



La Guía Práctica de RISC-V es una introducción concisa y referencia para programadores de sistemas embebidos, estudiantes y curiosos a una arquitectura moderna, popular y abierta. RISC-V abarca desde los microcontroladores embebidos más baratos de 32 bits hasta las computadoras más rápidas de 64 bits en la nube. El texto muestra cómo RISC-V siguió las buenas ideas de arquitecturas del pasado, evitando al mismo tiempo sus errores. Los aspectos destacados incluyen:

Los aspectos destacados incluyen:

- Introduce RISC-V en solo 100 páginas, incluyendo 75 figuras
- Una Tarjeta de Referencia de RISC-V de 2 páginas que resume todas las instrucciones
- Un Glosario de Instrucciones de 50 páginas que define todas las instrucciones detalladamente
- 75 resaltadores de buen diseño de arquitectura usando los íconos mostrados arriba
- 50 barras laterales con comentarios interesantes e historia de RISC-V
- 25 citas para transmitir la sabiduría de científicos e ingenieros reconocidos

Diez capítulos introducen cada componente del set de instrucciones modular de RISC-V—frecuentemente contrastando código compilado de C a RISC-V versus las arquitecturas anteriores ARM, Intel y MIPS—pero los lectores pueden comenzar a programar después del Capítulo 2.

“Me gusta RISC-V y este libro porque son elegantes—breves, al punto y completos.” C. Gordon Bell



David Patterson (Google y UC Berkeley) y Andrew Waterman (SiFive) son 2 de 4 arquitectos de RISC-V



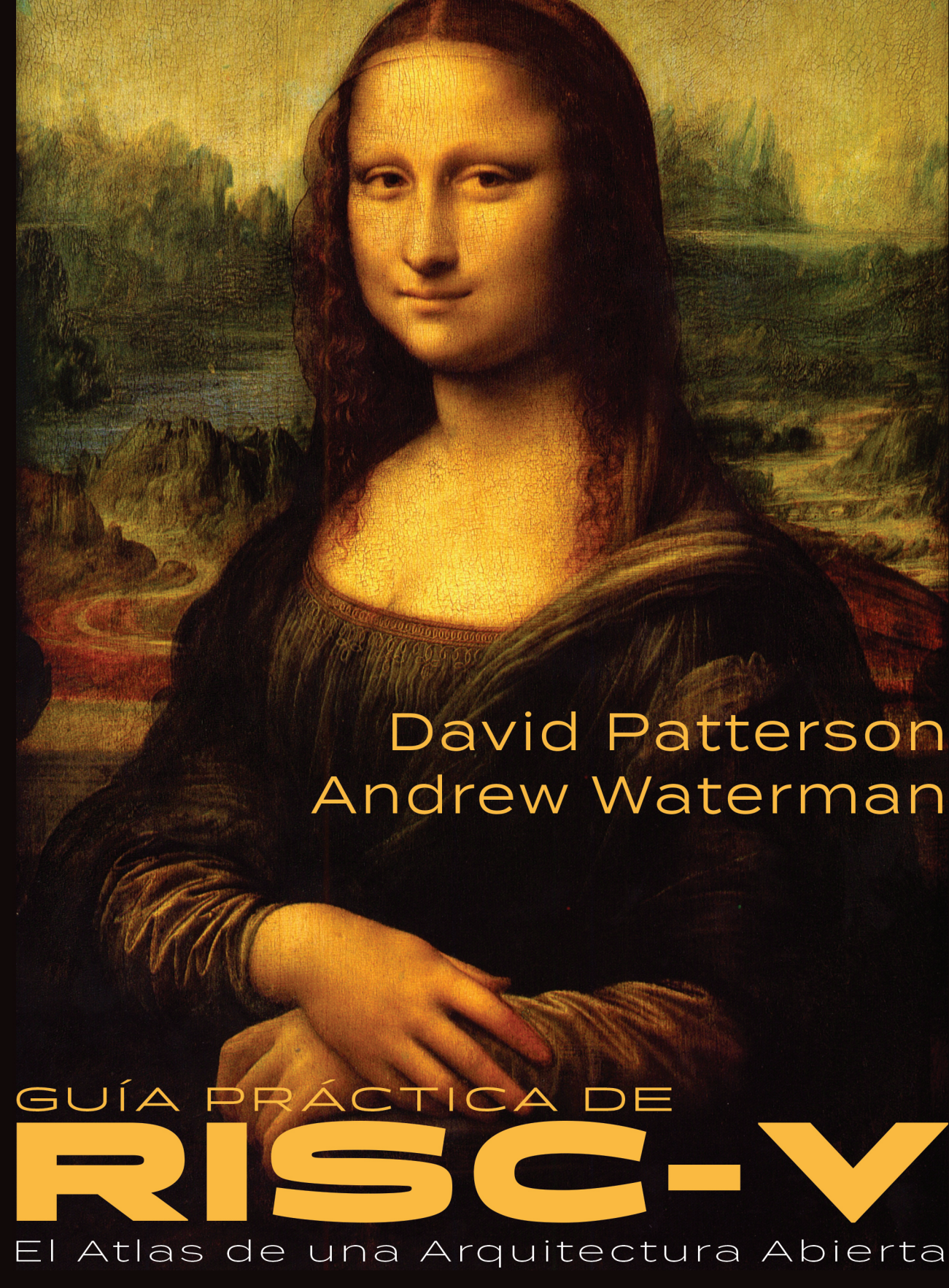
Strawberry Canyon LLC San Francisco, CA, USA



Guía Práctica de RISC-V



Patterson y Waterman



Elogios para La Guía Práctica de RISC-V

Este oportuno libro describe concisamente el simple, libre y abierto ISA RISC-V que está experimentando una rápida aceptación en muchos sectores diferentes de la computación. El libro también contiene muchas percepciones acerca de la arquitectura de computadoras en general, así como explica las decisiones de diseño particulares que tomamos al crear RISC-V. Puedo imaginar que este libro se convierta en una guía muy utilizada para muchos profesionales de RISC-V.

—Profesor Krste Asanović, University of California, Berkeley, uno de los cuatro arquitectos de RISC-V

Me gusta RISC-V y este libro porque son elegantes—breves, al punto y completos. Los comentarios del libro proveen historia, motivación y crítica de las arquitecturas.

—C. Gordon Bell, Microsoft y diseñador de las arquitecturas de set de instrucciones Digital PDP-11 y VAX-11

Este pequeño y útil libro resume fácilmente todos los elementos esenciales de la Arquitectura de Set de Instrucciones RISC-V, una guía de referencia perfecta para estudiantes y profesionales.

—Profesor Randy Katz, University of California, Berkeley, uno de los inventores del sistema de almacenamiento RAID

RISC-V es una excelente opción para que estudiantes puedan aprender acerca de arquitecturas de set de instrucciones y programación a nivel de ensamblador, los fundamentos básicos para el trabajo posterior en lenguajes de alto nivel. Este libro escrito con claridad ofrece una buena introducción a RISC-V, aumentado con comentarios esclarecedores de su historia evolutiva y comparaciones con otras arquitecturas conocidas. Basándose en experiencias del pasado con otras arquitecturas, los diseñadores de RISC-V pudieron evitar características innecesarias, comúnmente irregulares, resultando en pedagogía fácil. A pesar de ser simple, es lo suficientemente poderoso para amplio uso en aplicaciones reales. Hace mucho, yo enseñaba un primer curso de programación en ensamblador y si lo estuviera haciendo ahora, gustosamente utilizaría este libro.

—John Mashey, uno de los diseñadores de la arquitectura de set de instrucciones MIPS

Este libro cuenta qué puede hacer RISC-V y por qué sus diseñadores decidieron dotarlo de esas habilidades. Aun más interesante, los autores explican por qué RISC-V omite cosas presentes en máquinas más antiguas. Las razones son por lo menos tan interesantes como las dotaciones y omisiones de RISC-V.

—Ivan Sutherland, galardonado al Premio Turing, llamado el padre de la computación gráfica

RISC-V cambiará el mundo, y este libro te ayudará a ser parte de este cambio.

—Profesor Michael B. Taylor, University of Washington

Este libro será una referencia invaluable para quien trabaje con el ISA RISC-V. Los opcodes son presentados en varios formatos útiles para referencia rápida, facilitando la escritura e interpretación de código en lenguaje ensamblador. Adicionalmente, las explicaciones y ejemplos de cómo usar el ISA simplifican la labor del programador. Las comparaciones con otros ISAs son interesantes y muestran por qué los creadores de RISC-V tomaron las decisiones de diseño que tomaron.

—Megan Wachs, PhD, Ingeniera de SiFive

Tarjeta de Referencia para RISC-V Abierto



Instrucciones Base para Enteros: RV32I y RV64I					Instrucciones Privilegiadas RV				
Categoría	Nombre	Fmt	RV32I Base	+RV64I	Categoría	Nombre	Fmt	Mnemónico RV	
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2	Excep.	Mach-mode trap return	R	MRET	
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt		Supervisor-mode trap return	R	SRET	
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2		Interrupciones	Wait for Interrupt	R	WFI
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt			MMU	Virtual Memory FENCE	R
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2		Ejemplos de las 60 Pseudoinstrucciones RV			
Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt	Branch = 0 (BEQ rs,x0,imm)	J	BEQ rs,imm			
Aritmética	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2	Jump (uses JAL x0,imm)	J	J imm		
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm	MoVe (uses ADDI rd,rs,0)	R	MV rd,rs		
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2	RETurn (uses JALR x0,0,ra)	I	RET		
	Load Upper Imm	U	LUI rd,imm		Extensión Opcional de Instrucciones Comprimidas (16b): RV32C				
Add Upper Imm to PC	U	AUIPC rd,imm		Categoría	Nombre	Fmt	RVC	Equivalente RISC-V	
Lógica	XOR	R	XOR rd,rs1,rs2	Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4	
	XOR Immediate	I	XORI rd,rs1,imm		Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4	
	OR	R	OR rd,rs1,rs2		Float Load Word SP	CL	C.FLW rd',rs1',imm	FLW rd',rs1',imm*8	
	OR Immediate	I	ORI rd,rs1,imm		Float Load Word	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8	
	AND	R	AND rd,rs1,rs2		Float Load Double	CL	C.FLD rd',rs1',imm	FLD rd',rs1',imm*16	
AND Immediate	I	ANDI rd,rs1,imm	Float Load Double SP	CI	C.FLDSF rd,imm	FLD rd,sp,imm*16			
Comparación	Set <	R	SLT rd,rs1,rs2	Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4	
	Set < Immediate	I	SLTI rd,rs1,imm		Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4	
	Set < Unsigned	R	SLTU rd,rs1,rs2		Float Store Word SP	CS	C.FSW rs1',rs2',imm	FSW rs1',rs2',imm*8	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm		Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8	
Branches	Branch =	B	BEQ rs1,rs2,imm	Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16		
	Branch ≠	B	BNE rs1,rs2,imm	Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16		
	Branch >	B	BGT rs1,rs2,imm	Aritmética	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1	
	Branch >=	B	BGE rs1,rs2,imm		ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm		ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm		ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4	
Jump & Link	J&L	J	JAL rd,imm		SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1	
	Jump & Link Register	I	JALR rd,rs1,imm		AND	CR	C.AND rd,rs1	AND rd,rd,rs1	
Sinc.	Synch thread	I	FENCE	AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm		
	Synch Instr & Data	I	FENCE.I	OR	CR	C.OR rd,rs1	OR rd,rd,rs1		
Ambiente	CALL	I	ECALL	eXclusive OR	CR	C.XOR rd,rs1	AND rd,rd,rs1		
	BREAK	I	EBREAK	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0		
Control Status Register (CSR)	Read/Write	I	CSRRW rd,csr,rs1	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm		
	Read & Set Bit	I	CSRRS rd,csr,rs1	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm		
	Read & Clear Bit	I	CSRRC rd,csr,rs1	Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm	
	Read/Write Imm	I	CSRRWI rd,csr,imm		Shift Right Ari. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm	
	Read & Set Bit Imm	I	CSRRSI rd,csr,imm	Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm		
	Read & Clear Bit Imm	I	CSRRCI rd,csr,imm	Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm	
			Branch≠0		CB	C.BNEZ rs1',imm	BNE rs1',x0,imm		
Loads	Load Byte	I	LB rd,rs1,imm	Jump	Jump	CJ	C.J imm	JAL x0,imm	
	Load Halfword	I	LH rd,rs1,imm		Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0	
	Load Byte Unsigned	I	LBU rd,rs1,imm	Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm	
Load Half Unsigned	I	LHU rd,rs1,imm	Jump & Link Register		CR	C.JALR rs1	JALR ra,rs1,0		
Stores	Store Byte	S	SB rs1,rs2,imm	Sistema	Env. BREAK	CI	C.EBREAK	EBREAK	
	Store Halfword	S	SH rs1,rs2,imm						
	Store Word	S	SW rs1,rs2,imm						
+RV64I					Extensión Opcional Comprimida: RV64C				
LWU	rd,rs1,imm				Todo RV32C (excepto C.JAL, 4 word loads, 4 word stores) más:				
LD	rd,rs1,imm				ADD Word (C.ADDW)		Load Doubleword (C.LD)		
					ADD Imm. Word (C.ADDIW)		Load Doubleword SP (C.LDSF)		
					SUBtract Word (C.SUBW)		Store Doubleword (C.SD)		
							Store Doubleword SP (C.SDSF)		

Formatos de Instrucciones de 32 bits

	31	27	26	25	24	20	19	15	14	12	11	7	6	0
R	func7					rs2					rs1	func3	rd	opcode
I											rs1	func3	rd	opcode
S						rs2					rs1	func3	imm[4:0]	opcode
U						rs2					rs1	func3	imm[4:1][11]	opcode
B													rd	opcode
J													rd	opcode

Formatos de Instrucciones de 16 bits (RVC)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CR	func4										rd/rs1		rs2			op
CI	func3										rd/rs1		imm			op
CSS	func3										imm		rs2			op
CIW	func3										imm				rd'	op
CL	func3										imm	rs1'	imm		rd'	op
CS	func3										imm	rs1'	imm		rs2'	op
CB	func3										offset	rs1'			offset	op
CJ	func3														jump target	op

RISC-V Base-Enteros (RV32I/64I), privilegiado, y RV32/64C opcional. Registros x1-x31 y el PC son de 32 bits en RV32I y 64 en RV64I (x0=0). RV64I agrega 12 insts. para los datos anchos. Toda instrucción de 16 bits RVC se mapea a una instrucción RV existente de 32 bits.

Tarjeta de Referencia para RISC-V Abierto ②

Extensión Opcional de Instrucciones de Multiplicación y División: RVM						Extensión Opcional Vectorizada: RVV					
Categoría	Nombre	Fmt	RV32M (Mult.-Div.)	+RV64M		Nombre	Fmt	RV32V/R64V			
Multiplicación	MULTipty	R	MUL rd,rs1,rs2	MULW rd,rs1,rs2		SET Vector Len.	R	SETVL rd,rs1			
	MULTipty High	R	MULH rd,rs1,rs2			MULTipty High	R	VMULH rd,rs1,rs2			
	MULTipty High Sign/Uns	R	MULHSU rd,rs1,rs2			REMAinder	R	VREM rd,rs1,rs2			
	MULTipty High Uns	R	MULHU rd,rs1,rs2			Shift Left Log.	R	VSLL rd,rs1,rs2			
División	DIVide	R	DIV rd,rs1,rs2	DIVW rd,rs1,rs2		Shift Right Log.	R	VSRL rd,rs1,rs2			
	DIVide Unsigned	R	DIVU rd,rs1,rs2			Shift R. Arith.	R	VSRA rd,rs1,rs2			
Residuo	REMAinder	R	REM rd,rs1,rs2	REMW rd,rs1,rs2		Load	I	VLD rd,rs1,imm			
	REMAinder Unsigned	R	REMU rd,rs1,rs2	REMUW rd,rs1,rs2		Load Strided	R	VLDS rd,rs1,rs2			
Extensión Opcional de Instrucciones Atómicas: RVA						Load indexEd R VLDX rd,rs1,rs2					
Categoría	Nombre	Fmt	RV32A (Atómico)	+RV64A		STORE	S	VST rd,rs1,imm			
Load	Load Reserved	R	LR.W rd,rs1	LR.D rd,rs1		STORE Strided	R	VSTS rd,rs1,rs2			
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.D rd,rs1,rs2		STORE indexEd	R	VSTX rd,rs1,rs2			
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.D rd,rs1,rs2		AMO SWAP	R	AMOSWAP rd,rs1,rs2			
Suma	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.D rd,rs1,rs2		AMO ADD	R	AMOADD rd,rs1,rs2			
Lógica	XOR	R	AMOXR.W rd,rs1,rs2	AMOXR.D rd,rs1,rs2		AMO XOR	R	AMOXOR rd,rs1,rs2			
	AND	R	AMOAND.W rd,rs1,rs2	AMOAND.D rd,rs1,rs2		AMO AND	R	AMOAND rd,rs1,rs2			
	OR	R	AMoor.W rd,rs1,rs2	AMoor.D rd,rs1,rs2		AMO OR	R	AMoor rd,rs1,rs2			
Min/Max	MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.D rd,rs1,rs2		AMO MINimum	R	AMOMIN rd,rs1,rs2			
	MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.D rd,rs1,rs2		AMO MAXimum	R	AMOMAX rd,rs1,rs2			
	MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.D rd,rs1,rs2		Predicate =	R	VP EQ rd,rs1,rs2			
	MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.D rd,rs1,rs2		Predicate ≠	R	VP NE rd,rs1,rs2			
Dos Extensiones de Instrucciones Opcionales de Punto Flotante: RVF & RVD						Predicate <	R	VPLT rd,rs1,rs2			
Predicate ≥	R	VP GE rd,rs1,rs2				Predicate ≥	R	VP GE rd,rs1,rs2			
Predicate AND NOT	R	VP AND rd,rs1,rs2				Predicate AND NOT	R	VP AND rd,rs1,rs2			
Predicate OR	R	VP OR rd,rs1,rs2				Predicate OR	R	VP OR rd,rs1,rs2			
Predicate XOR	R	VP XOR rd,rs1,rs2				Predicate XOR	R	VP XOR rd,rs1,rs2			
Predicate NOT	R	VP NOT rd,rs1				Predicate NOT	R	VP NOT rd,rs1			
Predicate SWAP	R	VP SWAP rd,rs1				Predicate SWAP	R	VP SWAP rd,rs1			
Conversión de Llamadas						MOVE	R	VMOV rd,rs1			
ConVerT	ConVerT	R	VCVT rd,rs1	VCVT rd,rs1		ConVerT	R	VCVT rd,rs1			
ADD	ADD	R	VADD rd,rs1,rs2	VADD rd,rs1,rs2		ADD	R	VADD rd,rs1,rs2			
SUBtract	SUBtract	R	VSUB rd,rs1,rs2	VSUB rd,rs1,rs2		SUBtract	R	VSUB rd,rs1,rs2			
MULTipty	MULTipty	R	VMUL rd,rs1,rs2	VMUL rd,rs1,rs2		MULTipty	R	VMUL rd,rs1,rs2			
DIVide	DIVide	R	VDIV rd,rs1,rs2	VDIV rd,rs1,rs2		DIVide	R	VDIV rd,rs1,rs2			
SQure Root	SQure Root	R	VSQRT rd,rs1	VSQRT rd,rs1		SQure Root	R	VSQRT rd,rs1,rs2			
Multi-Suma	MULTipty-ADD	R	VFMADD rd,rs1,rs2,rs3	VFMADD rd,rs1,rs2,rs3		MULTipty-ADD	R	VFMADD rd,rs1,rs2,rs3			
	MULTipty-SUBtract	R	VFMSUB rd,rs1,rs2,rs3	VFMSUB rd,rs1,rs2,rs3		MULTipty-SUB	R	VFMSUB rd,rs1,rs2,rs3			
	Negative Multipty-SUBtract	R	VFNMSUB rd,rs1,rs2,rs3	VFNMSUB rd,rs1,rs2,rs3		Neg. Mul.-SUB	R	VFNMSUB rd,rs1,rs2,rs3			
	Negative Multipty-ADD	R	VFNMADD rd,rs1,rs2,rs3	VFNMADD rd,rs1,rs2,rs3		Neg. Mul.-ADD	R	VFNMADD rd,rs1,rs2,rs3			
Sign Injct	SIGN source	R	VSGNJ rd,rs1,rs2	VSGNJ rd,rs1,rs2		SIGN injct	R	VSGNJ rd,rs1,rs2			
	Negative SIGN source	R	VSGNJN rd,rs1,rs2	VSGNJN rd,rs1,rs2		Neg SIGN injct	R	VSGNJN rd,rs1,rs2			
	Xor SIGN source	R	VSGNJX rd,rs1,rs2	VSGNJX rd,rs1,rs2		Xor SIGN injct	R	VSGNJX rd,rs1,rs2			
Min/Max	MINimum	R	VMIN rd,rs1,rs2	VMIN rd,rs1,rs2		MINimum	R	VMIN rd,rs1,rs2			
	MAXimum	R	VMAX rd,rs1,rs2	VMAX rd,rs1,rs2		MAXimum	R	VMAX rd,rs1,rs2			
Comparación	compare Floa	R	VXOR rd,rs1,rs2	VXOR rd,rs1,rs2		XOR	R	VXOR rd,rs1,rs2			
	compare Float <	R	VOR rd,rs1,rs2	VOR rd,rs1,rs2		OR	R	VOR rd,rs1,rs2			
	compare Float ≤	R	VAND rd,rs1,rs2	VAND rd,rs1,rs2		AND	R	VAND rd,rs1,rs2			
Categoriz.	CLASSify type	R	VCLASS rd,rs1	VCLASS rd,rs1		CLASS	R	VCLASS rd,rs1			
Configuración	Read Status	R	zero	Hardwired zero		SET Data Conf.	R	VSETDCFG rd,rs1			
	Read Rounding Mode	R	ra	Return address		EXTRACT	R	VEXTRACT rd,rs1,rs2			
	Read Flags	R	sp	Stack pointer		MERGE	R	VMERGE rd,rs1,rs2			
	Swap Status Reg	R	gp	Global pointer		SELECT	R	VSELECT rd,rs1,rs2			
	Swap Rounding Mode	R	tp	Thread pointer							
	Swap Flags	R	t0-0,ft0-7	Temporaries							
	Swap Rounding Mode Imm	I	s0-11,fs0-11	Saved registers							
	Swap Flags Imm	I	a0-7,fa0-7	Function args							

Convención de llamadas RISC-V y 5 extensiones opcionales: 8 RV32M; 11 RV32A; 34 instrucciones de punto flotante c/u para datos de 32 y 64 bits (RV32F, RV32D); y 53 RV32V. Usando notación, {} significa conjunto, así FADD.{F|D} es FADD.F y FADD.D. RV32{F|D} agrega registros f0-f31, con el ancho de la precisión más ancha, y un registro de control y estado de punto flotante fcsr. RV32V agrega registros vectorizados v0-v31, registros de predicado vect. vp0-vp7, y registro de longitud de vector vl. RV64 agrega unas insts: RVM recibe 4, RVA 11, RVF 6, RVD 6, y RVV 0.

Guía Práctica de RISC-V:
El Atlas de una Arquitectura Abierta
Primera Edición, 1.0.5

David Patterson y Andrew Waterman

Traducido por Alí Lemus y Eduardo Corpeño

11 de Julio de 2018

Copyright 2017 Strawberry Canyon LLC. Todos los derechos reservados.

Ninguna parte de este libro o sus materiales relacionados puede ser reproducida en ninguna forma sin el permiso escrito de sus autores.

Versión del libro: 1.0.5

El fondo de la portada es una fotografía de la ***Mona Lisa***. Es un retrato de Lisa Gherardini, pintado entre 1503 y 1506, por Leonardo da Vinci. El Rey de Francia lo compró a Leonardo cerca de 1530, y ha estado en exhibición en el Museo del Louvre en París desde 1797. La Mona Lisa es considerada la mejor obra de arte conocida en el mundo. Mona Lisa representa elegancia, lo cual creemos que es una característica de RISC-V.

Este libro fue preparado con L^AT_EX. Los Makefiles necesarios, archivos de estilo y la mayor parte de scripts están disponibles bajo la Licencia BSD en github.com/armandofox/latex2ebook.

Arthur Klepchukov diseñó las portadas y gráficos para todas las versiones.

Catalogación-en-la-Fuente del Publicador

Nombres: Patterson, David A. | Waterman, Andrew, 1986-

Título: Guía práctica de RISC-V: el atlas de una arquitectura abierta

David Patterson y Andrew Waterman.

Descripción: Primera edición. | [Berkeley, California] : Strawberry Canyon LLC, 2017. |

Incluye referencias bibliográficas e índice.

Identificadores: ISBN 978-0-9992491-2-3

Temas: LCSH: Arquitectura de Computadoras. | Microprocesadores RISC. |

Lenguajes ensambladores (Computadoras electrónicas)

Clasificación: LCC QA76.9.A73 P38 2017 | DDC 004.22- -dc23

Dedicatoria

David Patterson dedica este libro a sus padres:

—Para mi padre David, de quien heredé inventiva, atletismo y el valor para luchar por lo correcto; y

—Para mi madre Lucie, de quien heredé inteligencia, optimismo y mi temperamento.

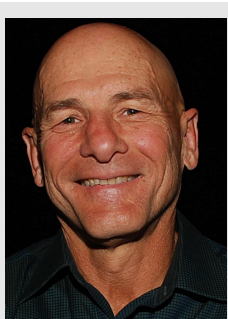
Gracias por ser tan buenos ejemplos, lo cual me enseñó lo que significa ser un buen esposo, padre y abuelo.



Andrew Waterman dedica este libro a sus padres, John y Elizabeth, quienes han sido enormemente alentadores, incluso a miles de millas de distancia.



Acerca de los Autores



David Patterson se retiró después de 40 años como Profesor de Ciencias de la Computación en UC Berkeley en el 2016, y luego se unió a Google Brain como ingeniero distinguido. También es Vicepresidente de la Junta de Directores de la Fundación RISC-V. En el pasado, fue nombrado Presidente de la División de Ciencias de la Computación de Berkeley y fue elegido para ser Presidente de la CRA (*Computing Research Association*) y Presidente de la ACM (*Association for Computing Machinery*). En los 1980s, lideró cuatro generaciones de proyectos *Reduced Instruction Set Computer* (RISC), lo cual inspiró a llamar al RISC más reciente de Berkeley “RISC Cinco”. Junto con Andrew Waterman, fue uno de los cuatro arquitectos de RISC-V. Más allá de RISC, sus proyectos más conocidos son *Redundant Arrays of Inexpensive Disks* (RAID) y *Networks of Workstations* (NOW). Esta investigación llevó a muchos artículos, 7 libros y más de 35 honores, incluyendo la elección a la NAE (*National Academy of Engineering*), la NAS (*National Academy of Sciences*) y el *Silicon Valley Engineering Hall of Fame* además de ser nombrado Miembro del Museo Histórico de Computadoras, ACM, IEEE y ambas organizaciones AAAS. Sus premios a la enseñanza incluyen el Premio a la Enseñanza Distinguida (UC Berkeley), el Premio Karlstrom al Educador Sobresaliente (ACM), la Medalla Mulligan a la Educación (IEEE) y el Premio a la Enseñanza de Pregrado (IEEE). También ganó Premios a la Excelencia en Libros de Texto (“Texty”) de la TAA (*Text and Academic Authors Association*) por un libro de arquitectura de computadoras y por un libro de ingeniería de software. Recibió todos sus títulos de UCLA, la cual le otorgó un Premio a Exalumnos Académicos de Ingeniería Sobresalientes. Creció en el Sur de California, y como diversión juega fútbol, monta bicicleta con sus hijos y camina en la playa con su esposa. Originalmente novios de secundaria, celebraron su 50 aniversario de bodas unos días después de que la edición Beta fuera publicada.



Andrew Waterman es el Jefe de Ingeniería y cofundador de SiFive. SiFive fue fundada por los creadores de la arquitectura RISC-V para proporcionar chips a la medida basados en RISC-V. Andrew recibió su PhD en Ciencias de la Computación de UC Berkeley, donde, cansado de los caprichos de las arquitecturas de set de instrucciones existentes, codiseñó el ISA RISC-V y el primer microprocesador RISC-V. Es uno de los principales contribuidores al generador de chips Rocket de código abierto y basado en RISC-V, el lenguaje de construcción de hardware Chisel, y las versiones RISC-V del kernel del sistema operativo Linux, el compilador de C GNU y Librería de C. También tiene un MS de UC Berkeley, que fue la base de la extensión RVC para RISC-V, y un BSE de la Universidad de Duke.

Acerca de los Traductores

Alí Lemus es el director del Laboratorio Turing de la Facultad de Ingeniería de Sistemas, Informática y Ciencias de la Computación de *Universidad Galileo* en Guatemala, donde se dedica a la investigación científica en las áreas de videojuegos, educación, supercomputación, *deep learning* e inteligencia artificial. Tiene un MS en Ciencias de la Computación Aplicada de la Universidad de Tohoku en Japón y un BSCS de la Universidad Francisco Marroquín en Guatemala. También imparte cursos de ciencias de la computación, programación en alto y bajo nivel y organización de computadoras. El laboratorio Turing concentra las mentes más talentosas de la facultad de ingeniería de sistemas con diversos proyectos disponibles en su sitio web (turing.galileo.edu). Alí es visto por sus amigos como un *hombre de mundo* debido a su gusto por conocer personas, lugares y actividades nuevas todo el tiempo. En su tiempo libre dedica tiempo de calidad a sus hijos.



Eduardo Corpeño es el director del área académica de electrónica en la Facultad de Ingeniería de Sistemas, Informática y Ciencias de la Computación de *Universidad Galileo* en Guatemala, donde imparte diversos cursos de ingeniería en las áreas de electrónica y arquitectura de computadoras, además de codirigir la Maestría en Electrónica Industrial (mei.galileo.edu). Tiene un MS en Ciencias de la Computación del Georgia Institute of Technology, y un BSECE de la Universidad Francisco Marroquín en Guatemala. Para Eduardo, las acciones hablan más fuerte que las palabras. Por eso valora la puntualidad y le apasiona terminar todos los proyectos que inicia. En su tiempo libre disfruta de la grata compañía de su esposa, su hijo y sus dos perros.



