and Status Register Write. Pseudoinstruction, RV32I e RV64I.
Copia x[*rs1*] para registrador de controle e status *csr*. Expande para **csrrw** x0, csr, rs1.

---

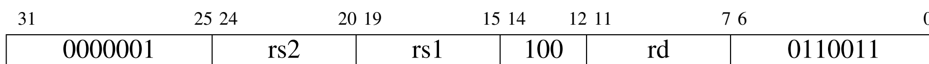
csrwi csr,zimm[4:0] `CSRs[csr] = zimm`

Control and Status Register Write Immediate. Pseudoinstruction, RV32I e RV64I.
Copia o imediato estendido de zero de 5 bits para o registrador de controle e status *csr*. Expande para **csrrwi** x0, csr, zimm.

---

div rd, rs1, rs2 `x[rd] = x[rs1]` $\div_s$ `x[rs2]`

Divide. R-type, RV32M and RV64M.
Divide x[*rs1*] por x[*rs2*], arredondando para zero, tratando os valores como números de complemento de dois, e escreve o quociente em x[*rd*].

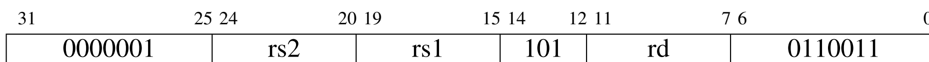


31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0110011

---

divu rd, rs1, rs2 `x[rd] = x[rs1]` $\div_u$ `x[rs2]`

Divide, Unsigned. R-type, RV32M and RV64M.
Divide x[*rs1*] por x[*rs2*], arredondando para zero, tratando os valores como números sem sinal, e escreve o quociente em x [*rd*].

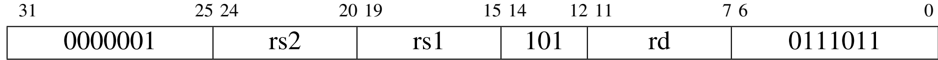


31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0110011

---

divuw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] ÷u x[rs2][31:0])`

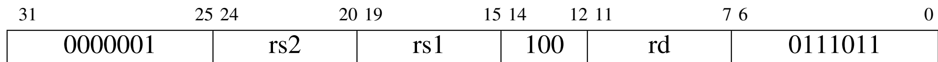
Divide Word, Unsigned. R-type, RV64M only.
Divide os 32 primeiros bits de x[*rs1*] pelos 32 primeiros bits de x[*rs2*], arredondando para zero, tratando os valores como números sem sinal e escreve o quociente de 32 bits com extensão de sinal para x[*rd*].



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] ÷s x[rs2][31:0])`

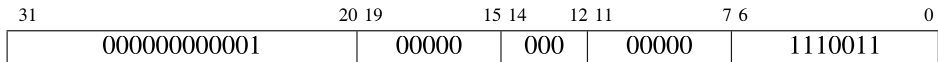
Divide Word. R-type, RV64M only.
Divide os 32 primeiros bits de x[*rs1*] pelos 32 primeiros bits de x[*rs2*], arredondando para zero, tratando os valores como números de complemento de dois e grava o quociente estendido de 32 bits para x[*rd*].



31–25	24–20	19–15	14–12	11–7	6–0
0000001	rs2	rs1	100	rd	0111011

ebreak `RaiseException(Breakpoint)`

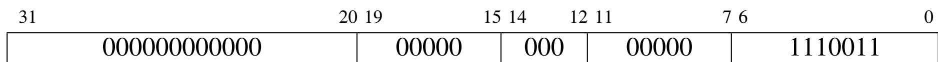
Environment Breakpoint. I-type, RV32I and RV64I.
Faz uma requisição do debugger levantando uma exceção de Breakpoint.



31–20	19–15	14–12	11–7	6–0
000000000001	00000	000	00000	1110011

ecall `RaiseException(EnvironmentCall)`

Environment Call. I-type, RV32I and RV64I.
Faz uma solicitação do ambiente de execução, levantando uma exceção de chamada de ambiente.



31–20	19–15	14–12	11–7	6–0
000000000000	00000	000	00000	1110011

fabs.d rd,rs1 `f[rd] = |f[rs1]|`

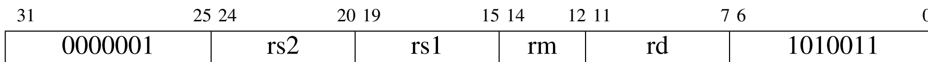
Floating-point Absolute Value, Double-Precision. Pseudoinstruction, RV32D e RV64D.
Escreve o valor absoluto do número de ponto flutuante de precisão dupla em f[*rs1*] para f[*rd*]. Expande para **fsgnjx.d** rd, rs1, rs1.

fabs.s rd,rs1 `f[rd] = |f[rs1]|`

Floating-point Absolute Value, Single-Precision. Pseudoinstruction, RV32F e RV64F.
Escreve o valor absoluto do número de ponto flutuante de precisão simples em f[*rs1*] para f[*rd*]. Expande para **fsgnjx.s** rd, rs1, rs1.

fadd.d rd, rs1, rs2 `f[rd] = f[rs1] + f[rs2]`

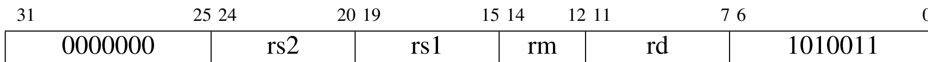
Floating-point Add, Double-Precision. R-type, RV32D and RV64D.
Adiciona os números de ponto flutuante de precisão dupla nos registradores f[*rs1*] e f[*rs2*] e grava a soma arredondada de precisão dupla em f[*rd*].



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]`

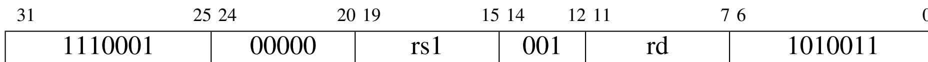
Floating-point Add, Single-Precision. R-type, RV32F and RV64F.
Adiciona os números de ponto flutuante de precisão simples em registra f[*rs1*] e f [*rs2*] e grava a soma arredondada de precisão simples para f[*rd*].



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] = classifica` $_d$`(f[rs1])`

Floating-point Classify, Double-Precision. R-type, RV32D and RV64D.
Escreve em x[*rd*] uma máscara indicando a classe do número de ponto flutuante de precisão dupla em f[*rs1*]. Veja a descrição de **fclass.s** para a interpretação do valor escrito em x[*rd*].

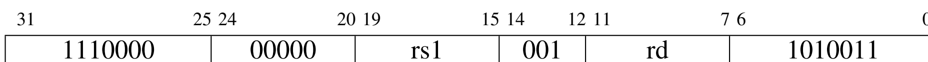


31–25	24–20	19–15	14–12	11–7	6–0
1110001	00000	rs1	001	rd	1010011

fclass.s rd, rs1, rs2 `x[rd] = classifica` $_s$`(f[rs1])`

Floating-point Classify, Single-Precision. R-type, RV32F and RV64F.
Escreve em x [*rd*] uma máscara indicando a classe do número de ponto flutuante de precisão simples em f[*rs1*]. Somente um bit em x [*rd*] é “setado”, conforme a tabela a seguir:

x[*rd*] bit	Significado
0	f[*rs1*] é $-\infty$.
1	f[*rs1*] é um número normal negativo.
2	f[*rs1*] é um número subnormal negativo.
3	f[*rs1*] é -0.
4	f[*rs1*] é $+0$.
5	f[*rs1*] é um número subnormal positivo.
6	f[*rs1*] é um número normal positivo.
7	f[*rs1*] é $+\infty$.
8	f[*rs1*] é uma sinalização NaN.
9	f[*rs1*] é um NaN silencioso.

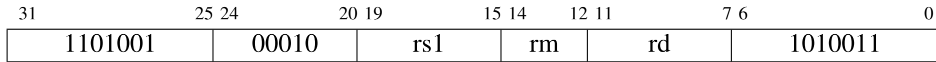


31–25	24–20	19–15	14–12	11–7	6–0
1110000	00000	rs1	001	rd	1010011

fcvt.dl rd, rs1, rs2

`f[rd] = f64` $_{s64}$`(x[rs1])`

Floating-point Convert to Double from Long. R-type, RV64D only.
Converte o inteiro de 64 bits complemento de dois em x[*rs1*] para um número de ponto flutuante de precisão dupla e grava-o em f[*rd*].

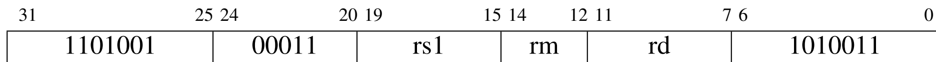


---

fcvt.d.lu rd, rs1, rs2

`f[rd] = f64` $_{u64}$`(x[rs1])`

Floating-point Convert to Double from Unsigned Long. R-type, RV64D only.
Converte o inteiro sem sinal de 64 bits em x[*rs1*] em um número de ponto flutuante de precisão dupla e grava-o em f[*rd*].

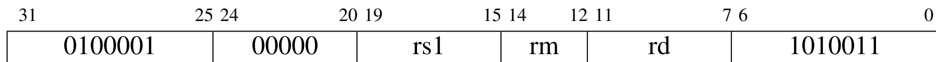


---

fcvt.d.s rd, rs1, rs2

`f[rd] = f64` $_{f32}$`(f[rs1])`

Floating-point Convert to Double from Single. R-type, RV32D and RV64D.
Converte o número de ponto flutuante de precisão simples em f[*rs1*] para um número de ponto flutuante de precisão dupla e grava-o em f[*rd*].

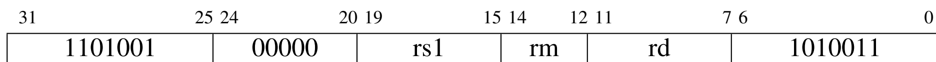


---

fcvt.d.w rd, rs1, rs2

`f[rd] = f64` $_{s32}$`(x[rs1])`

Floating-point Convert to Double from Word. R-type, RV32D and RV64D.
Converte o inteiro de 32 bits complemento de dois em x[*rs1*] para um número de ponto flutuante de precisão dupla e grava-o em f[*rd*].

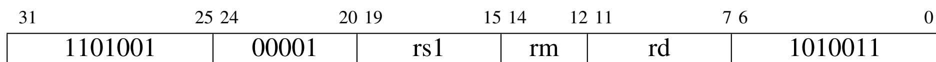


---

fcvt.d.wu rd, rs1, rs2

`f[rd] = f64` $_{u32}$`(x[rs1])`

Floating-point Convert to Double from Unsigned Word. R-type, RV32D and RV64D.
Converte o inteiro sem sinal de 32 bits em x [*rs1*] em um número de ponto flutuante de precisão dupla e grava-o em f[*rd*].

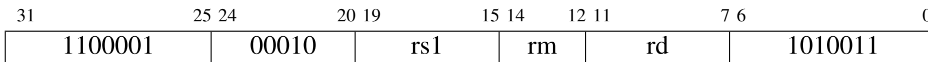


---

fcvt.l.d rd, rs1, rs2

x[rd] = s64 $_{f64}$(f[rs1])

Floating-point Convert to Long from Double. R-type, RV64D only.
Converte o número de ponto flutuante precisão dupla no registrador f[*rs1*] para um inteiro de 64 bits complemento de dois e grava-o em x[*rd*].



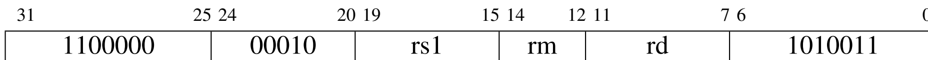
31 25	24 20	19 15	14 12	11 7	6 0
1100001	00010	rs1	rm	rd	1010011

---

fcvt.l.s rd, rs1, rs2

x[rd] = s64 $_{f32}$(f[rs1])

Floating-point Convert to Long from Single. R-type, RV64F only.
Converte o número de ponto flutuante precisão simples no registrador f[*rs1*] para um inteiro de 64 bits complemento de dois e grava-o em x[*rd*].



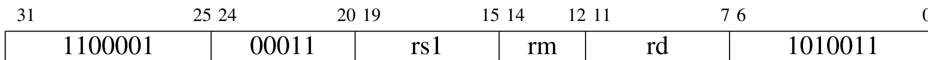
31 25	24 20	19 15	14 12	11 7	6 0
1100000	00010	rs1	rm	rd	1010011

---

fcvt.lu.d rd, rs1, rs2

x[rd] = u64 $_{f64}$(f[rs1])

Floating-point Convert to Unsigned Long from Double. R-type, RV64D only.
Converte o número de ponto flutuante de precisão dupla no registrador f[*rs1*] em um inteiro sem sinal de 64 bits e grava-o em x[*rd*].



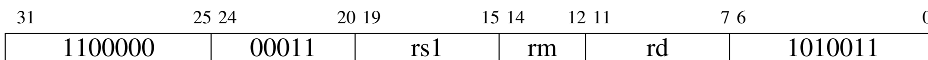
31 25	24 20	19 15	14 12	11 7	6 0
1100001	00011	rs1	rm	rd	1010011

---

fcvt.lu.s rd, rs1, rs2

x[rd] = u64 $_{f32}$(f[rs1])

Floating-point Convert to Unsigned Long from Single. R-type, RV64F only.
Converte o número de ponto flutuante de precisão simples no registrador f[*rs1*] em um inteiro sem sinal de 64 bits e grava-o em x[*rd*].



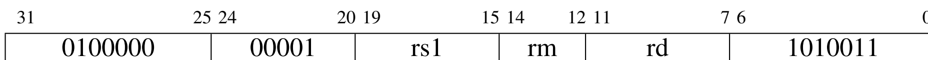
31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	rs1	rm	rd	1010011

---

fcvt.s.d rd, rs1, rs2

f[rd] = f32 $_{f64}$(f[rs1])

Floating-point Convert to Single from Double. R-type, RV32D and RV64D.
Converte o número de ponto flutuante precisão dupla em f[*rs1*] para um número de ponto flutuante de precisão simples e grava-o em f[*rd*].



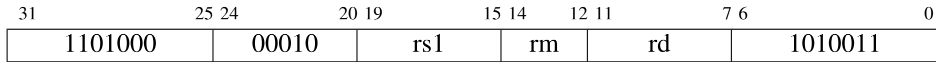
31 25	24 20	19 15	14 12	11 7	6 0
0100000	00001	rs1	rm	rd	1010011

---

fcvt.s.l rd, rs1, rs2

f[rd] = f32$_{s64}$(x[rs1])

Floating-point Convert to Single from Long. R-type, RV64F only.
Converte o inteiro de 64 bits complemento de dois em x[*rs1*] para um número ponto flutuante de precisão simples e grava-o em f[*rd*].



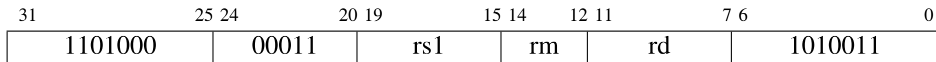
31–25	24–20	19–15	14–12	11–7	6–0
1101000	00010	rs1	rm	rd	1010011

---

fcvt.s.lu rd, rs1, rs2

f[rd] = f32$_{u64}$(x[rs1])

Floating-point Convert to Single from Unsigned Long. R-type, RV64F only.
Converte o inteiro sem sinal de 64 bits em x[*rs1*] em um número de ponto flutuante de precisão simples e grava-o em f[*rd*].



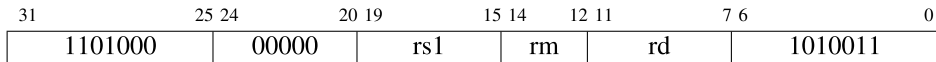
31–25	24–20	19–15	14–12	11–7	6–0
1101000	00011	rs1	rm	rd	1010011

---

fcvt.s.w rd, rs1, rs2

f[rd] = f32$_{s32}$(x[rs1])

Floating-point Convert to Single from Word. R-type, RV32F and RV64F.
Converte o inteiro de 32 bits complemento de dois em x[*rs1*] para um número de ponto flutuante de precisão simples e grava-o em f[*rd*].