Contenido Rápido

	Tarjeta de Referencia RISC-V	i
	Lista de Figuras	ix
	Prefacio	xiv
1	¿Por Qué RISC-V?	2
2	RV32I: ISA RISC-V Base para Números Enteros	16
3	Lenguaje Ensamblador RISC-V	34
4	RV32M: Multiplicación y División	48
5	RV32FD: Punto Flotante de Precisión Simple y Doble	52
6	RV32A: Atómico	64
7	RV32C: Instrucciones Comprimidas	68
8	RV32V: Vector	76
9	RV64: Instrucciones de Memoria de 64 bits	90
10	Arquitectura Privilegiada RV32/64	104
11	Futuras Extensiones RISC-V Opcionales	124
Apéndice A	Listados de Instrucciones RISC-V	128
Apéndice B	Transliteración de RISC-V	176
	Índice	184



Contenido

Lista de Figuras	xiii
Prefacio	xiv
1 ¿Por Qué RISC-V?	2
1.1 Introducción	2
1.2 ISAs Modulares vs. Incrementales	3
1.3 Introducción al Diseño del ISA	5
1.4 Un Vistazo al Libro	11
1.5 Observaciones Finales	12
1.6 Para Aprender Más	14
2 RV32I: ISA RISC-V Base para Números Enteros	16
2.1 Introducción	16
2.2 Formato de Instrucciones RV32I	16
2.3 Registros de RV32I	20
2.4 Computación Entera de RV32I	20
2.5 Loads y Stores de RV32I	23
2.6 Branches Condicionales de RV32I	23
2.7 Salto Incondicional de RV32I	25
2.8 Miscelánea de RV32I	25
2.9 Comparando RV32I, ARM-32, MIPS-32 y x86-32	25
2.10 Observaciones Finales	26
2.11 Para Aprender Más	27
3 Lenguaje Ensamblador RISC-V	34
3.1 Introducción	34
3.2 Convención de llamadas	34
3.3 Ensamblador	37
3.4 Linker	41
3.5 Linking Estático vs Dinámico	45
3.6 Loader	46
3.7 Observaciones Finales	46
3.8 Para Aprender Más	46

4	RV32M: Multiplicación y División	48
4.1	Introducción	48
4.2	Observaciones Finales	50
4.3	Para Aprender Más	50
5	RV32FD: Punto Flotante de Precisión Simple y Doble	52
5.1	Introducción	52
5.2	Registros de Punto Flotante	52
5.3	Loads, Stores y Aritmética de Punto Flotante	57
5.4	Moves y Converts de Punto Flotante	58
5.5	Instrucciones de Punto Flotante Misceláneas	58
5.6	Comparando RV32FD, ARM-32, MIPS-32 y x86-32 usando DAXPY	60
5.7	Observaciones Finales	60
5.8	Para Aprender Más	61
6	RV32A: Atómico	64
6.1	Introducción	64
6.2	Observaciones Finales	66
6.3	Para Aprender Más	66
7	RV32C: Instrucciones Comprimidas	68
7.1	Introducción	68
7.2	Comparando RV32GC, Thumb-2, microMIPS y x86-32	70
7.3	Observaciones Finales	71
7.4	Para Aprender Más	71
8	RV32V: Vector	76
8.1	Introducción	76
8.2	Instrucciones que Computan Vectores	77
8.3	Registros Vectorizados y Tipado Dinámico	78
8.4	Loads y Stores Vectorizados	80
8.5	Paralelismo Durante la Ejecución Vectorizada	80
8.6	Ejecución Condicional de Operaciones Vectoriales	81
8.7	Instrucciones Vectorizadas Misceláneas	82
8.8	Ejemplo Vectorizado: DAXPY en RV32V	83
8.9	Comparando RV32V, MIPS-32 MSA SIMD y x86-32 AVX SIMD	84
8.10	Observaciones Finales	86
8.11	Para Aprender Más	87
9	RV64: Instrucciones de Memoria de 64 bits	90
9.1	Introducción	90
9.2	Comparación con otros ISAs de 64 bits usando Ordenamiento por Inserción	94
9.3	Tamaño del Programa	97
9.4	Observaciones Finales	97
9.5	Para Aprender Más	98

10	Arquitectura Privilegiada RV32/64	104
10.1	Introducción	104
10.2	Modo Máquina para Sistemas Embebidos Simples	105
10.3	Manejo de Excepciones en Modo Máquina	107
10.4	Modo Usuario y Aislamiento de Procesos en Sistemas Embebidos	112
10.5	Modo Supervisor para Sistemas Operativos Modernos	113
10.6	Memoria Virtual Basada en Páginas	115
10.7	CSRs de Identificación y Rendimiento	120
10.8	Observaciones Finales	121
10.9	Para Aprender Más	123
11	Futuras Extensiones RISC-V Opcionales	124
11.1	Extensión Estándar “B” para Manipulación de Bits	124
11.2	Extensión Estándar “E” para Embebidos	124
11.3	Extensión de la Arquitectura Privilegiada “H” para Soporte de <i>Hypervisor</i>	124
11.4	Extensión Estándar “J” para Lenguajes Traducidos Dinámicamente	124
11.5	Extensión Estándar “L” para Punto Flotante Decimal	125
11.6	Extensión Estándar “N” para Interrupciones a Nivel de User	125
11.7	Extensión Estándar “P” para Instrucciones <i>Packed-SIMD</i>	125
11.8	Extensión Estándar “Q” para Punto Flotante de Precisión Cuádruple	125
11.9	Observaciones Finales	126
A	Listados de Instrucciones RISC-V	128
B	Transliteración de RISC-V	176
B.1	Introducción	176
B.2	Comparando RV32I, ARM-32 y x86-32 empleando Suma de Árboles	178
B.3	Conclusión	179
	Acrónimos	182
	Índice	184



Lista de Figuras

1.1	Miembros corporativos de la Fundación RISC-V	3
1.2	Crecimiento del set de instrucciones x86 a través de su existencia.	4
1.3	Descripción de la instrucción <i>ASCII Adjust after Addition</i> (aaa) del x86-32.	4
1.4	Una oblea de RISC-V de 8 pulgadas de diámetro diseñada por SiFive.	6
1.5	Tamaños de programa relativos para RV32G, ARM-32, x86-32, RV32C y Thumb-2.	9
1.6	Número de páginas y palabras en manuales de ISAs	12
2.1	Diagrama de las instrucciones RV32I.	17
2.2	Formato de instrucciones RISC-V.	17
2.3	El mapa de opcodes de RV32I tiene la estructura de la instrucción, opcodes, tipo de formato y nombres	18
2.4	Los registros de RV32I.	21
2.5	Ordenamiento por Inserción en C.	26
2.6	Número de instrucciones y tamaño del código para Ordenamiento por Inserción para estos ISAs.	26
2.7	Lecciones aprendidas por los arquitectos de RISC-V de ISAs anteriores.	28
2.8	Código RV32I para Ordenamiento por Inserción de la Figura 2.5.	29
2.9	Código ARM-32 para Ordenamiento por Inserción de la Figura 2.5.	30
2.10	Código MIPS-32 para Ordenamiento por Inserción de la Figura 2.5.	31
2.11	Código x86-32 para Ordenamiento por Inserción de la Figura 2.5.	32
3.1	Pasos para convertir desde código fuente hasta un programa en ejecución.	35
3.2	Mnemónicos de ensamblador para registros enteros y de punto flotante en RISC-V.	36
3.3	32 pseudo-instrucciones de RISC-V que dependen de x0, el registro cero.	38
3.4	28 pseudoinstrucciones de RISC-V que son independientes de x0, el registro cero.	39
3.5	Programa Hola Mundo en C (<code>hello.c</code>).	40
3.6	Programa Hola Mundo en lenguaje ensamblador de RISC-V (<code>hello.s</code>).	40
3.7	Programa Hola Mundo en lenguaje de máquina de RISC-V (<code>hello.o</code>).	42
3.8	Programa Hola Mundo en lenguaje de máquina de RISC-V luego de la etapa de <i>linking</i>	42
3.9	Directivas comunes del ensamblador de RISC-V.	43

3.10	Reserva de memoria para el programa y datos en RV32I.	44
4.1	Diagrama de las instrucciones RV32M.	48
4.2	El mapa de opcodes de RV32M tiene el formato: opcodes, tipo de formato y nombres	49
4.3	Código de RV32M que divide por una constante usando la multiplicación.	49
5.1	Diagrama de las instrucciones RV32F y RV32D.	53
5.2	El mapa de opcodes de RV32F tiene la estructura de la instrucción, opcodes, tipo de formato y nombres.	54
5.3	El mapa de opcodes de RV32D tiene la estructura de la instrucción, opcodes, tipo de formato y nombres.	55
5.4	Los registros de punto flotante de RV32F y RV32D.	56
5.5	Registro de control y estado de punto flotante.	57
5.6	Instrucciones de conversión de RV32F y RV32D.	58
5.7	DAXPY: un programa que usa punto flotante de manera exhaustiva en C.	60
5.8	Número de instrucciones y tamaño del código para los cuarto ISAs.	60
5.9	Código de RV32D para DAXPY en la Figura 5.7.	62
5.10	Código de ARM-32 para DAXPY en la Figura 5.7.	62
5.11	Código de MIPS-32 para DAXPY en la Figura 5.7.	63
5.12	Código de x86-32 para DAXPY en la Figura 5.7.	63
6.1	Diagrama de las instrucciones RV32A.	64
6.2	El mapa de opcodes de RV32A tiene la estructura de la instrucción, opcodes, tipo de formato y nombres.	65
6.3	Dos ejemplos de sincronización.	66
7.1	Diagrama de las instrucciones RV32C.	69
7.2	Instrucciones y tamaño del código para Ordenamiento por Inserción y DAXPY para ISAs comprimidos.	70
7.3	Código de RV32C para Ordenamiento por Inserción.	72
7.4	Código RV32DC para DAXPY.	73
7.5	Mapa de opcodes RV32C (bits[1 : 0] = 01) muestra la estructura, opcodes, formato y nombres.	73
7.6	Mapa de opcodes RV32C (bits[1 : 0] = 00) muestra la estructura, opcodes, formato y nombres.	74
7.7	Mapa de opcodes para RV32C (bits[1 : 0] = 10) muestra la estructura, opcodes, formato y nombres	74
7.8	Formatos de instrucciones comprimidas RVC de 16 bits.	75
8.1	Diagrama de las instrucciones RV32V.	77
8.2	Codificación de tipos de datos en registros vectorizados.	79
8.3	Código RV32V para DAXPY en la Figura 5.7.	83
8.4	Cantidad de instrucciones y tamaño del código de DAXPY para ISAs vectorizados.	85
8.5	Código MIPS-32 MSA para DAXPY de la Figura 5.7.	88
8.6	Código x86-32 AVX2 para DAXPY de la Figura 5.7.	89

9.1	Diagrama de las instrucciones RV64I.	91
9.2	Diagrama de las instrucciones RV64M y RV64A.	91
9.3	Diagrama de las instrucciones RV64F y RV64D.	92
9.4	Diagrama de las instrucciones RV64C.	92
9.5	Mapa de los opcodes de las instrucciones base y extensiones opcionales de RV64.	93
9.6	Número de instrucciones y tamaño del código de Ordenamiento por Inserción para los cuarto ISAs.	95
9.7	Tamaños del programa relativos para RV64G, ARM-64 y x86-64 versus RV64GC.	96
9.8	Código RV64I para Ordenamiento por Inserción en la Figura 2.5.	99
9.9	Código de ARM-64 para Ordenamiento por Inserción en la Figura 2.5.	100
9.10	Código de MIPS-64 para Ordenamiento por Inserción en la Figura 2.5.	101
9.11	Código x86-64 para Ordenamiento por Inserción en la Figura 2.5.	102
10.1	Diagrama de las instrucciones privilegiadas RISC-V.	104
10.2	Estructura de instrucciones privilegiadas RISC-V, opcodes, tipo de formato y nombre	105
10.3	Causas de excepciones e interrupciones en RISC-V.	106
10.4	El CSR <code>mstatus</code>	108
10.5	CSRs de interrupciones de máquina.	108
10.6	CSRs de causa para máquina y supervisor (<code>mcause</code> y <code>scause</code>).	108
10.7	CSRs de direcciones base de vectores de excepciones para máquina y supervisor (<code>mtvec</code> y <code>stvec</code>).	108
10.8	CSRs asociados con excepciones e interrupciones.	109
10.9	Niveles de privilegio de RISC-V y su codificación.	109
10.10	Código RISC-V para un manejador simple de interrupción de temporizador.	111
10.11	Un registro de dirección y configuración de PMP.	112
10.12	La estructura de las configuraciones de PMP en los CSRs <code>pmpcfg</code>	113
10.13	Los CSRs de delegación.	114
10.14	CSRs de interrupción de supervisor.	114
10.15	El CSR <code>sstatus</code>	115
10.16	Una entrada de la tabla de páginas (PTE) de RV32 Sv32.	116
10.17	Una entrada de la tabla de páginas (PTE) de RV64 Sv39.	117
10.18	El CSR <code>satp</code>	117
10.19	La codificación del campo MODE en el CSR <code>satp</code>	118
10.20	Diagrama del proceso de traducción de direcciones Sv32.	119
10.21	El CSR de ISA de Máquina misa reporta el ISA soportado.	121
10.22	El CSR <code>mvendorid</code> provee el ID de fabricante JEDEC del procesador.	121
10.23	Los CSRs de identificación de Máquina (<code>marchid</code> , <code>mimpid</code> , <code>mhartid</code>)	121
10.24	Los CSRs de Tiempo de Máquina (<code>mtime</code> y <code>mtimecmp</code>) miden tiempo	122
10.25	Los registros de habilitación de contadores <code>mcounteren</code> y <code>scounteren</code>	122
10.26	Los CSRs de monitoreo de rendimiento de hardware	122
10.27	El algoritmo completo para traducción de direcciones virtuales a físicas.	123
B.1	Instrucciones RV32I de acceso a memoria transliteradas a ARM-32 y x86-32.	176
B.2	Instrucciones aritméticas RV32I transliteradas a ARM-32 y x86-32.	177

B.3	Instrucciones de control de flujo de ejecución RV32I transliteradas a ARM-32 y x86-32.	178
B.4	Una rutina en C que suma los valores en un árbol binario, empleando un recorrido en-orden.	180
B.5	Código RV32I para recorrido en-orden del árbol.	180
B.6	Código ARM-32 para recorrido en-orden del árbol.	181
B.7	Código x86-32 para recorrido en-orden del árbol.	181



Prefacio

¡Bienvenidos!

RISC-V ha sido un fenómeno, rápidamente creciendo en popularidad desde su introducción en el 2011. Pensamos que una guía concisa para el programador ayudaría a impulsar su camino alentando a principiantes a entender por qué es un set de instrucciones atractivo y a ver cómo difiere de arquitecturas de set de instrucciones (ISAs) convencionales del pasado.

Libros de otros ISAs nos inspiraron, pero esperamos que la simpleza de RISC-V significara escribir mucho menos que las 500+ páginas de excelentes libros como *See MIPS Run*. A un tercio de esa longitud, al menos en esa medida tuvimos éxito. De hecho, los diez capítulos que introducen cada componente del set de instrucciones modular RISC-V ocupan tan solo 100 páginas—a pesar de tener casi una figura por página en promedio (75 en total)—lo cual facilita una lectura rápida.

Después de explicar los principios de diseño de sets de instrucciones, mostramos cómo los arquitectos de RISC-V aprendieron de los sets de instrucciones de los últimos 40 años para tomar prestadas sus buenas ideas y evitar sus errores. Los ISAs son juzgados tanto por lo que omiten como por lo que incluyen.

Luego introducimos cada componente de esta arquitectura modular en una secuencia de capítulos. Cada capítulo tiene un programa en el lenguaje ensamblador de RISC-V que demuestra el uso de las instrucciones introducidas en ese capítulo, lo cual facilita al programador de lenguaje ensamblador aprender el código RISC-V. También mostramos frecuentemente programas equivalentes en ARM, MIPS, y x86 que resaltan la simpleza y los beneficios en costo-energía-rendimiento de RISC-V.

Para hacer el libro más divertido, incluimos casi 50 barras laterales en los márgenes con lo que esperamos que sean comentarios interesantes acerca del texto. También incluimos cerca de 75 imágenes en los márgenes para enfatizar ejemplos de buen diseño de ISAs. (¡Hemos aprovechado nuestros márgenes!) Finalmente, para el lector dedicado, agregamos alrededor de 25 elaboraciones a lo largo del texto. Usted puede profundizar en estas secciones opcionales si le interesa un tema. Estas secciones no son requeridas para comprender el resto del material del libro, así que puede hacer caso omiso de ellos si no atrapan su interés. Para aficionados de la arquitectura de computadoras, citamos 25 artículos y libros que pueden ampliar sus horizontes. ¡Aprendimos mucho leyéndolos para escribir este libro!

¿Por Qué Tantas Citas?

Pensamos que las citas también hacen más divertida la lectura, así que hemos esparcido 25 a lo largo del texto. Similarmente son un mecanismo eficiente para transmitir sabiduría de mayores a novatos, y ayudan a establecer estándares culturales para un buen diseño de ISAs. También queremos que los lectores capten un poco de historia en el tema, por lo que presentamos citas de famosos científicos de computación e ingenieros a lo largo del texto.

Introducción y Referencia

Pretendemos que este delgado libro sirva como una introducción a RISC-V para estudiantes y programadores de sistemas embebidos interesados en escribir código RISC-V. Este libro asume que los lectores han visto al menos un set de instrucciones anteriormente. Si no, podría interesarle revisar nuestro libro introductorio de arquitectura basado en RISC-V: *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*.

Las referencias compactas en este libro incluyen:

- **Tarjeta de Referencia** – Esta descripción de RISC-V condensada en una página (dos lados) cubre RV32GCV y RV64GCV, que incluye la base y todas las extensiones definidas: RVI, RVM, RVA, RVF, RVD, RVC, e incluso RVV, a pesar de encontrarse aún en desarrollo.
- **Diagramas de Instrucciones** – Estas descripciones de cada extensión de instrucciones de media página, que son las primeras figuras de los capítulos, listan los nombres completos de todas las instrucciones RISC-V en un formato que permite ver fácilmente las variaciones de cada instrucción. Ver Figuras 2.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1, 9.2, 9.3 y 9.4.
- **Mapas de Opcodes** – Estas tablas muestran la estructura de las instrucciones, opcodes, tipo de formato y mnemónico para cada extensión de instrucciones en una fracción de página. Ver Figuras 2.3, 3.3, 3.4, 4.2, 5.2, 5.3, 6.2, 7.6, 7.5, 7.7, 9.5, y 10.1 (Los diagramas de instrucciones y mapas de opcodes inspiraron el uso de la palabra atlas en el subtítulo del libro).
- **Glosario de Instrucciones** – El Apéndice A es una descripción exhaustiva de todas las instrucciones y pseudoinstrucciones RISC-V.¹ Incluye todo: el nombre de la operación y operandos, una descripción en Español, una definición en lenguaje de transferencia de registros, en cuál extensión de RISC-V se encuentra, el nombre completo de la instrucción, el formato de la instrucción, un diagrama de la instrucción mostrando los opcodes, y referencias a versiones compactas de la instrucción. Asombrosamente, todo esto cabe en menos de 50 páginas.
- **Traductor de Instrucciones** – El Apéndice B ayuda a los programadores de lenguaje ensamblador experimentados proporcionando tablas que muestran las instrucciones ARM-32 o x86-32 equivalentes a instrucciones RV32I. También lista las salidas del

¹El comité encargado de definir RV32V no completaron su trabajo a tiempo para la edición Beta, así que omitimos esas instrucciones del Apéndice A. El Capítulo 8 es nuestra mejor estimación de cómo resultará RV32V, aunque es probable que cambie un poco.

compilador de C para un programa simple de recorrido de árboles para estas tres arquitecturas y describe las sorprendentemente pequeñas diferencias entre ellas. Concluye con consejos sobre cómo traducir código de las arquitecturas más antiguas a RISC-V, lo cual es más fácil de lo que uno pudiera pensar.

- **Índice** – Ayuda a encontrar la página que describe la explicación, definición, o diagrama de la instrucción ya sea por el nombre completo o por mnemónico. Está organizado como un diccionario.

Erratas y Contenido Suplementario

Pretendemos recolectar las Erratas y lanzar actualizaciones unas pocas veces al año. El sitio web del libro muestra la última versión del libro y una breve descripción de los cambios desde la versión anterior. Erratas anteriores pueden ser revisadas, y las nuevas reportadas, en el sitio web del libro (www.riscvbook.com). Nos disculpamos de antemano por los problemas que encuentre en esta edición, y esperamos sus comentarios sobre cómo mejorar este material.

Historia de este Libro

En el Sexto taller de RISC-V del 8 al 11 de Mayo de 2017 en Shanghai, vimos la necesidad de este libro. Comenzamos unas semanas después. Dada la gran experiencia de Patterson escribiendo libros, el plan era que él escribiera la mayor parte de los capítulos. Nosotros dos colaboramos en la organización y fuimos los primeros revisores mutuos de cada capítulo. Patterson es el autor de los Capítulos 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, la Tarjeta de Referencia, y este Prefacio, mientras que Waterman escribió el Capítulo 10, el Apéndice A—la sección más larga del libro—el Apéndice B, y escribió todos los programas en el libro. Waterman también administró las herramientas LaTeX de Armando Fox que nos permitió producir el libro.

Ofrecimos una edición Beta del libro para 800 estudiantes de UC Berkeley en el semestre de Otoño de 2017. Los lectores sólo encontraron algunos errores de escritura y de LaTeX, que arreglamos para la primera edición. También mejoramos los íconos de los márgenes para hacerlos más fáciles de recordar y actualizamos algunas figuras que no se veían tan bien en impresión como habíamos esperado.

Más significativamente, la primera edición extendió el Capítulo 10 para incluir 60+ Registros de Control y Estado y agregó el Apéndice B para ayudar a los programadores interesados en convertir programas de lenguaje ensamblador de ISAs antiguos a RISC-V.

La primera edición fue publicada a tiempo para estar disponible en el Séptimo Taller de RISC en Silicon Valley del 28 al 30 de Noviembre de 2017.

RISC-V fue un derivado de un proyecto de investigación en Berkeley¹ que estaba desarrollando tecnología para facilitar la construcción de hardware y software paralelo.

Reconocimientos

Deseamos agradecer a Armando Fox por el uso de sus herramientas LaTeX pipeline y consejos para navegar en el mundo de la autopublicación.

Dirigimos nuestros más profundos agradecimientos a las personas que leyeron los primeros borradores del libro y ofrecieron sugerencias útiles: Krste Asanović, Nikhil Athreya, C. Gordon Bell, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor y Megan Wachs.

¡Finalmente, agradecemos a los muchos estudiantes de UC Berkeley por su ayuda en el *debugging* y su constante interés en este material!

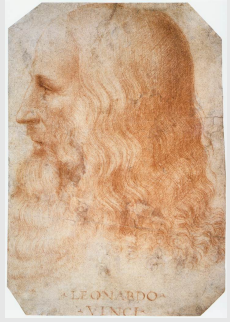
David Patterson y Andrew Waterman
16 de Noviembre de 2017
Berkeley, California

Notas

¹<http://parlab.eecs.berkeley.edu>

¿Por Qué RISC-V?

Leonardo da Vinci
(1452-1519) Arquitecto renacentista, ingeniero, escultor y pintor de la Mona Lisa.



Usamos recuadros en los márgenes para agregar comentarios que consideramos interesantes. Por ejemplo, RISC-V fue originalmente desarrollado para uso interno en investigación y cursos en UC Berkeley. Se volvió abierto porque personas externas comenzaron a utilizarlo por su cuenta. Los arquitectos de RISC-V se percataron cuando comenzaron a recibir quejas por cambios en el ISA en algunos de sus cursos disponibles en el Internet. Cuando los arquitectos entendieron la necesidad, decidieron volverlo un estándar abierto.

La sofisticación máxima es la simplicidad.

—Leonardo da Vinci

1.1 Introducción

El objetivo de *RISC-V* (“RISC cinco”) es convertirse en un *ISA* universal:

- Debe acoplarse a todos los procesadores, desde los microcontroladores más pequeños para sistemas embebidos, hasta las supercomputadoras más potentes.
- Debe funcionar bien con una amplia variedad de paquetes de software y lenguajes de programación populares.
- Debe poder implementarse en todo tipo de tecnologías: FPGAs (Field-Programmable Gate Arrays: Arreglos de compuertas programables), ASICs (Application-Specific Integrated Circuits: Circuitos integrados de propósito específico), chips personalizados, e incluso tecnologías aún no inventadas.
- Debe ser eficiente para todo tipo de micro-arquitecturas: microcódigo o control “hardwired”¹; distintos tipos de *pipelines*: en-orden, desacoplado, o fuera-de-orden²; emisión de instrucciones secuenciales simple o superescalar³; etcétera.
- Debe permitir un alto grado de especialización para utilizarse como base en aceleradores personalizados, los cuales cobran mayor importancia ahora que la Ley de Moore comienza a perder fuerza.
- Debe ser estable, implicando que el *ISA* base no debe cambiar. Aun más importante, no debe discontinuarse, como ha ocurrido en el pasado con *ISAs* propietarios como: AMD Am29000, Digital Alpha, Digital VAX, Hewlett Packard PA-RISC, Intel i860, Intel i960, Motorola 88000 y Zilog Z8000.

RISC-V es inusual no solo porque es un *ISA* reciente—nacido en esta década, mientras que la mayoría de las alternativas nacieron en los 1970s o 1980s— pero también porque es un *ISA* *abierto*. Contrastando con prácticamente todas las arquitecturas anteriores, su futuro no está atado al destino o capricho de alguna corporación, lo cual ha dañado a muchos

>\$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: Miembros corporativos de la Fundación RISC-V tomados en Mayo 2017 durante el Sexto Taller de RISC-V agrupado por ventas anuales. Las empresas de la columna izquierda, exceden los \$US 50B en ventas anuales, las empresas de la columna central venden menos de \$US 50B pero más de \$US 5B, la columna derecha son empresas con ingresos menores a \$US 5B pero más de \$US 0.5B. La fundación incluye otras 25 empresas más pequeñas, 5 startups (Antmicro Ltd, Blockstream, Esperanto Technologies, Greenwaves Technologies y SiFive), 4 organizaciones sin fines de lucro (CSEM, Draper Laboratory, ICT y lowRISC) y 6 universidades (ETH Zurich, IIT Madras, National University of Defense Technology, Princeton y UC Berkeley). La mayor parte de las 60 organizaciones tienen sus oficinas centrales fuera de EE.UU. Para saber más, dirigirse a www.riscv.org. Cantidades en Millardos de \$US (en inglés, *Billions*⁴).

ISAs en el pasado. Éste le pertenece a una fundación abierta, sin fines de lucro. El objetivo de la Fundación RISC-V es mantener la estabilidad de RISC-V, permitiéndole evolucionar únicamente por razones técnicas de manera lenta y cuidadosa, e intentar hacerlo tan popular en hardware como Linux lo es en sistemas operativos. Como muestra de su vitalidad, la Figura 1.1 enumera los miembros corporativos más importantes de la Fundación RISC-V.

1.2 ISAs Modulares vs. Incrementales

Intel apostó su futuro a un procesador de alto desempeño, sin embargo, el diseño de dicho procesador llevaría años. Para contrarrestar a Zilog, Intel desarrolló un procesador temporal llamado 8086. Dicho procesador duraría muy poco tiempo en el mercado y no tendría sucesores, pero la historia no fue esa. El procesador de alto desempeño llegó tarde al mercado y era muy lento. De esta manera, el 8086 siguió en el mercado— y evolucionó a un procesador de 32 bits y eventualmente a 64 bits. Los nombres fueron cambiando (80186, 80286, i386, i486, Pentium), pero, por cuestiones de compatibilidad, el set de instrucciones permaneció intacto.

—Stephen P. Morse, arquitecto del 8086 [Morse 2017]

El enfoque convencional en arquitectura de computadoras es desarrollar ISAs *incrementales*, en los cuales, los nuevos procesadores no solo implementan las nuevas extensiones, sino además todas las instrucciones de ISAs anteriores. El objetivo es mantener *compatibilidad-binaria* para que los programas ya compilados y en formato binario de décadas anteriores, aún puedan funcionar en los procesadores más recientes. Dicho requerimiento, combinado con la ventaja de mercadeo que daba anunciar instrucciones nuevas en cada nueva generación de procesadores, provocó un incremento sustancial en la cantidad y complejidad del ISA. Por ejemplo, la Figura 1.2 muestra el incremento en el número de instrucciones para uno de los ISAs dominantes actualmente: el 80x86. Comienza en 1978, y ha agregado alrededor de *tres instrucciones por mes* a través de su historia.

Dicha convención implica que cada implementación del x86-32 (nombre utilizado para la versión del x86 que utiliza direcciones de 32 bits) debe implementar los errores de exten-

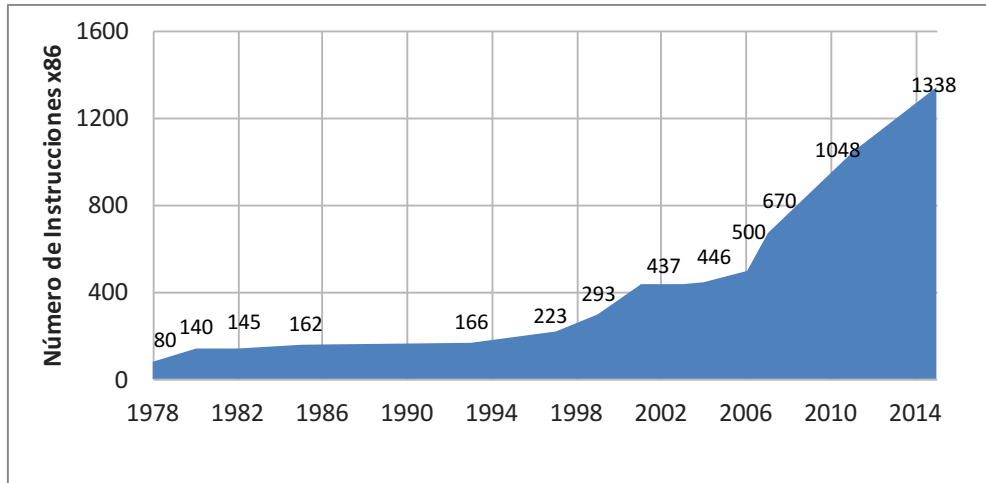


Figura 1.2: Crecimiento del set de instrucciones x86 a través de su existencia. x86 comenzó con 80 instrucciones en 1978. Para el 2015, había crecido 16X alcanzando 1338 instrucciones, y aún sigue en crecimiento. Sorprendentemente, esta gráfica es conservadora. Un blog de Intel indica que para el 2015, ya habían alcanzado las 3600 instrucciones [Rodgers and Uhlig 2017], lo cual implicaría una instrucción nueva de x86 cada cuatro días entre 1978 y 2015. Acá contamos instrucciones de lenguaje ensamblador, mientras que ellos presuntamente cuentan instrucciones de lenguaje de máquina. Como se explica en el capítulo 8, la mayor parte del crecimiento se debe a que el ISA del x86 utiliza instrucciones SIMD (Single Instruction Multiple Data: Una Instrucción, Múltiples Datos) para paralelismo a nivel de datos.

```

El registro AL es el origen y destino por defecto.
Si los 4 bits bajos del registro AL son > 9,
  o la bandera auxiliar de acarreo AF = 1,
Entonces
  Sumar 6 a los 4 bits bajos de AL y descartar overflow
  Incrementar el byte alto de AL
  Bandera de acarreo CF = 1
  Bandera auxiliar de acarreo AF = 1
De lo contrario
  CF = AF = 0
4 bits altos de AL = 0

```

Figura 1.3: Descripción de la instrucción *ASCII Adjust after Addition* (aaa) del x86-32. Ejecuta operaciones aritméticas en BCD (Binary Coded Decimal: Decimal Codificado en Binario), dichas operaciones se han vuelto obsoletas. La arquitectura x86 además cuenta con tres instrucciones similares para la resta (aas), multiplicación (aam) y división (aad). Dado que cada instrucción es de un byte, colectivamente ocupan el 1.6% (4/256) del valioso espacio para el opcode.

siones anteriores, aún cuando éstas ya no tengan sentido. Por ejemplo, Figura 1.3 describe la instrucción de x86: ASCII Adjust after Addition (aaa), la cual ya es obsoleta.

Como analogía, suponga un restaurante que solo sirve un platillo a un precio fijo, el cual comienza como una cena de una hamburguesa y un batido. Conforme pasa el tiempo, le agregan papas fritas, y luego un helado, seguido de ensalada, pie, vino, pasta vegetariana, carne, cerveza, ad infinitum hasta convertirse en un banquete gigantesco. Puede que no tenga sentido en general, pero los comensales pueden encontrar cualquier comida que hayan comido en el pasado en dicho restaurante. La mala noticia es que los comensales deben pagar el precio del gran banquete creciente en cada cena.

Además de ser reciente y abierto, RISC-V es inusual dado que, a diferencia de casi todos los ISAs anteriores, es *modular*. El núcleo fundamental del ISA es llamado *RV32I*, el cual ejecuta un stack de software completo. *RV32I* está congelado y nunca cambiará, lo cual provee un objetivo estable para desarrolladores de compiladores, sistemas operativos y programadores de lenguaje ensamblador. La modularidad viene de extensiones opcionales estándar que el hardware puede incorporar de acuerdo a las necesidades de cada aplicación.

Esta modularidad permite implementaciones muy pequeñas y de bajo consumo energético de RISC-V, lo cual puede ser crítico para aplicaciones embebidas. Al indicarle al compilador de RISC-V a través de banderas qué extensiones existen en hardware. La convención es concatenar las letras de extensión que son soportadas por dicho hardware. por ejemplo, *RV32IMFD* agrega la multiplicación (*RV32M*), punto flotante precisión simple (*RV32F*) y extensiones de punto flotante de precisión doble (*RV32D*) a las instrucciones base obligatorias (*RV32I*).

Regresando a nuestra analogía, RISC-V ofrece un menú en lugar de un buffet; el chef únicamente cocina lo que los clientes desean—no un gran banquete por cada comida—y los clientes únicamente pagan lo que ordenan. RISC-V no tiene necesidad de agregar instrucciones por cuestiones de mercadeo. La Fundación RISC-V decide cuándo agregar una nueva opción al menú, y lo hacen únicamente por razones técnicas sólidas luego de una discusión abierta por un comité de expertos en hardware y software.

Aun cuando opciones nuevas aparezcan en el menú, éstas permanecen como opcionales y no como un requerimiento para implementaciones futuras, como ISAs incrementales.

1.3 Introducción al Diseño del ISA

Antes de presentar el ISA del RISC-V, será útil entender los principios básicos y los sacrificios/compromisos que debe tomar un arquitecto de computadoras al momento de diseñar un ISA. A continuación se muestra un listado de las siete métricas. A un lado en los márgenes aparecen los íconos que utilizaremos en los siguientes capítulos para enfatizar el momento en que RISC-V hace referencia a dichas métricas (La contraportada del libro impreso tiene una ilustración con los íconos y su descripción).

- costo (ícono de la moneda de un USD)
- simplicidad (rueda)
- rendimiento (odómetro)

Si el software utiliza una instrucción omitida de RISC-V de una extensión opcional, el hardware captura el error y ejecuta la función deseada en software como parte de una librería estándar.