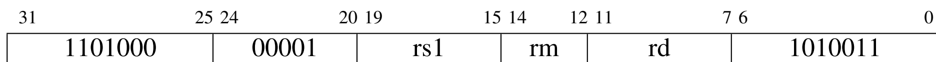
31–25	24–20	19–15	14–12	11–7	6–0
1101000	00000	rs1	rm	rd	1010011

---

fcvt.s.wu rd, rs1, rs2

f[rd] = f32$_{u32}$(x[rs1])

Floating-point Convert to Single from Unsigned Word. R-type, RV32F and RV64F.
Converte o inteiro sem sinal de 32 bits em x [*rs1*] em um número de ponto flutuante de precisão simples e grava-o em f[*rd*].



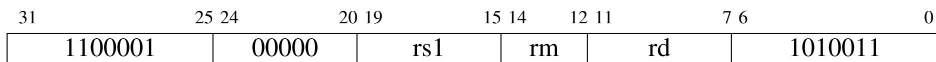
31–25	24–20	19–15	14–12	11–7	6–0
1101000	00001	rs1	rm	rd	1010011

---

fcvt.w.d rd, rs1, rs2

x[rd] = sext(s32$_{f64}$(f[rs1]))

Floating-point Convert to Word from Double. R-type, RV32D and RV64D.
Converte o número de ponto flutuante de precisão dupla no registrador f[*rs1*] em um inteiro de complemento de dois de 32 bits e grava o resultado com sinal estendido em x[*rd*].



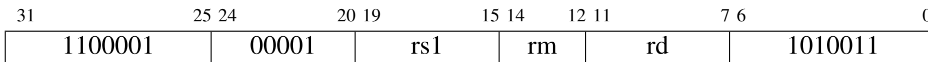
31–25	24–20	19–15	14–12	11–7	6–0
1100001	00000	rs1	rm	rd	1010011

---

fcvt.wu.d rd, rs1, rs2

x[rd] = sext($u32_{f64}$(f[rs1]))

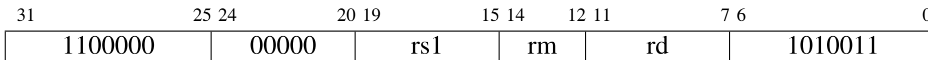
Floating-point Convert to Unsigned Word from Double. R-type, RV32D and RV64D.
Converte o número de ponto flutuante de precisão dupla do registrador f[*rs1*] em um inteiro sem sinal de 32 bits e grava o resultado em x[*rd*].



fcvt.w.s rd, rs1, rs2

x[rd] = sext($s32_{f32}$(f[rs1]))

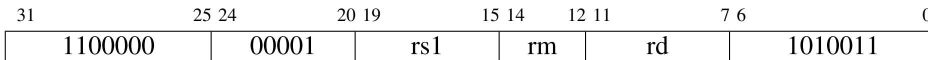
Floating-point Convert to Word from Single. R-type, RV32F and RV64F.
Converte o número de ponto flutuante de precisão simples no registrador f[*rs1*] em um inteiro de complemento de dois de 32 bits e grava o resultado com sinal estendido em x[*rd*].



fcvt.wu.s rd, rs1, rs2

x[rd] = sext($u32_{f32}$(f[rs1]))

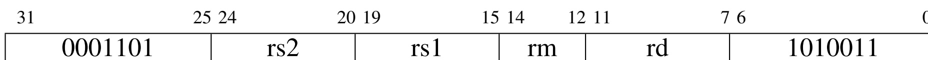
Floating-point Convert to Unsigned Word from Single. R-type, RV32F and RV64F.
Converte o número de ponto flutuante de precisão simples no registrador f[*rs1*] em um inteiro sem sinal de 32 bits e grava o resultado com sinal estendido em x[*rd*].



fdiv.d rd, rs1, rs2

f[rd] = f[rs1] ÷ f[rs2]

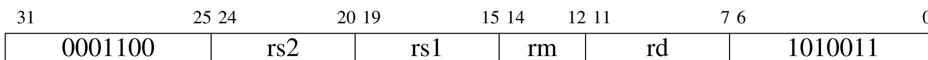
Floating-point Divide, Double-Precision. R-type, RV32D and RV64D.
Divide o número de ponto flutuante precisão dupla no registrador f[*rs1*] por f[*rs2*] e escreve o quociente de precisão dupla arredondado em f[*rd*].



fdiv.s rd, rs1, rs2

f[rd] = f[rs1] ÷ f[rs2]

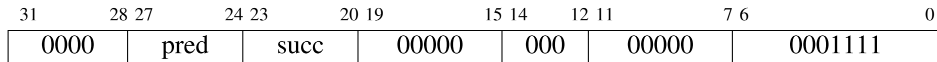
Floating-point Divide, Single-Precision. R-type, RV32F and RV64F.
Divide o número de ponto flutuante precisão simples no registrador f[*rs1*] por f[*rs2*] e escreve o quociente de precisão simples arredondado em f[*rd*].



fence pred, succ `Fence(pred, succ)`

Fence Memory and I/O. I-type, RV32I and RV64I.

Processa memória remanescente e acessos de E/S no conjunto *pred*ecessor, observável para outras threads e dispositivos, antes que a memória subsequente e os acessos de E/S no conjunto "*succ*essor" tornem-se observáveis. Os bits 3, 2, 1 e 0 nesses conjuntos correspondem aos dispositivos: **i** de entrada, **o** de saída, **r** de leitura de memória, e **w** de escrita, respectivamente. A instrução **fence** r, rw, por exemplo, ordena leituras antigas com novas leituras e escritas, e é codificada com *pred* = 0010 e *succ* = 0011. Se os argumentos forem omitidos, um **fence** iorw, iorw completo está implícito.

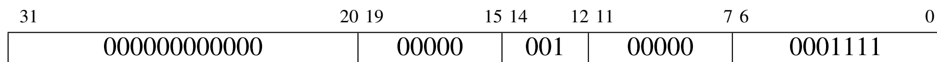


31 28	27 24	23 20	19 15	14 12	11 7	6 0
0000	pred	succ	00000	000	00000	0001111

fence.i `Fence(Store, Fetch)`

Fence Instruction Stream. I-type, RV32I and RV64I.

Processa stores para memória de instrução observável para buscas de instrução subsequentes.

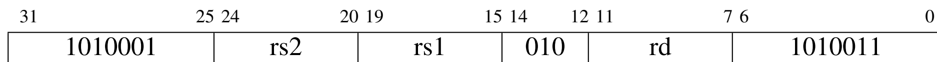


31 20	19 15	14 12	11 7	6 0
000000000000	00000	001	00000	0001111

feq.d rd, rs1, rs2 `x[rd] = f[rs1] == f[rs2]`

Floating-point Equals, Double-Precision. R-type, RV32D and RV64D.

Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão dupla em f[*rs1*] for igual ao número em f[*rs2*], e 0 caso contrário.

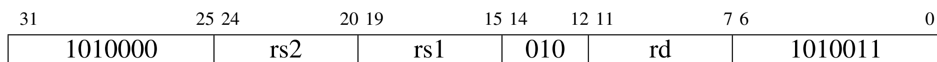


31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	010	rd	1010011

feq.s rd, rs1, rs2 `x[rd] = f[rs1] == f[rs2]`

Floating-point Equals, Single-Precision. R-type, RV32F and RV64F.

Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão simples em f[*rs1*] for igual ao número em f[*rs2*], e 0 caso contrário.

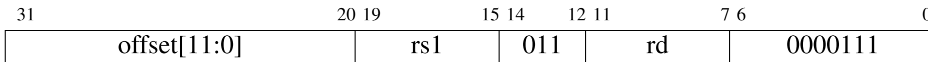


31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	010	rd	1010011

fld rd, offset(rs1)

`f[rd] = M[x[rs1] + sext(offset)][63:0]`

Floating-point Load Doubleword. I-type, RV32D and RV64D.
Carrega um número de ponto flutuante de precisão dupla do endereço de memória x[*rs1*] + *sign-extend*(deslocamento) e escreve em f[*rd*].
Formas compactadas: **c.fldsp** rd, offset; **c.fld** rd, offset (rs1)



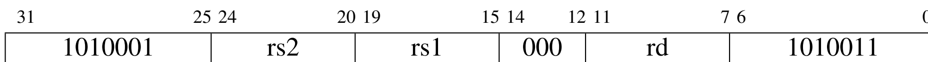
31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
offset[11:0]	rs1	011	rd	0000111

---

fle.d rd, rs1, rs2

`x[rd] = f[rs1] ≤ f[rs2]`

Floating-point Less Than or Equal, Double-Precision. R-type, RV32D and RV64D.
Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão dupla em f[*rs1*] for menor ou igual ao número em f[*rs2*], e 0 caso contrário.



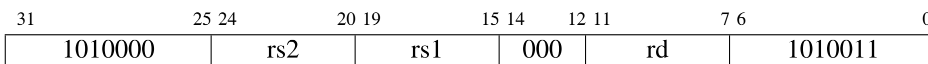
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
1010001	rs2	rs1	000	rd	1010011

---

fle.s rd, rs1, rs2

`x[rd] = f[rs1] ≤ f[rs2]`

Floating-point Less Than or Equal, Single-Precision. R-type, RV32F and RV64F.
Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão simples em f[*rs1*] for menor ou igual ao número em f[*rs2*], e 0 contrário.



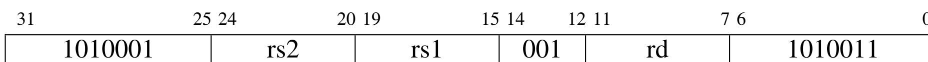
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
1010000	rs2	rs1	000	rd	1010011

---

flt.d rd, rs1, rs2

`x[rd] = f[rs1] < f[rs2]`

Floating-point Less Than, Double-Precision. R-type, RV32D and RV64D.
Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão dupla em f[*rs1*] for menor que o número em f [*rs2*], e 0 caso contrário.



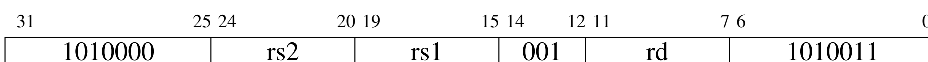
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
1010001	rs2	rs1	001	rd	1010011

---

flt.s rd, rs1, rs2

`x[rd] = f[rs1] < f[rs2]`

Floating-point Less Than, Single-Precision. R-type, RV32F and RV64F.
Escreve 1 em x[*rd*] se o número de ponto flutuante de precisão simples em f[*rs1*] for menor que o número em f[*rs2*], e 0 caso contrário.

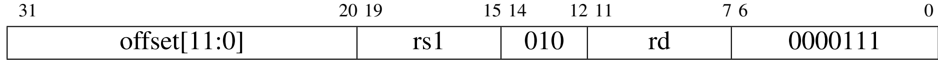


31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
1010000	rs2	rs1	001	rd	1010011

---

flw rd, offset(rs1) `f[rd] = M[x[rs1] + sext(offset)][31:0]`

Floating-point Load Word. I-type, RV32F and RV64F.
Carrega um número de ponto flutuante de precisão simples do endereço de memória x[*rs1*] + *sign-extend*(*deslocamento*) e grava-o em f[*rd*].
formas compactadas: **c.flwsp** rd, offset; **c.flw** rd, offset(rs1)

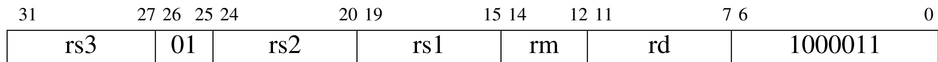


31–20	19–15	14–12	11–7	6–0
offset[11:0]	rs1	010	rd	0000111

---

fmadd.d rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]+f[rs3]`

Floating-point Fused Multiply-Add, Double-Precision. R4-type, RV32D and RV64D.
Multiplica os números de ponto flutuante de precisão dupla em f [*rs1*] e f [*rs2*], adiciona o produto não arredondado ao número de ponto flutuante de precisão dupla em f[*rs3*] e grava o resultado de precisão dupla arredondada em f[*rd*].

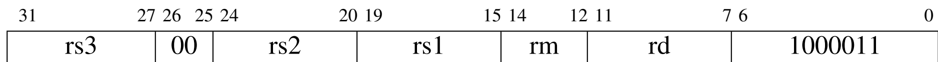


31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	01	rs2	rs1	rm	rd	1000011

---

fmadd.s rd, rs1, rs2, rs3 `f[rd] = f[rs1]×f[rs2]+f[rs3]`

Floating-point Fused Multiply-Add, Single-Precision. R4-type, RV32F and RV64F.
Multiplica os números de ponto flutuante de precisão simples em f[*rs1*] e f[*rs2*], adiciona o produto não arredondado ao número de ponto flutuante de precisão simples em f[*rs3*] e grava o resultado arredondado de precisão simples em f[*rd*].

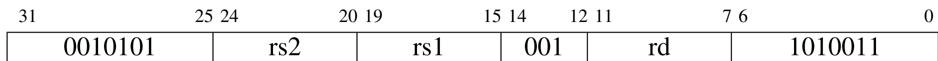


31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	00	rs2	rs1	rm	rd	1000011

---

fmax.d rd, rs1, rs2 `f[rd] = max(f[rs1], f[rs2])`

Floating-point Maximum, Double-Precision. R-type, RV32D and RV64D.
Copia o maior dos números de ponto flutuante de precisão dupla dos registradores f[*rs1*] e f [*rs2*] para f[*rd*].

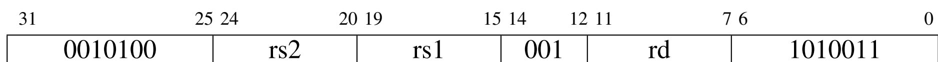


31–25	24–20	19–15	14–12	11–7	6–0
0010101	rs2	rs1	001	rd	1010011

---

fmax.s rd, rs1, rs2 `f[rd] = max(f[rs1], f[rs2])`

Floating-point Maximum, Single-Precision. R-type, RV32F and RV64F.
Copia o maior dos números de ponto flutuante de precisão simples dos registradores f[*rs1*] e f[*rs2*] para f[*rd*].



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])`

Floating-point Minimum, Double-Precision. R-type, RV32D and RV64D.
Copia o menor dos números de ponto flutuante de precisão dupla dos registradores f[*rs1*] e f[*rs2*] para f[*rd*].

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])`

Floating-point Minimum, Single-Precision. R-type, RV32F and RV64F.
Copia o menor dos números de ponto flutuante de precisão simples dos registradores f[*rs1*] e f[*rs2*] para f[*rd*].

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]`

Floating-point Fused Multiply-Subtract, Double-Precision. R4-type, RV32D and RV64D.
Multiplica os números de ponto flutuante de precisão dupla em f[*rs1*] e f[*rs2*], subtrai o resultado do produto não arredondado com o número de ponto flutuante de precisão dupla em f[*rs3*] e grava o resultado arredondado de precisão dupla em 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]`

Floating-point Fused Multiply-Subtract, Single-Precision. R4-type, RV32F and RV64F.
Multiplica os números de ponto flutuante de precisão simples em f[*rs1*] e f[*rs2*], subtrai o resultado do produto não arredondado com o número de ponto flutuante de precisão simples em f[*rs3*] e grava o resultado arredondado de precisão simples em 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]`

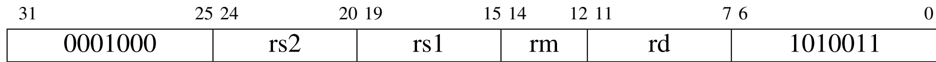
Floating-point Multiply, Double-Precision. R-type, RV32D and RV64D.
Multiplica os números de ponto flutuante de precisão dupla nos registradores f[*rs1*] e f[*rs2*] e grava o produto arredondado de precisão dupla em f[*rd*].

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]`

Floating-point Multiply, Single-Precision. R-type, RV32F and RV64F.
Multiplica os números de ponto flutuante de precisão simples nos registradores f[*rs1*] e f[*rs2*] e grava o produto arredondado de precisão simples em f[*rd*].



31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
0001000	rs2	rs1	rm	rd	1010011

---

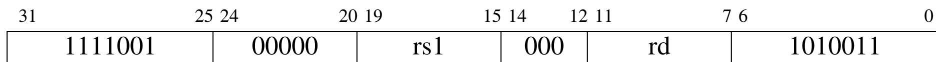
fmv.d rd, rs1 `f[rd] = f[rs1]`

Floating-point Move. Pseudoinstruction, RV32D and RV64D.
Copia o número de ponto flutuante de precisão dupla em f[*rs1*] para f[*rd*]. Expande para **fsgnj.d** rd, rs1, rs1.

---

fmv.d.x rd, rs1, rs2 `f[rd] = x[rs1][63:0]`

Floating-point Move Doubleword from Integer. R-type, RV64D only.
Copia o número de ponto flutuante de precisão dupla do registrador x[*rs1*] para f[*rd*].



31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
1111001	00000	rs1	000	rd	1010011

---

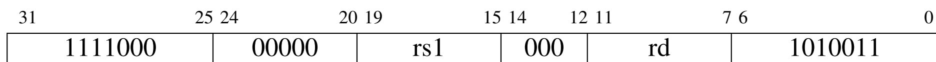
fmv.s rd, rs1 `f[rd] = f[rs1]`

Floating-point Move. Pseudoinstruction, RV32F and RV64F.
Copia o número de ponto flutuante de precisão simples em f[*rs1*] para f[*rd*]. Expande para **fsgnj.s** rd, rs1, rs1.

---

fmv.w.x rd, rs1, rs2 `f[rd] = x[rs1][31:0]`

Floating-point Move Word from Integer. R-type, RV32F and RV64F.
Copia o número de ponto flutuante de precisão simpels no registrador x[*rs1*] para f[*rd*].

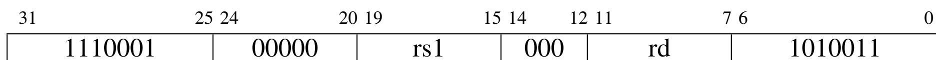


31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
1111000	00000	rs1	000	rd	1010011

---

fmv.x.d rd, rs1, rs2 `x[rd] = f[rs1][63:0]`

Floating-point Move Doubleword to Integer. R-type, RV64D only.
Copia o número de ponto flutuante de precisão dupla no registrador f[*rs1*] para x[*rd*].



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])`

Floating-point Move Word to Integer. R-type, RV32F and RV64F.
Copia o número de ponto flutuante de precisão única no registrador f[*rs1*] para x[*rd*], com extensão de sinal para RV64F.

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]`

Floating-point Negate. Pseudoinstruction, RV32D and RV64D.
Escreve o oposto do número de ponto flutuante de precisão dupla em f [*rs1*] a f [*rd*]. Expande para **fsgnjn.d** rd, rs1, rs1.

---

fneg.s rd, rs1 `f[rd] = -f[rs1]`

Floating-point Negate. Pseudoinstruction, RV32F and RV64F.
Escreve o oposto do número de ponto flutuante de precisão simples em f[*rs1*] a f[*rd*]. Expande para **fsgnjn.s** rd, rs1, rs1.

---

fnmadd.d rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]-f[rs3]`

Floating-point Fused Negative Multiply-Add, Double-Precision. R4-type, RV32D and RV64D.
Multiplica os números de ponto flutuante de precisão dupla em f[*rs1*] e f[*rs2*], nega o resultado, subtrai o número de ponto flutuante de precisão dupla em f[*rs3*] ao produto não-arredondado e grava o resultado de precisão dupla arredondada em 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]`

Floating-point Fused Negative Multiply-Add, Single-Precision. R4-type, RV32F and RV64F.
Multiplica os números de ponto flutuante de precisão simples em f[*rs1*] e f[*rs2*], nega o resultado, subtrai o número de ponto flutuante de precisão simples em f[*rs3*] ao produto não-arredondado e grava o resultado arredondado de precisão simples em 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]`

Floating-point Fused Negative Multiply-Subtract, Double-Precision. R4-type, RV32D and RV64D.
Multiplica os números de ponto flutuante de precisão dupla em f[*rs1*] e f[*rs2*], nega o resultado, adiciona o produto não arredondado ao número de ponto flutuante de precisão dupla em f [*]*rs3] e grava o resultado de precisão dupla arredondada em 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]`

Floating-point Fused Negative Multiply-Subtract, Single-Precision. R4-type, RV32F and RV64F.
Multiplica os números de ponto flutuante de precisão simples em f[*rs1*] e f[*rs2*], nega o resultado, adiciona o produto não arredondado ao número de ponto flutuante de precisão simples em f[*]*rs3] e grava o resultado arredondado de precisão simples em 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]`

Floating-point Read Control and Status Register. Pseudoinstruction, RV32F and RV64F.
Copia o valor de de ponto flutuante do registrador de controle e status para x[*rd*]. Expande para **csrrs** rd, fcsr, x0.

---

frflags rd

`x[rd] = CSRs[fflags]`

Floating-point Read Exception Flags. Pseudoinstruction, RV32F and RV64F.
Copia as *flags* de exceção de ponto flutuante para x[*rd*]. Expande para **csrrs** rd, fflags, x0.

---

frrm rd

`x[rd] = CSRs[frm]`

Floating-point Read Rounding Mode. Pseudoinstruction, RV32F and RV64F.
Copia o modo de arredondamento de ponto flutuante para x[*rd*]. Expande para **csrrs** rd, frm, x0.

---

fscsr rd, rs1

`t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t`

Floating-point Swap Control and Status Register. Pseudoinstruction, RV32F and RV64F.
Copia x[*rs1*] para o registrador de controle e status de ponto flutuante, e copia o valor anterior do registrado de controle e status de ponto flutuante para x[*rd*]. Expande para **csrrw** rd, fcsr, rs1. Se *rd* for omitido, x0 será adotado.

---

fsd rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = f[rs2][63:0]`

Floating-point Store Doubleword. S-type, RV32D and RV64D.
Armazena o número de ponto flutuante de precisão dupla do registrador f[*rs2*] na memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*).
Formas compactadas: **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`

Floating-point Swap Exception Flags. Pseudoinstruction, RV32F e RV64F.
Copia x[*rs1*] para o registrador de *flags* de exceção de ponto flutuante e, em seguida, copia as *flags* de exceção de ponto flutuante anteriores para x[*rd*]. Expande para **csrrw** rd, fflags, rs1. Se *rd* for omitido, x0 será adotado.

---

fsgnj.d rd, rs1, rs2

`f[rd] = {f[rs2][63], f[rs1][62:0]}`

Floating-point Sign Inject, Double-Precision. R-type, RV32D and RV64D.
Constrói um novo número de ponto flutuante de precisão dupla a partir do expoente e significando de f[*rs1*], pegando o sinal de f[*rs2*] e grava-o em 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]}`

Floating-point Sign Inject, Single-Precision. R-type, RV32F and RV64F.
Constrói um novo número de ponto flutuante de precisão simples a partir do expoente e significando de f[*rs1*], pegando o sinal de f[*rs2*] e grava-o em 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]}`

Floating-point Sign Inject-Negate, Double-Precision. R-type, RV32D and RV64D.
Constrói um novo número de ponto flutuante de precisão dupla a partir do expoente e significando de f[*rs1*], assumindo o sinal oposto de f[*rs2*] e grava-o em 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]}`

Floating-point Sign Inject-Negate, Single-Precision. R-type, RV32F and RV64F.
Constrói um novo número de ponto flutuante de precisão simples a partir do expoente e significando de f[*rs1*], assumindo o sinal oposto de f[*rs2*] e grava-o em 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]}`

Floating-point Sign Inject-XOR, Double-Precision. R-type, RV32D and RV64D.
Constrói um novo número de ponto flutuante de precisão dupla a partir do expoente e significando de f[*rs1*], tomando o sinal do XOR dos sinais de f[*rs1*] e f[*rs2*], e grava-o em 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]}`

Floating-point Sign Inject-XOR, Single-Precision. R-type, RV32F and RV64F.
Constrói um novo número de ponto flutuante de precisão simples a partir do expoente e significando de f[*rs1*], tomando o sinal do XOR dos sinais de f[*rs1*] e f [*rs2*], e gravo-o em 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 $f[rd] = \sqrt{f[rs1]}$

Floating-point Square Root, Double-Precision. R-type, RV32D and RV64D.
Calcula a raiz quadrada do número de ponto flutuante de precisão dupla no registrador f[*rs1*] e grava o resultado arredondado de precisão dupla em 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 $f[rd] = \sqrt{f[rs1]}$

Floating-point Square Root, Single-Precision. R-type, RV32F and RV64F.
Calcula a raiz quadrada do número de ponto flutuante de precisão simples no registrador f[*rs1*] e grava o resultado arredondado de precisão simples em f[*rd*].

31–25	24–20	19–15	14–12	11–7	6–0
0101100	00000	rs1	rm	rd	1010011

---

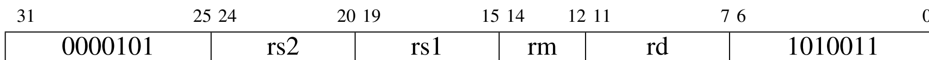
fsrm rd, rs1 `t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t`

Floating-point Swap Rounding Mode. Pseudoinstruction, RV32F e RV64F.
Copia x[*rs1*] para o registrador de modo de arredondamento de ponto flutuante e, em seguida, copia o modo de arredondamento de ponto flutuante anterior para x[*rd*]. Expande para **csrrw** rd, frm, rs1. Se *rd* for omitido, x0 será adotado.

---

fsub.d rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

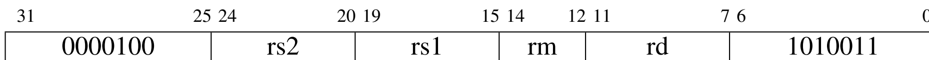
Floating-point Subtract, Double-Precision. R-type, RV32D and RV64D.
Subtrai o número de ponto flutuante de precisão dupla no registrador f[*rs2*] de f[*rs1*] e grava a diferença de precisão dupla arredondada em f[*rd*].



---

fsub.s rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

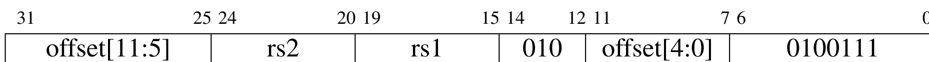
Floating-point Subtract, Single-Precision. R-type, RV32F and RV64F.
Subtrai o número de ponto flutuante de precisão simples no registrador f[*rs2*] de f[*rs1*] e grava a diferença arredondada de precisão simples para f[*rd*].



---

fsw rs2, offset(rs1) `M[x[rs1] + sext(offset)] = f[rs2][31:0]`

Floating-point Store Word. S-type, RV32F and RV64F.
Armazena o número de ponto flutuante de precisão simples do registrador f[*rs2*] na memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*).
Formas compactadas: **c.fswsp** rs2, offset; **c.fsw** rs2, offset (rs1)



---

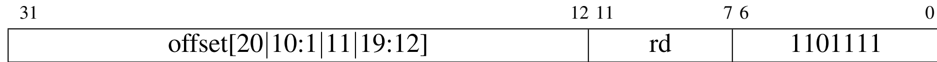
j offset `pc += sext(offset)`

Jump. Pseudoinstruction, RV32I and RV64I.
Atribui ao *PC* o atual valor do *PC* mais o valor de deslocamento com extensão de sinal. Expande para **jal** x0, offset.

---

jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

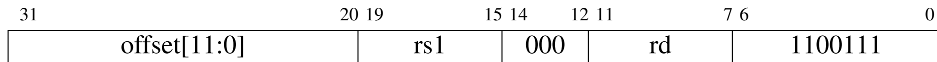
Jump and Link. J-type, RV32I and RV64I.
Escreve o endereço da próxima instrução (*PC* + 4) para x[*rd*] e, em seguida, atribui ao *PC* o atual valor de *PC* atual mais o valor de deslocamento com extensão de sinal. Se *rd* for omitido, x1 é adotado.
Formas compactadas: **c.j** offset; **c.jal** offset



---

jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&∼1; x[rd]=t`

Jump and Link Register. I-type, RV32I and RV64I.
Copia o *PC* para x[*rs1*] + *sign-extend*(*deslocamento*), mascarando o bit menos significativo do endereço calculado, então grava o valor anterior do *PC* + 4 em x[*rd*]. Se *rd* for omitido, x1 é assumido.
Formas compactadas: **c.jr** rs1; **c.jalr** rs1



---

jr rs1 `pc = x[rs1]`

Jump Register. Pseudoinstruction, RV32I and RV64I.
"Seta" o *PC* como x[*rs1*]. Expande para **jalr** x0, 0 (rs1).

---

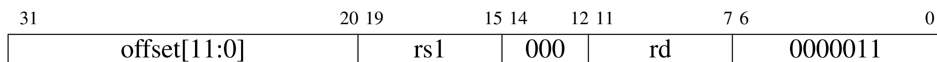
la rd, symbol `x[rd] = &symbol`

Load Address. Pseudoinstruction, RV32I and RV64I.
Carrega o endereço de *symbol* em x[*rd*]. Ao montar o código independente da posição, ele se expande em um load a partir da Tabela de Compensação Global: para RV32I, **auipc** rd, offsetHi e então **lw** rd, offsetLo(rd); para RV64I, **auipc** rd, offsetHi e depois **ld** rd, offsetLo(rd). Em último caso, ele se expande para **auipc** rd, offsetHi e depois **addi** rd, rd, offsetLo.

---

lb rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][7:0])`

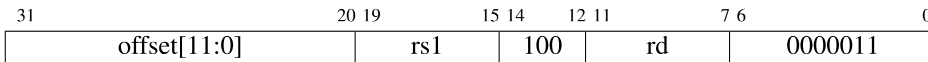
Load Byte. I-type, RV32I and RV64I.
Carrega um byte da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-o x[*rd*], realizando extensão de sinal.



---

lbu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][7:0]`

Load Byte, Unsigned. I-type, RV32I and RV64I.
Carrega um byte da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-o x[*rd*], realizando extensão de zero.

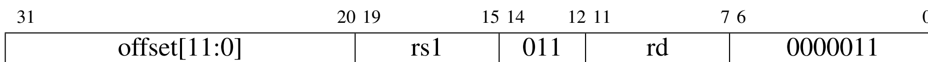


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
offset[11:0]	rs1	100	rd	0000011

---

ld rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][63:0]`

Load Doubleword. I-type, RV64I only.
Carrega oito bytes da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-os em x[*rd*].
Formas compactadas: **c.ldsp** rd, deslocamento; **c.ld** rd, offset (rs1)

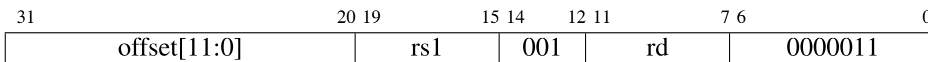


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
offset[11:0]	rs1	011	rd	0000011

---

lh rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

Load Halfword. I-type, RV32I and RV64I.
Carrega dois bytes da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-os em x[*rd*], realizando extensão de sinal.

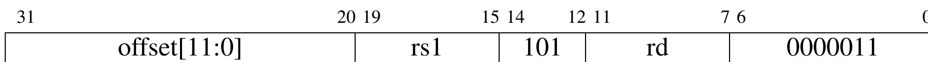


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
offset[11:0]	rs1	001	rd	0000011

---

lhu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][15:0]`

Load Halfword, Unsigned. I-type, RV32I and RV64I.
Carrega dois bytes da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-os em x[*rd*], realizando extensão de zeros.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
offset[11:0]	rs1	101	rd	0000011

---

li rd, immediate `x[rd] = immediate`

Load Immediate. Pseudoinstruction, RV32I and RV64I.
Carrega uma constante de imediato em x[*rd*], usando o menor número de instruções possível. Para o RV32I, ele se expande para **lui** e/ou **addi**; para RV64I, é tão grande quanto **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

---

lla rd, symbol `x[rd] = &symbol`

Load Local Address. Pseudoinstruction, RV32I and RV64I.

Carrega o endereço de *symbol* em x[*rd*]. Expande em **auipc** rd, offsetHi e depois **addi** rd, rd, offsetLo.

---

lr.d rd, (rs1) `x[rd] = LoadReserved64(M[x[rs1]])`

Load-Reserved Doubleword. R-type, RV64A only.

Carrega os oito bytes da memória no endereço x[*rs1*], grava-os em x[*rd*] e registra uma reserva nessa palavra dupla de memória.

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]])`

Load-Reserved Word. R-type, RV32A and RV64A.

Carrega os quatro bytes da memória no endereço x[*rs1*], grava-os em x[*rd*], realizando extensão de sinal no resultado, e registra uma reserva nessa palavra de memória.