Costo



Simplicidad



Rendimiento

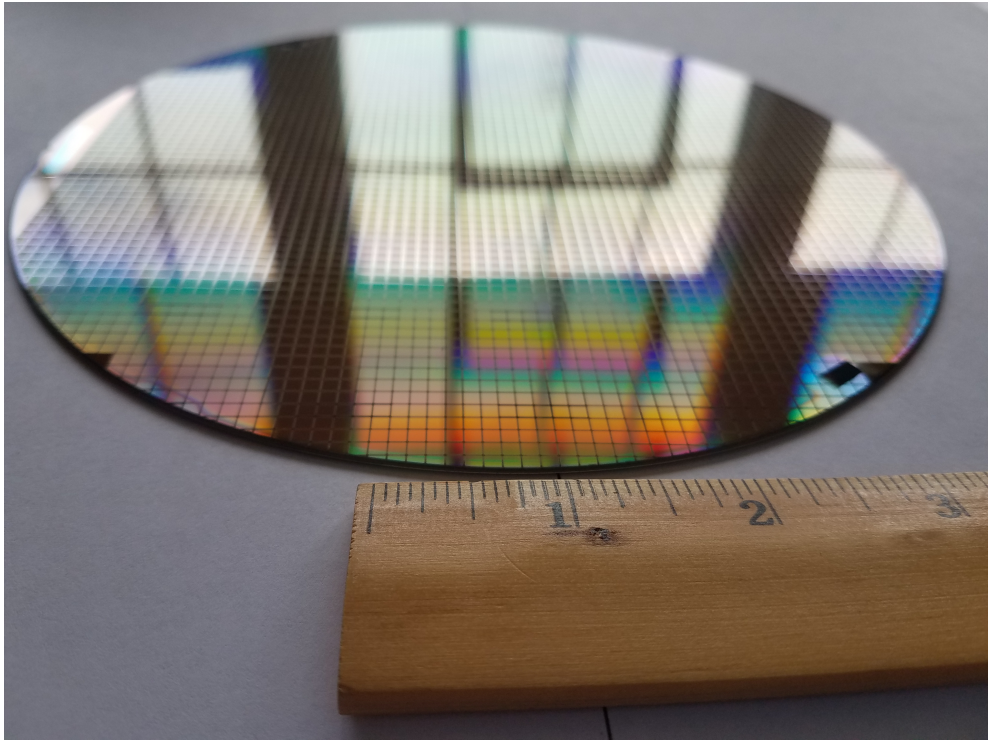


Figura 1.4: Una oblea de RISC-V de 8 pulgadas de diámetro diseñada por SiFive. Tiene dos tipos de procesadores RISC-V y utiliza una línea de procesamiento más antigua. El procesador FE310 es de $2.65\text{ mm} \times 2.72\text{ mm}$ y tiene un procesador de prueba SiFive de $2.89\text{ mm} \times 2.72\text{ mm}$. La oblea contiene 1846 chips de FE310 y 1866 de SiFive, para un total de 3712 chips.



Aislamiento de Arq e Impl



Espacio para Crecer



Tamaño del Programa



Programabilidad



Costo

- aislamiento de arquitectura e implementación (mitades aisladas de un círculo)
- espacio para crecer (acordeón)
- tamaño del programa (flechas opuestas comprimiendo una línea)
- facilidad de programar / compilar / *linkear* (“tan fácil como ABC”).

A manera ilustrativa, en esta sección mostraremos algunas decisiones que tomaron algunos ISAs que en retrospectiva entendemos que no fueron las mejores y cómo ahora con dicho conocimiento, se logró hacer una mejor labor en RISC-V.

Costo. Los procesadores son implementados como circuitos integrados, comúnmente denominados *chips* o *dies*. Se llaman “dies” (pronunciado “dais”) dado que comienzan como una oblea redonda, el cual es cortado (*diced* en inglés) en múltiples pedacitos. Figura 1.4 muestra una oblea de procesadores RISC-V. El costo es muy sensible al tamaño del die:

$$\text{costo} \approx f(\text{área del die}^2)$$

Obviamente, entre más pequeño el die, más dies por oblea, y la mayor parte del costo del die es el costo de la oblea. Un poco menos obvio es que entre más pequeño el die, hay un mayor rendimiento, el porcentaje de dies fabricados que funcionan. La razón para esto es que al fabricar la oblea, siempre hay algunos desperfectos que aparecen aleatoriamente, si el desperfecto cae sobre algún chip, el chip completo resulta defectuoso.

Un arquitecto desea conservar el ISA simple para reducir el tamaño del procesador que lo implementa. Como veremos en los siguientes capítulos, el ISA del RISC-V es mucho más sencillo que el ISA de ARM-32. Como un ejemplo concreto del impacto de la simplicidad, comparemos un procesador RISC-V Rocket a un ARM-32 Cortex-A5, ambos utilizando la misma tecnología (TSMC40GPLUS) y el mismo tamaño de cache (16 KiB). El die de RISC-V es 0.27 mm^2 vs 0.53 mm^2 para ARM-32. Dado que es casi el doble del área, el costo del die del ARM-32 Cortex-A5 es aproximadamente 4 veces (2^2) mayor que el del RISC-V Rocket. Aun una reducción del 10% del tamaño del die, reduce el costo en un factor de 1.2 (1.1^2).

Simplicidad. Dada la correlación costo-complejidad, los arquitectos prefieren un ISA simple para reducir el tamaño del die. La simplicidad también reduce el tiempo requerido para diseñar y validar, lo cual es un porcentaje importante del costo de desarrollo de un chip. Dichos costos deben ser agregados al costo del procesador siendo este incremento dependiente de la cantidad de procesadores vendidos. La simplicidad también reduce el costo de la documentación y la dificultad de hacer que los clientes entiendan cómo usar el ISA.

A continuación, un ejemplo de la complejidad del ISA de ARM-32:

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

Dicha instrucción significa LoaD Multiple, Increment-Address, on EQual. Ejecuta 5 loads de datos y escribe a 6 registros pero se ejecuta únicamente si la bandera EQ está activa. Además, escribe el resultado al PC, implicando un branch condicional. ¡Qué enredo!

Irónicamente, instrucciones más simples tienden a ser más utilizadas que las complejas. Por ejemplo, x86-32 incluye la instrucción `enter`, la cual supuestamente debe ser la primera instrucción a ejecutarse al entrar a un procedimiento para crear el stack frame (ver Capítulo 3). Sin embargo, la mayoría de los compiladores, utilizan estas dos simples instrucciones:

```
push ebp      # Hace push del frame pointer al stack
mov  ebp, esp # Copia el stack pointer al frame pointer
```

Rendimiento. A excepción de chips muy pequeños utilizados para aplicaciones embebidas, los arquitectos se preocupan tanto por rendimiento como costo. Tres factores que afectan el rendimiento son:

$$\frac{\text{instrucciones}}{\text{programa}} \times \frac{\text{ciclos promedio}}{\text{instrucción}} \times \frac{\text{tiempo}}{\text{ciclo de reloj}} = \frac{\text{tiempo}}{\text{programa}}$$

Aun cuando un ISA más sencillo ejecute más instrucciones que un ISA complejo, este programa puede ejecutarse más rápido si la frecuencia de reloj es más rápida o si promedian menos CPI (Cycles Per Instruction: Ciclos Por Instrucción).

Por ejemplo, para la *prueba de rendimiento*⁵ CoreMark [Gal-On and Levy 2012] (100,000 iteraciones), el rendimiento para el ARM-32 Cortex-A9 es

$$\frac{32.27 \text{ B instrucciones}}{\text{programa}} \times \frac{0.79 \text{ ciclos de reloj}}{\text{instrucción}} \times \frac{0.71 \text{ ns}}{\text{ciclos de reloj}} = \frac{18.15 \text{ segs}}{\text{programa}}$$

Procesadores de alta gama pueden obtener mayor desempeño combinando instrucciones simples sin sobrecargar implementaciones de gama baja con un ISA más complejo. A esta técnica se le denomina *macro-fusión*, dado que fusiona "macro" instrucciones.



Simplicidad

Un procesador simple es beneficioso en aplicaciones embebidas dado que es más fácil predecir el tiempo de ejecución. Programadores de lenguaje ensamblador para microcontroladores muchas veces quieren mantener tiempos precisos, por lo que confían en que su código se ejecutará en una cantidad de ciclos predecible, la cual pueda ser calculada con antelación.



Rendimiento

El último factor es el inverso de la frecuencia de reloj, así que una frecuencia de 1 GHz significa que el período es de 1 ns ($1/10^9$).

El promedio de ciclos puede ser menor a 1 dado que el A9 y BOOM [Celio et al. 2015] son procesadores *superescalares*, los cuales ejecutan más de una instrucción por ciclo de reloj.

Para la implementación de BOOM en RISC-V, la ecuación es:

$$\frac{29.51 \text{ B instrucciones}}{\text{programa}} \times \frac{0.72 \text{ ciclos de reloj}}{\text{instrucción}} \times \frac{0.67 \text{ ns}}{\text{ciclos de reloj}} = \frac{14.26 \text{ segs}}{\text{programa}}$$

El procesador ARM no ejecutó menos instrucciones que RISC-V en este caso. Como veremos, las instrucciones simples son además las más populares, así que un ISA más simple puede ganar en todas las métricas. Para este programa, el procesador RISC-V aventaja cerca de un 10% en cada uno de los tres factores, lo cual resulta en un rendimiento casi 30% más rápido. Si un ISA más simple además resulta en un chip más pequeño, su relación costo-rendimiento será muy buena.

Aislamiento de Arquitectura e Implementación. La distinción original entre *arquitectura e implementación*, la cual data de los 1960s, es que un programador de lenguaje máquina debe conocer la arquitectura para escribir programas correctos, pero tiene poco que ver con el rendimiento. Una gran tentación para un arquitecto es incluir instrucciones al ISA que mejoren el rendimiento o costo de una implementación particular, a costa de implementaciones futuras.

Para el ISA de MIPS-32, un ejemplo lamentable fue el branch retardado⁶. Branches condicionales dan problemas con el pipeline dado que el sistema necesita tener la siguiente instrucción lista para ejecutarse, sin embargo, dado que la siguiente instrucción depende si la condición se cumple (en cuyo caso ejecuta la instrucción a la que apunta el branch) de lo contrario ejecuta la siguiente instrucción. Para su primer procesador con un pipeline de 5 etapas, esta incertidumbre causaba un retraso de un ciclo de reloj. MIPS-32 resolvió este problema redefiniendo la instrucción para que el branch se tomara *después* de la siguiente instrucción. Esto implica que la instrucción que viene luego de un branch *siempre* se ejecuta. Es tarea del programador o compilador poner una instrucción “útil” luego del hueco de retardo⁷.

Dicha “solución” no benefició a implementaciones posteriores de procesadores MIPS-32 los cuales contaban con pipelines de muchas más etapas (lo que implicaba un atraso mucho mayor), pero dado que los ISAs incrementales requieren soporte de todas las arquitecturas anteriores (backward compatibility) (ver Sección 1.2), esto le hizo la vida más difícil a programadores de MIPS-32, escritores de compiladores y diseñadores de procesadores. Además, hace que MIPS-32 sea más difícil de entender (ver Figura 2.10 en página 31).

A pesar que los arquitectos no deberían agregar funciones que *ayudan* a una implementación específica, tampoco deberían agregar funciones que *afecten* algunas implementaciones. Por ejemplo, ARM-32 y otros ISAs tienen una instrucción de Load múltiple. Estas instrucciones pueden mejorar el rendimiento de diseños de pipelines que emiten una instrucción a la vez, pero empeoran el rendimiento de pipelines que emiten múltiples instrucciones. La razón es que la implementación más directa no permite realizar cargas múltiples en paralelo con otras instrucciones, reduciendo el throughput de instrucciones en dichos procesadores.

Espacio para Crecer. Con el desvanecimiento de la Ley de Moore, el único camino para incrementar significativamente el rendimiento es agregar instrucciones para tareas específicas, tales como redes neurales, realidad aumentada, optimizaciones combinatorias, gráficos, etc. Esto implica que es fundamental reservar espacio en el opcode para dichas instrucciones.

En los 1970s y 1980s, cuando la Ley de Moore estaba en su apogeo, casi nadie pensaba en reservar espacio para los opcodes futuros. Los arquitectos preferían direcciones más largas y valores inmediatos mayores para reducir el número de instrucciones por programa lo cual es el primer factor en rendimiento (página anterior).



Aislamiento de Arq e Impl

Pipelines modernos anticipan la instrucción a ejecutarse

utilizando predictores de hardware, los cuales exceden el 90% de exactitud y funcionan con cualquier cantidad de etapas. Únicamente necesitan un mecanismo para limpiar el pipeline en caso haya una predicción equivocada.



Espacio para Crecer

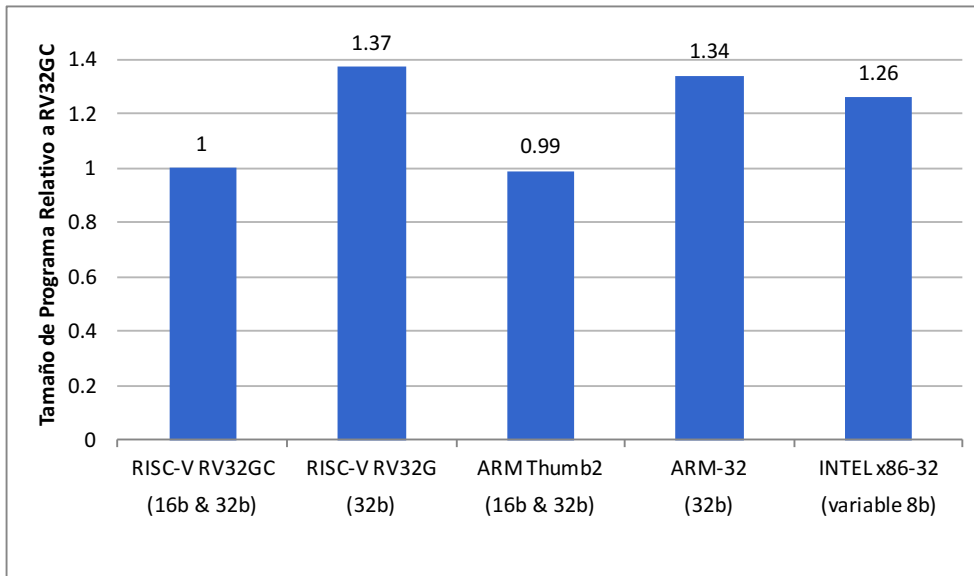


Figura 1.5: Tamaños de programa relativos para RV32G, ARM-32, x86-32, RV32C y Thumb-2. El enfoque de los últimos dos ISAs es código pequeño. Se utilizó la prueba de rendimiento SPEC CPU2006 y compiladores GCC. La leve ventaja de Thumb-2 sobre RV32C se debe al ahorro de *Load y Store Multiple* al entrar a funciones. RV32C las excluye para mantener un mapeo uno-a-uno con instrucciones de RV32G, el cual las omite para reducir la complejidad de implementación en procesadores de alto desempeño (ver abajo). El Capítulo 7 explica RV32C. RV32G indica una combinación popular de extensiones RISC-V (RV32M, RV32F, RV32D y RV32A), apropiadamente llamada RV32IMAFD. [Waterman 2016]

Un ejemplo del impacto en la escasez en opcodes fue cuando los arquitectos de ARM-32 intentaron reducir el tamaño del código agregando instrucciones de 16 bits al ISA (anteriormente uniforme) de 32 bits. Simplemente no había espacio disponible. La solución fue crear un nuevo ISA de 16 bits (Thumb) y luego un ISA nuevo con instrucciones de 16 y 32 bits (Thumb-2), el cual utilizaba un bit para cambiar entre ellos. Para cambiar de modalidad, el programador o compilador debe saltar a una dirección con un 1 en el bit menos significativo, lo cual funciona porque tanto las instrucciones de 16 y 32 bits deben tener un 0 allí.

Tamaño del Programa. Entre más pequeño el programa, menor el área del chip que ocupará la memoria, lo cual puede ser un costo significativo para sistemas embebidos. Por esto, los arquitectos de ARM agregaron instrucciones pequeñas en los ISAs Thumb y Thumb-2. Programas más pequeños además implican menos *instruction cache misses*, lo cual ahorra energía porque accesos a memoria DRAM externa consumen mucho más energía que accesos a SRAM en el chip. Generar código pequeño puede ser un objetivo del arquitecto del ISA.

El ISA x86-32 tiene instrucciones que van desde 1 byte hasta 15 bytes. Uno esperaría que el tamaño variable en instrucciones del x86 condujera a código más pequeño que ISAs donde todas las instrucciones son de 32 bits, como ARM-32 o RISC-V. Lógicamente, instrucciones variables de 8 bits deberían ser más pequeñas que ISAs que únicamente ofrecen instrucciones de 16 ó 32 bits, así como Thumb-2 y RISC-V usando extensiones RV32C (ver Capítulo 7).

La instrucción de ARM-32 `ldmiaeq` mencionada anteriormente es aun más complicada, dado que al hacer un salto también puede cambiar el modo de operación entre ARM-32 and Thumb/Thumb-2.



Tamaño del Programa

Un ejemplo de instrucción de x86-32 de 15 bytes es

```
lock add dword
ptr ds: [esi+ecx*4
+0x12345678] ,
0xefcdab89. Ensambla
en hexadecimal a: 67 66
f0 3e 81 84 8e 78563412
89abcdef. Los últimos 8
bytes son 2 direcciones
y los primeros 7 bytes
especifican operaciones
de memoria atómicas, la
operación suma, datos
de 32-bits, el registro
del segmento de datos,
los dos registros de di-
recciones y el modo de
direccionamiento escalado
indexado. Un ejemplo de
instrucción de 1-byte es
inc eax, que ensambla
a 40.
```



La Figura 1.5 muestra esto, mientras que el código de ARM-32 y RISC-V es del 6% al 9% mayor que el código del x86-32 cuando todas las instrucciones son de 32 bits, sorprendentemente, x86-32 es 26% *mayor* que las versiones comprimidas (RV32C y Thumb-2) que ofrecen instrucciones de 16 y 32 bits.

A pesar de que probablemente un nuevo ISA de instrucciones variables de 8 bits llevaría a código más compacto que RV32C y Thumb-2, los arquitectos del ISA x86 en los 70s tenían otras preocupaciones. Además, dado el requerimiento de retrocompatibilidad binaria de un ISA incremental (Sección 1.2), los cientos de instrucciones nuevas son mayores de lo esperado, dado que quedó muy poco espacio para el opcode en el x86 original.

Facilidad de programar, compilar y *linkear*. Dado que el acceso de datos en registros es mucho más rápido que en memoria, es crucial que los compiladores hagan una buena asignación de registros. Dicha tarea es más fácil entre más registros se tenga. Bajo esa perspectiva, ARM-32 tiene 16 registros y x86-32 únicamente 8. La mayoría de ISAs modernos, incluyendo RISC-V, tienen una cantidad relativamente generosa de 32 registros enteros. Más registros hacen la tarea más fácil para programadores de ensamblador y compiladores.

Otra complicación para programadores de ensamblador y compiladores es entender la velocidad de ejecución. Como veremos, las instrucciones de RISC-V normalmente se ejecutan en un ciclo de reloj (ignorando cache misses), pero como vimos antes, tanto ARM-32 como x86-32 tienen instrucciones que llevan varios ciclos de reloj aun sin cache misses. Es más, a diferencia de ARM-32 y RISC-V, las instrucciones aritméticas del x86-32 pueden tener operandos en memoria en vez de solo registros. Instrucciones complejas y operandos en memoria incrementan la dificultad para los diseñadores de procesadores en proveer estimaciones de desempeño.

Es útil si el ISA soporta PIC (Position Independent Code: Código independiente de su posición), dado que soporta *dynamic linking* (ver Sección 3.5), porque el código de la librería compartida puede estar en diferentes direcciones en distintos programas. Saltos relativos al PC y direccionamiento de datos son una ventaja para PIC. Mientras casi todos los ISAs proveen saltos relativos al PC, x86-32 y MIPS-32 omiten el direccionamiento de datos relativo al PC.

■ **Elaboración: ARM-32, MIPS-32 y x86-32**

Profundizaciones son secciones opcionales en las que el lector puede entrar en detalle si le interesa el tema, pero no son necesarias para entender el resto del libro. Por ejemplo, nuestros nombres de ISAs no son los oficiales. El ISA de ARM con direcciones de 32 bits tiene muchas versiones, la primera en 1986 y la última llamada ARMv7 en el 2005. ARM-32 generalmente se refiere al ISA ARM7. MIPS también tuvo muchas versiones de 32 bits, pero hacemos referencia al original llamado MIPS I. (“MIPS32” es un ISA más moderno al cual llamamos MIPS-32) La primer arquitectura de 16 bits de Intel fue el 8086 en 1978, el cual fue expandido a direcciones de 32 bits por el 80386 en 1985. Nuestra notación de x86-32 generalmente se refiere a IA-32. Dada la cantidad de variantes de estos ISAs, creemos que nuestra terminología poco estándar es menos confusa.

1.4 Un Vistazo al Libro

Este libro asume que el lector ha visto otros ISAs antes de RISC-V. De lo contrario, recomendamos nuestro libro introductorio basado en RISC-V [Patterson and Hennessy 2017].

El Capítulo 2 introduce RV32I, la base fija de instrucciones con enteros que son el fundamento de RISC-V. El Capítulo 3 explica el resto de instrucciones de ensamblador además de las indicadas en El Capítulo 2, incluyendo convenios para llamadas a funciones y algunos trucos inteligentes para *linking*. El lenguaje ensamblador incluye todas las instrucciones de RISC-V y algunas instrucciones externas útiles. Dichas *pseudo-instrucciones*, variantes ingeniosas de instrucciones reales, hacen más fácil programar en ensamblador sin complicar el ISA.

Los siguientes tres capítulos explican las extensiones estándar de RISC-V a las cuales, combinadas con RV32I, llamamos RV32G (G es de general):

- Capítulo 4: Multiplicación y División (RV32M)
- Capítulo 5: Punto Flotante (RV32F y RV32D)
- Capítulo 6: Atómicas (RV32A)

La “guía de referencia” de RISC-V en las páginas 3 y 4 es un resumen práctico de *todas* las instrucciones de RISC-V en este libro: RV32G, RV64G, y RV32/64V.

El Capítulo 7 describe las extensiones opcionales comprimidas RV32C, un excelente ejemplo de la elegancia de RISC-V. Restringiendo las instrucciones de 16 bits a versiones cortas de instrucciones existentes de 32 bits RV32G, son casi gratis. El ensamblador puede elegir el tamaño apropiado de la instrucción, permitiendo al programador y compilador despreocuparse de RV32C. El decodificador de hardware que traduce de instrucciones RV32C de 16 bits a RV32G de 32 bits únicamente requiere de 400 compuertas, lo cual es un porcentaje muy pequeño incluso en la implementación más simple de RISC V.

El Capítulo 8 introduce RV32V, la extensión vectorizada. Instrucciones vectorizadas son otro ejemplo de la elegancia del ISA en comparación con las innumerables instrucciones *SIMD* de ARM-32, MIPS-32 y x86-32. En efecto, cientos de las instrucciones agregadas a x86-32 en la Figura 1.2 eran SIMD y cientos más vendrán. RV32V es aun más simple que la mayoría de los ISAs vectorizados dado que asocia el tipo de dato y longitud con los registros vectorizados en lugar de los opcodes. Ciertamente, RV32V podría ser la razón más poderosa para cambiarse de un ISA basado en SIMD a RISC-V.

El Capítulo 9 muestra la versión de 64 bits de RISC-V, RV64G. Los arquitectos de RISC-V únicamente ampliaron los registros y modificaron un par de instrucciones de RV32G para soportar direcciones de 64 bits word, doubleword, or long

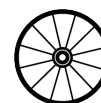
El Capítulo 10 explica instrucciones del sistema, mostrando cómo RISC-V maneja *paging* y los modos privilegiados Supervisor, Máquina y Usuario.

El último capítulo da una breve descripción de las extensiones que están bajo consideración por parte de la Fundación RISC-V.

Luego, la sección más grande del libro, el Apéndice A, da un resumen en orden alfabético del ISA RISC-V. Define todas las instrucciones, pseudo-instrucciones y extensiones mencionadas en unas 50 páginas, un testimonio de la simplicidad de RISC-V.

El Apéndice B muestra operaciones comunes en lenguaje ensamblador y su correspondencia con RV32I, ARM-32, y x86-32. Esas tres figuras son acompañadas por un pequeño

A la guía de referencia también le llamamos tarjeta verde porque en los años 60s, se utilizaba una página de cartón de color verde con el resumen de instrucciones. Preferimos utilizar un fondo blanco por legibilidad en lugar de verde por exactitud histórica.



Simplicidad

ISA	Páginas	Palabras	Horas para leer	Semanas para leer
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 y palabras en manuales de ISAs [Waterman and Asanović 2017a], [Waterman and Asanović 2017b], [Intel Corporation 2016], [ARM Ltd. 2014]. Horas y semanas para completar asumen leer 200 palabras por minuto, 40 horas a la semana. Basado en parte en [Baumann 2017].

programa en C y la salida del compilador para los tres ISAs. El apéndice cumple dos propósitos. Para aquellos lectores familiarizados con ARM-32 o x86-32, es una manera de aprender RISC-V, comparándolo con el ISA que ya conocen. Además, ayuda a programadores que están convirtiendo programas de otros ISAs hacia RISC-V.

Concluimos el libro con el índice.

1.5 Observaciones Finales

Utilizando métodos formales-lógicos es fácil mostrar que existen ciertos [sets de instrucciones] que pueden controlar y producir la ejecución de cualquier secuencia de operaciones... Lo realmente importante para el punto de vista actual al seleccionar un [set de instrucciones] tiene una naturaleza más práctica: simplicidad del equipo demandado por el [set de instrucciones], y la claridad de su aplicación para los problemas que realmente importan en conjunto con la velocidad de trabajar dichos problemas.

—[von Neumann et al. 1947, 1947]

RISC-V es un ISA reciente, abierto, minimalista y que arranca de cero informándose de los errores de de ISAs anteriores. El objetivo de los arquitectos de RISC-V es que sea eficaz para todos los dispositivos de cómputo, desde los más pequeños hasta los más rápidos. Siguiendo el consejo de von Neumann de hace más de 70 años, este ISA enfatiza simplicidad para mantener el costo bajo y muchos registros y velocidad de ejecución transparente para ayudar a compiladores y programadores a resolver problemas importantes de manera eficiente.

Un indicador de complejidad es el tamaño de la documentación. La Figura 1.6 muestra el tamaño del manual del set de instrucciones para RISC-V, ARM-32 y x86-32 medidos en páginas y palabras. Si leyeras manuales como un trabajo de tiempo completo—8 horas, 5 días a la semana—tomaría medio mes darle una pasada al manual de ARM-32, y el mes completo para x86-32. A este nivel de complejidad, probablemente nadie entiende completamente ARM-32 o x86-32. Utilizando esta métrica, RISC-V es $\frac{1}{12}$ más fácil que ARM-32 y de $\frac{1}{10}$ a $\frac{1}{30}$ más fácil que x86-32. Incluso, el resumen del ISA de RISC-V incluyendo todas las extensiones de únicamente dos páginas (ver guía de referencia).

Este ISA abierto y minimalista fue develado en el 2011 y actualmente es apoyado por una fundación encargada de evolucionarlo agregando extensiones opcionales basadas en justificaciones técnicas luego de debates prolongados. Esta apertura permite implementaciones gratis, compartidas de RISC-V, lo cual reduce costos y la posibilidad de secretos maliciosos ocultos en el procesador.

Sin embargo, únicamente el hardware no forma un sistema. Costos del desarrollo de software tienden a ser mayores que los de hardware, por lo que aunque hardware estable es importante, software estable lo es más. Necesita sistemas operativos, boot-loaders, software

Una versión anterior del excelente reporte de John von Neumann fue tan influyente que a este tipo de computador le llamamos *arquitectura von Neumann*, a pesar que este reporte está basado en el trabajo de otras personas. ¡Fue escrito tres años antes de que la primer computadora con programa almacenado fuera operacional!



Simplicidad

de referencia y herramientas de software populares. La fundación ofrece estabilidad para el ISA, y la base fija RV32I implica que el stack de software no cambiará. Dada su rápida adopción y apertura, RISC-V podría desafiar el dominio de los ISAs propietarios prevalecientes.

Elegancia es una palabra rara vez aplicada a los ISAs, pero, una vez concluido este libro, quizás concordemos que aplica a RISC-V. Resaltaremos características que creemos indican elegancia con un ícono de la Mona Lisa en los márgenes.



Elegancia

1.6 Para Aprender Más

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/>.

Notas

¹Hardwired: Implementado en hardware, por lo que no puede ser modificado.

²Pipelines en-orden, desacoplados y fuera de orden: En inglés, *In-order, decoupled, and out-of-order pipelines*.

³Emisión de instrucciones simple o superescalar: En inglés, *Single or superscalar instruction issue*.

⁴Tradicionalmente, la palabra del idioma inglés *Billion* significa miles de millones, en cambio la palabra *Billón* del idioma español significa millones de millones. En las cifras presentadas en este texto nos referimos a miles de millones, o sea *Millardos*. Por ejemplo, \$US 1B significa un millardo de dólares americanos.

⁵Prueba de rendimiento (en inglés, *Benchmark*): Conjunto de programas destinados a recaudar métricas de rendimiento de sistemas computacionales.

⁶Branch retardado: En inglés, *delayed branch*.

⁷Hueco de retardo: En inglés, *delay slot*.

RV32I: ISA RISC-V Base para Números Enteros

Frances Elizabeth “Fran” Allen (1932-) le fue otorgado el *Premio Turing*, principalmente por su trabajo optimizando compiladores. El *Premio Turing* es el galardón más importante en Ciencias de la Computación.



... la única manera realista de alcanzar las metas de rendimiento y hacerlas accesibles al usuario es diseñando el compilador y computador al mismo tiempo. De esta manera no se implementaría en hardware aquello que el software no pueda usar ...

—Frances Elizabeth “Fran” Allen, 1981

2.1 Introducción

La Figura 2.1 es una representación gráfica del set de instrucciones base RV32I. Es posible ver el set de instrucciones RV32I completo concatenando las letras subrayadas de izquierda a derecha para cada diagrama. La notación de conjunto usando { } lista las posibles variaciones de la instrucción, utilizando ya sea letras subrayadas o guión bajo $_$, lo cual significa no agregar letras en esta variante. Por ejemplo

$$\text{set less than } \left\{ \begin{array}{c} _ \\ \text{immediate} \end{array} \right\} \left\{ \begin{array}{c} _ \\ \text{unsigned} \end{array} \right\}$$

representa estas cuatro instrucciones RV32I: `slt`, `slti`, `sltu`, `sltiu`.

El objetivo de estos diagramas, los cuales serán la primera figura en los siguientes capítulos, es dar un vistazo rápido pero esclarecedor de las instrucciones del capítulo.

2.2 Formato de Instrucciones RV32I



Simplicidad



Costo



Rendimiento

La Figura 2.2 muestra los seis formatos de instrucciones básicos: tipo-R para operaciones entre registros; tipo-I para inmediatos cortos y loads; tipo-S para *stores*; tipo-B para *branches*; tipo-U para inmediatos largos; y tipo-J para saltos incondicionales. La Figura 2.3 muestra los opcodes para las instrucciones RV32I en la Figura 2.1 usando los formatos de la Figura 2.2.