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])`

Load Word. I-type, RV32I and RV64I.

Carrega quatro bytes da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-os em x[*rd*]. Para RV64I, o resultado é um sinal estendido.

Formas compactadas: **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]`

Load Word, Unsigned. I-type, RV64I only.

Carrega quatro bytes da memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*) e grava-os em x[*rd*], com extensão de zero no resultado.

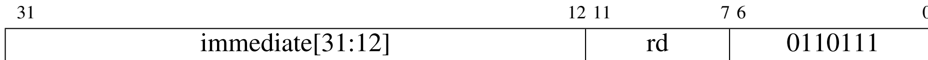
31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	110	rd	0000011

---

lui rd, immediate

`x[rd] = sext(immediate[31:12] << 12)`

Load Upper Immediate. U-type, RV32I and RV64I.
Escreve o valor imediato de 20 bits com sinal estendido e deslocado em 12 bits à esquerda para x[*rd*], zerando os 12 bits inferiores.
Forma compactada: **c.lui** rd, imm

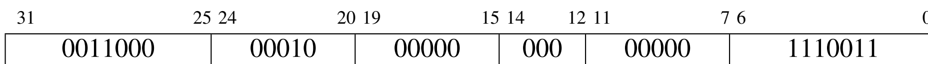


31–12	11–7	6–0
immediate[31:12]	rd	0110111

mret

`ExceptionReturn(Machine)`

Machine-mode Exception Return. R-type, RV32I and RV64I privileged architectures.
Retorna do *handler* de exceção de modo de máquina. "Seta" o *PC* para CSRs[mepc], o modo de privilégio para CSRs[mstatus].MPP, CSRs[mstatus].MIE para CSRs[mstatus].MPIE e CSRs[mstatus].MPIE para 1; e, se o modo de usuário for suportado, define CSRs[mstatus].MPP como 0.

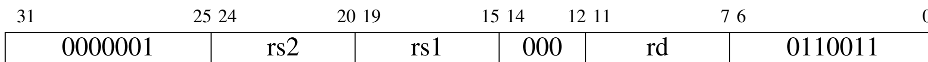


31–25	24–20	19–15	14–12	11–7	6–0
0011000	00010	00000	000	00000	1110011

mul rd, rs1, rs2

$x[rd] = x[rs1] \times x[rs2]$

Multiply. R-type, RV32M and RV64M.
Multiplica x [*rs1*] por x [*rs2*] e grava o produto em x[*rd*]. O overflow aritmético é ignorado.

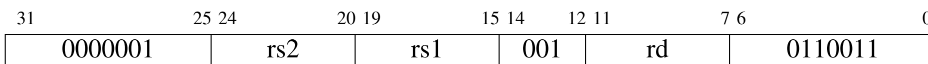


31–25	24–20	19–15	14–12	11–7	6–0
0000001	rs2	rs1	000	rd	0110011

mulh rd, rs1, rs2

$x[rd] = (x[rs1] \ {}_s\!\times_s \ x[rs2]) >>_s XLEN$

Multiply High. R-type, RV32M and RV64M.
Multiplica x[*rs1*] por x [*rs2*], tratando os valores como números de complemento de dois e grava a metade superior do produto em x[*rd*].

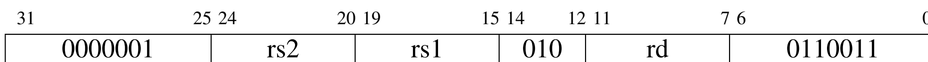


31–25	24–20	19–15	14–12	11–7	6–0
0000001	rs2	rs1	001	rd	0110011

mulhsu rd, rs1, rs2

$x[rd] = (x[rs1] \ {}_s\!\times_u \ x[rs2]) >>_s XLEN$

Multiply High Signed-Unsigned. R-type, RV32M and RV64M.
Multiplica x[*rs1*] por x[*rs2*], tratando x[rs1] como um número de complemento de dois e x[rs2] como um número sem sinal e grava a metade superior do produto em 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]) >>`$_u$ `XLEN`

Multiply High Unsigned. R-type, RV32M and RV64M.
Multiplica x[*rs1*] por x[*rs2*], tratando os valores como números sem sinal e grava a metade superior do produto em 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])`

Multiply Word. R-type, RV64M only.
Multiplica x[*rs1*] por x[*rs2*], trunca o valor do produto para 32 bits e grava o resultado com extensão de sinal em x[*rd*]. O overflow aritmético é ignorado.

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]`

Move. Pseudoinstruction, RV32I e RV64I.
Copia o valor do registrador x[*rs1*] para x[*rd*]. Expande para **addi** rd, rs1, 0.

---

neg rd, rs2 `x[rd] = -x[rs2]`

Negate. Pseudoinstruction, RV32I e RV64I.
Escreve o complemento de dois de x[*rs2*] em x[*rd*]. Expande para **sub** rd, x0, rs2.

---

negw rd, rs2 `x[rd] = sext((-x[rs2])[31:0])`

Negate Word. Pseudoinstruction, Somente RV64I.
Calcula o complemento de dois de x[*rs2*], trunca o resultado para 32 bits e grava o resultado de sinal estendido em x[*rd*]. Expande para **subw** rd, x0, rs2.

---

nop *Nothing*

No operation. Pseudoinstruction, RV32I and RV64I.
Apenas avança o *PC* para a próxima instrução. Expande para **addi** x0, x0, 0.

---

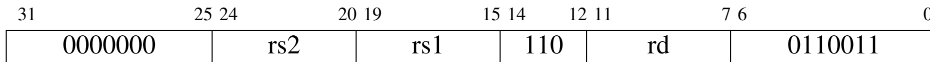
not rd, rs1 `x[rd] = ~x[rs1]`

NOT. Pseudoinstruction, RV32I e RV64I.
Escreve o complemento de um de x[*rs1*] em x[*rd*]. Expande para **xori** rd, rs1, -1.

---

or rd, rs1, rs2 `x[rd] = x[rs1] | x[rs2]`

OR. R-type, RV32I and RV64I.
Calcula o OR-inclusivo bit a bit dos registradores x[*rs1*] e x[*rs2*] e grava o resultado em x[*rd*].
Forma compactada: **c.or** rd, rs2

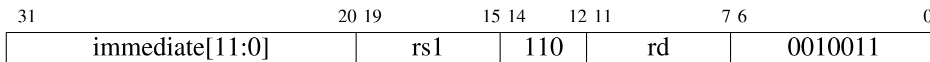


31–25	24–20	19–15	14–12	11–7	6–0
0000000	rs2	rs1	110	rd	0110011

---

ori rd, rs1, immediate `x[rd] = x[rs1] | sext(immediate)`

OR Immediate. I-type, RV32I and RV64I.
Calcula o OR-inclusivo bit a bit do imediato de sinal estendido com o registrador x[*rs1*] e grava o resultado em x[*rd*].



31–20	19–15	14–12	11–7	6–0
immediate[11:0]	rs1	110	rd	0010011

---

rdcycle rd `x[rd] = CSRs[cycle]`

Read Cycle Counter. Pseudoinstruction, RV32I e RV64I.
Escreve o número de ciclos que passaram em x[*rd*]. Expande para **csrrs** rd, cycle, x0.

---

rdcycleh rd `x[rd] = CSRs[cycleh]`

Read Cycle Counter High. Pseudoinstruction, Somente RV32I.
Escreve o número de ciclos que se passaram, com deslocamento à direita em 32 bits, em x[*rd*]. Expande para **csrrs** rd, cycleh, x0.

---

rdinstret rd `x[rd] = CSRs[instret]`

Read Instructions-Retired Counter. Pseudoinstruction, RV32I e RV64I.
Escreve o número de *retired-instructions* em x[*rd*]. Expande para **csrrs** rd, instret, x0.

---

rdinstreth rd `x[rd] = CSRs[instreth]`

Read Instructions-Retired Counter High. Pseudoinstruction, Somente RV32I.
Escreve o número de *retired-instructions*, deslocado em 32 bits à direita, em x[*rd*]. Expande para **csrrs** rd, instreth, x0.

---

rdtime rd `x[rd] = CSRs[time]`

Read Time. Pseudoinstruction, RV32I e RV64I.
Escreve a hora atual em x[*rd*]. A frequência do temporizador depende da plataforma. Expande para **csrrs** rd, time, x0.

---

rdtimeh rd

x[rd] = CSRs[timeh]

Read Time High. Pseudoinstruction, RV32I only.

Escreve a hora atual em x[*rd*], deslocada 32 bits à direita. A frequência do temporizador depende da plataforma. Expande para **csrrs** rd, time, x0.

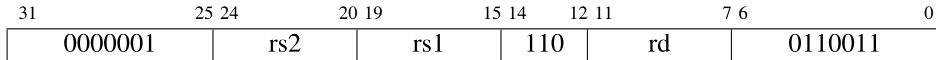
---

rem rd, rs1, rs2

x[rd] = x[rs1] $\%_s$ x[rs2]

Remainder. R-type, RV32M and RV64M.

Divide x[*rs1*] por x[*rs2*], arredondando para zero, tratando os valores como números de complemento de dois e grava o resto da divisão em x[*rd*].



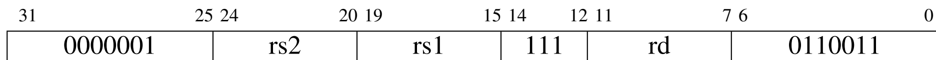
---

remu rd, rs1, rs2

x[rd] = x[rs1] $\%_u$ x[rs2]

Remainder, Unsigned. R-type, RV32M and RV64M.

Divide x[*rs1*] por x[*rs2*], arredondando para zero, tratando os valores como números sem sinal, e gravo o resto da divisão em x[*rd*].



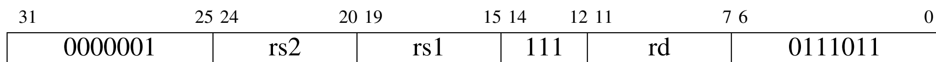
---

remuw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\%_u$ x[rs2][31:0])

Remainder Word, Unsigned. R-type, RV64M only.

Divide os primeiros 32 bits de x[*rs1*] pelos primeiros 32 bits de x[*rs2*], arredondando para zero, tratando os valores como números sem sinal e grava o resto de 32 bits com extensão de sinal em x[*rd*].



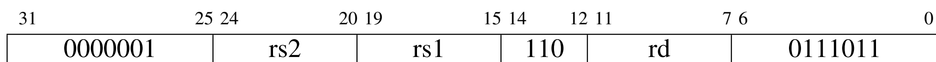
---

remw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\%_s$ x[rs2][31:0])

Remainder Word. R-type, RV64M only.

Divide os primeiros 32 bits de x[*rs1*] pelos primeiros 32 bits de x[*rs2*], arredondando para zero, tratando os valores como números de complemento de dois e escreve o resto da divisão de 32 bits com extensão de sinal em x[*rd*].



---

ret

pc = x[1]

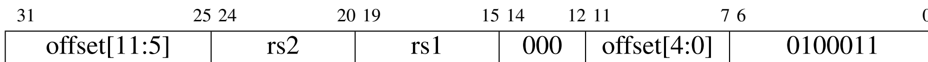
Return. Pseudoinstruction, RV32I and RV64I.

Retorna de uma sub-rotina. Expande para **jalr** x0, 0 (x1).

---

sb rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][7:0]`

Store Byte. S-type, RV32I and RV64I.
Armazena o byte menos significativo no registrador x[*rs2*] para a memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*).

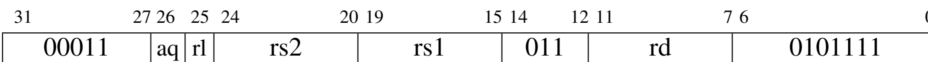


31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011

---

sc.d rd, rs2, (rs1) `x[rd] = StoreConditional64(M[x[rs1]], x[rs2])`

Store-Conditional Doubleword. R-type, RV64A only.
Armazena os oito bytes do registrador x[*rs2*] na memória no endereço x[*rs1*], desde que já exista um load reservado para este endereço de memória. Grava 0 em x[*rd*] se o store obtiver êxito, e um código de erro caso contrário.

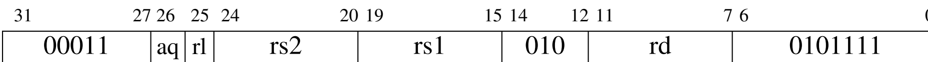


31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	011	rd	0101111

---

sc.w rd, rs2, (rs1) `x[rd] = StoreConditional32(M[x[rs1]], x[rs2])`

Store-Conditional Word. R-type, RV32A and RV64A.
Armazena os quatro bytes do registrador x[*rs2*] na memória no endereço x[*rs1*], desde que já exista um load reservado para este endereço de memória. Grava 0 em x[*rd*] se o store obtiver êxito, e um código de erro caso contrário.

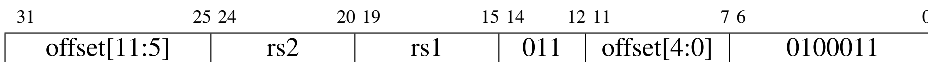


31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	010	rd	0101111

---

sd rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][63:0]`

Store Doubleword. S-type, RV64I only.
Armazena os oito bytes do registrador x[*rs2*] na memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*).
Formas compactadas: **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)`

Set if Equal to Zero. Pseudoinstruction, RV32I and RV64I.
Escreve 1 em x[*rd*] se x[*rs1*] for igual a 0, ou 0 se não for. Expande para **sltiu** rd, rs1, 1.

---

sext.w rd, rs1 `x[rd] = sext(x[rs1][31:0])`

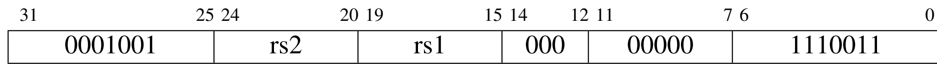
Sign-extend Word. Pseudoinstruction, RV64I only.
Lê os primeiros 32 bits de x[*rs1*], grava o resultado com extensão de sinal em x[*rd*]. Expande para **addiw** rd, rs1, 0.

---

sfence.vma rs1, rs2

`Fence(Store, AddressTranslation)`

Fence Virtual Memory. R-type, RV32I and RV64I privileged architectures.
Ordena stores encaminhados à tabela de páginas com seus respectivas traduções de endereço virtual. Quando *rs2* = 0, as traduções para todos os espaços de endereçamento são afetadas; caso contrário, somente as traduções para o espaço de endereço identificado por x[*rs2*] serão ordenadas. Quando *rs1* = 0, as traduções para todos os endereços virtuais nos espaços de endereço selecionados são ordenadas; caso contrário, somente as traduções para a página que contém o endereço virtual x[*rs1*] nos espaços de endereço selecionados serão ordenadas.



31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

sgtz rd, rs2

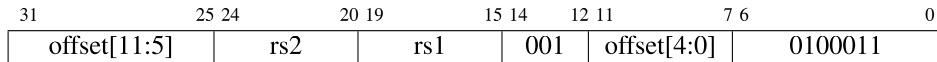
`x[rd] = (x[rs2] >_s 0)`

Set if Greater Than to Zero. Pseudoinstruction, RV32I e RV64I.
Escreve 1 em x[*rd*] se x[*rs2*] for maior que 0, ou 0 se não for. Expande para **slt** rd, x0, rs2.

sh rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = x[rs2][15:0]`

Store Halfword. S-type, RV32I and RV64I.
Armazena os dois bytes menos significativos do registrador x[*rs2*] na memória no endereço x[*rs1*] + *sign-extend(*deslocamento*)*.

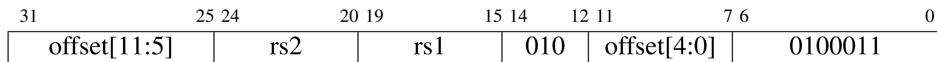


31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011

sw rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = x[rs2][31:0]`

Store Word. S-type, RV32I and RV64I.
Armazena os quatro bytes menos significativos do registrador x[*rs2*] na memória no endereço x[*rs1*] + *sign-extend*(*deslocamento*).
Formas compactadas: **c.swsp** rs2, offset; **c.sw** rs2, offset (rs1)

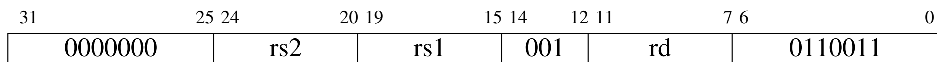


31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011

sll rd, rs1, rs2

`x[rd] = x[rs1] << x[rs2]`

Shift Left Logical. R-type, RV32I and RV64I.
Desloca o conteúdo do registrador x[*rs1*] em x[*rs2*] posições para à esquerda. Os bits vazios são preenchidos com zeros e o resultado é gravado em x[*rd*]. Os cinco bits menos significativos de x[*rs2*](ou seis bits para o RV64I) formam o conjunto deslocado; os bits remanescentes são ignorados.