Incluso el formato de instrucciones muestra ejemplos donde el diseño simple de RISC-V mejora el costo-rendimiento. Primero, únicamente hay seis formatos y todas las instrucciones son de 32 bits, simplificando la decodificación de instrucciones. ARM-32 y particularmente x86-32 tienen muchos formatos, haciendo la decodificación costosa en implementaciones de gama baja y un reto para procesadores de gama alta y media. Segundo, las instrucciones de RISC-V ofrecen operandos de tres registros, en vez de tener un campo compartido para origen y destino, como lo hace x86-32. Cuando una operación tiene tres operandos

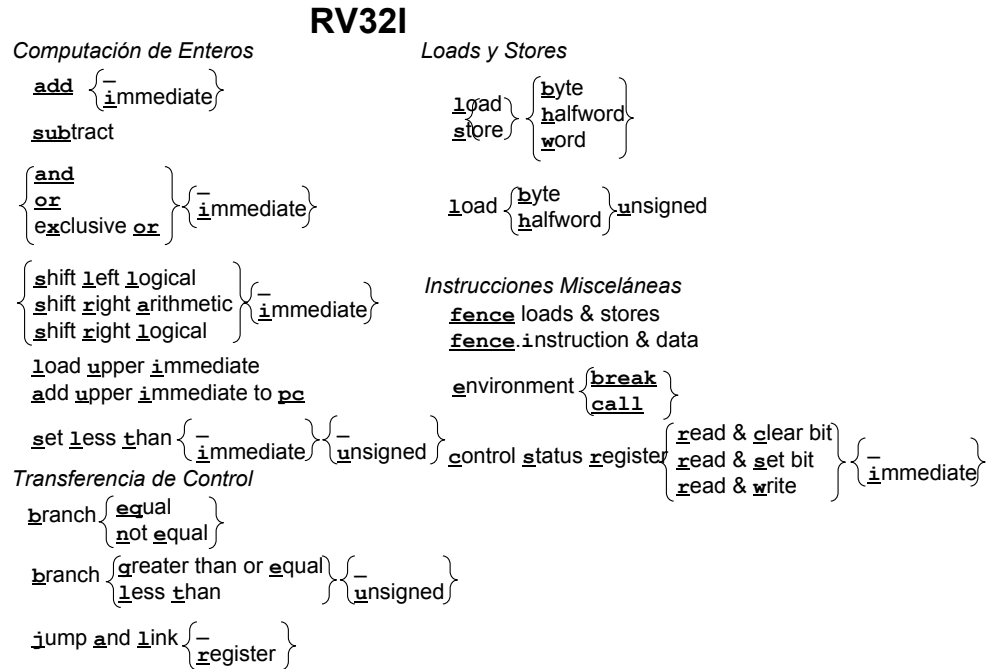


Figura 2.1: Diagrama de las instrucciones RV32I. Las letras subrayadas son concatenadas de izquierda a derecha para formar instrucciones RV32I. La notación de llaves { } implica que cada ítem vertical en el conjunto es una variante distinta de la instrucción. El guión bajo _ en un conjunto significa que una opción es simplemente el nombre de la instrucción hasta el momento, sin agregar letras de este conjunto. Por ejemplo, la notación cercana a la esquina superior izquierda representa las siguientes instrucciones: and, or, xor, andi, ori, xori.

	31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
	funct7			rs2			rs1	funct3		rd			opcode			Tipo R
	imm[11:0]						rs1	funct3	rd			opcode			Tipo I	
	imm[11:5]			rs2			rs1	funct3	imm[4:0]			opcode			Tipo S	
	imm[12]	imm[10:5]			rs2			rs1	funct3	imm[4:1]	imm[11]	opcode			Tipo B	
	imm[31:12]						rd			opcode			Tipo U			
	imm[20]	imm[10:1]			imm[11]	imm[19:12]			rd			opcode			Tipo J	

Figura 2.2: Formato de instrucciones RISC-V. Etiquetamos cada subcampo inmediato con la posición del bit (imm[x]) en el valor inmediato producido, en vez de la acostumbrada posición del bit en el campo inmediato de la instrucción. El Capítulo 10 explica cómo las instrucciones del *control status register* utilizan el formato tipo-I de otra manera. (La Figura 2.2 de Waterman and Asanović 2017 es la base de esta 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 csrww		
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: El mapa de opcodes de RV32I tiene la estructura de la instrucción, opcodes, tipo de formato y nombres (La Tabla 19.2 de [Waterman and Asanović 2017] es la base de esta figura).

distintos, pero el ISA solo provee una instrucción con dos, el compilador o programador de ensamblador deben usar una instrucción adicional para preservar el operando destino. Tercero, en RISC-V los bits de los registros a ser leídos y escritos van en la misma posición para todas las instrucciones, implicando que se puede comenzar a acceder a dichos registros antes de la decodificación. Muchos ISAs (ej. ARM-32 y MIPS-32) utilizan un campo como origen en algunas instrucciones y como destino en otras, forzando el agregar hardware extra en sitios potencialmente críticos (en tiempo) para seleccionar el campo correcto. Cuarto, los campos inmediatos en estos formatos siempre son extendidos en signo, y el bit del signo siempre está en el bit más significativo de la instrucción. Esta decisión implica que la extensión de signo del inmediato (lo cual también puede estar en un área crítica en el tiempo), puede continuar antes de la decodificación.

■ **Elaboración: ¿Formatos tipo B y J?**

El campo inmediato se rota 1 bit para instrucciones de branch, una variación del formato S que renombramos *formato B*. El campo inmediato también es rotado para instrucciones de salto, una variación del formato U renombrado *formato J*. Por lo tanto, en realidad hay cuatro formatos básicos, pero siendo conservadores podemos decir que hay 6 formatos en RISC-V.

Para ayudar al programador, una instrucción con todos los bits en cero es ilegal en RV32I. Por esta razón, un salto erróneo a regiones de memoria en cero inmediatamente levantará una excepción, lo cual ayuda en el debugging. Similarmente, una instrucción con todos los bits en uno también es ilegal, lo cual capturará otros errores comunes como dispositivos de memoria no volátil no programados, buses de memoria desconectados o chips de memoria descompuestos.

Para dejar suficiente espacio para extensiones del ISA, el ISA RV32I base usa menos de $\frac{1}{8}$ del espacio de codificación de los 32 bits. Los arquitectos además eligieron cuidadosamente los opcodes de RV32I para que instrucciones con *datapaths* similares compartan la mayor cantidad de bits posible, simplificando así la lógica de control. Finalmente, como veremos, las direcciones de los branches y jumps en los formatos B y J deben ser corridas un bit hacia la izquierda para multiplicar la dirección por 2, dándole a ambas mayor rango. RISC-V rota los bits en los operandos inmediatos desde un posicionamiento natural para reducir el *fanout* de la señal de instrucciones y el costo de la multiplexión inmediata casi por un factor de dos, lo cual a su vez simplifica la lógica del datapath en implementaciones de gama baja.

¿Qué es Diferente? Terminaremos cada sección en este y los siguientes capítulos describiendo cómo RISC-V difiere de otros ISAs. Generalmente contrastamos contra lo que le falta a RISC-V. Los arquitectos demuestran su buen gusto tanto por las características que incluyen como por las que omiten.

El campo inmediato de 12 bits de ARM-32 no es simplemente una constante, sino la entrada de una función que produce una constante: 8 bits se extienden con ceros y luego se rotan hacia la derecha usando el valor de los cuatro bits restantes multiplicado por 2. La esperanza era que codificando mejores constantes en 12 bits reduciría las instrucciones ejecutadas. ARM-32 también dedica cuatro preciados bits en la mayoría de formatos de instrucciones para ejecución condicional. A pesar de que casi nunca se usa, la ejecución condicional agrega complejidad a *procesadores fuera-de-orden*.

La extensión de signo en inmediatos también ayudan a las instrucciones lógicas.

Por ejemplo, `x & 0xfffff0` usa únicamente la instrucción `andi` en RISC-V, pero requiere dos instrucciones en MIPS-32 (`addiu` para cargar la constante, luego `and`), dado que MIPS extiende con cero los inmediatos lógicos. ARM-32 tenía que agregar una instrucción adicional, `bic`, que ejecuta `rx & inmediato` para compensar la extensión con ceros.



Programabilidad



Espacio para Crecer



Costo

Todas las implementaciones de RISC-V usan los mismos opcodes para extensiones opcionales como RV32M, RV32F, etc.

Extensiones no estándar que son únicas a un procesador tienen un espacio de opcode reservado y restringido.



Rendimiento

■ Elaboración: Los procesadores fuera-de-orden

son procesadores con pipeline de alta velocidad que ejecutan instrucciones oportunísticamente en lugar de bloquearse y continuar secuencialmente. Una característica crítica de estos procesadores es *renombrar registros*, la cual mapea los nombres de los registros en el programa a una cantidad mayor de registros internos físicos. El problema con la ejecución condicional es que el nuevo registro físico debe escribirse independientemente si se cumple la condición, por lo que el valor anterior del registro destino debe leerse como un tercer operando, en caso sea necesario restaurarlo si la condición no se cumple. Dicho operando adicional incrementa el costo del *register file*, del renombrador de registros y del hardware de ejecución *fuera-de-orden*.

2.3 Registros de RV32I



Simplicidad

Pipelining es usado en casi todos los procesadores actuales para obtener buen desempeño. Como una línea de ensamble, logran un mejor *throughput* traslapando la ejecución de varias instrucciones al mismo tiempo. Para lograrlo, los procesadores predicen el resultado de branches con una exactitud mayor a 90%. Cuando falla la predicción, re-ejecutan instrucciones. Procesadores primitivos tenían pipelines de 5 etapas, o sea que ejecutaban 5 instrucciones traslapadas. Los más recientes tienen más de 10 etapas. ARM v8, sucesor de ARM-32 dejó de usar el PC como registro de propósito general, admitiendo que fue un error.



Simplicidad

La Figura 2.4 muestra los registros de RV32I y sus nombres determinados por el ABI (Application Binary Interface: Interfaz Binaria de Aplicaciones) de RISC-V. En nuestros ejemplos de código usaremos los nombres ABI para facilitar su lectura. Para el agrado de los programadores de ensamblador y compiladores, RV32I tiene 31 registros, más x0, que siempre tiene el valor 0. ¡ARM-32 tiene apenas 16 registros mientras que x86-32 solo tiene 8!

¿Qué es diferente? Tener un registro a cero tiene un impacto tremendo en simplificar el ISA de RISC-V. La Figura 3.3 en la página 38 en el Capítulo 3 provee varios ejemplos de operaciones que son nativas en ARM-32 y x86-32, que no tienen el registro cero, pero pueden ser creadas a partir de instrucciones de RV32I simplemente usando el registro cero como operando.

El PC es uno de los 16 registros de ARM-32, lo cual implica que cualquier instrucción que modifica un registro puede ser, como efecto secundario, una instrucción de branch. El PC como registro complica la predicción de branches, lo cual es vital para un buen rendimiento del pipeline, dado que cualquier instrucción puede ser un branch en lugar del 10–20% de las instrucciones típicamente ejecutadas por programas. Además implica un registro de propósito general menos.

2.4 Computación Entera de RV32I

El Apéndice A detalla todas las instrucciones de RISC-V, incluyendo formatos y opcodes. En esta sección, y secciones similares posteriores, damos una vista general del ISA que debería ser suficiente para programadores de ensamblador experimentados, además de resaltar las características que demuestran las siete métricas del ISA mencionadas en el Capítulo 1.

Las instrucciones aritméticas sencillas (add, sub), instrucciones lógicas (and, or, xor), e instrucciones de corrimiento (sll, srl, sra) en la Figura 2.1 son lo que se esperaría de cualquier ISA. Leen dos valores de registros de 32 bits y escriben el resultado al registro destino también de 32 bits. RV32I además ofrece versiones inmediatas de estas instrucciones. A diferencia de ARM-32, a los valores inmediatos siempre se les hace *sign-extension*, por lo que pueden ser negativos, razón por la cual no hay necesidad de sub.

Los programas pueden generar valores Booleanos del resultado de una comparación. Para permitir dichos casos, RV32I provee la instrucción *set less than*, la cual guarda un 1 en el registro si el primer operando es menor que el segundo, o 0 en caso contrario. Como es de esperarse, hay una versión con signo (slt) y una sin signo (sltu) para enteros *signed* y

31	0	
		x0 / zero
		x1 / ra
		x2 / sp
		x3 / gp
		x4 / tp
		x5 / t0
		x6 / t1
		x7 / t2
		x8 / s0 / fp
		x9 / s1
		x10 / a0
		x11 / a1
		x12 / a2
		x13 / a3
		x14 / a4
		x15 / a5
		x16 / a6
		x17 / a7
		x18 / s2
		x19 / s3
		x20 / s4
		x21 / s5
		x22 / s6
		x23 / s7
		x24 / s8
		x25 / s9
		x26 / s10
		x27 / s11
		x28 / t3
		x29 / t4
		x30 / t5
		x31 / t6
		32
31	0	
		pc
		32

Figura 2.4: Los registros de RV32I. El Capítulo 3 explica la convención de llamadas de RISC-V, la idea detrás de los varios punteros (sp, gp, tp, fp), registros guardados (en inglés, *saved*) (s0-s11), y Temporales (t0-t6) (La Figura 2.1 y Tabla 20.1 de [Waterman and Asanović 2017] son la base de esta figura).

unsigned, así como versiones inmediatas para ambas (*slti*, *sltiu*). Como veremos, a pesar de que RV32I puede validar todas las relaciones entre dos registros, algunas expresiones condicionales involucran relaciones entre muchos pares de registros. El compilador o programador podría usar *slt* y las instrucciones lógicas *and*, *or*, *xor* para resolver expresiones condicionales más elaboradas.

Las dos instrucciones enteras restantes en la Figura 2.1 ayudan con el ensamblaje y *linking*. *Load upper immediate* (*lui*) carga una constante de 20 bits a los 20 bits más significativos de un registro. Seguidamente se puede utilizar la instrucción estándar inmediata para producir una constante de 32 bits utilizando dos instrucciones de RV32I. *Add upper immediate to PC* (*auipc*) soporta secuencias de dos instrucciones para acceder a desplazamientos arbitrarios desde el PC, tanto para control de flujo como acceso de datos. La combinación de *auipc* y el valor inmediato de 12 bits en *jalr* (ver abajo) pueden transferir el control a cualquier dirección de 32 bits relativa al PC, mientras que *auipc* más el valor inmediato de 12 bits en instrucciones de *load* y *store* pueden acceder a cualquier dirección de 32 bits.

¿Qué es Diferente? Primero, no hay operaciones enteras para bytes o *half-words*. Las operaciones siempre son del ancho del registro. Accesos a memoria consumen energía en órdenes de magnitud superiores a operaciones aritméticas, por lo que accesos pequeños a datos pueden ahorrar energía, pero las operaciones aritméticas pequeñas no ahorran. ARM-32 tiene la extraña característica de poder hacer un corrimiento de un operando en la mayoría de las operaciones aritmético-lógicas, lo cual complica el datapath y casi nunca es usado [Hohl and Hinds 2016]; RV32I tiene instrucciones separadas de corrimiento de bits.

RV32I tampoco incluye multiplicación ni división; éstas son parte de la extensión opcional RV32M (ver Capítulo 4). A diferencia de ARM-32 y x86-32, todo el *software stack* de RISC-V puede ejecutarse sin ellas, lo cual puede reducir el tamaño de los chips embebidos. Aunque no es una cuestión de hardware, el *ensamblador* de MIPS-32 puede reemplazar una multiplicación por una secuencia de sumas y corrimientos para mejorar el rendimiento, lo cual puede confundir al programador al ver la ejecución de instrucciones que no están en el programa. RV32I también omite instrucciones de rotación y detección de *overflow* aritmético. Ambas pueden ser calculadas con un par de instrucciones RV32I (ver Sección 2.6).



Simplicidad

ARM v8, el sucesor de ARM-32, ya no utiliza corrimiento opcional para instrucciones de ALU, nuevamente sugiriendo que era un error tenerlo en ARM32.



Costo

■ **Elaboración: Instrucciones de manipulación de bits (“Bit twiddling” en inglés)**

como *rotate* están bajo consideración por la Fundación RISC-V como parte de una extensión opcional llamada RV32B (ver Capítulo 11).

■ **Elaboración: *xor* permite hacer un truco de magia.**

¿Es posible intercambiar dos valores sin usar un registro adicional! Este código hace intercambio de los valores de *x1* y *x2*. Dejamos la demostración en manos del lector. Pista: or exclusivo es conmutativo ($a \oplus b = b \oplus a$), asociativo ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), su propio inverso ($a \oplus a = 0$), y tiene identidad ($a \oplus 0 = a$).

```
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 más fascinante que sea, RISC-V provee tantos registros que los compiladores generalmente encuentran un registro temporal, rara vez utilizando intercambio con XOR.

2.5 Loads y Stores de RV32I

Además de proveer loads y stores de palabras de 32 bits (lw , sw), la Figura 2.1 muestra que RV32I puede cargar bytes y halfwords, ya sea en su versión *signed* o *unsigned* (lb , lbu , lh , lhu) y guardar bytes y halfwords (sb , sh). Bytes y *halfwords* con signo hacen *sign-extension* a 32 bits y son escritos en el registro destino. Esta extensión de datos permite que las operaciones aritméticas subsiguientes operen correctamente con 32 bits aun cuando el dato original es más corto. Bytes y halfwords sin signo, útiles para texto y números sin signo, se extienden con cero a 32 bits antes de ser escritos al registro destino.

El único modo de direccionamiento para loads y stores es sumando un valor inmediato de 12 bits (*extendido en signo*¹) a un registro, denominado en x86-32 “modo de direccionamiento con desplazamiento” [Irvine 2014].

¿Qué es Diferente? RV32I omitió los modos de direccionamiento sofisticados de ARM-32 y x86-32. Desafortunadamente, todos los modos de direccionamiento de ARM-32 no están disponibles para todos los tipos de datos, pero los modos de direccionamiento de RV32I no discriminan a ningún tipo de dato. RISC-V puede imitar algunos modos de direccionamiento del x86-32. Por ejemplo, dejando el valor inmediato en cero, es igual al modo registro-indirecto. A diferencia del x86-32, RISC-V no tiene instrucciones de stack específicas. Utilizando un registro como stack pointer (ver Figura 2.4), el modo de direccionamiento estándar obtiene la mayoría de los beneficios de las instrucciones push y pop sin agregarle complejidad al ISA. Opuesto a MIPS-32, RISC-V rechazó el *load retardado*. Similar en estilo a *branches retardados*, MIPS-32 redefinió el load para que el dato esté disponible dos instrucciones después, cuando apareciera en un pipeline de 5 etapas. Cualquier beneficio que haya tenido se evaporó con los pipelines modernos de más etapas.

Mientras que ARM-32 y MIPS-32 requieren que los datos estén alineados en memoria, RISC-V no lo exige. Accesos desalineados a veces son requeridos cuando migramos código antiguo. Una opción es no permitir accesos desalineados en el ISA base y proveer instrucciones separadas para soportar accesos desalineados, similar a Load Word Left y Load Word Right de MIPS-32. Sin embargo, esta opción complicaría el acceso a registros, ya que $lw1$ y $lw2$ requieren escribir únicamente a partes de registros en lugar de registros completos. Permitir que los loads y stores pudieran acceder a memoria desalineada simplificaba el diseño general.

■ *Elaboración: Endianness*

RISC-V eligió *little-endian* dado que es el ordenamiento de bytes predominante comercialmente: todos los sistemas x86-32, Apple iOS, Google Android OS y Microsoft Windows para ARM son *little-endian*. Sin embargo, esto afecta a pocos programadores ya que el *endianness* sólo es importante cuando se accede al mismo dato en modo word y byte.

2.6 Branches Condicionales de RV32I

RV32I puede comparar dos registros y saltar si el resultado es igual (beq), distinto (bne), mayor o igual (bge), o menor (blt). Los últimos dos casos son comparaciones con signo, pero RV32I también ofrece versiones sin signo: $bgeu$ y $bltu$. Las dos relaciones restantes (“mayor que” y “menor o igual”) se obtienen intercambiando los argumentos, dado que $x < y$ implica $y > x$ y $x \geq y$ equivale a $y \leq x$.



Simplicidad



Programabilidad



Costo

$bltu$ permite revisar los límites de un arreglo en una sola instrucción, dado que cualquier número negativo será mayor que cualquier tamaño de arreglo no negativo!



Programabilidad

Dado que las instrucciones de RISC-V deben ser múltiplos de dos bytes—ver Capítulo 7 para aprender de instrucciones opcionales de 2-bytes—el modo de direccionamiento de branches multiplica el valor inmediato de 12 bits por 2, le extiende el signo y lo suma al PC. Direccionamiento relativo al PC ayuda con *código independiente de posición*, reduciendo de esta manera el trabajo del linker y del loader (Capítulo 3).

¿Qué es Diferente? Como mencionamos anteriormente, RISC-V excluyó al infame *branch retardado* de MIPS-32, Oracle SPARC y otros. Además evitó los códigos de condición de ARM-32 y x86-32 para branches condicionales. Éstos agregan estados adicionales que son puestos implícitamente por muchas instrucciones, lo cual complica el cálculo de dependencias en ejecución fuera-de-orden. Finalmente, se omitieron las instrucciones de *loop* del x86-32: `loop`, `loope`, `loopz`, `loopne`, `loopnz`.



Simplicidad

■ *Elaboración: Sumar números de múltiples words sin códigos de condición*

se hace de la siguiente manera en RV32I usando `sltu` para calcular el acarreo de salida:

```
add a0,a2,a4 # sumar la parte baja (32 bits): a0 = a2 + a4
sltu a2,a0,a2 # a2' = 1 if (a2+a4) < a2, a2' = 0 else
add a5,a3,a5 # sumar la parte alta 32 bits: a5 = a3 + a5
add a1,a2,a5 # sumar el acarreo de la parte baja
```

■ *Elaboración: Leyendo el PC*

El PC actual se puede obtener poniendo el campo inmediato U de la instrucción `auipc` a 0. En x86-32, para leer el PC debemos llamar una función (lo cual guarda el PC en el stack); la función llamada lee el PC del stack, y finalmente retorna el PC. ¡Así que leer el PC lleva 1 store, 2 loads y 2 jumps!

■ *Elaboración: Verificando el overflow en software*

La mayoría, pero no todos los programas ignoran el desbordamiento (overflow) aritmético de enteros, por lo que RISC-V hace dicha validación en software. Suma sin signo requiere solamente un branch adicional luego de la suma: `addu t0, t1, t2; bltu t0, t1, overflow`.

Para suma con signo, si se sabe el signo de un operando, validar el desbordamiento requiere un solo branch luego de la suma: `addi t0, t1, +imm; blt t0, t1, overflow`.

Esto incluye el caso común de la suma con un operando inmediato. En general, para validar el desbordamiento en suma con signo, tres instrucciones adicionales son requeridas, sabiendo que la suma debería ser menor que uno de los operandos si y solo si el otro operando es 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 si (t2<0) && (t1+t2>=t1)
                    #           || (t2>=0) && (t1+t2<t1)
```

2.7 Salto Incondicional de RV32I

La instrucción *jump and link* (`jal`) en la Figura 2.1 cumple dos propósitos. En llamadas a funciones, almacena la dirección de la siguiente instrucción PC+4 en el registro destino, normalmente el registro de la dirección de retorno `ra` (ver Figura 2.4). Para saltos incondicionales, utilizamos el registro cero (`x0`) en lugar de `ra` como el registro destino, dado que éste no cambia. Al igual que los *branches*, `jal` multiplica su dirección de 20 bits por 2, extiende el signo y suma el resultado al PC para obtener la dirección a saltar.

La versión de *jump and link* (`jalr`) con registro es también multipropósito. Puede hacer una llamada a función a una dirección de memoria calculada dinámicamente o simplemente retornar de la función usando a `ra` como registro origen, y el registro cero (`x0`) como destino. Enunciados de *switch* o *case*, que calculan la dirección a saltar, también pueden usar `jalr` con el registro cero como destino.

¿Qué es Diferente? RV32I rechazó llamadas a funciones complejas, tales como las instrucciones de x86-32 `enter` y `leave`, o *register windows* como las que aparecen en Intel Itanium, Oracle SPARC y Cadence Tensilica.

2.8 Miscelánea de RV32I

Las instrucciones del *control status register* (`csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi`) en la Figura 2.1 proveen acceso fácil a registros que ayudan a medir el rendimiento de un programa. Dichos contadores de 64 bits, que pueden ser leídos en 32 bits, miden el tiempo, ciclos ejecutados y número de instrucciones retiradas.

La instrucción `ecall` hace solicitudes al ambiente de ejecución, tales como llamadas al sistema. Los *debuggers* usan la instrucción `ebreak` para transferir el control al ambiente de *debugging*.

La instrucción `fence` coordina accesos a dispositivos de I/O y memoria vistos desde otros *threads* y dispositivos externos o coprocesadores. La instrucción `fence.i` sincroniza el flujo de instrucciones y datos. RISC-V no garantiza que el store a la memoria de instrucciones sea visible al *instruction fetch* en el mismo procesador hasta que la instrucción `fence.i` sea ejecutada.

El Capítulo 10 cubre las instrucciones de sistema de RISC-V.

¿Qué es Diferente? RISC-V utiliza I/O mapeado a memoria en lugar de las instrucciones `in`, `ins`, `insb`, `insw` y `out`, `outs`, `outsb`, `outsw` de x86-32. Soporta strings utilizando `byte loads` y `stores` en lugar de las 16 instrucciones especiales de x86-32 `rep`, `movs`, `coms`, `scas`, `lods`,

2.9 Comparando RV32I, ARM-32, MIPS-32 y x86-32 usando Ordenamiento por Inserción

Hemos introducido el set de instrucciones base de RISC-V, y comentamos acerca de sus decisiones en comparación con ARM-32, MIPS-32 y x86-32. Ahora haremos una comparación mano-a-mano. La Figura 2.5 muestra Ordenamiento por Inserción² en C, que será nuestra referencia. La Figura 2.6 resume el número de instrucciones y cantidad de bytes para Ordenamiento por Inserción en los ISAs.



Simplicidad

Register windows aceleraba llamadas a funciones por tener mucho más de 32 registros. Una función nueva obtenía una *ventana* de 32 registros en la llamada. Para enviar argumentos, las ventanas se traslapan, implicando que algunos registros estaban en dos ventanas adyacentes.



Simplicidad

```

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: Ordenamiento por Inserción en C. Aunque simple, Ordenamiento por Inserción tiene muchas ventajas sobre algoritmos más complicados: es eficiente con la memoria y rápido para arreglos pequeños, manteniéndose adaptable, estable y en línea. GCC produjo el código para las siguientes 4 figuras. Activamos las banderas de optimización para reducir el tamaño del código, lo cual produjo el código más legible.

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

Figura 2.6: Número de instrucciones y tamaño del código para Ordenamiento por Inserción para estos ISAs. El Capítulo 7 describe ARM Thumb-2, microMIPS y RV32C.

Trasladamos los ejemplos de código para el final del capítulo para mantener la fluidez en este y capítulos subsiguientes.

Las Figuras 2.8 hasta 2.11 muestran el código compilado para RV32I, ARM-32, MIPS-32 y x86-32. A pesar del énfasis en simplicidad, la versión de RISC-V usa las mismas o menos instrucciones, y los tamaños del código para las distintas arquitecturas son similares. En este ejemplo, los branches de comparación y ejecución de RISC-V ahorran tantas instrucciones como los modos de direccionamiento más sofisticados e instrucciones de push y pop de ARM-32 y x86-32 en las Figuras 2.9 y 2.11.

La genealogía de todas las instrucciones de RISC-V está detallada en [Chen and Patterson 2016].

2.10 Observaciones Finales

Aquellos que no recuerdan el pasado están condenados a repetirlo.

—George Santayana, 1905

El efecto Lindy [Lin 2017] indica que la expectativa de vida futura de una idea o tecnología es proporcional a su edad. Ha sobrevivido la prueba del tiempo, entre más ha sobrevivido en el pasado, mayor será su probabilidad de sobrevivir en el futuro. En caso de mantenerse dicha hipótesis, la arquitectura RISC podría ser una buena idea por mucho tiempo.

La Figura 2.7 usa las siete métricas de diseño del ISA del Capítulo 1 para organizar las lecciones aprendidas de ISAs previos en secciones anteriores, y muestra los resultados positivos para RV32I. *No* estamos afirmando que RISC-V sea el primero en tener esos beneficios. En efecto, RV32I los hereda de RISC-I, su tatarabuelo [Patterson 2017]:

- Espacio de memoria de 32 bits direccionable por bytes
- Todas las instrucciones son de 32 bits
- 31 registros, todos de 32 bits, y el registro 0 alambrado a cero
- Todas las operaciones son entre registros (ninguna es de registro a memoria)
- Load/Store word, más load/store byte y halfword (*signed* y *unsigned*)
- Opción de inmediatos en todas las instrucciones aritméticas, lógicas y de corrimientos

- A los valores inmediatos siempre se les hace *sign-extension*
- Un modo de direccionamiento (registro + inmediato) y branching relativo al PC
- No hay instrucciones de multiplicación ni división
- Una instrucción que carga un valor inmediato de 20 bits a la parte alta del registro para cargar una constante de 32 bits en 2 instrucciones

RISC-V se beneficia comenzando un tercio o un cuarto de siglo después de otros ISAs populares, lo cual ayudó a los arquitectos de RISC-V a seguir el consejo de Santayana de usar las buenas ideas y no repetir los errores del pasado —incluyendo los de RISC-I— en RISC-V. Además, la Fundación RISC-V hará crecer el ISA lentamente a través de extensiones opcionales para prevenir el incrementalismo que ha plagado ISAs exitosos del pasado.



Elegancia

■ **Elaboración: ¿Es RV32I único?**

Procesadores del pasado tenían diferentes chips para aritmética de punto flotante, haciendo esas instrucciones opcionales. La ley de Moore pronto unificó todo en un solo chip, y la modularidad desvaneció de los ISAs. Usar subconjuntos del ISA en procesadores más simples y atrapar excepciones en software para emularlas data de hace décadas, con ejemplos como IBM 360 y Digital Equipment microVAX. RV32I es distinto en que el stack de software completo solamente necesita las instrucciones base, por lo que un procesador RV32I no tiene que generar excepciones constantemente para emular instrucciones de RV32G. Posiblemente el ISA más parecido en ese respecto es el Tensilica Xtensa, que estaba enfocado a aplicaciones embebidas. Su ISA base de 80 instrucciones fue diseñado para que los usuarios crearan sus propias instrucciones para acelerar sus aplicaciones. RV32I tiene un ISA base más simple, tiene una versión de direccionamiento de 64 bits y ofrece extensiones tanto para supercomputadoras como para microcontroladores.

2.11 Para Aprender Más

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 1: User-Level ISA, Version 2.2*. May 2017. URL <https://riscv.org/specifications/>.

Notas

¹Extendido en signo: En inglés, *Sign extended*.

²Ordenamiento por Inserción: En inglés, *Insertion Sort*.

	Errores del Pasado			Lecciones Aprendidas
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
Costo	Multiplicación de enteros obligatoria	Multiplicación y división de enteros obligatorias	Operaciones de 8 y 16 bits. Multiplicación y división de enteros obligatorias	No tiene operaciones de 8 ni 16 bits. Multiplicación y división de enteros opcional (RV32M)
Simplicidad	No tiene registro cero. Ejecución condicional de instrucciones. Modos de direccionamiento complejos. Instrucciones de stack (push/pop). Opción de corrimiento para instrucciones aritmético-lógicas	Inmediatos extendidos en signo y cero. Algunas instrucciones aritméticas pueden causar excepciones de <i>overflow</i>	No tiene registro cero. Instrucciones complejas para llamadas y retorno de funciones (enter/leave). Instrucciones de stack (push/pop). Modos de direccionamiento complejos. Instrucciones de <i>Loop</i>	Registro x0 dedicado a 0. Inmediatos únicamente <i>sign-extended</i> . Un modo de direccionamiento. No tiene ejecución condicional. No tiene instrucciones complejas de stack o de llamadas y retorno. No tiene excepciones de <i>overflow</i> aritmético. Instrucciones separadas de corrimiento
Rendimiento	Códigos de condición para branches. Registros origen y destino varían en el formato de instrucción. Load múltiple. Inmediatos calculados. El PC es un registro de propósito general	Registros origen y destino varían en el formato de instrucción.	Códigos de condición para branches. Como máximo, 2 registros por instrucción	Instrucciones compare y branch (sin códigos de condición). 3 registros por instrucción. No tiene load múltiple. Registros origen y destino fijos en el formato de instrucción. Inmediatos constantes. El PC no es de propósito general
Aislamiento de arquitectura con implementación	La longitud del pipeline es expuesta al escribir al PC como registro de propósito general	Branch y Load retardado. Registros HI y LO solo para multiplicación y división	Registros de propósito no-general (AX, CX, DX, DI, SI tienen propósitos únicos)	No tiene branch ni load retardado. Registros de propósito general
Espacio para crecer	Espacio limitado disponible para el opcode	Espacio limitado disponible para el opcode		Espacio generoso disponible para el opcode
Tamaño del programa	Solo instrucciones de 32 bits (+Thumb-2 como un ISA aparte)	Solo instrucciones de 32 bits (+microMIPS como un ISA aparte)	Instrucciones variables en bytes, pero con malas elecciones	Instrucciones de 32 bits + extensión RV32C de 16 bits
Facilidad de programar / compilar / <i>linkear</i>	Solo 15 registros. Datos alineados en memoria. Modos de direccionamiento irregulares. Contadores de rendimiento inconsistentes	Datos alineados en memoria. Contadores de rendimiento inconsistentes	Solo 8 registros. No tiene direccionamiento relativo al PC. Contadores de rendimiento inconsistentes	31 registros. Los datos pueden estar desalineados. Direccionamiento relativo al PC. Direccionamiento simétrico. Contadores de rendimiento definidos en la arquitectura

Figura 2.7: Lecciones aprendidas por los arquitectos de RISC-V de ISAs anteriores. Muchas veces la lección era simplemente evitar “optimizaciones” del pasado. Las lecciones y errores son clasificados usando las siete métricas del Capítulo 1. Muchas características que aparecen bajo costo, simplicidad y rendimiento podrían intercambiarse de lugar, dado que es una cuestión de gustos, pero son importantes sin importar dónde estén.