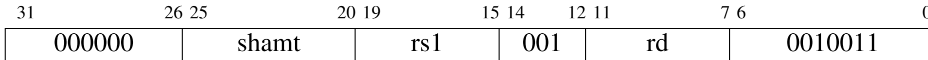


31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	001	rd	0110011

slli rd, rs1, shamt

x[rd] = x[rs1] << shamt

Shift Left Logical Immediate. I-type, RV32I and RV64I.
Desloca o conteúdo do registrador x[*rs1*] em *shamt* posições para à esquerda. Os bits vazios são preenchidos com zeros e o resultado é gravado em x[*rd*]. Para RV32I, a instrução só é permitida quando *shamt*[5] = 0.
Forma compactada: **c.slli** rd, shamt

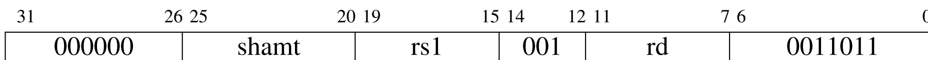


---

slliw rd, rs1, shamt

x[rd] = sext((x[rs1] << shamt)[31:0])

Shift Left Logical Word Immediate. I-type, RV64I only.
Desloca o conteúdo do registrador x[*rs1*] em *shamt* posições para à esquerda. Os bits vazios são preenchidos com zeros, o resultado é truncado para 32 bits e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. A instrução só é legal quando *shamt*[5] = 0.

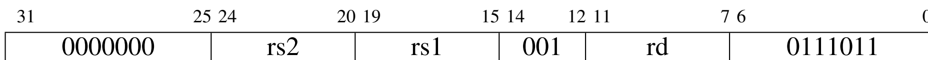


---

sllw rd, rs1, rs2

x[rd] = sext((x[rs1] << x[rs2][4:0])[31:0])

Shift Left Logical Word. R-type, RV64I only.
Desloca os primeiros 32 bits de x[*rs1*] em x[*rs2*] posições para à esquerda. Os bits vazios são preenchidos com zeros, e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. Os cinco bits menos significativos de x[*rs2*] formam o conjuto deslocado; os bits remanescentes são ignorados.

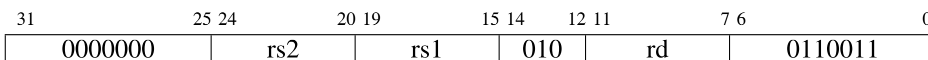


---

slt rd, rs1, rs2

x[rd] = x[rs1] $<_s$ x[rs2]

Set if Less Than. R-type, RV32I and RV64I.
Compara x[*rs1*] e x[*rs2*] como números de complemento de dois e escreve 1 em x[*rd*] se x[*rs1*] for menor, e 0 caso contrário.

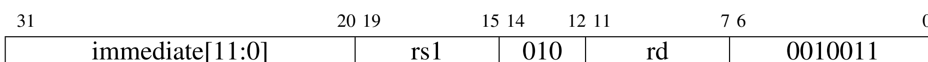


---

slti rd, rs1, immediate

x[rd] = x[rs1] $<_s$ sext(immediate)

Set if Less Than Immediate. I-type, RV32I and RV64I.
Compara x[*rs1*] e o valor de imediato com extensão de sinal como números de complemento de dois e escreve 1 em x[*rd*] se x[*rs1*] for menor, e 0 caso contrário.

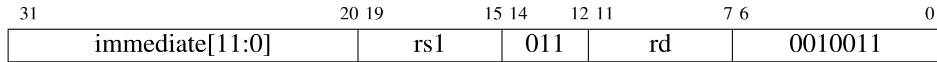


---

sltiu rd, rs1, immediate

$x[rd] = x[rs1] <_u sext(immediate)$

Set if Less Than Immediate, Unsigned. I-type, RV32I and RV64I.
Compara x[*rs1*] e o valor de imediato com extensão de sinal como números sem sinal e escreve 1 em x[*rd*] se x[*rs1*] for menor, e 0 caso contrário.

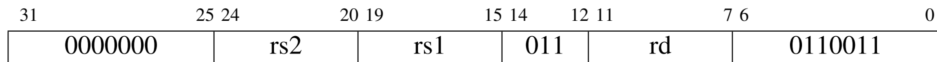


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
immediate[11:0]	rs1	011	rd	0010011

sltu rd, rs1, rs2

$x[rd] = x[rs1] <_u x[rs2]$

Set if Less Than, Unsigned. R-type, RV32I and RV64I.
Compara x[*rs1*] e x[*rs2*] como números sem sinal e escreve 1 em x[*rd*] se x[*rs1*] for menor, e 0 caso contrário.



31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
0000000	rs2	rs1	011	rd	0110011

sltz rd, rs1

$x[rd] = (x[rs1] <_s 0)$

Set if Less Than to Zero. Pseudoinstruction, RV32I e RV64I.
Escreve 1 em x[*rd*] se x[*rs1*] for menor que zero, e 0 caso contrário. Expande para **slt** rd, rs1, x0.

snez rd, rs2

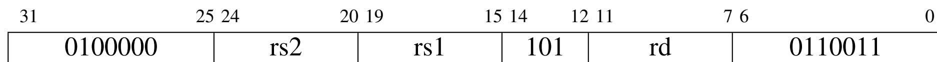
$x[rd] = (x[rs2] \neq 0)$

Set if Not Equal to Zero. Pseudoinstruction, RV32I e RV64I.
Escreve 0 em x[*rd*] se x[*rs2*] for igual a zero, e 1 caso contrário. Expande para **sltu** rd, x0, rs2.

sra rd, rs1, rs2

$x[rd] = x[rs1] >>_s x[rs2]$

Shift Right Arithmetic. R-type, RV32I and RV64I.
Desloca o conteúdo de x[*rs1*] em x[*rs2*] posições para à direita. Os bits vazios são preenchidos com cópias do bit mais significativo de x[*rs1*], e o resultado é gravado em x[*rd*]. Os cinco bits menos significativos de x[*rs2*](ou seis bits para RV64I) formam o conjunto deslocado; os bits remanescentes são ignorados.



31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
0100000	rs2	rs1	101	rd	0110011

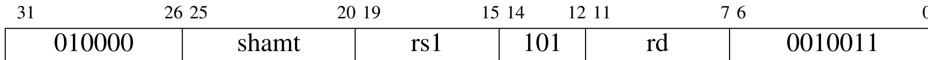
srai rd, rs1, shamt

`x[rd] = x[rs1] >>`$_s$ `shamt`

Shift Right Arithmetic Immediate. I-type, RV32I and RV64I.

Desloca o conteúdo do registrador x[*rs1*] em *shamt* posições para à direita. Os bits vazios são preenchidos com cópias do bit mais significativo de x[*rs1*], e o resultado é gravado em x[*rd*]. Para RV32I, a instrução só é permitida quando *shamt*[5] = 0.

Forma compactada: **c.srai** rd, shamt



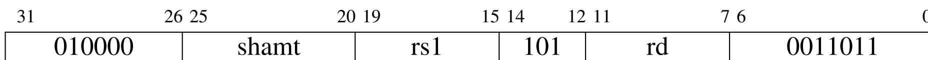
31 26	25 20	19 15	14 12	11 7	6 0
010000	shamt	rs1	101	rd	0010011

sraiw rd, rs1, shamt

`x[rd] = sext(x[rs1][31:0] >>`$_s$ `shamt)`

Shift Right Arithmetic Word Immediate. I-type, RV64I only.

Desloca os primeiros 32 bits de x[*rs1*] em *shamt* posições para à direita. Os bits vazios são preenchidos com cópias de x[*rs1*][31], e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. A instrução só é permitida quando *shamt*[5] = 0.



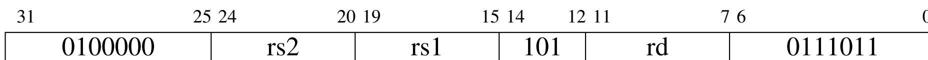
31 26	25 20	19 15	14 12	11 7	6 0
010000	shamt	rs1	101	rd	0011011

sraw rd, rs1, rs2

`x[rd] = sext(x[rs1][31:0] >>`$_s$ `x[rs2][4:0])`

Shift Right Arithmetic Word. R-type, RV64I only.

Desloca os primeiros 32 bits de x[*rs1*] para a direita pelas posições de bit x[*rs2*]. Os bits vazios são preenchidos com x[*rs1*][31] e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. Os cinco bits menos significativos de x [*rs2*] formam o conjunto deslocado; os bits remanescentes são ignorados.



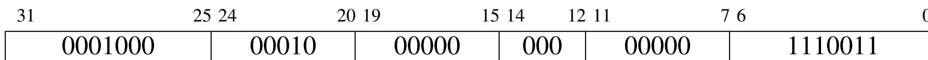
31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0111011

sret

`ExceptionReturn(Supervisor)`

Supervisor-mode Exception Return. R-type, RV32I and RV64I privileged architectures.

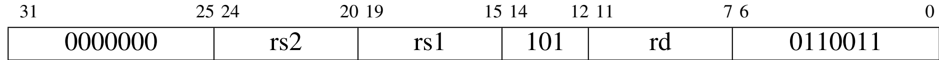
Retorna de um *handler* de exceção do modo supervisor. "Seta" o *PC* para CSRs[sepc], o modo de privilégio para CSRs[sstatus].SPP, CSRs[sstatus].SIE para CSRs[sstatus].SPIE, CSRs[sstatus].SPIE para 1 e CSRs[sstatus].SPP para 0.



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] >>`$_u$ `x[rs2]`

Shift Right Logical. R-type, RV32I and RV64I.
Desloca o conteúdo do registrador x[*rs1*] em x[*rs2*] posições para à direita. Os bits vazios são preenchidos com zeros e o resultado é gravado em x[*rd*]. Os cinco bits menos significativos de x[*rs2*](ou seis bits para RV64I) formam o conjunto deslocado; os bits remanescentes são ignorados.

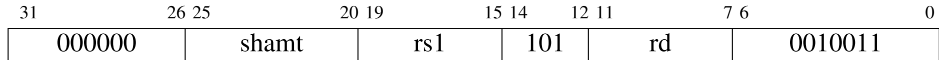


31–25	24–20	19–15	14–12	11–7	6–0
0000000	rs2	rs1	101	rd	0110011

---

srli rd, rs1, shamt — `x[rd] = x[rs1] >>`$_u$ `shamt`

Shift Right Logical Immediate. I-type, RV32I and RV64I.
Desloca o conteúdo do registrador x[*rs1*] em *shamt* posições para à direita. Os bits vazios são preenchidos com zeros e o resultado é gravado em x[*rd*]. Para RV32I, a instrução só é permitida quando *shamt*[5] = 0.
Forma compactada: **c.srli** rd, shamt

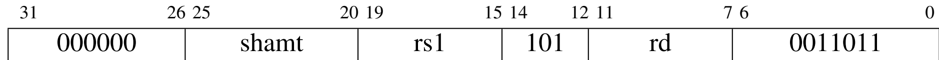


31–26	25–20	19–15	14–12	11–7	6–0
000000	shamt	rs1	101	rd	0010011

---

srliw rd, rs1, shamt — `x[rd] = sext(x[rs1][31:0] >>`$_u$ `shamt)`

Shift Right Logical Word Immediate. I-type, RV64I only.
Desloca os primeiros 32 bits de x[*rs1*] em *shamt* posições para à direita. Os bits vazios são preenchidos com zeros, e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. A instrução só é permitida quando *shamt*[5] = 0.

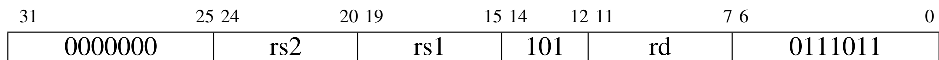


31–26	25–20	19–15	14–12	11–7	6–0
000000	shamt	rs1	101	rd	0011011

---

srlw rd, rs1, rs2 — `x[rd] = sext(x[rs1][31:0] >>`$_u$ `x[rs2][4:0])`

Shift Right Logical Word. R-type, RV64I only.
Desloca os primeiros 32 bits de x[*rs1*] em x[*rs2*] posições para à direita. Os bits vazios são preenchidos com zeros, e o resultado de 32 bits com extensão de sinal é gravado em x[*rd*]. Os cinco bits menos significativos de x[*rs2*] formam o conjunto deslocado; os bits remanescentes são ignorados.

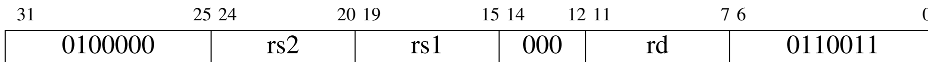


31–25	24–20	19–15	14–12	11–7	6–0
0000000	rs2	rs1	101	rd	0111011

---

sub rd, rs1, rs2 `x[rd] = x[rs1] - x[rs2]`

Subtract. R-type, RV32I and RV64I.
Subtrai o registrador x[*rs2*] do registrador x[*rs1*] e grava o resultado em x[*rd*]. O overflow aritmético é ignorado.
Forma compactada: **c.sub** rd, rs2

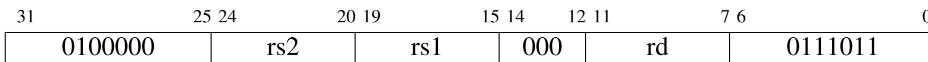


31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0110011

---

subw rd, rs1, rs2 `x[rd] = sext((x[rs1] - x[rs2])[31:0])`

Subtract Word. R-type, RV64I only.
Subtrai o registrador x[*rs2*] do registrador x[*rs1*], trunca o resultado para 32 bits e grava o resultado com extensão de sinal em x[*rd*]. O overflow aritmético é ignorado.
Forma compactada: **c.subw** rd, rs2



31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

---

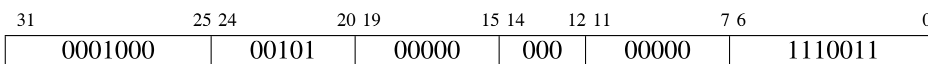
tail symbol `pc = &symbol; clobber x[6]`

Tail call. Pseudoinstruction, RV32I and RV64I.
"Seta" o *PC* como *symbol*, sobrescrevendo x[6] no processo. Expande para **auipc** x6, offsetHi e **jalr** x0, offsetLo (x6).

---

wfi `while (noInterruptsPending) idle`

Wait for Interrupt. R-type, RV32I and RV64I privileged architectures.
Ocupa o processador para economizar energia se nenhuma interrupção ativa estiver pendente no momento.

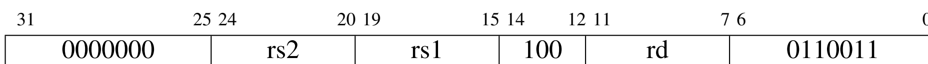


31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

---

xor rd, rs1, rs2 `x[rd] = x[rs1] ^ x[rs2]`

Exclusive-OR. R-type, RV32I and RV64I.
Calcula o OR-exclusivo bit a bit dos registradores x[*rs1*] e x[*rs2*] e grava o resultado em x[*rd*].
Forma compactada: **c.xor** rd, rs2



31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	100	rd	0110011

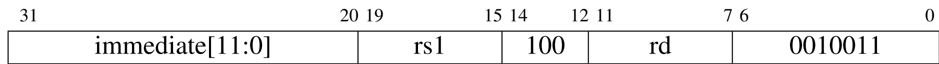
---

xori rd, rs1, immediate

`x[rd] = x[rs1] ^ sext(immediate)`

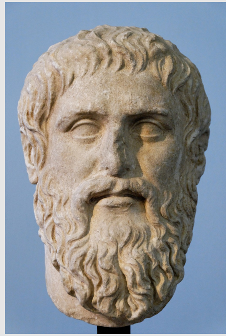
Exclusive-OR Immediate. I-type, RV32I and RV64I.

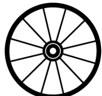
Calcula o OR-exclusivo bit a bit do *immediate* com extensão de sinal com o registrador x[*rs1*] e grava o resultado em x[*rd*].



B Transliteração do RISC-V

Plato (428–348 BCE) foi um filósofo grego que estabeleceu as bases para a matemática, a filosofia e a ciência ocidentais.





Simplicidade

Beauty of style and harmony and grace and good rhythm depend on simplicity.

— Plato, *The Republic*.

B.1 Introdução

Este apêndice inclui tabelas que transliteram instruções comuns e expressões idiomáticas em RV32I para códigos equivalentes em ARM-32 e x86-32. Nosso objetivo ao escrever este apêndice é ajudar programadores que não estão familiarizados com RISC-V, mas se sentem confortáveis com o ARM-32 ou x86-32, para ajudá-los a aprender RISC-V e a traduzir códigos em ISAs antigas em código básico RISC-V. O apêndice conclui com uma rotina C que percorre uma árvore binária e com código de assembly comentado para todas as três ISAs. Nós agendamos as três instruções de implementação o mais parecido possível para esclarecer suas similaridades.

As instruções de transferência de dados na Figura B.1 mostram a similaridade entre os loads e stores do RV32I e ARM-32 para o modo de endereçamento mais popular. Dada a orientação do registrador de memória do ISA x86, em vez da orientação de load-store das ISAs RV32I e ARM-32, o x86 transfere dados usando instruções de movimentação.

Além das instruções aritméticas de inteiros padrão, lógicas e de deslocamento, a Figura B.2 mostra como algumas operações comuns são feitas em cada ISA. Por exemplo, zerar um registrador usa a pseudo-instrução `li` em RV32I, uma instrução imediata de movimento em ARM-32 e por aplicação de OR exclusivo em um registrador com seu próprio

Descrição	RV32I	ARM-32	x86-32
Carrega palavra (Load word)	`lw t0, 4(t1)`	`ldr r0, [r1, #4]`	`mov eax, [edi+4]`
Carrega meia palavra (Load halfword)	`lh t0, 4(t1)`	`ldrsh r0, [r1, #4]`	`movsx eax,WORD PTR[edi+4]`
Carrega meia palavra sem sinal (Load halfword un.)	`lhu t0, 4(t1)`	`ldrh r0, [r1, #4]`	`movzx eax,WORD PTR[edi+4]`
Carrega byte (Load byte)	`lb t0, 4(t1)`	`ldrsb r0, [r1, #4]`	`movsx eax,BYTE PTR[edi+4]`
Carrega byte sem sinal (Load byte unsigned)	`lbu t0, 4(t1)`	`ldrb r0, [r1, #4]`	`movzx eax,BYTE PTR[edi+4]`
Armazena byte (Store byte)	`sb t0, 4(t1)`	`strb r0, [r1, #4]`	`mov [edi+4], al`
Armazena meia palavra (Store halfword)	`sh t0, 4(t1)`	`strh r0, [r1, #4]`	`mov [edi+4], ax`
Armazena palavra (Store word)	`sw t0, 4(t1)`	`str r0, [r1, #4]`	`mov [edi+4], eax`

Figura B.1: Instruções de acesso à memória RV32I transliteradas para ARM-32 e x86-32.

Descrição	RV32I	ARM-32	x86-32
Zera registrador	`li t0, 0`	`mov r0, #0`	`xor eax, eax`
Move registrador	`mv t0, t1`	`mov r0, r1`	`mov eax, edi`
Complementa registrador	`not t0, t1`	`mvn r0, r1`	`not eax, edi`
Nega registrador	`neg t0, t1`	`rsb r0, r1, #0`	`mov eax, edi` `neg eax`
Carrega constante grande	`lui t0, 0xABCDE` `addi t0, t0, 0x123`	`movw r0, #0xE123` `movt r0, #0xABCD`	`mov eax, 0xABCDE123`
Move o PC para o Registrador	`auipc t0, 0`	`ldr r0, [pc, #-8]`	`call 1f` `1: pop eax`
Adição	`add t0, t1, t2`	`add r0, r1, r2`	`lea eax, [edi+esi]`
Add (im.)	`addi t0, t0, 1`	`add r0, r0, #1`	`add eax, 1`
Subtração	`sub t0, t0, t1`	`sub r0, r0, r1`	`sub eax, edi`

"Seta" reg. para (reg=0)	`sltiu t0, t1, 1`	`rsbs r0, r1, #1`
		`movcc r0, #0`
			`xor eax, eax`
			`test edx, edx`
			`sete al`

"Seta" reg. para (reg≠0)	`sltu t0, x0, t1`	`adds r0, r1, #0`
		`movne r0, #1`
			`xor eax, eax`
			`test edx, edx`
			`setne al`

OR Bitwise	`or t0, t0, t1`	`orr r0, r0, r1`	`or eax, edi`
AND Bitwise	`and t0, t0, t1`	`and r0, r0, r1`	`and eax, edi`
XOR Bitwise	`xor t0, t0, t1`	`eor r0, r0, r1`	`xor eax, edi`
OR Bitwise (im.)	`ori t0, t0, 1`	`orr r0, r0, #1`	`or eax, 1`
AND Bitwise (im.)	`andi t0, t0, 1`	`and r0, r0, #1`	`and eax, 1`
XOR Bitwise (im.)	`xori t0, t0, 1`	`eor r0, r0, #1`	`xor eax, 1`
Deslocamento para esquerda	`sll t0, t0, t1`	`lsl r0, r0, r1`	`sal eax, cl`
Deslocamento lógico para direita	`srl t0, t0, t1`	`lsr r0, r0, r1`	`shr eax, cl`
Deslocamento aritmético à direita.	`sra t0, t0, t1`	`asr r0, r0, r1`	`sar eax, cl`
Deslocamento a esquerda (im.)	`slli t0, t0, 1`	`lsl r0, r0, #1`	`sal eax, 1`
Deslocamento lógico à direita (im.)	`srli t0, t0, 1`	`lsr r0, r0, #1`	`shr eax, 1`
Deslocamento aritmético à direita (im.)	`srai t0, t0, 1`	`asr r0, r0, #1`	`sar eax, 1`

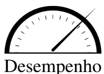
Figura B.2: Instruções aritméticas RV32I transliteradas em ARM-32 e x86-32. O formato de instrução de dois operandos x86-32 geralmente precisa de mais instruções do que o formato de instrução de três operandos de ARM-32 e RV32I.

Descrição	RV32I	ARM-32	x86-32
Desvia se $=$	`beq t0, t1, foo`	`cmp r0, r1` `beq foo`	`cmp eax, esi` `je foo`
Desvia se $\neq$	`bne t0, t1, foo`	`cmp r0, r1` `bne foo`	`cmp eax, esi` `jne foo`
Desvia se $<$	`blt t0, t1, foo`	`cmp r0, r1` `blt foo`	`cmp eax, esi` `jl foo`
Desvia se $\geq_s$	`bge t0, t1, foo`	`cmp r0, r1` `bge foo`	`cmp eax, esi` `jge foo`
Desvia se $<_u$	`bltu t0, t1, foo`	`cmp r0, r1` `bcc foo`	`cmp eax, esi` `jb foo`
Desvia se $\geq_u$	`bgeu t0, t1, foo`	`cmp r0, r1` `bcs foo`	`cmp eax, esi` `jnb foo`
Desvia se $=0$	`beqz t0, foo`	`cmp r0, #0` `beq foo`	`test eax, eax` `je foo`
Desvia se $\neq 0$	`bnez t0, foo`	`cmp r0, #0` `bne foo`	`test eax, eax` `jne foo`
Salto direto ou chamada no final (tail call)	`jal x0, foo`	`b foo`	`jmp foo`
Chamada de sub-rotina	`jal ra, foo`	`bl foo`	`call foo`
Retorno de sub-rotina	`jalr x0, 0(ra)`	`bx lr`	`ret`
Chamada indireta	`jalr ra, 0(t0)`	`blx r0`	`call eax`
Salto indireto ou chamada no final (tail call)	`jalr x0, 0(t0)`	`bx r0`	`jmp eax`

Figura B.3: Instruções de fluxo de controle RV32I transliteradas em ARM-32 e x86-32. A instrução de comparação e desvio do RV32I leva metade do número de instruções dos desvios baseadas no código de condição do ARM-32 e do x86-32.

valor em x86-32. O limite de dois operandos de instruções x86-32 significa mais instruções em alguns casos, embora o formato de instrução de comprimento variável permita carregar uma grande constante em uma única instrução. As instruções convencionais add, subtract, logical e shift - que são responsáveis pela maioria das instruções executadas - mapeiam-se um para um entre as ISAs.

A Figura B.3 lista as instruções de desvio e chamadas condicionais e incondicionais. A abordagem do código de condição para desvios condicionais requer duas instruções para o ARM-32 e x86-32, enquanto o RV32I precisa de apenas uma. Como o Capítulo 2 ilustra, da Figura 2.5 até 2.11, apesar de sua abordagem minimalista ao projeto de conjuntos de instruções, os desvios de comparação e execução do RISC-V reduzem o número de instruções na Ordenação por Inserção tanto quanto os modos de endereço mais sofisticados e as instruções push e pop do ARM-32 e x86-32 .



B.2 Comparando RV32I, ARM-32 e x86-32 usando o algoritmo de soma de árvore

A Figura B.4 é o nosso programa C de exemplo que usamos para comparar os três ISAs lado a lado, da Figura B.5 até B.7. Ele soma os valores em uma árvore binária, usando uma travessia de árvore em ordem. As árvores são uma estrutura de dados fundamental e, embora essa operação de árvore possa parecer excessivamente simplista, nós a escolhemos porque demonstra tanto a recursão quanto a iteração em apenas algumas instruções assembly.

A rotina utiliza uma abordagem recursiva para calcular a soma da subárvore esquerda, mas usa a iteração para calcular a soma da subárvore direita, o que reduz o espaço de memória utilizado e o número de instruções. A otimização de compiladores pode transformar o código totalmente recursivo nessa versão; mostramos explicitamente a iteração para maior clareza.

As maiores diferenças entre o tamanho dos três programas em linguagem assembly estão na entrada e saída da função. O RISC-V usa quatro instruções para salvar e restaurar três registradores na pilha e ajustar o ponteiro da pilha. x86-32 salva e restaura apenas dois registradores na pilha, pois pode realizar operações aritméticas em operandos de memória, em vez de carregá-los para registradores. Ele também os salva e restaura usando as instruções push e pop, que ajusta implicitamente o ponteiro da pilha em vez de explicitamente como no RISC-V. O ARM-32 pode salvar três registradores mais o registrador de link com o endereço de retorno na pilha em uma única instrução push e restaurá-los com uma única instrução pop.

O RISC-V executa o próprio loop principal em sete instruções, em vez de oito nas outras ISAs, porque como mostra a Figura B.3, ele pode comparar e desviar em uma única instrução enquanto essa operação leva duas instruções para o ARM-32 e x86-32. O restante das instruções no loop mapeia um-para-um entre o RV32I e o ARM-32, como ilustram as Figuras B.1 e B.2. Uma diferença é que as instruções `call` e `ret` do x86 implicitamente adicionam e removem o endereço de retorno da pilha, enquanto as outras ISAs o fazem explicitamente em seus prólogos e epílogos (salvando e restaurando `ra` no RV32I, ou adicionando `lr` e movendo para o `PC` no ARM-32). Além disso, como a convenção de chamada x86-32 transmite argumentos na pilha, o código x86-32 tem uma instrução `push` e `pop` no loop que as outras ISAs podem evitar. A transferência extra de dados reduz o desempenho.

B.3 Conclusão

Facilidade de Programação

Apesar das filosofias ISA amplamente diferentes, os programas resultantes são bastante semelhantes, tornando simples a conversão das versões do programa das arquiteturas mais antigas para o RISC-V. Possuir 32 registradores no RISC-V, versus 16 no ARM-32, e 8 no x86-32 simplifica a conversão para RISC-V, o que seria muito mais difícil na outra direção. Primeiro, ajuste os prólogos e epílogos de função, depois altere os desvios condicionais do código de condição orientado para as instruções de comparação e desvio e, finalmente, substitua todos os nomes de registradores e instruções pelos equivalentes RISC-V. Podem haver mais alguns ajustes restantes, como o tratamento de constantes longas e endereços no ISA x86-32 de comprimento variável, ou a adição de instruções RISC-V para realizar os modos de endereçamento extravagantes se usado em transferências de dados, mas a sua tradução está bem próxima após seguir apenas esses três passos.

```
struct tree_node {
  struct tree_node *left;
  struct tree_node *right;
  long value;
};

long tree_sum(const struct tree_node *node)
{
  long result = 0;
  while (node) {
    result += tree_sum(node->left);
    result += node->value;
    node = node->right;
  }
  return result;
}
```

Figura B.4: Uma rotina em C que soma os valores em uma árvore binária, usando uma travessia em ordem.

```
  addi sp,sp,-16  # Aloca quadro de pilha
  sw   s1,4(sp)   # Preserva s1
  sw   s0,8(sp)   # Preserva s0
  sw   ra,12(sp)  # Preserva ra
  li   s1,0       # soma = 0
  beqz a0,.L1     # Pula o laço se nodo == 0
  mv   s0,a0      # s0 = node
.L3:
  lw   a0,0(s0)   # a0 = nodo->esquerda
  jal  tree_sum   # Recurso; resulta em a0
  lw   a5,8(s0)   # a5 = nodo->valor
  lw   s0,4(s0)   # nodo = nodo->direita
  add  s1,a0,s1   # soma += a0
  add  s1,s1,a5   # soma += a5
  bnez s0,.L3     # Laço se nodo != 0
.L1:
  mv   a0,s1      # Retorna a soma em a0
  lw   s1,4(sp)   # Restaura s1
  lw   s0,8(sp)   # Restaura s0
  lw   ra,12(sp)  # Restaura ra
  addi sp,sp,16   # Desaloca quadro de pilha
  ret             # Retorno
```

Figura B.5: Código RV32I para percorrer da árvore em ordem. O loop principal é menor do que as versões para os outros dois ISAs devido a instrução de comparar e desviar `bnez`.

```
  push {r4, r5, r6, lr}  # Preserva regs
  mov  r5, #0            # soma = 0
  subs r4, r0, #0        # r4 = nodo; nodo == 0?
  beq  .L1               # Pula o laço se sim
.L3:
  ldr  r0, [r4]          # r0 = nodo->esquerda
  bl   tree_sum          # Recurso; resulta em r0
  ldr  r3, [r4, #8]      # r3 = nodo->valor
  ldr  r4, [r4, #4]      # r4 = nodo->direita
  add  r5, r0, r5        # soma += r0
  add  r5, r5, r3        # soma += r3
  cmp  r4, #0            # nodo == 0?
  bne  .L3               # Laço se não
.L1:
  mov  r0, r5            # Retorna soma em r0
  pop  {r4, r5, r6, pc}  # Restaura regs e retorna
```

Figura B.6: Código ARM-32 para passagem de árvore em ordem. As instruções push e pop multiword reduzem o tamanho do código para o ARM-32 comparado às outras ISAs.

```
  push esi             # Preserva esi
  push ebx             # Preserva ebx
  xor  esi, esi        # soma = 0
  mov  ebx, [esp+12]   # ebx = nodo
  test ebx, ebx        # nodo == 0?
  je   .L1             # Pula se sim
.L3:
  push [ebx]           # Carrega nodo->esquerda; empurra para a pilha
  call tree_sum        # Recurso; resulta em eax
  pop  edx             # Dá Pop no argumento antigo e descarta
  add  esi, [ebx+8]    # soma += nodo->valor
  mov  ebx, [ebx+4]    # node = nodo->direita
  add  esi, eax        # soma += eax
  test ebx, ebx        # nodo == 0?
  jne  .L3             # Laço se não
.L1:
  mov  eax, esi        # Retorna soma em eax
  pop  ebx             # Restaura ebx
  pop  esi             # Restaura esi
  ret                  # Retorno
```

Figura B.7: Código x86-32 para percorrer a árvore em ordem. O loop principal tem instruções push e pop não encontradas nas versões do programa para os outros ISAs, que criam tráfego extra de dados.