```

# RV32I (19 instrucciones, 76 bytes, o 52 bytes con RVC)
# a1 es n, a3 apunta a a[0], a4 es i, a5 es j, a6 es x
  0: 00450693  addi  a3,a0,4   # a3 apunta a a[i]
  4: 00100713  addi  a4,x0,1   # i = 1
Outer Loop:
  8: 00b76463  bltu  a4,a1,10   # si i < n, saltar a Continue Outer loop
Exit Outer Loop:
  c: 00008067  jalr  x0,x1,0   # retornar de función
Continue Outer Loop:
 10: 0006a803  lw    a6,0(a3)  # x = a[i]
 14: 00068613  addi  a2,a3,0   # a2 apunta a a[j]
 18: 00070793  addi  a5,a4,0   # j = i
Inner Loop:
1c: ffc62883  lw    a7,-4(a2) # a7 = a[j-1]
20: 01185a63  bge  a6,a7,34   # si a[j-1] <= a[i], saltar a Exit Inner Loop
24: 01162023  sw    a7,0(a2)  # a[j] = a[j-1]
28: fff78793  addi  a5,a5,-1  # j--
2c: ffc60613  addi  a2,a2,-4  # decrementar a2 para apuntar a a[j]
30: fe0796e3  bne  a5,x0,1c  # si j != 0, saltar a Inner Loop
Exit Inner Loop:
34: 00279793  slli  a5,a5,0x2 # multiplicar a5 por 4
38: 00f507b3  add  a5,a0,a5   # a5 es ahora el byte address 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   # incrementar a3 para apuntar a a[i]
48: fc1ff06f  jal  x0,8      # saltar a Outer Loop

```

Figura 2.8: Código RV32I para Ordenamiento por Inserción de la Figura 2.5. La dirección en hexadecimal está a la izquierda, seguidamente viene la instrucción codificada en hexadecimal y luego la instrucción en lenguaje ensamblador seguida de un comentario. RV32I reserva dos registros que apuntan a $a[j]$ y $a[j-1]$. Dado que tiene suficientes registros, el ABI reserva algunos para llamadas a funciones. A diferencia de los otros ISAs, no tiene que acceder a la memoria para guardar y restaurar registros. A pesar de que el código generado es mayor que el de x86-32, el usar RV32C (ver Capítulo 7) reduce la brecha. Nótese que el uso de comparación y branch evita las tres comparaciones que hacen ARM-32 y x86-32.

```

# ARM-32 (19 instrucciones, 76 bytes; o 18 inst/46 bytes con Thumb-2)
# r0 apunta a a[0], r1 es n, r2 es j, r3 es i, r4 es x
  0: e3a03001 mov  r3, #1          # i = 1
  4: e1530001 cmp  r3, r1         # i vs. n (¿innecesario?)
  8: e1a0c000 mov  ip, r0         # ip = a[0]
  c: 212ffff1e bxcS lr          # impedir que la dir. de ret. cambie ISAs
 10: e92d4030 push {r4, r5, lr}   # guardar r4, r5, dirección de retorno
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!   # x = a[i] ; incrementar ip
 18: e1a02003 mov  r2, r3         # j = i
 1c: e1a0e00c mov  lr, ip        # lr = a[0] (usando lr como scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]   # r5 = a[j-1]
 24: e1550004 cmp  r5, r4         # comparar a[j-1] vs. x
 28: da000002 ble  38            # si a[j-1] <= a[i], ir a Exit Inner Loop
 2c: e2522001 subs r2, r2, #1     # j--
 30: e40e5004 str  r5, [lr], #-4  # a[j] = a[j-1]
 34: 1afffff9 bne  20            # if j != 0, saltar a Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1     # i++
 3c: e1530001 cmp  r3, r1         # i vs. n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14            # if i < n, saltar a Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}   # restaurar r4, r5 y dirección de retorno

```

Figura 2.9: Código ARM-32 para Ordenamiento por Inserción de la Figura 2.5. La dirección en hexadecimal está a la izquierda, seguidamente viene la instrucción codificada en hexadecimal y luego la instrucción en lenguaje ensamblador seguida de un comentario. Corto de registros, ARM-32 guarda dos en el stack además de la dirección de retorno. Usa un modo de direccionamiento que escala a i y j para tener acceso a bytes. Dado que un branch tiene el potencial de cambiar entre ARM-32 y Thumb-2, `bxcS` primero pone a cero el bit menos significativo de la dirección de retorno antes de ser almacenada. Los códigos de condición ahorran una comparación en j luego de decrementarla, aun así, aparecen otras tres comparaciones en otras partes.


```

# MIPS-32 (24 instrucciones, 96 bytes, o 56 bytes con microMIPS)
# a1 es n, a3 apunta a a[0], v0 es j, v1 es i, t0 es x
  0: 24860004 addiu a2,a0,4 # a2 apunta a a[i]
  4: 24030001 li    v1,1    # i = 1
Outer Loop:
  8: 0065102b sltu  v0,v1,a1 # es 1 cuando i < n
  c: 14400003 bnez  v0,1c    # si i<n, saltar a Continue Outer Loop
 10: 00c03825 move  a3,a2    # a3 apunta a a[j] (llena el hueco)
 14: 03e00008 jr    ra      # retornar de función
 18: 00000000 nop                    # hueco de retardo de branch vacío
Continue Outer Loop:
 1c: 8cc80000 lw    t0,0(a2) # x = a[i]
 20: 00601025 move  v0,v1    # j = i
Inner Loop:
 24: 8ce9fffc lw    t1,-4(a3) # t1 = a[j-1]
 28: 00000000 nop                    # hueco de retardo de load vacío
 2c: 0109502a slt   t2,t0,t1 # es 1 cuando a[i] < a[j-1]
 30: 11400005 beqz  t2,48    # si a[j-1]<=a[i], saltar a Exit Inner Loop
 34: 00000000 nop                    # hueco de retardo de branch vacío
 38: 2442ffff addiu v0,v0,-1 # j--
 3c: ace90000 sw    t1,0(a3) # a[j] = a[j-1]
 40: 1440fff8 bnez  v0,24    # si j != 0, saltar a Inner Loop
 44: 24e7fffc addiu a3,a3,-4 # decr. a2 para apuntar a a[j] (hueco lleno)
Exit Inner Loop:
 48: 00021080 sll   v0,v0,0x2 #
 4c: 00821021 addu  v0,a0,v0 # v0 ahora es el byte address de a[j]
 50: ac480000 sw    t0,0(v0) # a[j] = x
 54: 24630001 addiu v1,v1,1  # i++
 58: 1000ffeb b     8        # saltar a Outer Loop
 5c: 24c60004 addiu a2,a2,4  # incr. a2 para apuntar a a[i] (hueco lleno)

```

Figura 2.10: Código MIPS-32 para Ordenamiento por Inserción de la Figura 2.5. La dirección en hexadecimal está a la izquierda, seguidamente viene la instrucción codificada en hexadecimal y luego la instrucción en lenguaje ensamblador seguida de un comentario. El código de MIPS-32 tiene tres instrucciones *nop*, incrementando su tamaño. Dos son por *branching retardado* y la tercera por *load retardado*. El compilador no encontró instrucciones que pudiera poner en los *huecos de retardo*¹. Los *branches retardados* hacen el código más difícil de entender, dado que la instrucción que le sigue a un *jump* o *branch* siempre se ejecuta. Por ejemplo, la última instrucción (*addiu*) en la dirección 5c es parte del ciclo a pesar de que está después del *branch*.

```

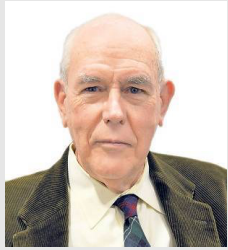
# x86-32 (20 instrucciones, 45 bytes)
# eax es j, ecx es x, edx es i
# puntero a a[0] está en la dirección de memoria esp+0xc; n en esp+0x10
0: 56          push esi          # guardar esi en stack (usado abajo)
1: 53          push ebx          # guardar ebx en stack (usado abajo)
2: ba 01 00 00 00 mov  edx,0x1      # i = 1
7: 8b 4c 24 0c  mov  ecx,[esp+0xc] # ecx apunta a a[0]
Outer Loop:
b: 3b 54 24 10  cmp  edx,[esp+0x10] # comparar i vs. n
f: 73 19          jae  2a <Exit Loop> # si i >= n, saltar a Exit Outer Loop
11: 8b 1c 91       mov  ebx,[ecx+edx*4] # x = a[i]
14: 89 d0          mov  eax,edx       # j = i
Inner Loop:
16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
1a: 39 de          cmp  esi,ebx       # comparar a[j-1] vs. x
1c: 7e 06          jle  24 <Exit Loop> # si a[j-1] <= a[i], Exit Inner Loop
1e: 89 34 81       mov  [ecx+eax*4],esi # a[j] = a[j-1]
21: 48             dec  eax           # j--
22: 75 f2          jne  16 <Inner Loop> # si j != 0, saltar a Inner Loop
Exit Inner Loop:
24: 89 1c 81       mov  [ecx+eax*4],ebx # a[j] = x
27: 42             inc  edx           # i++
28: eb e1          jmp  b <Outer Loop> # saltar a Outer Loop
Exit Outer Loop:
2a: 5b           pop  ebx           # restaurar ebx del stack
2b: 5e           pop  esi           # restaurar esi del stack
2c: c3           ret                # retornar de la función

```

Figura 2.11: Código x86-32 para Ordenamiento por Inserción de la Figura 2.5. La dirección en hexadecimal está a la izquierda, seguidamente viene la instrucción codificada en hexadecimal y luego la instrucción en lenguaje ensamblador seguida de un comentario. Carente de registros, x86-32 guarda dos en el stack. Además, dos de las variables que en RV32I están en registros, acá se almacenan en memoria (n y el puntero a a[0]). Utiliza de buena manera el modo de direccionamiento escalado indexado para acceder a a[i] y a[j]. Siete de las 20 instrucciones de x86-32 son de un byte, permitiéndole al x86-32 un código pequeño para este programa simple. Hay dos versiones populares de lenguaje ensamblador x86: Intel/Microsoft y AT&T/Linux. Utilizamos la sintaxis de Intel, en parte porque usa el mismo orden de operandos que RISC-V, ARM-32 y MIPS-32 con el destino a la izquierda y los operandos a la derecha, mientras que para AT&T los operandos están al revés (y se les agrega un % a los registros). Este detalle, por trivial que parezca, se ha vuelto un asunto cuasi religioso para algunos programadores. La pedagogía y no la ortodoxia nos hicieron elegir.

Lenguaje Ensamblador RISC-V

Ivan Sutherland (1938-) es llamado el padre de la computación gráfica dada la invención de Sketchpad—el precursor de la interfaz gráfica de usuario usada actualmente creada en 1962—por lo cual obtuvo el Premio Turing.



Es muy satisfactorio encontrar una solución simple a un problema que considerábamos complejo. Las mejores soluciones siempre son simples.

—Ivan Sutherland

3.1 Introducción

La Figura 3.1 muestra los cuatro pasos básicos para convertir un programa escrito en C a lenguaje de máquina listo para ejecutarse en un computador. Este capítulo cubre los últimos tres pasos, pero comenzaremos con el rol que desempeña el ensamblador en la convención de llamadas de RISC-V.

3.2 Convención de llamadas

Hay seis etapas generales al llamar una función [Patterson and Hennessy 2017]:

1. Poner los argumentos en algún lugar donde la función pueda acceder a ellos.
2. Saltar a la función (utilizando `jal` de RV32I).
3. Reservar el espacio de memoria requerido por la función, almacenando los registros que se requiera.
4. Realizar la tarea requerida de la función.
5. Poner el resultado de la función en un lugar accesible por el programa que invocó a la función, restaurando los registros y liberando la memoria.
6. Dado que una función puede ser llamada desde varias partes de un programa, retornar el control al punto de origen (usando `ret`).

Para obtener un buen rendimiento, es preferible mantener las variables en registros y no en memoria, y por otro lado, evitar accesos a memoria para guardar y restaurar estos registros.

Afortunadamente, RISC-V tiene suficientes registros para dar lo mejor de ambos mundos: mantener los operandos en registros y reducir la necesidad de guardarlos y restaurarlos. La clave es tener algunos registros que *no* se garantiza que conserven su valor a través de una llamada a una función, llamados *temporary registers*, y otros que sí se conservan, llamados

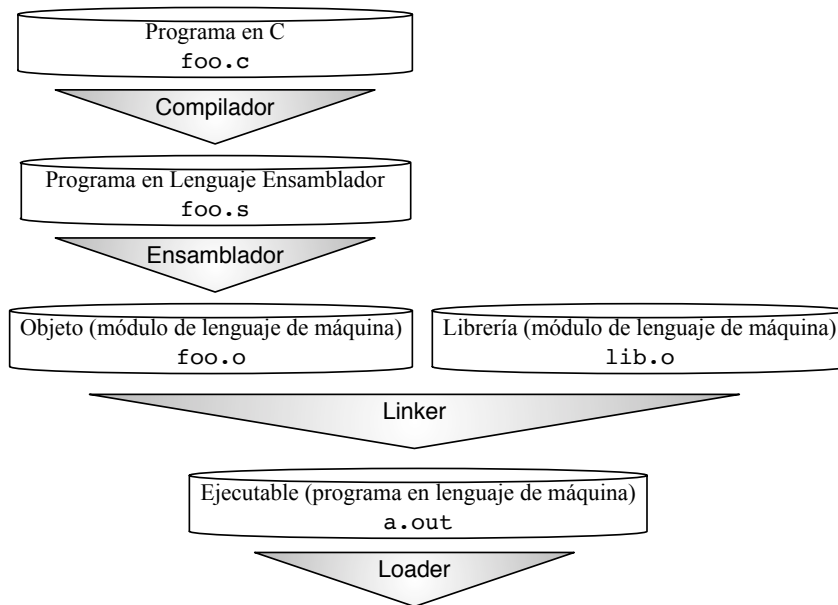


Figura 3.1: Pasos para convertir desde código fuente hasta un programa en ejecución. Estos son los pasos lógicos, aunque algunos pasos son combinados para acelerar la traducción. Utilizamos la convención de nombres de Unix para los tipos de archivos. Los equivalentes para MS-DOS son .C, .ASM, .OBJ, .LIB, y .EXE.

saved registers. Funciones que no llaman a otras funciones son llamadas funciones *hoja*. Cuando una función hoja tiene pocos argumentos y variables locales, podemos guardar todo en registros sin “derramarlos” a memoria. Si estas condiciones se cumplen, el programa no necesita guardar los valores de los registros en memoria, y una fracción sorprendente de llamadas a funciones caen en este afortunado caso.

Otros registros en la llamada a función deben ser considerados ya sea en la misma categoría como *saved registers*, los cuales se preservan a través de llamadas a funciones, o en la misma categoría que los *temporary registers*, que no se preservan. Una función modificará los registros que almacenan los valores de retorno, por lo que estos son registros temporales. No hay necesidad de preservar la dirección de retorno ni los argumentos, por lo que estos registros también son temporales. Quien llama la función puede confiar en que el stack pointer no se modificará a través de llamadas a funciones. La Figura 3.2 muestra los nombres de los registros ABI de RISC-V y la convención de preservar a través de llamadas a funciones o no.

Dadas las convenciones ABI, podemos ver el código estándar en RV32I para entrada y salida de una función. Este sería el *prólogo* de la función:

```

entry_label:
    addi sp,sp,-framesize    # Reservar espacio para el stack frame
                             # ajustando el stack pointer (registro sp)
    sw   ra,framesize-4(sp)  # Almacenar la dirección de retorno (registro ra)
    # almacenar otros registros al stack si fuera necesario
    ... # cuerpo de la función
  
```

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alternativo	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

Figura 3.2: Mnemónicos de ensamblador para registros enteros y de punto flotante en RISC-V. RISC-V tiene suficientes registros que el ABI puede reservar registros para ser usados por funciones hoja sin necesidad de guardarlos y restaurarlos. Los registros preservados a través de llamadas a funciones también son llamados *caller saved* versus *callee saved*, los cuales no se conservan. El Capítulo 5 explica los registros de punto flotante f (La Tabla 20.1 de [Waterman and Asanović 2017] es la base para esta figura).

Si hay demasiados argumentos y variables en la función para caber en los registros, el prólogo reserva espacio en el stack para la función, a esto se le llama *frame*. Luego que la función ha concluido, el *epílogo* restaura el stack frame y regresa al punto de origen:

```
# restaurar registros del stack de ser necesario
lw ra,framesize-4(sp) # Restaurar el registro con la dirección de retorno
addi sp,sp, framesize # Liberar el espacio del stack frame
ret # Retornar a donde se invocó la función
```

Pronto veremos un ejemplo basado en este ABI, pero primero debemos explicar las demás tareas del ensamblador más allá de la conversión de nombres a números de registro.

■ Elaboración: Los registros *saved* y *temporary* no son contiguos

para soportar RV32E, una versión de RISC-V embebida que solo tiene 16 registros (ver Capítulo 11). Simplemente usa números de registro del `x0` al `x15`, así que algunos registros *temporary* y *saved* están en este rango, y el resto están en los últimos 16 registros. RV32E es más pequeño, pero aún no tiene soporte del compilador dado que no coincide con RV32I.

3.3 Ensamblador

La entrada para esta etapa en Unix es un archivo con extensión `.s`, ej. `foo.s`; para MS-DOS sería `.ASM`.

La tarea del ensamblador en la Figura 3.1 no es simplemente producir código objeto a partir de instrucciones que el procesador pueda ejecutar, sino además extenderlas para incluir operaciones útiles para el programador de lenguaje ensamblador o el escritor de compiladores. Esta categoría, basada en configuraciones ingeniosas de instrucciones normales es llamada *pseudoinstrucciones*. Las Figuras 3.3 y 3.4 enumeran las pseudoinstrucciones de RISC-V. En la primera figura, todas dependen en que el registro `x0` siempre sea cero, mientras que en el segundo listado no dependen de eso. Por ejemplo, la instrucción `ret` mencionada anteriormente es en realidad una pseudoinstrucción que el ensamblador reemplaza por `jalr x0, x1, 0` (ver Figura 3.3). La mayoría de las pseudoinstrucciones de RISC-V dependen de `x0`. Como pueden ver, apartar uno de los 32 registros para que esté alambrado a cero, simplifica significativamente el set de instrucciones de RISC-V permitiendo muchas operaciones populares—tales como: `jump`, `return` y `branch on equal to zero`—como pseudoinstrucciones.



Simplicidad

Pseudoinstrucción	Instrucción/Instrucciones Base	Significado
nop	addi x0, x0, 0	No operation
neg rd, rs	sub rd, x0, rs	Complemento a 2
negw rd, rs	subw rd, x0, rs	Complemento a 2 (word)
snez rd, rs	sltu rd, x0, rs	Poner en 1 si \neq cero
sltz rd, rs	slt rd, rs, x0	Poner en 1 si $<$ cero
sgtz rd, rs	slt rd, x0, rs	Poner en 1 si $>$ cero
beqz rs, offset	beq rs, x0, offset	Branch si = cero
bnez rs, offset	bne rs, x0, offset	Branch si \neq cero
blez rs, offset	bge x0, rs, offset	Branch si \leq cero
bgez rs, offset	bge rs, x0, offset	Branch si \geq cero
bltz rs, offset	blt rs, x0, offset	Branch si $<$ cero
bgtz rs, offset	blt x0, rs, offset	Branch si $>$ cero
j offset	jal x0, offset	Jump
jr rs	jalr x0, rs, 0	Jump a registro
ret	jalr x0, x1, 0	Retornar de subrutina
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	<i>Tail call</i> subrutina lejana
rdinstret[h] rd	csrrs rd, instret[h], x0	Leer el contador de instrucciones retiradas
rdcycle[h] rd	csrrs rd, cycle[h], x0	Leer el contador de ciclos
rdtime[h] rd	csrrs rd, time[h], x0	Leer <i>real-time clock</i>
csrr rd, csr	csrrs rd, csr, x0	Leer CSR
csrw csr, rs	csrrw x0, csr, rs	Escribir CSR
csrs csr, rs	csrrs x0, csr, rs	Poner bits en 1 en CSR
csrc csr, rs	csrrc x0, csr, rs	Poner bits en 0 en CSR
csrwi csr, imm	csrrwi x0, csr, imm	Escribir CSR, inmediato
csrsi csr, imm	csrrsi x0, csr, imm	Poner bits en 1 en CSR, inmediato
csrci csr, imm	csrrci x0, csr, imm	Poner bits en 0 en CSR, inmediato
frcsr rd	csrrs rd, fcsr, x0	Leer <i>FP control/status register</i>
fscsr rs	csrrw x0, fcsr, rs	Escribir <i>FP control/status register</i>
frfm rd	csrrs rd, frm, x0	Leer <i>FP rounding mode</i>
fsrm rs	csrrw x0, frm, rs	Escribir <i>FP rounding mode</i>
frflags rd	csrrs rd, fflags, x0	Leer <i>FP exception flags</i>
fsflags rs	csrrw x0, fflags, rs	Escribir <i>FP exception flags</i>

Figura 3.3: 32 pseudo-instrucciones de RISC-V que dependen de x0, el registro cero. El Apéndice A incluye tanto las instrucciones reales como pseudoinstrucciones de RISC-V. Las que leen los contadores de 64 bits pueden leer los 32 bits de la parte alta utilizando la versión “h” de la instrucción y la versión normal para leer la parte baja (Las Tablas 20.2 y 20.3 de [Waterman and Asanović 2017] son la base de esta figura).

Pseudoinstrucción	Instrucción/Instrucciones Base	Significado
lla rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load de dirección local
la rd, symbol	<i>PIC</i> : auipc rd, GOT[symbol][31:12] l{w d} rd, rd, GOT[symbol][11:0] <i>No-PIC</i> : Igual que lla rd, symbol	Load de dirección
l{b h w d} rd, symbol	auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0](rd)	Load global
s{b h w d} rd, symbol, rt	auipc rt, symbol[31:12] s{b h w d} rd, symbol[11:0](rt)	Store global
fl{w d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w d} rd, symbol[11:0](rt)	Load global de punto flotante
fs{w d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0](rt)	Store global de punto flotante
li rd, immediate	<i>Muchas secuencias</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copiar registro
not rd, rs	xori rd, rs, -1	Complemento a uno
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Poner en 1 si = cero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copiar registro de precisión simple
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Valor absoluto de precisión simple
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Negación de precisión simple
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copiar registro de precisión doble
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Valor absoluto de precisión doble
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Negación de precisión doble
bgt rs, rt, offset	blt rt, rs, offset	Branch si >
ble rs, rt, offset	bge rt, rs, offset	Branch si ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch si >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch si ≤, unsigned
jal offset	jal x1, offset	Jump and link
jalr rs	jalr x1, rs, 0	Jump and link a registro
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	Llamar subrutina lejana
fence	fence iorw, iorw	<i>Fence</i> en toda la memoria e I/O
fscsr rd, rs	csrrw rd, fcsr, rs	Swap con <i>FP control/status register</i>
fsrm rd, rs	csrrw rd, frm, rs	Swap con <i>FP rounding mode</i>
fsflags rd, rs	csrrw rd, fflags, rs	Swap con <i>FP exception flags</i>

Figura 3.4: 28 pseudoinstrucciones de RISC-V que son independientes de x0, el registro cero. Para la, GOT significa Global Offset Table, y mantiene las direcciones de los símbolos en las librerías *linkeadas* dinámicamente. El Apéndice A incluye tanto las instrucciones reales como pseudoinstrucciones de RISC-V (Las Tablas 20.2 y 20.3 de [Waterman and Asanović 2017] son la base para esta figura).

```

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

```

Figura 3.5: Programa Hola Mundo en C (hello.c).

```

.text                # Directiva: ingresar a sección text
.align 2             # Directiva: alinear código a 2^2 bytes
.globl main          # Directiva: declarar símbolo global main
main:                # etiqueta para inicio de main
    addi sp,sp,-16   # reservar stack frame
    sw ra,12(sp)     # guardar dirección de retorno
    lui a0,%hi(string1) # calcular dirección de
    addi a0,a0,%lo(string1) # string1
    lui a1,%hi(string2) # calcular dirección de
    addi a1,a1,%lo(string2) # string2
    call printf      # llamar función printf
    lw ra,12(sp)     # restaurar dirección de retorno
    addi sp,sp,16    # liberar stack frame
    li a0,0          # cargar valor de retorno 0
    ret              # retornar
.section .rodata     # Directiva: ingresar a sección read-only data
.balign 4            # Directiva: alinear sección data a 4 bytes
string1:             # etiqueta para el primer string
    .string "Hello, %s!\n" # Directiva: null-terminated string
string2:             # etiqueta para el segundo string
    .string "world"   # Directiva: null-terminated string

```

Figura 3.6: Programa Hola Mundo en lenguaje ensamblador de RISC-V (hello.s).

La Figura 3.5 muestra el clásico programa “Hola Mundo” en C. El compilador produce el código en ensamblador de la Figura 3.6 usando la convención de llamadas de la Figura 3.2 y las pseudoinstrucciones de las Figuras 3.3 y 3.4.

Los comandos que comienzan con un punto son *directivas del ensamblador*. Estos son comandos para el ensamblador y no código a ser traducido. Le indican al ensamblador dónde poner código y datos, especifican constantes de texto y datos para uso en el programa, etcétera. La Figura 3.9 muestra las directivas de ensamblador para RISC-V. Para la Figura 3.6, las directivas son:

- `.text`—Comienza la sección de código.
- `.align 2`—Alinear el siguiente código a 2^2 bytes.
- `.globl main`—Declarar el símbolo global “main”.
- `.section .rodata`—Comienza la sección de datos de solo lectura.
- `.balign 4`—Alinear la sección de datos a 4 bytes.
- `.string ‘Hello, %s!\n’`—Crear este string que termina en null.
- `.string ‘world’`—Crear este string que termina en null.

El ensamblador produce el archivo objeto de la Figura 3.7 usando el Formato ELF (Executable and Linkable Format: Formato Ejecutable y *Linkeable*) [TIS Committee 1995].

3.4 Linker

En lugar de compilar todo el código fuente cada vez que cambia un archivo, el linker permite que archivos individuales puedan ser ensamblados por separado. Luego “une” el código objeto nuevo con otros módulos precompilados, tales como librerías. Deriva su nombre a partir de una de sus tareas, la de editar todos los links de las instrucciones de jump and link en el archivo objeto. En realidad, linker es un nombre corto de “link editor”, el nombre histórico para este paso de la Figura 3.1. En sistemas Unix, la entrada del linker son archivos con la extensión `.o` (e.j., `foo.o`, `libc.o`), y su salida es el archivo `a.out`. Para MS-DOS, las entradas son archivos con extensión `.OBJ` o `.LIB` y la salida es un archivo `.EXE`.

La Figura 3.10 muestra las direcciones de las regiones de memoria reservadas para código y datos en un programa típico de RISC-V. El linker debe ajustar las direcciones tanto del programa como de datos en el archivo objeto para apegarse a las direcciones de esta figura. Es más fácil para el linker si los archivos de entrada son *position independent code (PIC)*. PIC significa que todos los branches a instrucciones y referencias a datos en un archivo son correctos independientemente de dónde sea puesto el código. Como se mencionó en el Capítulo 2, el branch relativo a PC de RV32I hace esto mucho más fácil.

Además de las instrucciones, cada archivo objeto contiene una tabla de símbolos, la cual contiene todas las etiquetas en el programa que deben ser modificadas como parte del proceso de *linking*. Esta lista incluye etiquetas al área de datos y código. La Figura 3.6 tiene dos etiquetas en el área de datos (`string1` y `string2`), y otras dos etiquetas en el área de código (`main` y `printf`). Una dirección de 32 bits es difícil de codificar en una instrucción de 32 bits, por lo que el linker debe ajustar dos instrucciones por etiqueta en el código de RV32I, como se muestra en la Figura 3.6: `lui` y `addi` para direcciones de datos, y `auipc` y `jalr` para direcciones de código. La Figura 3.8 es la versión final de `a.out` luego de hacer el linking del archivo objeto en la Figura 3.7.

El programa “Hola Mundo” es típicamente el primer programa ejecutado en un procesador recién diseñado. Los arquitectos tradicionalmente ejecutan el sistema operativo lo suficientemente bien para imprimir “Hola Mundo” como una prueba de que el nuevo procesador en gran parte funciona. Envían la salida del programa por email a sus colegas y directivos, y luego se van a celebrar.

```

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 Hola Mundo en lenguaje de máquina de RISC-V (`hello.o`). Las seis instrucciones que luego son modificadas por el linker (posiciones 8 a 1c) tienen cero en su campo de dirección. La tabla de símbolos incluida en el archivo objeto guarda las etiquetas y direcciones de todas las instrucciones que deben ser editadas por el 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 Hola Mundo en lenguaje de máquina de RISC-V luego de la etapa de *linking*. En sistemas Unix el archivo se llamaría `a.out`.

Directiva	Descripción
<code>.text</code>	Ítems subsiguientes son almacenados en la sección <code>text</code> (código de máquina).
<code>.data</code>	Ítems subsiguientes son almacenados en la sección <code>data</code> (variables globales).
<code>.bss</code>	Ítems subsiguientes son almacenados en la sección <code>bss</code> (variables globales inicializadas a 0).
<code>.section .foo</code>	Ítems subsiguientes son almacenados en la sección llamada <code>.foo</code> .
<code>.align n</code>	Alinear el siguiente dato en un límite de 2^n bytes. Por ejemplo, <code>.align 2</code> alinea el próximo valor en un límite de <code>word</code> .
<code>.balign n</code>	Alinear el siguiente dato en un límite de n bytes. Por ejemplo, <code>.balign 4</code> alinea el próximo valor en un límite de <code>word</code> .
<code>.globl sym</code>	Declara que la etiqueta <code>sym</code> es global y se le puede hacer referencia desde otros archivos.
<code>.string "str"</code>	Almacenar el string <code>str</code> en memoria y terminarlo en <code>null</code> .
<code>.byte b1,..., bn</code>	Almacenar las n cantidades de 8 bits en bytes sucesivos de memoria.
<code>.half w1,..., wn</code>	Almacenar las n cantidades de 16 bits en halfwords sucesivos de memoria.
<code>.word w1,..., wn</code>	Almacenar las n cantidades de 32 bits en words sucesivos de memoria.
<code>.dword w1,..., wn</code>	Almacenar las n cantidades de 64 bits en doublewords sucesivos de memoria.
<code>.float f1,..., fn</code>	Almacenar los n números de punto flotante de precisión simple en words sucesivos de memoria.
<code>.double d1,..., dn</code>	Almacenar los n números de punto flotante de precisión doble en doublewords sucesivos de memoria.
<code>.option rvc</code>	Comprimir las instrucciones subsiguientes (ver Capítulo 7).
<code>.option norvc</code>	No comprimir las instrucciones subsiguientes.
<code>.option relax</code>	Permitir relajación del linker para las instrucciones subsiguientes.
<code>.option norelax</code>	No permitir relajación del linker para las instrucciones subsiguientes.
<code>.option pic</code>	Las instrucciones subsiguientes son <i>position-independent code</i> .
<code>.option nopic</code>	Las instrucciones subsiguientes son <i>position-dependent code</i> .
<code>.option push</code>	Hacer Push del estado de todos los <code>.options</code> a un stack, para que un posterior <code>.option pop</code> restaure sus valores.
<code>.option pop</code>	Hacer Pop del stack de opciones, restaurando todos los <code>.options</code> a su estado en el momento del último <code>.option push</code> .

Figura 3.9: Directivas comunes del ensamblador de RISC-V.

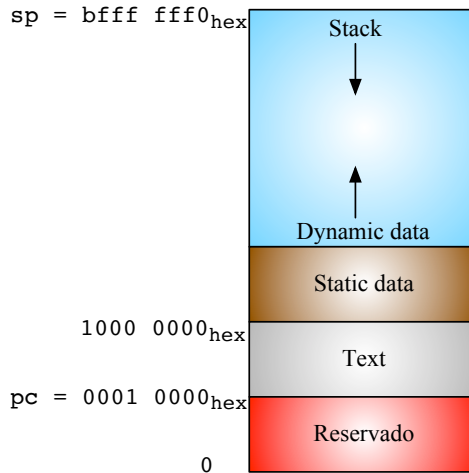


Figura 3.10: Reserva de memoria para el programa y datos en RV32I. Las direcciones más altas aparecen en la parte superior de la figura y las direcciones más bajas en la parte inferior. En esta convención del software de RISC-V, el stack pointer (sp) comienza en bfff fff0_{hex} y crece hacia abajo hacia el área de *Static data*. El área de *text* (código del programa) comienza en 0001 0000_{hex} e incluye las librerías *linkeadas* estáticamente. El área de *Static data* comienza inmediatamente después del área de *text*; en este ejemplo, asumimos que es la dirección 1000 0000_{hex} . Los datos dinámicos, reservados en C usando `malloc()`, están justo después del *Static data*. Llamado el *heap*, crece hacia el área de *stack* e incluye las librerías *linkeadas* dinámicamente.