Índice

ABI, *see also* application binary interface
Add, 20, *129*
  immediate, 20, *129*
  immediate word, 90, *129*
  upper immediate to PC, *133*
  word, 90, *129*
add, 20, *see also* c.add, *129*
Add upper immediate to PC, 22
addi, *see also* Add immediate, *see also* c.addi16sp, *see also* c.addi4spn, *see also* c.addi, *see also* c.li, *129*
addiw, *see also* Add immediate word, *see also* c.addiw, *129*
addw, *see also* Add word, *see also* c.addw, *129*
ALGOL, *124*
Allen, Fran, *16*
AMD64, 96
amoadd.d, *see also* Atomic Memory Operation Add Doubleword, *129*
amoadd.w, *see also* Atomic Memory Operation Add Word, *130*
amoand.d, *see also* Atomic Memory Operation And Doubleword, *130*
amoand.w, *see also* Atomic Memory Operation And Word, *130*
amomax.d, *see also* Atomic Memory Operation Maximum Doubleword, *130*
amomax.w, *see also* Atomic Memory Operation Maximum Word, *130*
amomaxu.d, *see also* Atomic Memory Operation Maximum Unsigned Doubleword, *131*
amomaxu.w, *see also* Atomic Memory Operation Maximum Unsigned Word, *131*
amomin.d, *see also* Atomic Memory Operation Minimum Doubleword, *131*
amomin.w, *see also* Atomic Memory Operation Minimum Word, *131*
amominu.d, *see also* Atomic Memory Operation Minimum Unsigned Doubleword, *131*
amominu.w, *see also* Atomic Memory Operation Minimum Unsigned Word, *132*
amoor.d, *see also* Atomic Memory Operation Or Doubleword, *132*
amoor.w, *see also* Atomic Memory Operation Or Word, *132*
amoswap.d, *see also* Atomic Memory Operation Swap Doubleword, *132*
amoswap.w, *see also* Atomic Memory Operation Swap Word, *132*
amoxor.d, *see also* Atomic Memory Operation Exclusive Or Doubleword, *133*
amoxor.w, *see also* Atomic Memory Operation Exclusive Or Word, *133*
And, 20, *133*
  immediate, 20, *133*
and, *see also* c.and, *133*
andi, 20, *see also* And immediate, *see also* c.andi, *133*
application binary interface, 20, 29, 36, 53
Application Specific Integrated Circuits, *2*
architecture, *8*
ARM
  code size, 9, 96
  Cortex-A5, 7
  Cortex-A9, 8
  DAXPY, **59**
  Insertion Sort, **26**
  instruction reference manual
    number of pages, 13
  Load Multiple, 7, 9
  number of registers, 11
  Thumb, 9
  Thumb-2, 9
  Tree Sum, **180**
ARMv8, **96**
ASIC, *see also* Application Specific Integrated Circuits
assembler directives, *42*, 42
Atomic Memory Operation
  Add
    Doubleword, 90, *129*
    Word, 64, *130*
  And
    Doubleword, 90, *130*
    Word, 64, *130*
  Exclusive Or
    Doubleword, 90, *133*
    Word, 64, *133*
  Maximum
    Doubleword, 90, *130*
    Word, 64, *130*
  Maximum Unsigned
    Doubleword, 90, *131*
    Word, 64, *131*
  Minimum
    Doubleword, 90, *131*
    Word, 64, *131*
  Minimum Unsigned
    Doubleword, 90, *131*
    Word, 64, *132*
  Or
    Doubleword, 90, *132*
    Word, 64, *132*
  Swap
    Doubleword, 90, *132*
    Word, 64, *132*
auipc, *see also* Add upper immediate to PC, *133*

backwards binary-compatibility, *3*
Bell, C. Gordon, 50, *90*
beq, *see also* Branch if equal, *see also* c.beqz, *134*

beqz, *37*, *134*
bge, *see also* Branch if greater or equal, *134*
bgeu, *see also* Branch if greater or equal unsigned, *134*
bgez, *37*, *134*
bgt, *37*, *134*
bgtu, *37*, *134*
bgtz, *37*, *135*
ble, *37*, *135*
bleu, *37*, *135*
blez, *37*, *135*
blt, *see also* Branch if less than, *135*
bltu, *see also* Branch if less than unsigned, *135*
bltz, *37*, *135*
bne, *see also* Branch if not equal, *see also* c.bnez, *136*
bnez, *37*, *136*
Branch
  if equal, 24, *134*
  if greater or equal, 24, *134*
  if greater or equal unsigned, 24, *134*
  if less than, 24, *135*
  if less than unsigned, 24, *135*
  if not equal, 24, *136*
branch prediction, 8, 20
Brooks, Fred, 120
Browning, Robert, *60*

C programs
  DAXPY, **59**
  Insertion Sort, **26**
  Tree Sum, **180**
c.add, *see also* add, *136*
c.addi, *see also* addi, *136*
c.addi16sp, *see also* addi, *136*
c.addi4spn, *see also* addi, *136*
c.addiw, 90, *see also* addiw, *137*
c.addw, 90, *see also* addw, *137*
c.and, *see also* and, *137*
c.andi, *see also* andi, *137*
c.beqz, *see also* beq, *137*
c.bnez, *see also* bne, *137*
c.ebreak, *see also* ebreak, *138*
c.fld, *see also* fld, *138*
c.fldsp, *see also* fld, *138*
c.flw, *see also* flw, *138*
c.flwsp, *see also* flw, *138*
c.fsd, *see also* fsd, *138*
c.fsdsp, *see also* fsd, *139*
c.fsw, *see also* fsw, *139*
c.fswsp, *see also* fsw, *139*
c.j, *see also* jal, *139*
c.jal, *see also* jal, *139*
c.jalr, *see also* jalr, *139*
c.jr, *see also* jalr, *140*
c.ld, 90, *see also* ld, *140*
c.ldsp, 90, *see also* ld, *140*
c.li, *see also* addi, *140*
c.lui, *see also* lui, *140*
c.lw, *see also* lw, *140*
c.lwsp, *see also* lw, *141*
c.mv, *see also* add, *141*
c.or, *see also* or, *141*
c.sd, 90, *see also* sd, *141*
c.sdsp, 90, *see also* sd, *141*
c.slli, *see also* slli, *141*
c.srai, *see also* srai, *142*
c.srli, *see also* srli, *142*
c.sub, *see also* sub, *142*
c.subw, 90, *see also* subw, *142*
c.sw, *see also* sw, *142*
c.swsp, *see also* sw, *142*
c.xor, *see also* xor, *143*
call, *37*, *143*
Callee saved registers, *36*
Caller saved registers, *36*
Calling conventions, **34**
Chanel, Coco, 128
chip, *see also* die, *7*
Compilers
  Turing Award, *124*
context switch, *79*
Control and Status Register
  `marchid`, *120*, 120
  `mcause`, *107*, 107, 109, 110
  `mcounteren`, *120*, 120
  `mcycle`, *120*, 120
  `medeleg`, *114*, 114
  `mepc`, *107*, 109, 110
  `mhartid`, *120*, 120
  `mhpmcounter`*i*, *120*, 120
  `mhpmevent`*i*, *120*, 120
  `mideleg`, *114*
  `mie`, *107*, 107, 109, 110
  `mimpid`, *120*, 120
  `minstret`, *120*, 120
  `mip`, *107*, 107, 109, 110
  `misa`, 119, *120*
  `mscratch`, *107*, 109, 110
  `mstatus`, *107*, 107, 109, 110, 112
  `mtimecmp`, *120*, 120
  `mtime`, *120*, 120
  `mtval`, *107*, 109, 110
  `mtvec`, *107*, 107, 109, 110
  `mvendorid`, 119, *120*
  `pmpcfg`, *113*
  `satp`, *117*, 117
  `scause`, *107*, 115
  `scounteren`, *120*, 120
  `sedeleg`, *114*
  `sepc`, *107*, 115
  `sideleg`, *114*
  `sie`, *114*, 114
  `sip`, *114*, 114
  `sscratch`, *107*, 115
  `sstatus`, *114*, 115
  `stval`, *107*, 115
  `stvec`, *107*, 115
  read and clear, 25, *143*
  read and clear immediate, 25, *144*
  read and set, 25, *144*
  read and set immediate, 25, *144*
  read and write, 25, *144*
  read and write immediate, 25, *144*
CoreMark benchmark, 8
cost, *see also* instruction set architecture, principles of design, cost
Cray, Seymour, *76*, 96
CSR, *see* Control and Status Register
csrc, *37*, *143*
csrci, *37*, *143*
csrr, *37*, *143*
csrrc, *see also* Control and Status Register read and clear, *143*
csrrci, *see also* Control and Status Register read and clear immediate, *144*
csrrs, *see also* Control and Status Register read and set, *144*
csrrsi, *see also* Control and Status Register read and set immediate, *144*
csrrw, *see also* Control and Status Register read and write, *144*
csrrwi, *see also* Control and Status Register read and write immediate, *144*
csrs, *37*, *145*
csrsi, *37*, *145*
csrw, *37*, *145*
csrwi, *37*, *145*

da Vinci, Leonardo, *2*
data-level parallelism, *76*
DAXPY, **59**
de Saint-Exupéry, Antoine, *52*
delay slot, 8
delayed branch, 8
die, *see also* chip, *7*
  yield, *7*
div, *145*
Divide, 49, *145*
  unsigned, 49, *145*
  unsigned word, 90, *146*
  using shift right, 49
  word, 90, *146*
divu, *see also* Divide unsigned, *145*
divuw, *see also* Divide unsigned word, *146*
divw, *see also* Divide word, *146*
dynamic linking, **45**
dynamic register typing, *78*, 94

ease of programming, compiling, and linking, *see also* instruction set architecture, principles of design, ease of programming, compiling, and linking
ebreak, *146*
ecall, *146*
Einstein, Albert, *64*
ELF, *see also* executable and linkable format
endianness, 24
epilogue, *see also* function epilogue
Exception, *105*
Exception Return
  Machine, *110*, *165*
  Supervisor, *115*, *173*
Exclusive Or, 20, *175*
  immediate, 20, *176*
executable and linkable format, *42*

fabs.d, *37*, *146*
fabs.s, *37*, *146*
fadd.d, *see also* Floating-point Add double-precision, *147*
fadd.s, *see also* Floating-point Add single-precision, *147*
fclass.d, *see also* Floating-point Classify double-precision, *147*
fclass.s, *see also* Floating-point Classify single-precision, *147*
fcvt.d.l, *see also* Floating-point Convert double from long
fcvt.d.lu, *see also* Floating-point Convert double from long unsigned, *148*
fcvt.d.s, *see also* Floating-point Convert double from single, *148*
fcvt.d.w, *see also* Floating-point Convert double from word, *148*
fcvt.d.wu, *see also* Floating-point Convert double from word unsigned, *148*
fcvt.dl, *148*
fcvt.l.d, *see also* Floating-point Convert long from double, *149*
fcvt.l.s, *see also* Floating-point Convert long from single, *149*
fcvt.lu.d, *see also* Floating-point Convert long unsigned from double, *149*
fcvt.lu.s, *see also* Floating-point Convert long unsigned from single, *149*
fcvt.s.d, *see also* Floating-point Convert single from double, *149*
fcvt.s.l, *see also* Floating-point Convert single from long, *150*
fcvt.s.lu, *see also* Floating-point Convert single from long unsigned, *150*
fcvt.s.w, *150*
fcvt.s.wu, *see also* Floating-point Convert single from word unsigned, *150*
fcvt.w.d, *see also* Floating-point Convert word from double, *150*
fcvt.w.s, *see also* Floating-point Convert word from single, *151*
fcvt.wu.d, *see also* Floating-point Convert word unsigned from double, *151*
fcvt.wu.s, *see also* Floating-point Convert word unsigned from single, *151*
fdiv.d, *see also* Floating-point Divide double-precision, *151*
fdiv.s, *see also* Floating-point Divide single-precision, *151*
Fence
  Instruction Stream, *152*
  Memory and I/O, *152*
  Virtual Memory, *119*, *170*
fence, *37*, *see also* Fence Memory and I/O, *152*
fence.i, *see also* Fence Instruction Stream, *152*
feq.d, *see also* Floating-point Equals double-precision, *152*
feq.s, *see also* Floating-point Equals single-precision, *152*
Field-Programmable Gate Array, *2*
fld, *see also* c.fldsp, *see also* c.fld, *see also* Floating-point load doubleword, *153*
fle.d, *see also* Floating-point Less or Equals double-precision, *see also* Floating-point Less Than double-precision, *153*
fle.s, *see also* Floating-point Less or Equals single-precision, *see also* Floating-point Less Than single-precision, *153*
Floating-Point
  dynamic rounding mode, *53*
  fused multiply-add, *57*
  IEEE 754-2008 floating-point standard, 52
  sign-injection, *58*
  static rounding mode, *53*
Floating-point, **52**
  Add
    double-precision, 52, *147*
    single-precision, 52, *147*
  binary128, **60**
  binary16, **60**
  binary256, **60**
  binary32, **60**
  binary64, **60**
  Classify
    double-precision, 52, *147*
    single-precision, 52, *147*
  Convert
    double from long, 52, 90, *148*
    double from long unsigned, 52, 90, *148*
    double from single, 52, *148*
    double from word, 52, *148*
    double from word unsigned, 52, *148*
    long from double, 52, 90, *149*
    long from single, 52, 90, *149*
    long unsigned from double, 52, 90, *149*
    long unsigned from single, 52, 90, *149*
    single from double, 52, *149*
    single from long, 52, 90, *150*
    single from long unsigned, 52, 90, *150*
    single from word unsigned, 52, *150*
    word from double, 52, *150*
    word from single, 52, *151*
    word unsigned from double, 52, *151*
    word unsigned from single, 52, *151*
  decimal128, **60**
  decimal32, **60**
  decimal64, **60**
  Divide
    double-precision, 52, *151*
    single-precision, 52, *151*
  Equals
    double-precision, 52, *152*
    single-precision, 52, *152*
  Fused multiply-add
    double-precision, 52, *154*
    single-precision, 52, *154*
  Fused multiply-subtract
    double-precision, 52, *155*
    single-precision, 52, *155*
  Fused negative multiply-add
    double-precision, 52, *157*
    single-precision, 52, *157*
  Fused negative multiply-subtract
    double-precision, 52, *158*
    single-precision, 52, *158*
  half precision, **60**
  Less or Equals
    double-precision, 52, *153*
    single-precision, 52, *153*
  Less Than
    double-precision, 52, *153*
    single-precision, 52, *153*
  Load
    doubleword, 52, *153*
    word, 52, *154*