Los compiladores de RISC-V soportan varios ABIs, dependiendo si las extensiones F y D están presentes. Para RV32, los ABIs son llamados `ilp32`, `ilp32f` y `ilp32d`. `ilp32` significa que los tipos de datos de C `int`, `long`, y `pointer` son todos de 32 bits; el sufijo opcional indica cómo se envían los argumentos de punto flotante. En `ilp32`, los argumentos de punto flotante se envían en registros enteros. En `ilp32f`, los argumentos de punto flotante precisión-simple se envían en registros de punto flotante. En `ilp32d`, argumentos de punto flotante precisión-doble también se pasan en registros de punto flotante.

Naturalmente, para enviar un argumento de punto flotante en un registro de punto flotante, es necesario tener la extensión F o D en el ISA (ver Capítulo 5). Así que para compilar código para RV32I (GCC con la bandera `'-march=rv32i'`), se debe utilizar el ABI `ilp32` (GCC con la bandera `'-mabi=ilp32'`). Por otro lado, tener instrucciones de punto flotante no implica que la convención de llamadas requiere su utilización. Por ejemplo, RV32IFD es compatible con los tres ABIs: `ilp32`, `ilp32f`, e `ilp32d`.

El linker revisa que el ABI del programa sea compatible con todas sus librerías. A pesar de que el compilador soporta muchas combinaciones de ABIs y extensiones del ISA, solamente un par de librerías podrían estar instaladas. Por eso, un error común es intentar hacer linking de un programa sin tener las librerías compatibles instaladas. El linker no producirá mensajes de error apropiados en este caso; simplemente intentará hacer link con una librería incompatible y luego informará acerca de la incompatibilidad. Este error generalmente se da únicamente cuando se compila desde una arquitectura hacia otra (*cross compiling*).

■ Elaboración: Relajación del Linker

La instrucción `jump and link` tiene un campo de dirección relativo a PC de 20 bits, así que una sola instrucción puede saltar bastante lejos. A pesar de que el compilador produce dos instrucciones por cada llamada a función externa, muchas veces una sola instrucción basta. Dado que esta optimización ahorra tiempo y espacio, los linkers hacen varias pasadas por el código reemplazando las dos instrucciones cada vez que puedan. Dado que cada iteración puede reducir la distancia entre la llamada y la función, para que quepa en una única instrucción, el linker continúa iterando hasta que ya no haya cambios. A este proceso le llamamos *relajación del linker*, nombre tomado de técnicas matemáticas para resolver sistemas de ecuaciones. Además de las llamadas a funciones, el linker de RISC-V también relaja las direcciones de datos a usar el *global pointer* cuando un dato está entre ± 2 KiB de `gp`, eliminando un `lui` o `auipc`. De manera similar, relaja el direccionamiento del TLS (Thread-Local Storage: Almacenamiento local del thread) cuando el dato está dentro de ± 2 KiB de `tp`.

3.5 Linking Estático vs Dinámico

La sección anterior describe *linking estático*, en el cual todo el código potencial de una librería se unifica con el programa y se carga a memoria antes de ejecutarse. Dichas librerías tienden a ser bastante grandes, por lo que unir librerías populares a múltiples programas desperdicia memoria. Además, las librerías se adjuntan al momento del linking—aun cuando luego son actualizadas para arreglar bugs—forzando al programa a utilizar versiones antiguas que pueden tener bugs.

Para evitar ambos problemas, la mayoría de los sistemas usan *linking dinámico*, donde la función externa es cargada y unida al programa luego que se llama por primera vez; si nunca se llama, no es cargada ni *linkeada*. Cada llamada subsiguiente usa un link rápido, por lo que el costo adicional solo se paga una vez. Cuando el programa arranca, se crea un link hacia la versión actual de la librería de funciones que necesita, obteniendo así la versión más reciente. Además, si múltiples programas usan la misma librería, todos los programas hacen referencia al mismo espacio de memoria.

El código que genera el compilador es similar al de *linking* estático. En lugar de saltar a la función real, salta a una pequeña función *stub* de tres instrucciones. La función *stub* carga la dirección real de la función de una tabla en memoria, luego salta a ella. Sin embargo la primera vez que se invoca, la tabla no tiene la dirección real de la función, sino la dirección de la rutina de *linking* dinámico. En ese instante, el linker dinámico utiliza la tabla de símbolos para encontrar la función real, la carga a memoria y actualiza la tabla con la información real. Cada llamada posterior únicamente paga el precio de las tres instrucciones de la función *stub*.

Los arquitectos generalmente miden el rendimiento de sus procesadores utilizando pruebas de rendimiento con linking estático a pesar que la mayoría de los programas usan linking dinámico. La excusa es que usuarios interesados en rendimiento deberían usar linking estático, pero esta es una justificación pobre. Tiene más sentido acelerar el rendimiento en programas reales y no en pruebas de rendimiento.

3.6 Loader

Un programa como el que aparece en la Figura 3.8 es un archivo ejecutable almacenado. Cuando se desea ejecutar, el trabajo del loader es cargarlo a memoria y saltar a la dirección de inicio. Actualmente, el “loader” es el sistema operativo; dicho de otra manera, cargar a `a.out` es una de las muchas tareas del sistema operativo.

Loading es un poco más complicado para programas con linking dinámico. En lugar de simplemente arrancar el programa, el sistema operativo arranca el linker dinámico. Éste a su vez carga el programa deseado y se encarga de las llamadas externas (solo la primera vez), carga a memoria las funciones, y modifica el programa para apuntar a la dirección correcta.

3.7 Observaciones Finales

Keep it simple, stupid.

—Kelly Johnson, ingeniero aeronáutico que acuñó el término “KISS Principle,” 1960

El ensamblador enriquece el ISA simple de RISC-V con 60 pseudoinstrucciones que hacen el código de RISC-V más fácil de leer y escribir sin incrementar costos de hardware. Dedicar un registro de RISC-V a cero permite muchas de estas operaciones útiles. Las instrucciones Load Upper Immediate (`lui`) y Add Upper Immediate to PC (`auipc`) hacen que sea más fácil para el compilador y el linker ajustar las direcciones para datos y funciones externas, y el branching relativo al PC ayuda al linker con Código de Posicionamiento Independiente (PIC). Tener bastantes registros permite una convención de llamadas que hace las llamadas y retornos de funciones más rápidas, reduciendo el derramamiento de registros (almacenarlos y restaurarlos de memoria).

RISC-V ofrece una colección sofisticada de mecanismos simples, pero de alto impacto que reducen costos, mejoran el rendimiento y lo hacen más fácil de programar.

3.8 Para Aprender Más

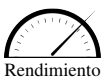
D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

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/>.

Notas

¹Hueco de retardo: En inglés, *delay slot*.



RV32M: Multiplicación y División

William de Ockham

(1287-1347) fue un teólogo Inglés promotor de lo que ahora se conoce como “la navaja de Ockham”, una preferencia por la simplicidad en el método científico.



srl puede hacer división sin signo para 2^i . Por ejemplo, si $a2 = 16 (2^4)$ entonces `srl i t2, a1, 4` produce el mismo valor que `divu t2, a1, a2`.

No hay que multiplicar los entes sin necesidad.

—William de Ockham, 1320

4.1 Introducción

RV32M agrega las instrucciones de multiplicación y división a RV32I. La Figura 4.1 es una representación gráfica de la extensión RV32M al set de instrucciones y la Figura 4.2 lista sus opcodes.

La división es simple. Recordemos que

$$\text{Cociente} = (\text{Dividendo} - \text{Residuo}) \div \text{Divisor}$$

o alternativamente

$$\text{Dividendo} = \text{Cociente} \times \text{Divisor} + \text{Residuo}$$

$$\text{Residuo} = \text{Dividendo} - (\text{Cociente} \times \text{Divisor})$$

RV32M tiene instrucciones tanto para enteros con signo como sin signo: división (`div`) y división unsigned (`divu`), las cuales almacenan el cociente en el registro destino. Menos frecuentemente, los programadores quieren el Residuo (también llamado *resto*) en lugar del

RV32M

multiply

multiply high $\left\{ \begin{array}{l} \text{— unsigned} \\ \text{— signed } \underline{\text{u}}\text{nsigned} \end{array} \right\}$

$\left\{ \begin{array}{l} \underline{\text{d}}\text{ivide} \\ \underline{\text{r}}\text{em} \text{ainder} \end{array} \right\} \left\{ \begin{array}{l} \text{— unsigned} \end{array} \right\}$

Figura 4.1: Diagrama de las instrucciones 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: El mapa de opcodes de RV32M tiene el formato: opcodes, tipo de formato y nombres (La Tabla 19.2 de [Waterman and Asanović 2017] es la base para esta figura).

```
# Calcular división unsigned de a0 entre 3 usando multiplicación.
0: aaaab2b7    lui    t0,0xaaaab # t0 = 0xaaaaaaaaab
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 de RV32M que divide por una constante usando la multiplicación. Se requiere de un análisis numérico cuidadoso para demostrar que este algoritmo funciona para cualquier dividendo, y para algunos otros divisores, el paso correctivo es más complicado. La prueba de validez (correctness) y el algoritmo para generar los recíprocos y los pasos correctivos está en [Granlund and Montgomery 1994].

cociente, así que RV32M provee las instrucciones: (`rem`) y residuo unsigned (`remu`), que escriben el residuo en lugar del cociente.

La ecuación de la multiplicación es simplemente:

$$\text{Producto} = \text{Multiplicando} \times \text{Multiplicador}$$

Es más complicado que la división porque el tamaño del producto es la suma del tamaño del multiplicador y el multiplicando; multiplicar dos números de 32 bits produce un producto de 64 bits. Para producir un número de 64 bits apropiado, RISC-V cuenta con cuatro instrucciones de multiplicación. Para obtener el producto de 32 bits—la parte baja del producto total—se usa `mul`. Si ambos operandos son números con signo, se usa `mulh` para obtener los 32 bits de la parte alta, se usa `mulhu` si ambos operandos son *unsigned* y `mulhsu` si un número es *signed* y el otro *unsigned*. Dado que sería complicado para el hardware escribir el producto de 64 bits en dos registros de 32 bits en una sola instrucción, RV32M requiere de dos instrucciones de multiplicación para producir el producto de 64 bits.

En muchos procesadores, la división entera es relativamente lenta. Como se mencionó anteriormente, corrimientos hacia la derecha pueden reemplazar divisiones enteras sin signo en potencias de 2. También se pueden optimizar divisiones por otras constantes multiplicándolas por el recíproco aproximado y luego aplicando una corrección a la parte alta del producto. Por ejemplo, la Figura 4.3 muestra el código para división entera entre 3.

¿Qué es Diferente? Por mucho tiempo, ARM-32 tuvo una instrucción de multiplicación pero no de división. La división no se volvió obligatoria hasta el 2005, casi 20 años después del primer procesador ARM. MIPS-32 usa registros especiales (HI y LO) como los únicos registros destino para instrucciones de multiplicación y división. Mientras que este diseño simplificó la implementación de los primeros procesadores MIPS, requiere una instrucción

sll puede hacer multiplicación con signo o sin signo por 2ⁱ. Por ejemplo, si $a2 = 16$ (2^4) entonces `slli t2,a1,4` produce el mismo valor que `mul t2,a1,a2`.



Para casi todos los procesadores, las multiplicaciones son más lentas que los corrimientos y las sumas y las divisiones son mucho más lentas que las multiplicaciones.

adicional de `move` para utilizar el resultado de la multiplicación o división, potencialmente reduciendo el rendimiento. Los registros HI y LO también incrementan el estado de la arquitectura, haciéndolo un poco más lento cambiar entre tareas.

■ **Elaboración:** *`mulh` y `mulhu` pueden validar el overflow en la multiplicación.*

No hay overflow cuando usamos `mul` para multiplicación sin signo si el resultado de `mulhu` es cero. Similarmente, no hay overflow cuando usamos `mul` para multiplicación con signo si todos los bits en el resultado de `mulh` coinciden con el bit de signo del resultado de `mul`, es decir, igual a 0 si es positivo, o `ffff_ffffhex` si es negativo.

■ **Elaboración:** *También es fácil validar la división entre cero.*

Simplemente validando el divisor con `beqz` antes de la división. RV32I no genera una excepción con división entre cero porque pocos programadores desean ese comportamiento, y los que lo requieren pueden hacerlo fácilmente en software. Por supuesto, divisiones por constantes nunca necesitan validaciones.

■ **Elaboración:** *`mulhsu` es útil para multiplicación con signo de múltiples words.*

`mulhsu` genera la parte alta del producto cuando el multiplicador tiene signo y el multiplicando no. Es un paso intermedio de una multiplicación con signo de múltiples words cuando se multiplica el word más significativo del multiplicador (que contiene el bit del signo) con los words menos significativos (que son unsigned). Esta instrucción mejora el rendimiento de la multiplicación de múltiples words aproximadamente en un 15%.

4.2 Observaciones Finales

Los componentes más baratos, rápidos y confiables son lo que no están allí.

—C. Gordon Bell, arquitecto de minicomputadoras prominentes.



Para ofrecer los procesadores RISC-V más pequeños para aplicaciones embebidas, la multiplicación y división son parte de la primera extensión estándar de RISC-V. Aun así, muchos procesadores RISC-V incluirán RV32M.

4.3 Para Aprender Más

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/>.

RV32F y RV32D: Punto Flotante de Precisión Simple y Doble

Antoine de Saint-Exupéry (1900-1944) Fue un escritor y aviador francés mejor conocido por su libro *El Principito*.



La perfección no se alcanza cuando no hay nada más que añadir, sino cuando no hay nada más que quitar.

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

5.1 Introducción

A pesar de que RV32F y RV32D son dos extensiones opcionales del set de instrucciones diferentes, normalmente se incluyen juntas. Dado que para casi todas las instrucciones de punto flotante hay versiones de precisión simple y doble (32 y 64 bits), por brevedad, los presentamos en el mismo capítulo. La Figura 5.1 es una representación gráfica de las extensiones al set de instrucciones RV32F y RV32D. La Figura 5.2 muestra los opcodes de RV32F y la Figura 5.3 muestra los opcodes de RV32D. Como la mayoría de los ISAs modernos, RISC-V sigue el estándar de punto flotante IEEE 754-2008 [IEEE Standards Committee 2008].

5.2 Registros de Punto Flotante

RV32F y RV32D usan registros separados llamados *f* en lugar de los registros *x*. La razón principal para dos conjuntos de registros es que los procesadores pueden mejorar el rendimiento duplicando la capacidad de los registros y el ancho de banda al tener dos conjuntos de registros sin incrementar la cantidad de bits usados para especificar los registros en el formato de la instrucción. El mayor impacto en el set de instrucciones es tener nuevas instrucciones para load y store de los registros *f* y transferencia entre registros *x* y *f*. La Figura 5.4 muestra los registros RV32F y RV32D.

Si un procesador implementa RV32F y RV32D, las operaciones de precisión simple únicamente usan los 32 bits menos significativos de los registros *f*. A diferencia del registro *x0* en RV32I, *f0* no está alambrado a cero y es un registro alterable al igual que los 31 registros *f* restantes.

El estándar IEEE 754-2008 provee varios métodos de redondear aritmética de punto-flotante, los cuales son útiles para determinar límites de error y escribir librerías numéricas. El más preciso y común es aproximar al entero más cercano (*round to nearest even: RNE*). La modalidad de redondeo se activa en el registro de control *fcsr*. La Figura 5.5 muestra *fcsr* y lista las opciones. Además almacena las excepciones acumuladas que exige el estándar.



RV32F y RV32D

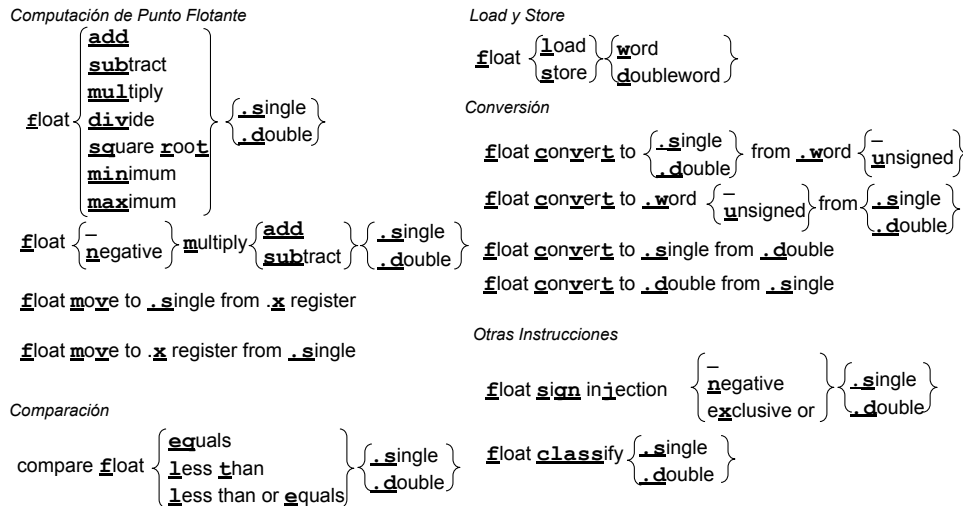


Figura 5.1: Diagrama de las instrucciones RV32F y RV32D.

¿Qué es Diferente? Tanto ARM-32 como MIPS-32 tienen 32 registros de punto flotante pero únicamente 16 registros de precisión doble. Ambos utilizan dos registros de precisión simple contiguos para formar un registro de precisión doble de 64 bits. La aritmética de punto flotante del x86-32 no tenía registros, por lo que usaba el stack. Se utilizan espacios en el stack de 80 bits para mejorar la precisión, por lo que los loads convierten los operandos de 32 ó 64 bits a 80 bits, y vice versa para stores. Una versión posterior de x86-32 agregó 8 registros de punto flotante de 64 bits con sus respectivas instrucciones. A diferencia de RV32FD y MIPS-32, ARM-32 y x86-32 omitieron instrucciones para mover datos directamente entre registros enteros y de punto flotante. La única opción es almacenar el registro de punto flotante en memoria y luego cargarlo a un registro entero, y vice versa.

Tener únicamente 16 registros de precisión doble fue el error más doloroso en el ISA de MIPS de acuerdo a John Mashey, uno de sus arquitectos.

■ **Elaboración:** RV32FD permite que el método de redondeo sea por instrucción.

Llamado *redondeo estático*, mejora el rendimiento cuando se desea cambiar el modo de redondeo para una instrucción. Por defecto se utiliza el modo dinámico indicado en `fcsr`. El redondeo estático se especifica como un último argumento opcional, i.e. `fadd.s ft0, ft1, ft2, rtz` redondeará hacia cero, independientemente del valor de `fcsr`. La Figura 5.5 muestra los modos de redondeo.

31		27	26	25	24	20		19	15	14	12	11	7	6	0	
imm[11:0]						rs1		rs1	010			rd	0000111		I flw	
imm[11:5]		rs2				rs1		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: El mapa de opcodes de RV32F tiene la estructura de la instrucción, opcodes, tipo de formato y nombres. La diferencia principal en codificación entre esta y la siguiente figura es que el bit 12 es 0 para las primeras dos instrucciones y el bit 25 es 0 para el resto, mientras que ambos bits son 1 en RV32D (La Tabla 19.2 de [Waterman and Asanović 2017] el la base para esta 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: El mapa de opcodes de RV32D tiene la estructura de la instrucción, opcodes, tipo de formato y nombres. Hay algunas instrucciones en estas dos figuras que no difieren únicamente en el tamaño de los datos. Esta figura en particular tiene *fcvt.s.d* y *fcvt.d.s* mientras que la otra tiene *fmv.x.w* y *fmv.w.x* (La Tabla 19.2 de [Waterman and Asanović 2017] es la base para esta figura).

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

Figura 5.4: Los registros de punto flotante de RV32F y RV32D. Los registros de precisión simple ocupan la parte derecha de los 32 registros de precisión doble. El Capítulo 3 explica la convención de llamadas para registros de punto flotante en RISC-V, la idea detrás de los registros para argumentos FP (fa0-fa7), registros *saved* FP (fs0-fs11) y temporales FP (ft0-ft11) (La Tabla 20.1 de [Waterman and Asanović 2017] es la base para esta figura).

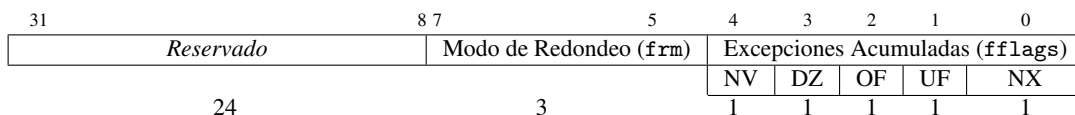


Figura 5.5: Registro de control y estado de punto flotante. Almacena el modo de redondeo y las banderas de excepciones. Los modos de redondeo son: aproximar al más cercano, empate hacia par (*rte*, 000 en *frm*); aproximar hacia cero (*rtz*, 001); aproximar hacia abajo, hacia $-\infty$ (*rdn*, 010); aproximar hacia arriba, hacia $+\infty$ (*rup*, 011); y aproximar al más cercano, empate a máxima magnitud (*rmm*, 100). Las cinco banderas de excepción indican las condiciones de excepción acumuladas en cualquier operación de punto flotante desde la última vez que el campo fue restablecido por software: NV es Operación Inválida; DZ es División entre Cero; OF es *Overflow*; UF es *Underflow*; y NX es Inexacto (La Figura 8.2 de [Waterman and Asanović 2017] el la base para esta figura).

5.3 Loads, Stores y Aritmética de Punto Flotante

RISC-V tiene dos instrucciones de load (*flw*, *fld*) y dos de store (*fsw*, *fsd*) para RV32F y RV32D. Ambas tienen el mismo modo de direccionamiento y formato de instrucción que *lw* and *sw*.

Además de las operaciones aritméticas estándar (*fadd.s*, *fadd.d*, *fsub.s*, *fsub.d*, *fmul.s*, *fmul.d*, *fdiv.s*, *fdiv.d*), RV32F y RV32D incluyen raíz cuadrada (*fsqrt.s*, *fsqrt.d*). También tienen mínimo y máximo (*fmin.s*, *fmin.d*, *fmax.s*, *fmax.d*), que escriben el valor máximo o mínimo del par de operandos fuente sin usar una instrucción de branch.

Muchos algoritmos de punto flotante como la multiplicación de matrices, multiplican e inmediatamente después hacen una suma o resta. Por lo tanto, RISC-V provee instrucciones que multiplican dos operandos y luego suman (*fmadd.s*, *fmadd.d*) o restan (*fmsub.s*, *fmsub.d*) un tercer operando al producto antes de escribir el resultado. También tiene versiones que niegan el producto antes de sumar o restar el tercer operando: *fnmadd.s*, *fnmadd.d*, *fnmsub.s*, *fnmsub.d*. Estas instrucciones *fusionadas* de multiplicación-suma son requeridas por el estándar IEEE 754-2008 dada su precisión: redondean solamente una vez (luego de la suma) en vez de dos veces (luego de la multiplicación, y nuevamente luego de la suma). Omitir este redondeo intermedio tiene un gran impacto cuando el producto y sumando tienen magnitudes similares pero signos opuestos, lo que causa que muchos bits de la mantisa se cancelen en la resta. Estas instrucciones necesitan un formato nuevo que especifique 4 registros, llamado R4. La Figura 5.2 y 5.3 muestra el formato R4, una variación del formato R.

En vez de branches de punto flotante, RV32F y RV32D proveen instrucciones de comparación de registros de punto flotante que activan un bit de un registro entero: *feq.s*, *feq.d*, *flt.s*, *flt.d*, *fle.s*, *fle.d*. Dichas instrucciones permiten usar un branch entero para cambiar el flujo basado en una condición de punto flotante. Por ejemplo, este código salta a *Exit* si $f1 < f2$:

```
flt x5, f1, f2 # x5 = 1 si f1 < f2; de lo contrario x5 = 0
bne x5, x0, Exit # si x5 != 0, salta a Exit
```

A diferencia de aritmética entera, el tamaño del producto de números de punto flotante es igual que sus operandos. Además, RV32F y RV32D omiten instrucciones de residuo.



Rendimiento

Hacia	Desde			
	Entero de 32b signed (w)	Entero de 32b unsigned (wu)	Punto flotante de 32b (s)	Punto flotante de 64b (d)
Entero de 32b signed (w)	–	–	<code>fcvt.w.s</code>	<code>fcvt.w.d</code>
Entero de 32b unsigned (wu)	–	–	<code>fcvt.wu.s</code>	<code>fcvt.wu.d</code>
Punto flotante de 32b (s)	<code>fcvt.s.w</code>	<code>fcvt.s.wu</code>	–	<code>fcvt.s.d</code>
Punto flotante de 64b (d)	<code>fcvt.d.w</code>	<code>fcvt.d.wu</code>	<code>fcvt.d.s</code>	–

Figura 5.6: Instrucciones de conversión de RV32F y RV32D. Las columnas muestran el tipo de dato origen y las filas el tipo de dato destino convertido.

5.4 Moves y Converts de Punto Flotante

RV32F y RV32D tienen todas las combinaciones útiles de instrucciones que convierten entre enteros con signo de 32 bits, enteros sin signo de 32 bits, punto flotante de 32 bits y punto flotante de 64 bits. La Figura 5.6 muestra estas 10 instrucciones por tipo de dato origen, y tipo de dato destino convertido.

RV32F además provee instrucciones para mover datos desde registros *x* hacia *f* (`fmv.x.w`) y vice versa (`fmv.w.x`).

5.5 Instrucciones de Punto Flotante Misceláneas

RV32F y RV32D ofrece instrucciones inusuales que ayudan a la librería matemática y proveen pseudoinstrucciones útiles (El estándar de punto flotante IEEE 754 requiere una manera de copiar y manipular los signos y clasificar datos de punto flotante, lo cual inspiró estas instrucciones).

Las primeras son las instrucciones *sign-injection*, las cuales copian todo de un registro hacia el otro a excepción del bit de signo. El valor del bit de signo depende de la instrucción:

1. Float sign inject (`fsgnj.s`, `fsgnj.d`): el bit de signo resultante es el mismo de *rs2*.
2. Float sign inject negative (`fsgnjn.s`, `fsgnjn.d`): el bit de signo resultante es el opuesto de *rs2*.
3. Float sign inject exclusive-or (`fsgnjx.s`, `fsgnjx.d`): el bit de signo resultante es el XOR entre los bits de signo de *rs1* y *rs2*.

Además de ayudar con la manipulación del bit de signo para librerías de matemática, las instrucciones de *sign-injection* proveen tres pseudoinstrucciones (ver Figura 3.4 en la página 39):

1. Copiar registro de punto flotante:
`fmv.s rd,rs` es en realidad `fsgnj.s rd,rs,rs` y
`fmv.d rd,rs` es en realidad `fsgnj.d rd,rs,rs`.
2. Negación:
`fneg.s rd,rs` se traduce a `fsgnjn.s rd,rs,rs` y
`fneg.d rd,rs` traduce a `fsgnjn.d rd,rs,rs`.
3. Valor absoluto (dado que $0 \oplus 0 = 0$ y $1 \oplus 1 = 0$):
`fabs.s rd,rs` se convierte en `fsgnjx.s rd,rs,rs` y
`fabs.d rd,rs` se convierte en `fsgnjx.d rd,rs,rs`.

La segunda instrucción inusual de punto flotante es `classify` (`fclass.s`, `fclass.d`). Las instrucciones de `classify` también son un gran apoyo a las librerías matemáticas. Prueban un operando para ver cuáles de las 10 propiedades de punto flotante aplican (ver tabla de abajo), luego escriben una máscara en los 10 bits más bajos al registro destino con la respuesta. Nueve bits se ponen en 0 y únicamente uno se pone en 1.

bit $x[rd]$	Significado
0	$f[rs]$ es $-\infty$.
1	$f[rs]$ es un número negativo normal.
2	$f[rs]$ es un número negativo subnormal.
3	$f[rs]$ es -0 .
4	$f[rs]$ es $+0$.
5	$f[rs]$ es un número positivo subnormal.
6	$f[rs]$ es número positivo normal.
7	$f[rs]$ es $+\infty$.
8	$f[rs]$ es un NaN de señalización.
9	$f[rs]$ es un NaN silencioso.

```

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: DAXPY: un programa que usa punto flotante de manera exhaustiva en C.

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

Figura 5.8: Número de instrucciones y tamaño del código para los cuatro ISAs. Muestra el número de instrucciones por ciclo y el total. El Capítulo 7 describe ARM Thumb-2, microMIPS, y RV32C.

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

El nombre DAXPY

viene de la fórmula: Doble-precisión A por X Plus Y. La versión de precisión simple es llamada SAXPY.

Haremos ahora una comparación mano-a-mano utilizando DAXPY como referencia de punto flotante (Figura 5.7). Calcula $Y = a \times X + Y$ en precisión doble, donde X y Y son vectores, y a es un escalar. La Figura 5.8 resume el número de instrucciones y bytes para el programa DAXPY en los cuatro ISAs. Su código está en las Figuras 5.9 a la 5.12.

Como fue el caso para Ordenamiento por Inserción en el Capítulo 2, a pesar de su énfasis en simplicidad, la versión de RISC-V tiene nuevamente las mismas o menos instrucciones, y los tamaños del código en las arquitecturas son similares. En este ejemplo, los branches de comparación-ejecución de RISC-V ahorran tantas instrucciones como los modos de direccionamiento estafalarios e instrucciones push y pop de ARM-32 y x86-32.



Simplicidad



Rendimiento

5.7 Observaciones Finales

Menos es Más.

—Robert Browning, 1855. La escuela minimalista de arquitectura (de construcción) adoptó este poema como un axioma en los 1980s.

El estándar IEEE 754-2008 de punto flotante [IEEE Standards Committee 2008] define los tipos de datos de punto flotante, la precisión de la computación y las operaciones requeridas. Su éxito reduce significativamente la dificultad de migrar programas de punto flotante, y también significa que los ISAs de punto flotante probablemente sean más uniformes que su equivalente en otros capítulos.

■ Elaboración: Aritmética de punto flotante de 16 bits, 128 bits y decimal

El estándar revisado de punto flotante (IEEE 754-2008) describe varios formatos más allá de la precisión simple y doble, denominadas *binary32* y *binary64*. La incorporación menos sorprendente es la precisión cuádruple, llamada *binary128*. Para esto, RISC-V tiene planeada una extensión tentativa llamada RV32Q (ver Capítulo 11). El estándar provee dos tamaños más para el intercambio binario de datos, indicando que los programadores pueden almacenar estos números en memoria pero no se debería computar en estos tamaños. Estos tamaños son precisión media (*binary16*) y precisión óctuple (*binary256*). A pesar de la intención del estándar, los GPUs computan y almacenan en precisión-media. El plan para RISC-V es incluir precisión-media en instrucciones vectorizadas (RV32V en el Capítulo 8), siempre y cuando los procesadores que soporten instrucciones vectorizadas de precisión media también agreguen instrucciones escalares de precisión media. La incorporación más sorprendente al estándar revisado es punto flotante decimal, para lo cual RISC-V ha apartado RV32L (ver Capítulo 11). Los tres formatos decimales son llamados *decimal32*, *decimal64*, y *decimal128*.

5.8 Para Aprender Más

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 insts en loop; 11 insts/44 bytes en total; 28 bytes RVC)
# a0 es n, a1 apunta a x[0], a2 apunta a y[0], fa0 es a
0: 02050463 beqz    a0,28          # si n == 0, saltar a Exit
4: 00351513 slli    a0,a0,0x3       # a0 = n*8
8: 00a60533 add     a0,a2,a0        # a0 = dirección 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++ (incrementar puntero a y)
18: 00858593 addi   a1,a1,8         # a1++ (incrementar puntero a 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         # si i != n, saltar a Loop
Exit:
28: 00008067 ret     # retornar

```

Figura 5.9: Código de RV32D para DAXPY en la Figura 5.7. La dirección en hexadecimal está a la izquierda, luego viene el código en lenguaje de máquina en hexadecimal, por último la instrucción en ensamblador seguida de un comentario. Las instrucciones de compare-and-branch evitan las dos instrucciones de comparación de ARM-32 y x86-32.

```

# ARM-32 (6 insts en loop; 10 insts/40 bytes en total; 28 bytes Thumb-2)
# r0 es n, d0 es a, r1 apunta a x[0], r2 apunta a y[0]
0: e3500000 cmp     r0, #0              # comparar n con 0
4: 0a000006 beq     24 <daxpy+0x24> # si n == 0, saltar a Exit
8: e0820180 add     r0, r2, r0, lsl #3 # r0 = dirección de x[n] (último elemento)
Loop:
c: ecb16b02 vldmia  r1!,{d6}        # d6 = x[i], incrementar puntero a 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 a y
1c: e1520000 cmp     r2, r0          # i vs. n
20: 1afffff9 bne     c <daxpy+0xc>    # si i != n, saltar a Loop
Exit:
24: e12ffffe bx     lr              # retornar

```

Figura 5.10: Código de ARM-32 para DAXPY en la Figura 5.7. El modo de direccionamiento *autoincremento* de ARM-32 ahorra dos instrucciones en comparación con RISC-V. A diferencia de Ordenamiento por Inserción, no hay necesidad de usar push y pop para DAXPY en ARM-32.


```

# MIPS-32 (7 insts en loop; 12 insts/48 bytes en total; 32 bytes microMIPS)
# a0 es n, a1 apunta a x[0], a2 apunta a y[0], f12 es a
0: 10800009 beqz   a0,28 <daxpy+0x28> # si n == 0, saltar a Exit
4: 000420c0 sll    a0,a0,0x3          # a0 = n*8 (ocupa el hueco de retardo)
8: 00c42021 addu   a0,a2,a0          # a0 = dirección de x[n] (último elemento)
Loop:
c: 24c60008 addiu  a2,a2,8          # a2++ (incrementar puntero a y)
10: d4a00000 ldc1   $f0,0(a1)        # f0 = x[i]
14: 24a50008 addiu  a1,a1,8          # a1++ (incrementar puntero a 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> # si i != n, saltar a Loop
24: f4c0fff8 sdc1   $f0,-8(a2)       # y[i] = a*x[i] + y[i] (llena el hueco)
Exit:
28: 03e00008 jr    ra              # retornar
2c: 00000000 nop                    # (hueco de retardo de branch vacío)

```

Figura 5.11: Código de MIPS-32 para DAXPY en la Figura 5.7. Dos de los tres *huecos de retardo de branches*¹ son llenados con instrucciones útiles. La posibilidad de comparar dos registros evita las dos instrucciones de comparación de ARM-32 y x86-32. A diferencia de los loads enteros, los loads de punto flotante no tienen hueco de retardo.

```

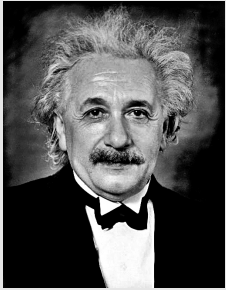
# x86-32 (6 insts en loop; 16 insts/50 bytes en total)
# eax es i, n está en memoria en esp+0x8, a está en memoria en esp+0xc
# puntero a x[0] está en memoria en esp+0x14
# puntero a y[0] está en memoria en esp+0x18
0: 53                push     ebx                # guardar ebx
1: 8b 4c 24 08        mov     ecx,[esp+0x8]      # ecx tiene copia de n
5: c5 fb 10 4c 24 0c vmovsd  xmm1,[esp+0xc]        # xmm1 tiene copia de a
b: 8b 5c 24 14        mov     ebx,[esp+0x14]    # ebx apunta a x[0]
f: 8b 54 24 18        mov     edx,[esp+0x18]    # edx apunta a y[0]
13: 85 c9             test    ecx,ecx           # comparar n con 0
15: 74 19             je     30 <daxpy+0x30>    # si n==0, saltar a Exit
17: 31 c0             xor     eax,eax           # i = 0 (porque x^x==0)
Loop:
19: c5 fb 10 04 c3    vmovsd  xmm0,[ebx+eax*8]   # xmm0 = x[i]
1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
24: c5 fb 11 04 c2    vmovsd  xmm0,xmm1,[edx+eax*8] # y[i] = a*x[i] + y[i]
29: 83 c0 01         add     eax,0x1           # i++
2c: 39 c1             cmp     ecx,eax          # comparar i vs n
2e: 75 e9             jne    19 <daxpy+0x19>    # si i!=n, saltar a Loop
Exit:
30: 5b                pop     ebx                # restaurar ebx
31: c3                ret                    # retornar

```

Figura 5.12: Código de x86-32 para DAXPY en la Figura 5.7. La escasez de registros en x86-32 es evidente en este ejemplo, con cuatro variables almacenadas en memoria que se encuentran en registros en los otros ISAs. Además nos muestra cómo comparar un registro con cero (`test ecx,ecx`) en x86-32 o cómo ponerlo a cero (`xor eax,eax`).

RV32A: Instrucciones Atómicas

Albert Einstein (1879-1955) fue el científico más famoso del siglo XX. Inventó la teoría de la relatividad y promovió la construcción de la bomba atómica para la Segunda Guerra Mundial.



Todo debería hacerse lo más simple posible, pero no más simple.

—Albert Einstein, 1933

6.1 Introducción

Asumiremos que el lector ya entiende el soporte necesario del ISA para multiprocesamiento, así que simplemente explicaremos las instrucciones RV32A y qué hacen. En caso se requiera aprender el tema o recordarlo, recomendamos estudiar “synchronization (computer science)” en Wikipedia ([https://en.wikipedia.org/wiki/Synchronization_\(computer_science\)](https://en.wikipedia.org/wiki/Synchronization_(computer_science))) o leer la Sección 2.1 de nuestro libro relevante a la arquitectura RISC-V [Patterson and Hennessy 2017]. RV32A tiene dos tipos de operaciones atómicas para sincronización:

- AMO (Atomic Memory Operations: Operaciones de Memoria Atómicas), y
- carga reservada / almacenamiento condicional².

La Figura 6.1 es una representación gráfica de la extensión al set de instrucciones llamada RV32A y la Figura 6.2 lista sus opcodes y formato de instrucciones.

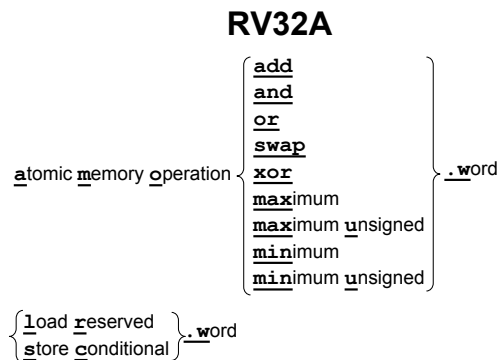


Figura 6.1: Diagrama de las instrucciones RV32A.

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: El mapa de opcodes de RV32A tiene la estructura de la instrucción, opcodes, tipo de formato y nombres. (La Tabla 19.2 de [Waterman and Asanović 2017] es la base para esta figura).

Las instrucciones AMO ejecutan atómicamente una operación a un operando en memoria y almacenan en el registro destino el valor original en memoria. Atómico significa que no puede haber interrupciones entre la lectura y escritura a memoria, tampoco otros procesadores pueden modificar los valores de memoria entre la lectura y escritura de la instrucción AMO.

Load reserved y store conditional proveen una operación atómica a través de dos instrucciones. Load reserved lee un word de memoria, lo escribe en el registro destino, y registra una reserva en ese espacio de memoria. Store conditional guarda un word en la dirección indicada en el registro fuente *siempre y cuando exista una reserva hecha por load reserved en esa dirección*. Escribe cero en el registro destino si logró hacer el store, o un código de error distinto de cero en caso contrario.

Una pregunta obvia es: ¿Por qué RV32A tiene dos maneras de implementar operaciones atómicas? La respuesta es que existen dos casos de uso muy distintos.

Los desarrolladores de lenguajes de programación asumen que la arquitectura puede ejecutar atómicamente la operación compare-and-swap: Comparar el valor de un registro al valor almacenado en la dirección de memoria dada por otro registro, y si son iguales, hacer *swap* de un tercer registro con el valor en memoria. Se asume esto dado que es una primitiva universal de sincronización, o sea que cualquier otra operación de sincronización *single-word* se puede generar a partir de compare-and-swap [Herlihy 1991].

A pesar de que es un argumento poderoso para agregar dicha instrucción al ISA, requiere tres registros fuente en una instrucción. Desafortunadamente, pasar de dos a tres operandos fuente complicaría la interfaz de memoria del sistema, el control y datapath de enteros, y el formato de instrucciones (Los tres operandos fuente de las instrucciones multiply-add de RV32FD afectan únicamente el datapath de punto flotante, no al datapath de enteros). Afortunadamente, load reserved y store conditional tienen únicamente dos registros fuente y pueden implementar compare and swap atómicamente (ver la mitad superior de la Figura 6.3).

El motivo fundamental de tener instrucciones AMO es que escalan mejor en grandes sistemas de multiprocesamiento que load reserved y store conditional. Además pueden usarse para implementar operaciones de reducción eficientemente. Las AMOs también son útiles para comunicarse con dispositivos de I/O, dado que ejecutan una lectura y escritura en una sola transacción de bus atómica. Esta atomicidad puede simplificar los drivers de dispositivos y mejorar el rendimiento de I/O. La parte baja de la Figura 6.3 muestra cómo escribir la sección crítica usando un swap atómico.

AMOs y LR/SC requieren direcciones de memoria alineadas naturalmente porque es oneroso para el hardware garantizar la atomicidad a través de los límites de un bloque de cache.



Simplicidad



Rendimiento

```

# Compare-and-swap (CAS) al word en memoria M[a0] usando lr/sc.
# Valor anterior esperado en a1; valor nuevo deseado en a2.
0: 100526af    lr.w  a3,(a0)    # Cargar valor anterior
4: 06b69e63    bne  a3,a1,80   # ¿Valor anterior igual a a1?
8: 18c526af    sc.w  a3,a2,(a0) # Swap del valor nuevo en caso afirmativo
c: fe069ae3    bnez  a3,0      # Reintentar si falló el store
    ... aquí va el código posterior a un CAS exitoso ...
80:                                     # CAS fallido.

# Sección crítica resguardada por test-and-set spinlock usando una AMO.
0: 00100293    li    t0,1      # Inicializar el valor del lock
4: 0c55232f    amoswap.w.aq t1,t0,(a0) # Intentar adquirir el lock
8: fe031ee3    bnez  t1,4      # Reintentar en caso fallido
    ... aquí va la sección crítica ...
20: 0a05202f    amoswap.w.rl x0,x0,(a0) # Liberar el lock.

```

Figura 6.3: Dos ejemplos de sincronización. El primero usa load reserved/store conditional `lr.w,sc.w` para implementar compare-and-swap, y el segundo utiliza swap atómico `amoswap.w` para implementar un mutex.

■ *Elaboración: Modelos de consistencia de memoria*

RISC-V usa un modelo de consistencia de memoria relajado, por lo que otros *threads* pueden ver algunos accesos a memoria fuera de orden. La Figura 6.2 muestra que todas las instrucciones de RV32A tienen un *acquire bit* (`aq`) y un *release bit* (`rl`). Una operación atómica con el bit `aq` en 1 garantiza que otros *threads* verán las AMO en orden con accesos a memoria *posteriores*. Si el bit `rl` es 1, los demás *threads* verán la operación atómica en orden con accesos a memoria *previos*. Para aprender más, [Adve and Gharachorloo 1996] es un excelente tutorial del tema.

¿Qué es Diferente? El ISA MIPS-32 original no tenía un mecanismo para sincronización, pero los arquitectos agregaron instrucciones load reserved / store conditional en versiones posteriores del ISA de MIPS.

6.2 Observaciones Finales

RV32A es opcional, y un procesador RISC-V que no lo implemente es más simple. Sin embargo, como dijo Einstein, todo debería ser lo más simple *posible*, pero no más simple. Muchas circunstancias requieren RV32A.

6.3 Para Aprender Más

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/>.

Notas

¹Hueco de retardo de branch: En inglés, *branch delay slot*.

²Carga Reservada y Almacenamiento Condicional: En inglés, *Load Reserved & Store Conditional*.

RV32C: Instrucciones Comprimidas

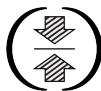
E. F. Schumacher

(1911-1977) escribió este libro de economía que promueve el uso de tecnologías apropiadas, descentralizadas y de escala humana. Traducido a múltiples idiomas, fue nombrado uno de los libros más influyentes desde la Segunda Guerra Mundial.

**small
is
beautiful**

a study of economics
as if people mattered

EF Schumacher



Tamaño del Programa



Simplicidad

Pequeño es Hermoso.

—E. F. Schumacher, 1973

7.1 Introducción

ISAs anteriores expandieron significativamente el número de instrucciones y formatos para reducir el tamaño del código: agregando instrucciones cortas de dos operandos en vez de tres, campos inmediatos pequeños y así sucesivamente. ARM y MIPS inventaron ISAs completos dos veces para reducir el código: ARM Thumb y Thumb-2 además MIPS16 y microMIPS. Estos nuevos ISAs complicaron al procesador y al compilador, e incrementaron la carga cognitiva para el programador de ensamblador.

RV32C tiene un enfoque innovador: *cada* instrucción corta *debe* mapearse a *una* sola instrucción estándar de RISC-V de 32-bits. Además, solo el ensamblador y el linker conocen las instrucciones de 16 bits, y es responsabilidad de ellos reemplazar una instrucción ancha por su prima angosta. El escritor del compilador y el programador de lenguaje ensamblador pueden ignorar las instrucciones RV32C y sus formatos, excepto por el beneficio de programas más cortos. La Figura 7.1 es una representación gráfica de la extensión RV32C al set de instrucciones.

Los arquitectos de RISC-V eligieron las instrucciones en la extensión RVC para obtener una buena compresión de código a través de una amplia gama de programas, usando tres observaciones para acoplarlas a 16 bits. Primero, los diez registros más populares (a0–a5, s0–s1, sp y ra) se acceden más que el resto. Segundo, muchas instrucciones sobrescriben uno de sus operandos. Tercero, los operandos inmediatos tienden a ser pequeños, y algunas instrucciones favorecen ciertos valores inmediatos. Así, muchas instrucciones RV32C solo acceden a registros populares; algunas instrucciones implícitamente sobrescriben un operando; y casi todos los inmediatos se reducen en tamaño, con loads y stores usando únicamente offsets sin signo en múltiplos del tamaño del operando.

Las Figuras 7.3 y 7.4 muestran el código RV32C para Ordenamiento por Inserción y DAXPY. Presentamos las instrucciones RV32C para mostrar el impacto de la compresión explícitamente, pero normalmente estas instrucciones son invisibles en el programa de ensamblador. Los comentarios muestran las instrucciones de 32 bits equivalentes entre paréntesis. El Apéndice A incluye la instrucción RISC-V de 32 bits que corresponde a cada instrucción RV32C de 16 bits.

RV32C

Computación de Enteros

$c.add \left\{ \begin{array}{l} - \\ \text{immediate} \end{array} \right\}$
 $c.add \text{immediate} * 16 \text{ to stack pointer}$
 $c.add \text{immediate} * 4 \text{ to stack pointer nondestructive}$
 $c.subtract$
 $c. \left\{ \begin{array}{l} \text{shift left logical} \\ \text{shift right arithmetic} \\ \text{shift right logical} \end{array} \right\} \text{immediate}$
 $c.and \left\{ \begin{array}{l} - \\ \text{immediate} \end{array} \right\}$
 $c.or$
 $c.move$
 $c.exclusive \text{ or}$
 $c.load \left\{ \begin{array}{l} - \\ \text{upper} \end{array} \right\} \text{immediate}$

Loads y Stores

$c. \left\{ \begin{array}{l} \text{float} \\ \text{float} \end{array} \right\} \left\{ \begin{array}{l} \text{load} \\ \text{store} \end{array} \right\} \text{word} \left\{ \begin{array}{l} - \\ \text{using stack pointer} \end{array} \right\}$
 $c.float \left\{ \begin{array}{l} \text{load} \\ \text{store} \end{array} \right\} \text{doubleword} \left\{ \begin{array}{l} - \\ \text{using stack pointer} \end{array} \right\}$

Transferencia de Control

$c.branch \left\{ \begin{array}{l} \text{equal} \\ \text{not equal} \end{array} \right\} \text{to zero}$
 $c.jump \left\{ \begin{array}{l} - \\ \text{and link} \end{array} \right\}$
 $c.jump \left\{ \begin{array}{l} - \\ \text{and link} \end{array} \right\} \text{register}$

Otras Instrucciones

$c.environment \text{ break}$

Figura 7.1: Diagrama de las instrucciones RV32C. Los campos inmediatos de las instrucciones `shift` y `c.addi4spn` se extienden con cero y se usa extensión de signo para las demás instrucciones.

Por ejemplo, en la dirección 4 de Ordenamiento por Inserción en la Figura 7.3, el ensamblador reemplazó la siguiente instrucción de 32 bits RV32I:

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

por esta instrucción RV32C de 16-bits:

```
c.li a4,1 # (se expande a addi a4,x0,1) i = 1
```

La instrucción `load immediate` RV32C es más corta porque solo debe especificar un registro y un inmediato pequeño. El código de máquina para `c.li` son solo cuatro dígitos hexadecimales en la Figura 7.3, mostrando que la instrucción `c.li` en efecto ocupa dos bytes.

Otro ejemplo es en la dirección 10 en la Figura 7.3, donde el ensamblador reemplazó:

```
add a2,x0,a3 # a2 es un puntero a a[j]
```

por esta instrucción RV32C de 16-bits:

```
c.mv a2,a3 # (se expande a add a2,x0,a3) a2 es el puntero a a[j]
```

La instrucción `move` de RV32C solo ocupa 16 bits porque solo especifica dos registros.

Si bien el diseñador de hardware no puede ignorar las instrucciones RV32C, un truco hace su implementación menos costosa: un decodificador traduce todas las instrucciones de 16 bits a su equivalente en 32 bits *antes* de ejecutarse. Las Figuras 7.6 a 7.8 muestran los formatos de instrucciones RV32C y los opcodes que el decodificador traduce. El decodificador utiliza 400 compuertas lógicas mientras que el procesador de 32 bits más pequeño—sin ninguna extensión de RISC-V—es de 8000 compuertas. Si es solo el 5% de un diseño tan pequeño, el decodificador casi desaparece en un procesador regular que con caches llega a unas 100,000 compuertas.



Costo

Referencia	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Ordenamiento por Inserción	Instrucciones	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instrucciones	10	12	16	11
	Bytes	28	32	50	28

Figura 7.2: Instrucciones y tamaño del código para Ordenamiento por Inserción y DAXPY para ISAs comprimidos.

¿Qué es Diferente? No hay instrucciones de byte o halfword en RV32C dado que otras instrucciones tienen mayor influencia en el tamaño del código. La pequeña ventaja en tamaño de Thumb-2 sobre RV32C en la Figura 1.5 en la página 9 se debe a las instrucciones Load y Store Multiple en la entrada y salida de procedimientos. RV32C las excluye para mantener un mapeo uno-a-uno con instrucciones RV32G, que las omite para reducir la complejidad de implementación en procesadores de gama alta. Dado que Thumb-2 es un ISA aparte de ARM-32, pero el procesador debe entender ambos, el hardware debe tener dos decodificadores de instrucciones: uno para ARM-32 y otro para Thumb-2. RV32GC es un solo ISA, así que procesadores RISC-V solo necesitan un decodificador.

■ **Elaboración: ¿Por qué algunos arquitectos no implementarían RV32C?**

La decodificación de instrucciones puede ser un cuello de botella para procesadores superescalares que intentan hacer fetch de múltiples instrucciones en un ciclo de reloj. Otro ejemplo es la *macrofusión*, donde el decodificador de instrucciones combina instrucciones de RISC-V para formar instrucciones más poderosas en ejecución (ver Capítulo 1). Una mezcla de instrucciones RV32C de 16 bits y RV32I de 32 bits puede hacer una decodificación sofisticada más difícil de completar en un ciclo de reloj en implementaciones de alto rendimiento.

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

La Figura 7.2 resume el tamaño de Ordenamiento por Inserción y DAXPY para estos cuatro ISAs.

De las 19 instrucciones RV32I originales en Ordenamiento por Inserción, 12 se vuelven RV32C, por lo que el código se reduce de $19 \times 4 = 76$ bytes a $12 \times 2 + 7 \times 4 = 52$ bytes, ahorrando $24/76 = 32\%$. DAXPY se reduce de $11 \times 4 = 44$ bytes a $8 \times 2 + 3 \times 4 = 28$ bytes, o $16/44 = 36\%$.

Los resultados para estos dos pequeños ejemplos son sorprendentemente congruentes con la Figura 1.5 en la página 9, Capítulo 2, la cual muestra que el código RV32G es alrededor de 37% más grande que el código RV32GC, para un conjunto mucho mayor de programas más grandes. Para alcanzar ese nivel de ahorro, más de la mitad de las instrucciones debieron ser RV32C.

■ **Elaboración:** *¿Es RV32C realmente único?*

Las instrucciones RV32I son indistinguibles en RV32IC. Thumb-2 es en realidad un ISA separado con instrucciones de 16-bits, además de muchas pero no todas las instrucciones de ARMv7. Por ejemplo, *Compare and Branch on Zero* está en Thumb-2 pero no en ARMv7, y vice versa para *Reverse Subtract with Carry*. MicroMIPS32 tampoco es un superconjunto de MIPS32. Por ejemplo, microMIPS multiplica los desplazamientos de los branches por dos, pero se multiplican por cuatro en MIPS32. RISC-V *siempre* multiplica por dos.

7.3 Observaciones Finales

Hubiese escrito una carta más corta, pero no tenía tiempo.

—Blaise Pascal, 1656.

Fue un matemático que construyó una de las primeras calculadoras mecánicas, lo que condujo al ganador del *Premio Turing* Niklaus Wirth bautizar un lenguaje de programación en su nombre.

RV32C le da a RISC-V uno de los tamaños de código más pequeños a la fecha. Casi pueden considerarse pseudoinstrucciones asistidas por hardware. Sin embargo, el ensamblador las esconde del programador de lenguaje ensamblador y compilador, en lugar de, como en el Capítulo 3, expandir el set de instrucciones con operaciones populares que hacen el código de RISC-V más fácil de usar y entender. Ambos enfoques benefician la productividad del programador.

Consideramos a RV32C como uno de los mejores ejemplos en RISC-V de un mecanismo simple pero poderoso que mejora su costo-rendimiento.

7.4 Para Aprender Más

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/>.



Tamaño del Programa



Elegancia

```

# RV32C (19 instrucciones, 52 bytes)
# a1 es n, a3 apunta a a[0], a4 es i, a5 es j, a6 es x
0: 00450693 addi  a3,a0,4  # a3 apunta a a[i]
4: 4705      c.li  a4,1    # (extendido: addi a4,x0,1) i = 1
Outer Loop:
6: 00b76363 bltu  a4,a1,c  # si i < n, saltar a Continue Outer loop
a: 8082      c.ret                # (extendido: jalr x0,ra,0) retornar
Continue Outer Loop:
c: 0006a803 lw   a6,0(a3)  # x = a[i]
10: 8636     c.mv  a2,a3    # (extendido: add a2,x0,a3) a2 apunta a a[j]
12: 87ba     c.mv  a5,a4    # (extendido: add a5,x0,a4) j = i
InnerLoop:
14: ffc62883 lw   a7,-4(a2) # a7 = a[j-1]
18: 01185763 ble  a7,a6,26  # si a[j-1] <= a[i], saltar a Exit InnerLoop
1c: 01162023 sw   a7,0(a2)  # a[j] = a[j-1]
20: 17fd     c.addi a5,-1    # (extendido: addi a5,a5,-1) j--
22: 1671     c.addi a2,-4    # (extendido: addi a2,a2,-4) decr a2 para -> a[j]
24: fbe5     c.bnez a5,14    # (extendido: bne a5,x0,14) si j!=0, InnerLoop
Exit InnerLoop:
26: 078a     c.slli a5,0x2    # (extendido: slli a5,a5,0x2) multiplicar a5 * 4
28: 97aa     c.add  a5,a0    # (extendido: add a5,a5,a0)a5 = byte addr de a[j]
2a: 0107a023 sw   a6,0(a5)  # a[j] = x
2e: 0705     c.addi a4,1    # (extendido: addi a4,a4,1) i++
30: 0691     c.addi a3,4    # (extendido: addi a3,a3,4) incr a3 para -> a[i]
32: bfd1     c.j    6      # (extendido: jal x0,6) saltar a Outer Loop

```

Figura 7.3: Código de RV32C para Ordenamiento por Inserción. Las 12 instrucciones de 16 bits hacen el código 32% más pequeño. El tamaño de cada instrucción es evidente por el número de caracteres hexadecimales en la segunda columna. Las instrucciones RV32C (comenzando con c.) se muestran explícitamente en este ejemplo, pero normalmente los programadores de ensamblador y compiladores no las pueden ver.