Maximum
double-precision, 52, *154*
single-precision, 52, *154*
Minimum
double-precision, 52, *155*
single-precision, 52, *155*
Move
doubleword from integer, 52, *156*
doubleword to integer, 52, *156*
word from integer, 52, *156*
word to integer, 52, *157*
Multiply
double-precision, 52, *155*
single-precision, 52, *156*
octuple precision, **60**
quadruple precision, **60**
Sign-inject
double-precision, 52, *159*
single-precision, 52, *159*
Sign-inject negative
double-precision, 52, *159*
single-precision, 52, *160*
Sign-inject XOR
double-precision, 52, *160*
single-precision, 52, *160*
Square root
double-precision, 52, *160*
single-precision, 52, *160*
Store
doubleword, 52, *159*
word, 52, *161*
Subtract
double-precision, 52, *161*
single-precision, 52, *161*
floating-point
control and status register, *53*
flt.d, *153*
flt.s, *153*
flw, *see also* c.flwsp, *see also* c.flw, *see also* Floating-point load word, *154*
fmadd.d, *see also* Floating-point fused multiply-add double-precision, *154*
fmadd.s, *see also* Floating-point fused multiply-add single-precision, *154*
fmax.d, *see also* Floating-point maximum double-precision, *see also* Floating-point maximum single-precision, *154*
fmax.s, *154*
fmin.d, *see also* Floating-point minimum double-precision, *155*
fmin.s, *see also* Floating-point minimum single-precision, *155*
fmsub.d, *see also* Floating-point fused multiply-subtract double-precision, *155*
fmsub.s, *see also* Floating-point fused multiply-subtract single-precision, *155*
fmul.d, *see also* Floating-point Multiply double-precision, *155*
fmul.s, *see also* Floating-point Multiply single-precision, *156*, *see also* Floating-point Subtract single-precision
fmv.d, *37*, *156*
fmv.d.x, *see also* Floating-point move doubleword from integer, *156*
fmv.s, *37*, *156*
fmv.w.x, *see also* Floating-point move word from integer, *156*
fmv.x.d, *see also* Floating-point move doubleword to integer, *156*
fmv.x.w, *see also* Floating-point move word to integer, *157*
fneg.d, *37*, *157*
fneg.s, *37*, *157*
fnmadd.d, *see also* Floating-point fused negative multiply-add double-precision, *see also* Floating-point fused negative multiply-add single-precision, *157*
fnmadd.s, *157*
fnmsub.d, *see also* Floating-point fused negative multiply-subtract double-precision, *158*
fnmsub.s, *see also* Floating-point fused negative multiply-subtract single-precision, *158*
FPGA, *see also* Field-Programmable Gate Array, *2*
frcsr, *37*, *158*
frflags, *37*, *158*
frrm, *37*, *158*
fscsr, *37*, *158*
fsd, *see also* c.fsdsp, *see also* c.fsd, *see also* Floating-point store doubleword, *159*
fsflags, *37*, *159*
fsgnj.d, *see also* Floating-point Sign-inject double-precision, *159*
fsgnj.s, *see also* Floating-point Sign-inject single-precision, *159*
fsgnjn.d, *see also* Floating-point Sign-inject negative double-precision, *159*
fsgnjn.s, *see also* Floating-point Sign-inject negative single-precision, *160*
fsgnjx.d, *see also* Floating-point Sign-inject XOR double-precision, *160*
fsgnjx.s, *see also* Floating-point Sign-inject XOR single-precision, *160*
fsqrt.d, *see also* Floating-point Square Root double-precision, *160*
fsqrt.s, *see also* Floating-point Square Root single-precision, *160*
fsrm, *37*, *161*
fsub.d, *see also* Floating-point Subtract double-precision, *161*
fsub.s, *161*
fsw, *see also* c.fswsp, *see also* c.fsw, *see also* Floating-point store word, *161*
function epilogue, *37*
function prologue, *37*
Fused multiply-add, *57*

gather, *80*

Hart, *105*

IEEE 754-2008 floating-point standard, 52
Illiac IV, *85*
implementation, *8*
Insertion Sort, **26**
Instruction diagram
Privileged instructions, **105**
RV32A, **64**
RV32C, **68**
RV32D, **52**
RV32F, **52**
RV32I, **16**
RV32M, **48**
RV64A, **90**
RV64C, **90**
RV64D, **90**
RV64F, **90**
RV64I, **90**
RV64M, **90**
instruction set architecture, *2*
backwards binary-compatibility, *3*
elegance, 14, 27, 46, 71, 86, 98, 126
incremental, **3**
metrics of design, **7**
cost, **7**, 16, 19, 22, 23, 27, 50, 69, 96, 119
ease of programming, compiling, and linking, **9**, 19, 20, 22–24, 27, 58, 76, 78, 80, 86, 94–96, 110, 120
isolation of architecture from implementation, **8**, 27, 76, 86, 105, 125
performance, **7**, 16, 19, 27, 49, 52, 57, 59, 65, 76, 80–82, 84, 86, 95, 96
program size, **9**, 27, 68, 71, 94, 96, 98
room for growth, **9**, 19, 27, 98

simplicity, **7**, 12, 13, 20, 22–27, 59, 65, 68, 77, 86, 110, 115, 120, 125
mistakes of the past, **27**
modularity, **5**
open, **3**
principles of design
cost, 46
ease of programming, compiling, and linking, 42, 46, 124
performance, 34, 46, 124, 125
room for growth, 125
simplicity, 37
Interrupt, *107*
ISA, *see* instruction set architecture
isolation of architecture from implementation, *see also* instruction set architecture, principles of design, isolation of architecture from implementation
Itanium, **95**

j, *37*, *161*
jal, *37*, *see also* c.jal, *see also* c.j, *see also* Jump and link, *162*
jalr, *37*, *see also* c.jalr, *see also* c.jr, *see also* Jump and link register, *162*
Johnson, Kelly, *46*
jr, *37*, *162*
Jump and link, 25, *162*
register, 25, *162*

la, *162*
lb, *see also* Load byte, *162*
lbu, *see also* Load byte unsigned, *163*
ld, *see also* c.ldsp, *see also* c.ld, *see also* Load doubleword, *163*
leaf function, *36*
lh, *see also* Load halfword, *163*
lhu, *see also* Load halfword unsigned, *163*
li, *37*, *163*
Lindy effect, *27*
linker relaxation, **45**
little-endian, 24
lla, *164*
Load
byte, 23, *162*
byte unsigned, 23, *163*
doubleword, 90, *163*
halfword, 23, *163*
halfword unsigned, 23, *163*
reserved
doubleword, 90, *164*
word, 64, *164*
upper immediate, 22, *165*
word, 23, *164*
word unsigned, 23, *164*
Load upper immediate, 22

lr.d, *see also* Load reserved doubleword, *164*
lr.w, *see also* Load reserved word, *164*
lui, *see also* c.lui, *see also* Load upper immediate, *165*
lw, *see also* c.lwsp, *see also* c.lw, *see also* Load word, *164*
lwu, *see also* Load word unsigned, *164*

Machine mode, *105*
macrofusion, *7*, 7, 70
marchid, *see* Control and Status Register
mcause, *see* Control and Status Register
mcounteren, *see* Control and Status Register
mcycle, *see* Control and Status Register
mepc, *see* Control and Status Register
metrics of ISA design, *see also* instruction set architecture, metrics of design
mhartid, *see* Control and Status Register
mhpmcounter*i*, *see* Control and Status Register
mhpmevent*i*, *see* Control and Status Register
microMIPS, 59
mie, *see* Control and Status Register
mimpid, *see* Control and Status Register
minstret, *see* Control and Status Register
mip, *see* Control and Status Register
MIPS
assembler, 22
DAXPY, **59**
delayed branch, 8, 24, 29, 60, 98
delayed load, 23, 29, 98
Insertion Sort, **26**
MIPS MSA, 85
MIPS-IV, **96**
misa, *see* Control and Status Register
Moore's Law, 2
mret, *see also* Exception Return Machine, *165*
mscratch, *see* Control and Status Register
mstatus, *see* Control and Status Register
mtime, *see* Control and Status Register
mtimecmp, *see* Control and Status Register

mtval, *see* Control and Status Register
mtvec, *see* Control and Status Register
mul, *see also* Multiply, *165*
mulh, *see also* Multiply high, *165*
mulhsu, *see also* Multiply high signed-unsigned, *165*
mulhu, *see also* Multiply high unsigned, *166*
Multiply, 49, *165*
high, 49, *165*
high signed-unsigned, 49, *165*
high unsigned, 49, *166*
multi-word, 50
using shift left, 49
word, 90, *166*
mulw, *see also* Multiply word, *166*
mv, *37*, *166*
mvendorid, *see* Control and Status Register

neg, *37*, *166*
negw, *37*, *166*
nop, *37*, *166*
not, *37*, *166*

Occam, William of, *48*
Or, 20, *167*
immediate, 20, *167*
or, *see also* c.or, *167*
ori, 20, *see also* Or immediate, *167*

Page, *115*
Page fault, *115*
Page table, *115*
Pascal, Blaise, 71
performance, *see also* instruction set architecture, principles of design, performance
CoreMark benchmark, 8
equation, 8
Perlis, Alan, *124*
PIC, *see also* position independent code
pipelined processor, *20*
Plato, *178*
position independent code, 11, 24, *42*, 95
Privilege mode, *104*
Machine mode, *105*
User mode, *112*
processadores fora de ordem, *20*
program size, *see also* instruction set architecture, principles of design, program size
Programming languages
Turing Award, *124*
prologue, *see also* function prologue
Pseudoinstruction, **37**

beqz, *37*, *134*
bgez, *37*, *134*
bgt, *37*, *134*
bgtu, *37*, *134*
bgtz, *37*, *135*
ble, *37*, *135*
bleu, *37*, *135*
blez, *37*, *135*
bltz, *37*, *135*
bnez, *37*, *136*
call, *37*, *143*
csrc, *37*, *143*
csrci, *37*, *143*
csrr, *37*, *143*
csrs, *37*, *145*
csrsi, *37*, *145*
csrw, *37*, *145*
csrwi, *37*, *145*
fabs.d, *37*, *146*
fabs.s, *37*, *146*
fence, *37*
fmv.d, *37*, *156*
fmv.s, *37*, *156*
fneg.d, *37*, *157*
fneg.s, *37*, *157*
frcsr, *37*, *158*
frflags, *37*, *158*
frrm, *37*, *158*
fscsr, *37*, *158*
fsflags, *37*, *159*
fsrm, *37*, *161*
j, *37*, *161*
jr, *37*, *162*
la, *162*
li, *37*, *163*
lla, *164*
mv, *37*, *166*
neg, *37*, *166*
negw, *37*, *166*
nop, *37*, *166*
not, *37*, *166*
rdcycle, *37*, *167*
rdcycleh, *37*, *167*
rdinstret, *37*, *167*
rdinstreth, *37*, *167*
rdtime, *37*, *167*
rdtimeh, *37*, *168*
ret, *37*, *168*
seqz, *37*, *169*
sext.w, *37*, *169*
sgtz, *37*, *170*
sltz, *37*, *172*
snez, *37*, *172*
tail, *37*, *175*

rdcycle, *37*, *167*
rdcycleh, *37*, *167*
rdinstret, *37*, *167*
rdinstreth, *37*, *167*
rdtime, *37*, *167*
rdtimeh, *37*, *168*
registers
number of, 11
rem, *see also* Remainder, *168*
Remainder, 49, *168*
unsigned, 49, *168*
unsigned word, 90, *168*
word, 90, *168*
remu, *see also* Remainder unsigned, *168*
remuw, *see also* Remainder unsigned word, *168*
remw, *see also* Remainder word, *168*
ret, *37*, *168*
RISC-V
application binary interface, 20, 29, 36, 53
assembler directives, *42*
BOOM, 8
Calling conventions, **37**
code size, 9, 96
DAXPY, **59**
Foundation, **2**
function epilogue, *37*
function prologue, *37*
heap region, **42**
Insertion Sort, **26**
instruction reference manual
number of pages, 13
instruction set naming scheme, **5**
lessons learned, **27**
Linker, **42**
Loader, **45**
long, *94*
macrofusion, 7
memory allocation, **42**
modularity, *5*
number of registers, 11
pseudoinstruction, **37**
pseudoinstructions, 11
Rocket, 7
RV128, *98*
RV32A, **64**
RV32C, 9, 12, **68**
RV32D, **52**
RV32F, **52**
RV32G, *9*, 12
RV32I, **16**
RV32M, **48**
RV32V, 12, **76**
RV64A, **90**
RV64C, **90**, **96**
RV64D, **90**
RV64F, **90**
RV64G, 12
RV64I, **90**
RV64M, **90**
saved registers, *36*
stack region, **42**
static region, **42**
temporary registers, *36*
text region, **42**
Tree Sum, **180**
RISC-V ABI, *see* RISC-V Application Binary Interface, *44*
RISC-V Application Binary Interface
ilp32, *44*
ilp32d, *44*
ilp32f, *44*
lp64, *94*
lp64d, *94*
lp64f, *94*
RISC-V Foundation, **2**
room for growth, *see also* instruction set architecture, principles of design, room for growth
RV128, *98*
RV32C, 59
RV32V, 85

Santayana, George, 27
sb, *see also* Store byte, *169*
sc.d, *see also* Store conditional doubleword, *169*
sc.w, *see also* Store conditional word, *169*
scatter, *80*
scause, *see* Control and Status Register
Schumacher, E. F., *68*
scounteren, *see* Control and Status Register
sd, *see also* c.sdsp, *see also* c.sd, *see also* Store doubleword, *169*
sepc, *see* Control and Status Register
seqz, *37*, *169*
Set less than, 22, *171*
immediate, 22, *171*
immediate unsigned, 22, *172*
unsigned, 22, *172*
seven metrics of ISA design, *see also* instruction set architecture, metrics of design
sext.w, *37*, *169*
sfence.vma, *see also* Fence Virtual Memory, *170*
sgtz, *37*, *170*
sh, *see also* Store halfword, *170*
Shift
left logical, 20, *170*
left logical immediate, 20, *171*
left logical immediate word, 90, *171*
left logical word, 90, *171*
right arithmetic, 20, *172*
right arithmetic immediate, 20, *173*

right arithmetic immediate word, 90, *173*
right arithmetic word, 90, *173*
right logical, 20, *174*
right logical immediate, 20, *174*
right logical immediate word, 90, *174*
right logical word, 90, *174*
SIMD, *see also* Single Instruction Multiple Data
simplicity, *see also* instruction set architecture, principles of design, simplicity
Single Instruction Multiple Data, 3, *12*, 76
sll, *see also* Shift left logical, *170*
slli, *see also* c.slli, *see also* Shift left logical immediate, *171*
slliw, *see also* Shift left logical immediate word, *171*
sllw, *see also* Shift left logical word, *171*
slt, *see also* Set less than, *171*
slti, *see also* Set less than immediate, *171*
sltiu, *see also* Set less than immediate unsigned, *172*
sltu, *see also* Set less than unsigned, *172*
sltz, *37*, *172*
Small is Beautiful, *68*
Smith, Jim, *85*
snez, *37*, *172*
sra, *see also* Shift right arithmetic, *172*
srai, *see also* c.srai, *see also* Shift right arithmetic immediate, *173*
sraiw, *see also* Shift right arithmetic immediate word, *173*
sraw, *see also* Shift right arithmetic word, *173*
sret, *see also* Exception Return Supervisor, *173*
srl, *see also* Shift right logical, *174*
srli, *see also* c.srli, *see also* Shift right logical immediate, *174*
srliw, *see also* Shift right logical immediate word, *174*
srlw, *see also* Shift right logical word, *174*
sscratch, *see* Control and Status Register
static linking, **45**
Store
byte, 23, *169*
conditional
doubleword, 90, *169*
word, 64, *169*
doubleword, 90, *169*
halfword, 23, *170*
word, 23, *170*
strip mining, *84*
stval, *see* Control and Status Register
stvec, *see* Control and Status Register
sub, *see also* Subtract, 20, *see also* c.sub, *see also* Subtract, *175*
Subtract, 20, *175*
word, 90, *175*
subw, *see also* c.subw, *see also* Subtract word, *175*
superscalar, 2, *8*, 70
Sutherland, Ivan, 34
sw, *see also* c.swsp, *see also* c.sw, *see also* Store word, *170*

tail, *37*, *175*
Thoreau, Henry David, 125
Thumb-2, 59
TLB, *119*
TLB shootdown, *119*
Translation Lookaside Buffer, *119*
Tree Sum, **180**
Turing Award
Allen, Fran, *16*
Brooks, Fred, 120
Dijkstra, Edsger W., *104*
Perlis, Alan, *124*
Sutherland, Ivan, 34
Wirth, Niklaus, 71

User mode, *112*

Vector
gather, *80*
indexed load, *80*
indexed store, *80*
scatter, *80*
strided load, *80*
strided store, *80*
strip-mining, *84*
vectorizable, *81*, 86
Vector architecture, **77**
context switch, *79*
dynamic register typing, *78*, 94
type encoding, *79*
vectorizable, *81*, 86
Virtual address, *115*
Virtual memory, *115*
von Neumann architecture, *13*
von Neumann, John, *13*

Wait for Interrupt, *110*, *175*
wfi, *see also* Wait for Interrupt, *175*
William of Occam, *48*
Wirth, Niklaus, 71

x86
aaa instruction, *3*
aad instruction, *3*
aam instruction, *3*
aas instruction, *3*
code size, 9, 96
DAXPY, **59**
enter instruction, 7
Insertion Sort, **26**
instruction reference manual
number of pages, 13
ISA growth, 3
number of registers, 11
position independent code, 11
Tree Sum, **180**
x86-32 AVX2, 85
x86-64
AMD64, **95**
XLEN, *105*
xor, 20, *see also* c.xor, *see also* Exclusive Or, *175*
xor properties, 23
xor register exchange, 23
xori, 20, *see also* Exclusive Or immediate, *176*

yield, *7*