```

# RV32DC (11 instrucciones, 28 bytes)
# a0 es n, a1 apunta a x[0], a2 apunta a y[0], fa0 es a
0: cd09      c.beqz a0,1a      # (extendido: beq a0,x0,1a) si n==0, ir a Exit
2: 050e      c.slli a0,a0,0x3     # (extendido: slli a0,a0,0x3) a0 = n*8
4: 9532      c.add a0,a2       # (extendido: add a0,a0,a2) a0 = dir. de x[n]
Loop:
6: 2218      c.fld fa4,0(a2)           # (extendido: fld fa4,0(a2) ) fa5 = x[]
8: 219c      c.fld fa5,0(a1)           # (extendido: fld fa5,0(a1) ) fa4 = y[]
a: 0621      c.addi a2,8              # (extendido: addi a2,a2,8) a2++ (incr. ptr a y)
c: 05a1      c.addi a1,8              # (extendido: addi a1,a1,8) a1++ (incr. ptr a 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              # si i != n, saltar a Loop
Exit:
1a: 8082      ret                      # (extendido: jalr x0,ra,0) retornar

```

|

Figura 7.4: Código RV32DC para DAXPY. Las 8 instrucciones de 16-bits reducen el código en un 36%. El tamaño de cada instrucción es evidente por el número de caracteres hexadecimales en la segunda columna.

Las instrucciones RV32C (comenzando con c.) se muestran explícitamente en este ejemplo, pero normalmente los programadores de ensamblador y compiladores no las pueden ver.

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

Figura 7.5: Mapa de opcodes RV32C (bits[1 : 0] = 01) muestra la estructura, opcodes, formato y nombres. rd', rs1' y rs2' hacen referencia a los 10 registros populares a0–a5, s0–s1, sp y ra (La Tabla 12.5 de Waterman and Asanović 2017) es la base para esta figura).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000	0										0	00	CIW <i>Instrucción ilegal</i>			
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: Mapa de opcodes RV32C (bits[1 : 0] = 00) muestra la estructura, opcodes, formato y nombres. rd', rs1' y rs2' hacen referencia a los 10 registros populares a0–a5, s0–s1, sp y ra (La Tabla 12.4 de Waterman and Asanović 2017) es la base para esta 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: Mapa de opcodes para RV32C (bits[1 : 0] = 10) muestra la estructura, opcodes, formato y nombres (La Tabla 12.6 de Waterman and Asanović 2017) es la base para esta figura).

Formato	Significado	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
CR	Registro	funct4				rd/rs1				rs2				op					
CI	Inmediato	funct3		imm		rd/rs1				imm				op					
CSS	Store relativo al Stack	funct3		imm								rs2				op			
CIW	Inmediato Ancho	funct3		imm								rd'				op			
CL	Load	funct3		imm		rs1'				imm		rd'		op					
CS	Store	funct3		imm		rs1'				imm		rs2'		op					
CB	Branch	funct3		offset				rs1'		offset				op					
CJ	Jump	funct3		jump target												op			

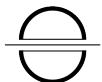
Figura 7.8: Formatos de instrucciones comprimidas RVC de 16 bits. rd', rs1' y rs2' hacen referencia a los 10 registros populares a0–a5, s0–s1, sp y ra (La Tabla 12.1 de Waterman and Asanović 2017) es la base para esta figura).

Seymour Cray (1925-1996) Fue el arquitecto de la Cray-1 en 1976, la primer supercomputadora comercialmente exitosa que usaba una arquitectura vectorizada. La Cray-1 era una joya; era la computadora más rápida aun *sin usar* instrucciones vectorizadas.



Rendimiento

Las Extensiones Multimedia (MMX) de Intel en 1997 popularizaron a SIMD. Fueron integradas y expandidas con *Streaming SIMD Extensions* (SSE) en 1999 y *Advanced Vector Extensions* (AVX) en el 2010. La fama de MMX fue impulsada por una campaña publicitaria mostrando trabajadores de líneas de producción de procesadores vestidos de técnico y bailando disco (<https://www.youtube.com/watch?v=paU16B-bZEA>).



Aislamiento de Arq e Impl

Me encanta la simplicidad. Si es muy complicado no lo puedo entender.

—Seymour Cray

8.1 Introducción

Este capítulo se concentra en *paralelismo a nivel de datos*¹, donde hay muchos datos en los cuales la aplicación puede computar simultáneamente. Los arreglos son un ejemplo popular. Fundamentales para aplicaciones científicas, los programas multimedia también usan arreglos. Los primeros usan datos de punto-flotante de precisión simple y doble, mientras los segundos generalmente usan datos enteros de 8 y 16 bits.

La arquitectura más conocida para paralelismo a nivel de datos es *Single Instruction Multiple Data (SIMD)*. SIMD inicialmente se volvió popular haciendo particiones de registros de 64 bits en muchas partes de 8, 16 ó 32 bits y luego los operaba en paralelo. El opcode indicaba el tamaño del dato y la operación. Las transferencias de datos son simplemente loads y stores de un registro SIMD de 64 bits.

El primer paso de hacer particiones de los registros de 64 bits es tentador dado que es simple y directo. Para hacer SIMD más rápido, los arquitectos simplemente ensanchan los registros para computar más particiones concurrentemente. Dado que los ISAs SIMD pertenecen a la escuela de diseño incremental, y el opcode especifica el ancho del dato, expandir los registros SIMD también expande el set de instrucciones. Cada paso subsiguiente de duplicar el ancho de los registros SIMD y del set de instrucciones incrementa la complejidad del ISA, la cual deben soportar los diseñadores de procesadores, escritores de compiladores y programadores de ensamblador.

Otra alternativa más antigua y en nuestra opinión más elegante de trabajar con paralelismo a nivel de datos es la arquitectura *vectorizada*. Este capítulo explica nuestros motivos para utilizar vectores en lugar de SIMD en RISC-V.

Las computadoras vectorizadas cargan objetos de memoria y los almacenan secuencialmente en registros vectorizados anchos. Unidades de ejecución pipelined computan eficientemente sobre los registros vectorizados. Seguidamente esparcen los resultados desde los registros vectorizados hacia la memoria principal. El tamaño de los registros vectorizados es determinado por la implementación, en lugar del opcode como en SIMD. Como veremos, *separar la longitud del vector y la cantidad máxima de operaciones por ciclo de reloj de la codificación de instrucciones es el punto crucial de las arquitecturas vectorizadas*: el mi-

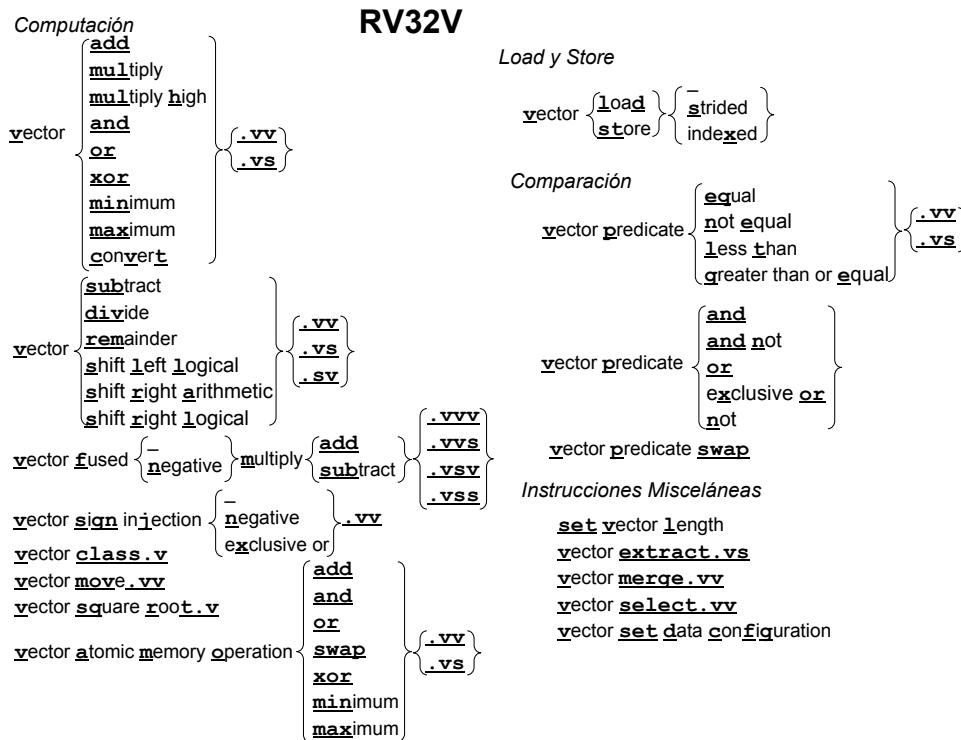


Figura 8.1: Diagrama de las instrucciones RV32V. Dado que el tipo de registro es dinámico, este diagrama de instrucciones también aplica a RV64V en el Capítulo 9.

croarquitecto encargado de la parte vectorizada puede diseñar hardware con paralelismo de datos sin afectar al programador, y el programador puede aprovechar vectores más largos sin reescribir el código. Además, las arquitecturas vectorizadas tienen menos instrucciones que arquitecturas SIMD.

Es más, a diferencia de SIMD, las arquitecturas vectorizadas tienen tecnologías de compiladores bien establecidas.

Las arquitecturas vectorizadas son menos frecuentes que las arquitecturas SIMD, por lo que menos lectores conocen ISAs vectorizadas. Por esa razón, este capítulo tendrá un tono más de tutorial que otros. Si desea profundizar más en arquitecturas vectoriales, lea el Capítulo 4 y el Apéndice G de [Hennessy and Patterson 2011]. RV32V también tiene características innovadoras que simplifican el ISA, lo cual requiere mayor explicación aun para personas familiarizadas con arquitecturas vectoriales.

8.2 Instrucciones que Computan Vectores

La Figura 8.1 es una representación gráfica de la extensión RV32V al set de instrucciones. La codificación completa de RV32V aún no está terminada, por lo que esta edición no incluye el acostumbrado diagrama de estructura de instrucciones.

Virtualmente todas las instrucciones que computan números enteros o de punto flotante de capítulos anteriores tienen una versión vectorizada: La Figura 8.1 hereda operaciones de

RV32I, RV32M, RV32F, RV32D y RV32A. Hay varios tipos de instrucciones vectorizadas dependiendo si los operandos son todos vectores (sufijo `.vv`) o un vector operado con un escalar (sufijo `.vs`). Un sufijo escalar significa un registro `x` o `f` operado con un registro vector (`v`). Por ejemplo, nuestro programa DAXPY (Figura 5.7 página 60 Capítulo 5) calcula $Y = a \times X + Y$, donde X y Y son vectores, y a es un escalar. Para operaciones vector-escalar, el campo `rsl` especifica el registro escalar a usarse.

Operaciones asimétricas como la resta y la división ofrecen una tercera variante de instrucciones vectorizadas, donde el primer operando es escalar y el segundo es vector (sufijo `.sv`). Operaciones como $Y = a - X$ las utilizan. Estas son superfluas para operaciones simétricas como suma y multiplicación, así que estas instrucciones no tienen versiones `.sv`. Las instrucciones fusionadas de multiplicación-suma tienen tres operandos, por lo que tienen la mayor combinación de opciones de vector y escalar: `.vvv`, `.vvs`, `.vsv`, y `.vss`.

Los lectores notarán que la Figura 8.1 ignora el tipo de dato y el ancho de las operaciones vectorizadas. La próxima sección explica por qué.

8.3 Registros Vectorizados y Tipado Dinámico

RV32V agrega 32 registros vectorizados, cuyos nombres comienzan con `v`, pero el número de *elementos* por registro vector varía. Dicho número depende tanto del ancho de las operaciones y de la cantidad de memoria dedicada a registros vectorizados, lo cual depende del diseñador del procesador. Por ejemplo, si el procesador reservó 4096 bytes para registros vectorizados, sería suficiente para que los 32 registros vectorizados soporten 16 elementos de 64 bits cada uno, 32 elementos de 32 bits, 64 elementos de 16 bits o 128 elementos de 8 bits.

Para mantener la flexibilidad en la cantidad de elementos de un ISA vectorizado, un procesador vectorizado calcula la *longitud máxima del vector* (`mv1`) la cual usan los programas para ejecutarse apropiadamente en procesadores con distintas cantidades de memoria asignada a los registros vectorizados. El registro de longitud de vector (`v1`) asigna la cantidad de elementos por vector para una operación particular, lo cual ayuda a los programas cuando la dimensión del arreglo no es múltiplo de `mv1`. Mostraremos el uso de `mv1`, `v1` y los ocho registros predicados (`vpi`) en mayor detalle en las siguientes secciones.

RV32V tiene el enfoque innovador de asociar el tipo de dato y la longitud con los *registros vectorizados* en lugar de los *opciones de las instrucciones*. El programa marca los registros vectoriales con su tipo de dato y longitud antes de computar las instrucciones vectorizadas. Utilizar *tipado dinámico de registros*² reduce drásticamente el número de instrucciones vectorizadas, lo cual es importante dado que normalmente hay seis versiones enteras y tres de punto flotante por cada instrucción vectorizada como se muestra en la Figura 8.1.

Como veremos en la Sección 8.9 cuando afrontemos varias instrucciones SIMD, una arquitectura vectorizada con tipado dinámico reduce la carga cognitiva para el programador de ensamblador y la dificultad del generador de código del compilador.

Otra ventaja de tener tipos de datos dinámicos es que los programas pueden deshabilitar registros vectoriales no utilizados. Esta característica asigna toda la memoria vectorial a los registros vectorizados que estén habilitados. Por ejemplo, supongamos que únicamente dos registros están habilitados, ambos floats de 64 bits, y el procesador tiene 1024 bytes de memoria para registros vectorizados. El procesador asignaría la mitad de esa memoria a cada registro vector, asignando 512 bytes o $512/8 = 64$ elementos por consiguiente escribiendo 64 en `mv1`. Así pues, `mv1` es dinámico, pero su valor es asignado por el procesador y no puede ser modificado directamente por software.

Tipo	Punto Flotante		Entero Signed		Entero Unsigned	
Ancho	Nombre	vetype	Nombre	vetype	Nombre	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: Codificación de tipos de datos en registros vectorizados. Los tres bits de la derecha definen el ancho de los datos, los dos bits de la izquierda definen su tipo. X64 y U64 están disponibles solo para RV64V. F16 y F32 requieren la extensión RV32F y F64 requiere RV32F y RV32D. F16 se adhiere al estándar de punto flotante de 16 bits IEEE 754-2008 (binary16). Escribir 00000 a vetype deshabilita los registros vectorizados (La Tabla 17.4 de [Waterman and Asanović 2017] es la base para esta figura).

Los registros fuente y destino determinan el tipo y tamaño de la operación y su resultado, por lo que el tipado dinámico hace conversiones implícitas. Por ejemplo, un procesador puede multiplicar un vector de números de punto flotante de precisión doble por un número escalar de precisión simple sin necesidad de convertir los operandos. Esta característica reduce el número total de instrucciones vectorizadas así como la cantidad de instrucciones ejecutadas.

La instrucción `vsetdcfg` asigna los tipos a los registros vectorizados. La Figura 8.2 muestra los tipos disponibles en RV32V para los registros vectorizados además de los tipos para RV64V (ver Capítulo 9). RV32V requiere que las operaciones vectorizadas de punto flotante también tengan versiones escalares. Por esto, se debe tener al menos RV32FV para usar el tipo F32 y RV32FDV para usar el tipo F64. RV32V introduce un formato tipo F16 para punto flotante de 16 bits. Si una implementación soporta ambos RV32V y RV32F, entonces debe soportar los formatos F16 y F32.

■ **Elaboración:** *RV32V puede cambiar de contexto rápidamente.*

Una razón por la que las arquitecturas vectorizadas fueron menos populares que las SIMD era la preocupación de que agregar registros vectorizados largos incrementaría el tiempo usado para guardar y restaurar el programa en caso de una interrupción, llamado un *cambio de contexto*³. Tener registros de tipo dinámico es útil. El programador debe indicarle al procesador qué registros vectorizados están siendo utilizados, lo que significa que el procesador únicamente debe guardar y restaurar dichos registros en un cambio de contexto. La convención de RV32V es deshabilitar *todos* los registros vectorizados cuando no se usa instrucciones vectorizadas, lo que significa que el procesador puede tener el beneficio de los registros vectorizados y únicamente debe pagar el tiempo adicional en un cambio de contexto si una interrupción ocurre mientras se ejecute instrucciones vectorizadas. Arquitecturas vectorizadas anteriores debían pagar el tiempo adicional del peor caso de cambio de contexto (de guardar y restaurar todos los registros vectorizados) siempre que ocurriera una interrupción.

La preocupación por un cambio de contexto lento hizo que Intel evitara agregar registros en la extensión SIMD MMX original. Simplemente reutilizó los registros de punto flotante existentes, lo que significó que no había contexto nuevo que cambiar, pero el programa no podía mezclar instrucciones de punto flotante con instrucciones multimedia.

8.4 Loads y Stores Vectorizados

Cada load y store tiene un offset inmediato de 7 bits sin signo que se escala por el tipo de elemento en el registro destino para loads y registro fuente para stores.

El caso más sencillo para loads y stores vectorizados es con arreglos de una dimensión que son almacenados secuencialmente en memoria. La instrucción de load vectorizado llena un registro vector con datos secuenciales en memoria comenzando con la dirección `v1d`. El tipo de dato asociado con el registro vector determina el tamaño de los elementos y el registro de longitud de vector `v1` indica la cantidad de elementos a cargar. Store vectorizado `vst` hace la operación inversa de `v1d`.

Por ejemplo, si `a0` tiene 1024, y el tipo de `v0` es X32, entonces `v1d v0, 0(a0)` generará las direcciones 1024, 1028, 1032, 1036, ... hasta llegar al límite dado por `v1`.

Para arreglos multidimensionales, algunos accesos no serán secuenciales. Si se almacenan fila por fila, accesos secuenciales a columnas en un arreglo bidimensional quieren los elementos separados por el tamaño de la fila. Las arquitecturas vectorizadas soportan este tipo de accesos utilizando transferencias de datos por *zancadas*⁴: `vlds` y `vsts`. A pesar de que uno podría obtener el mismo efecto de `v1d` y `vst` poniendo el tamaño de la zancada igual al valor del elemento en `vlds` y `vsts`, las instrucciones `v1d` y `vst` garantizan que todos los accesos serán secuenciales, lo cual facilita proveer una alta transferencia de datos a memoria. Otro motivo para proveer `v1d` y `vst` es que reduce el tamaño del código y número de instrucciones para el caso común donde la zancada es uno. Estas instrucciones indican dos registros fuente, uno indica la dirección donde comenzará y el otro indica el tamaño de la zancada.

Por ejemplo, asuma que la dirección inicial `a0` fuera 1024, y el tamaño de la fila en `a1` fuera 64 bytes. `vlds v0, a0, a1` enviaría esta secuencia de direcciones a memoria: 1024, 1088 ($1024 + 1 \times 64$), 1152 ($1024 + 2 \times 64$), 1216 ($1024 + 3 \times 64$), y así sucesivamente hasta que el registro de longitud de vector `v1` le indique que debe parar. Los datos retornados son almacenados en elementos secuenciales del registro vector destino.

Hasta el momento, hemos asumido que el programa trabaja con arreglos densos. Para soportar arreglos dispersos, las arquitecturas vectoriales ofrecen transferencias de datos *indexados*: `vldx` y `vstx`. En estas instrucciones, un registro especifica un registro vector y un registro escalar. El registro escalar tiene la dirección donde comienza el arreglo disperso, y cada elemento del registro vector contiene el índice de los elementos distintos de cero el arreglo disperso.

Supongamos que la dirección inicial del arreglo `a0` fuese 1024, y que el registro vector `v1` tuviera estos índices en sus cuatro primeros elementos: 16, 48, 80, 160. `vldx v0, a0, v1` enviaría esta secuencia de direcciones a memoria: 1040 ($1024 + 16$), 1072 ($1024 + 48$), 1104 ($1024 + 80$), 1184 ($1024 + 160$). Carga los datos retornados en elementos secuenciales del registro vector destino.

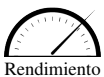
Utilizamos arreglos dispersos como motivación para loads y stores indexados, pero hay muchos otros algoritmos que acceden a datos indirectamente usando tablas de índices.

8.5 Paralelismo Durante la Ejecución Vectorizada

Si bien un procesador vectorizado simple puede ejecutar un elemento de un vector a la vez, las operaciones de elementos son independientes por definición, así que un procesador podría teóricamente ejecutar todas simultáneamente. El tipo de dato más ancho en RV32G es de 64 bits, y hoy en día los procesadores vectorizados normalmente ejecutan dos, cuatro u ocho elementos de 64 bits por ciclo de reloj. Hardware especializado se encarga de los ca-



Load indexado también es llamado gather y store indexado también se llama scatter.



tos marginales donde la longitud del vector no es un múltiplo del número de elementos a ejecutarse por ciclo de reloj.

Similar a SIMD, la cantidad de operaciones con datos más pequeños es la relación de tamaños del dato pequeño al grande. Así pues, un procesador vectorial que computa 4 operaciones de 64 bits por ciclo de reloj normalmente haría 8 operaciones de 32 bits, 16 de 16 bits y 32 de 8-bits.

En SIMD, el arquitecto del ISA determina la cantidad máxima de operaciones paralelas por ciclo de reloj y la cantidad de elementos por registro. En contraste, el diseñador del procesador de RV32V elige ambos sin modificar el ISA o el compilador, mientras que cada duplicación el tamaño de los registros en SIMD duplica la cantidad de instrucciones y requiere modificar los compiladores SIMD. Esta flexibilidad oculta significa que el mismo programa de RV32V corre sin cambios en los procesadores vectorizados más sencillos y en los más agresivos.



8.6 Ejecución Condicional de Operaciones Vectoriales

Algunas computaciones vectoriales incluyen condiciones *if*. En lugar de usar *branches* condicionales, las arquitecturas vectorizadas incluyen una máscara que suprime operaciones en algunos elementos del vector. Las instrucciones predicadas en la Figura 8.1 ejecutan pruebas condicionales entre dos vectores o un vector y un escalar y le escriben a cada elemento de la máscara vectorial un 1 si la condición se cumple o un 0 de lo contrario (La máscara vectorial debe tener la misma cantidad de elementos que el registro vector). Cualquier instrucción vectorizada posterior puede usar dicha máscara, donde un 1 en el bit *i* significa que el elemento *i* es modificado por operaciones vectoriales, y un 0 significa que el elemento *i* no se modifica.



RV32V provee 8 *registros predicados vectoriales*⁵ (*vp*i**) que sirven como máscaras vectorizadas. Las instrucciones *vpand*, *vpandn*, *vp_{or}*, *vp_{xor}* y *vpnot* ejecutan instrucciones lógicas para ser combinadas y permitir el procesamiento eficiente de expresiones condicionales anidadas.

Las instrucciones RV32V especifican ya sea *vp0* o *vp1* para usarse como la máscara que controla una operación vectorial. Para ejecutar una operación normal a todos los elementos, uno de esos dos registros predicados debe tener todos sus bits en uno. Para intercambiar rápidamente de uno de los seis registros predicados hacia *vp0* o *vp1*, RV32V tiene la instrucción *vpswap*. Los registros predicados también se habilitan dinámicamente, y al deshabilitarlos se limpian todos los registros predicados rápidamente.

Por ejemplo, supongamos que todos los elementos pares en un registro vector *v3* fueran enteros negativos y que todos los impares fueran enteros positivos. El resultado de este código:

```

vplt.v3, v3, 0 # poner en 1 los bits de la máscara cuando
               # los elementos de v3 < 0
add.vv, vp0, v0, v1, v2 # modificar los elementos de v0 a v1+v2
                       # si la máscara es 1

```

pondría todos los bits pares de *vp0* a 1, todos los impares a 0, y reemplazaría todos los elementos pares de *v0* con la suma de los elementos *v1* y *v2*. Los elementos impares de *v0* no cambiarían.

Se dice que un programa es *vectorizable* si la mayoría de las operaciones son instrucciones vectorizadas. *Gather*, *scatter* e instrucciones predicadas aumentan la cantidad de programas vectorizables.

8.7 Instrucciones Vectorizadas Misceláneas

Además de la instrucción que configura los tipos de datos en registros vectorizados (`vsetdcfg`) mencionada anteriormente, `setvl` escribe en el registro de longitud de vector (`v1`) y en el registro destino, el menor entre el operando fuente y la longitud máxima de vector (`mv1`). La razón para elegir el mínimo es para saber si en un ciclo el código vectorizado puede ejecutarse a la longitud de vector máxima (`mv1`) o si debe ejecutarse a un valor menor para cubrir los elementos restantes. Así pues, la instrucción `setvl` se ejecuta en cada iteración para este caso marginal.

RV32V tiene tres instrucciones que manipulan elementos dentro de un registro vector.

Vector select (`vselect`) produce un nuevo vector tomando elementos de un vector fuente de datos utilizando los índices especificados en el segundo vector:

```
# vindices contiene los valores desde 0..mv1-1
# que seleccionan elementos de vsrc

vselect vdest, vsrc, vindices
```

Así, si los cuatro primeros elementos de `v2` contienen 8, 0, 4, 2, entonces `vselect v0, v1, v2` reemplazará el elemento cero de `v0` con el octavo elemento de `v1`, el primer elemento de `v0` (índice 1) con el elemento cero de `v1`, el segundo elemento de `v0` con el cuarto elemento de `v1`, y el tercer elemento de `v0` con el segundo elemento de `v1`.

Vector merge (`vmerge`) es similar a vector select, pero usa un registro vector predicado para saber cuales de las fuentes usar. Produce un nuevo vector, resultado de guardar los elementos de algún registro fuente basado en el registro predicado. El nuevo elemento viene de `vsrc1` si el elemento en el registro predicado es 0, o de `vsrc2` si es 1:

```
# el bit i de determina si el nuevo elemento i de vdest
# viene de vsrc1 (si el bit i == 0) o vsrc2 (si el bit i == 1)
vmerge, vp0 vdest, vsrc1, vsrc2
```

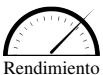
Así, si los primeros cuatro elementos de `vp0` contienen 1, 0, 0, 1, los primeros cuatro elementos de `v1` son 1, 2, 3, 4, y los primeros cuatro elementos de `v2` son 10, 20, 30, 40, entonces `vmerge, vp0 v0, v1, v2` produciría: `v0 = 10, 2, 3, 40`.

La instrucción vector extract toma elementos del medio de un vector y los copia al principio del segundo registro vector.

```
# start es un reg escalar indicando el elemento inicial de vsrc
vextract vdest, vsrc, start
```

Por ejemplo, si la longitud del vector `v1` es 64 y `a0` es 32, entonces `vextract v0, v1, a0` copiaría los últimos 32 elementos de `v1` a los primeros 32 de `v0`.

La instrucción `vextract` ayuda en reducciones siguiendo un enfoque de partir a la mitad recursivamente para cualquier operador binario asociativo. Por ejemplo, para sumar todos los elementos de un registro vector, usamos vector extract para copiar la última mitad de un registro vector hacia la primera de otro y dividimos la longitud a la mitad. Luego, sumamos ambos registros y repetimos la partición recursiva y la suma hasta que el vector tenga longitud igual a 1. El resultado en el elemento cero será la suma de todos los elementos en el registro vector original.



```

# a0 es n, a1 apunta a x[0], a2 apunta a y[0], fa0 es a
0: li t0, 2<<25
4: vsetdcfg t0          # habilitar 2 registros Fl.Pt. de 64b
loop:
8: setv1 t0, a0         # v1 = t0 = min(mv1, n)
c: vld v0, a1          # cargar vector x
10: slli t1, t0, 3      # t1 = v1 * 8 (en bytes)
14: vld v1, a2         # cargar vector y
18: add a1, a1, t1      # incrementar puntero C a x por v1*8
1c: vfmadd v1, v0, fa0, v1 # v1 += v0 * fa0 (y = a * x + y)
20: sub a0, a0, t0      # n -= v1 (t0)
24: vst v1, a2         # almacenar Y
28: add a2, a2, t1      # incrementar puntero C a y por v1*8
2c: bnez a0, loop      # repetir si n != 0
30: ret                # retornar

```

Figura 8.3: Código RV32V para DAXPY en la Figura 5.7. El lenguaje máquina no aparece porque los opcodes de RV32V aún no han sido definidos.

8.8 Ejemplo Vectorizado: DAXPY en RV32V

La Figura 8.3 muestra el programa en ensamblador de RV32V para DAXPY (La Figura 5.7 en la página 60, Capítulo 5), el cual explicaremos paso a paso.

DAXPY en RV32V comienza habilitando los registros vectorizados usados en la función. Requiere solamente dos registros para almacenar partes de x e y , que son números de punto flotante de precisión doble de 8 bytes. La primera instrucción crea una constante y la segunda la escribe al registro de control y estado que configura los registros vectorizados (`vcfgd`) para tener dos registros de tipo F64 (ver Figura 8.2). Por definición, el hardware genera los registros en orden numérico, resultando en `v0` and `v1`.

Asumamos que nuestro procesador RV32V tiene 1024 bytes de memoria dedicada a registros vectorizados. El hardware reparte la memoria entre los dos registros vectorizados, que almacenan números de punto flotante de precisión doble (8 bytes). Cada registro vector tiene $512/8 = 64$ elementos, por lo que el procesador define la longitud máxima del vector ($mv1=64$) para esta función.

La primera instrucción en el ciclo asigna la longitud del vector para las siguientes instrucciones vectorizadas. La instrucción `setv1` escribe el menor entre $mv1$ y n en `v1` y `t0`. La idea es que si la cantidad de iteraciones del ciclo fuera mayor que n , lo más rápido que puede ejecutarse son 64 valores simultáneos, así que asigna $v1 = mv1$. Si n es menor que $mv1$, no podemos leer ni escribir pasado el límite de x e y , por lo que deberíamos computar solo los últimos n elementos en esta última iteración. `setv1` también escribe a `t0` para ayudar luego en la logística del ciclo en la dirección 10.

La instrucción `vld` en la dirección `c` es un load vectorial desde la dirección `x` en el registro escalar `a1`. Transfiere $v1$ elementos de x desde memoria a `v0`. La siguiente instrucción de corrimiento `slli` multiplica la longitud del vector por el ancho de los datos en bytes (8) para luego incrementar los punteros de x e y .

La instrucción en la dirección 14 (`vld`) carga $v1$ elementos de y de memoria hacia `v1` y la siguiente instrucción (`add`) incrementa el valor del puntero a x .

Con la instrucción en la dirección 1c nos ganamos la lotería. `vfmadd` multiplica $v1$ ele-

La V en RISC-V también es por vector. Los arquitectos de RISC-V tenían mucha experiencia positiva con arquitecturas vectorizadas y estaban frustrados porque SIMD dominaba los microprocesadores. Entonces, la V es por el quinto proyecto RISC en Berkeley y porque su ISA enfatizaría vectores.

Arquitecturas vectoriales sin `setv1` tienen código extra llamado *strip-mining* que asigna a `v1` los últimos n elementos del ciclo y revisa si n es inicialmente cero.

mentos de x ($v0$) por el escalar a ($f0$) y suma cada producto a los $v1$ elementos de y ($v1$) y almacena esas $v1$ sumas de regreso en y ($v1$).

Lo único que queda es almacenar los resultados en memoria y un poco de *overhead* del ciclo. La instrucción en la dirección 20 (*sub*) decrementa n ($a0$) por $v1$ para llevar un conteo de la cantidad de operaciones realizadas en esta iteración del ciclo. La próxima instrucción (*vst*) almacena $v1$ resultados y a memoria. La instrucción en la dirección 28 (*add*) incrementa el puntero a y , y la siguiente instrucción repite el ciclo si n ($a0$) no es cero. Si n es cero, la última instrucción *ret* retorna de la función.

El poder de la arquitectura vectorizada es que cada iteración de este ciclo de 10 instrucciones ejecuta $3 \times 64 = 192$ accesos a memoria y $2 \times 64 = 128$ sumas y multiplicaciones de punto flotante (asumiendo que n es al menos 64). Eso promedia unos 19 accesos a memoria y 13 operaciones por instrucción. Como veremos en la siguiente sección, este rendimiento es un orden de magnitud mejor que SIMD.



Rendimiento

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

ARM-32 tiene dos extensiones SIMD llamadas NEON pero no soporta instrucciones de punto flotante de precisión doble, así que no ayuda a DAXPY.

Ahora veremos el contraste de cómo las arquitecturas SIMD y vectorizada ejecutan DAXPY. Si inclina la cabeza, podrá notar que SIMD es una arquitectura vectorizada restringida con registros pequeños—ocho “elementos” de 8 bits—pero no tiene registro de longitud de vector ni transferencia de datos por zancadas ni indexada.

MIPS SIMD. La Figura 8.5 en la página 88 muestra la versión de DAXPY para MSA (MIPS SIMD Architecture: Arquitectura SIMD de MIPS). Cada instrucción SIMD MSA puede operar con dos números de punto flotante dado que los registros MSA son de 128 bits.

A diferencia de RV32V, dado que no hay un registro de longitud de vector, MSA requiere datos e instrucciones adicionales de validación para detectar problemas con n . Cuando n es impar, hay que ejecutar código adicional para computar una multiplicación-suma de punto flotante dado que MSA *debe* operar con *pares* de operandos. Dicho código aparece desde 3c hasta 4c en la Figura 8.5. En el raro pero posible caso en que n sea cero, el branch en la dirección 10 se saltará el ciclo principal.

Si dicho branch no se salta el ciclo principal, la instrucción en la dirección 18 (*splat.i.d*) pone una copia de a en ambas mitades del registro SIMD $w2$. Para sumar datos escalares en SIMD, necesitamos replicarlo para que sea tan ancho como el registro SIMD.

Dentro del ciclo, la instrucción *ld.d* en la dirección 1c carga dos elementos de y al registro SIMD $w0$ y luego incrementa el puntero a y . Luego carga dos elementos de x al registro SIMD $w1$. La siguiente instrucción en la dirección 28 incrementa el puntero a x . Luego tenemos la provechosa instrucción de multiplicación-suma en la dirección 2c.

El branch (retardado) al final del ciclo revisa si el puntero hacia y ha excedido el límite del último elemento de y . Si no lo ha hecho, se repite el ciclo. La instrucción SIMD *guardar en el hueco de retardo de branch* en la dirección 34 escribe el resultado a dos elementos de y .

Luego que termina el ciclo principal, el código revisa si n es impar. Si lo es, ejecuta la última multiplicación-suma usando instrucciones escalares vistas en el Capítulo 5. La última instrucción retorna de la función.

El ciclo de 7 instrucciones del código DAXPY para MIPS MSA ejecuta 6 accesos a memoria de precisión doble y 4 sumas y multiplicaciones de punto flotante. El promedio está alrededor de 1 acceso a memoria y 0.5 operaciones por instrucción.

Dicho código de validación es considerado parte de *strip mining* en arquitecturas vectorizadas. Como lo explica el texto en la Figura 8.5, el registro de longitud de vector $v1$ hace innecesario el código de SIMD en RV32V. Las arquitecturas vectorizadas tradicionales usan código especial para manejar el caso marginal de $n = 0$. RV32V simplemente hace que las instrucciones vectorizadas se comporten como *nops* cuando $n = 0$.

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instrucciones (estáticas)	22	29	13
Bytes (estáticos)	88	92	52
Instrucciones por Main Loop	7	6	10
Resultados por Main Loop	2	4	64
Instrucciones (dinámicas, n=1000)	3511	1517	163

Figura 8.4: Cantidad de instrucciones y tamaño del código de DAXPY para ISAs vectorizados. Muestra el total de instrucciones (estáticas), tamaño del código, número de instrucciones y resultados por ciclo, y cantidad de instrucciones ejecutadas (n = 1000). microMIPS con MSA reduce el tamaño del código a 64 bytes y RV32FDCV lo reduce a 40 bytes.

SIMD x86. Intel ha pasado por muchas generaciones de extensiones SIMD, lo cual se aprecia en el código de la Figura 8.6 en la página 89. La expansión SSE hacia SIMD de 128 bits llevó a los registros `xmm` y a las instrucciones que los usan, y la expansión hacia SIMD de 256 bits como parte de AVX creó los registros `ymm` y sus instrucciones.

El primer grupo de instrucciones en las direcciones de la 0 a la 25 carga las variables desde memoria, hace cuatro copias de `a` los registros `ymm` de 256 bits, y se asegura de que `n` sea al menos 4 antes de entrar al ciclo principal. Usa dos instrucciones SSE y una AVX (El texto de la Figura 8.6 explica cómo, a mayor detalle).

El ciclo principal es el corazón de la computación de DAXPY. La instrucción AVX `vmovapd` en la dirección 27 carga 4 elementos de `x` hacia `ymm0`. La instrucción AVX `vfmadd213pd` en la dirección 2c multiplica 4 copias de `a` (`ymm2`) por 4 elementos de `x` (`ymm0`), suma 4 elementos de `y` (en memoria en la dirección `ecx+edx*8`), y guarda las 4 sumas en `ymm0`. La siguiente instrucción AVX en la dirección 32, `vmovapd`, almacena los 4 resultados en `y`. Las siguientes tres instrucciones incrementan contadores y repiten el ciclo si fuera necesario.

Al igual que con MIPS MSA, el código “marginal” entre las direcciones 3e y 57 (etiquetado `Fringe`) se encarga de los casos cuando `n` no es múltiplo de 4. Utiliza tres instrucciones SSE.

Las 6 instrucciones del ciclo principal en el código DAXPY para x86-32 AVX2 ejecuta 12 accesos a memoria de precisión doble y 8 multiplicaciones y sumas de punto flotante. Promedia 2 accesos a memoria y una operación por instrucción.

■ **Elaboración: La Illiac IV fue la primera en mostrar la complejidad de compilar para SIMD.**

Con 64 FPU's (Floating Point Units: Unidades de Punto Flotante) paralelas de 64 bits, la Illiac IV planeaba tener mas de un millón de compuertas lógicas antes que Moore publicara su ley. Sus arquitectos originalmente predijeron 1000 MFLOPS (Mega floating-point operations per second: millones de operaciones de punto flotante por segundo), pero el rendimiento real fue de 15 MFLOPS a lo sumo. Los costos escalaron del estimado de \$8M en 1966 hasta \$31M en 1972, a pesar de que se construyó solo 64 de los 256 FPU's planeados. El proyecto comenzó en 1965 pero fue hasta 1976 que ejecutó su primera aplicación real, el mismo año en que la Cray-1 fue develada. Posiblemente la supercomputadora más infame, alcanzó uno de los 10 primeros lugares en una lista de desastres de ingeniería [Falk 1976].

8.10 Observaciones Finales

Si el código es vectorizable, la mejor arquitectura es la vectorizada.

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

La Figura 8.4 resume el número de instrucciones y cantidad de bytes en DAXPY para programas en RV32IFDV, MIPS-32 MSA y x86-32 AVX2. El código de computación de SIMD es opacado por el código adicional que conlleva. De dos tercios a tres cuartos del código de MIPS-32 MSA y x86-32 AVX2 es *overhead* de SIMD, ya sea para preparar los datos para el ciclo principal de SIMD o para manejar los casos marginales cuando n no es un múltiplo de la cantidad de números de punto flotante en un registro SIMD.

El código RV32V en la Figura 8.3 no tiene ese overhead, reduciendo a la mitad la cantidad de instrucciones. A diferencia de SIMD, tiene un registro con la longitud del vector, permitiendo a las instrucciones vectorizadas trabajar con cualquier valor de n . Podría pensarse que RV32V tendría un problema si n es 0. No lo tiene dado que las instrucciones vectorizadas en RV32V no modifican nada si $v1 = 0$.

Sin embargo, la diferencia más significativa entre SIMD y procesamiento vectorizado no es el tamaño del código estático. Las instrucciones SIMD ejecutan de 10 a 20 veces más instrucciones que RV32V dado que cada ciclo de SIMD solo opera con 2 ó 4 elementos en vez de 64 como en el caso vectorial. El hecho de que haya más fetches y decodes significa que se usa más energía para realizar la misma tarea.

Comparando los resultados de la Figura 8.4 con la versión escalar de DAXPY en la Figura 5.8, página 31, Capítulo 5, notamos que SIMD prácticamente duplica el tamaño del código en instrucciones y bytes, pero el tamaño del ciclo es casi igual. La reducción en la cantidad dinámica de instrucciones ejecutadas es un factor de 2 ó 4, dependiendo del ancho de los registros SIMD. ¡Sin embargo, el tamaño del código vectorizado de RV32V incrementa por un factor de 1.2 (el ciclo principal a 1.4X) pero la ejecución dinámica de instrucciones se reduce en un factor de 43!

A pesar de que el conteo de instrucciones dinámicas es una gran diferencia, en nuestra opinión esa es la segunda disparidad más significativa entre arquitecturas SIMD y vectorizadas. La falta del registro de longitud de vector incrementa la cantidad de instrucciones así como el código de mantenimiento. ISAs como MIPS-32 y x86-32 que siguen la doctrina incrementalista deben duplicar todas las instrucciones SIMD definidas para registros más angostos cada vez que los agrandan. Seguramente, cientos de instrucciones de MIPS-32 y x86-32 fueron creadas durante muchas generaciones de ISAs SIMD y otros cientos estarán en su futuro. La carga cognitiva para el programador de ensamblador de este enfoque de fuerza bruta en la evolución del ISA debe ser agobiante. ¿Cómo recordar qué significa `vfmadd213pd` y cuándo usarlo?

En comparación, el código de RV32V no se ve afectado por el tamaño de la memoria de los registros vectorizados. No solo RV32V permanece sin cambios si se expande el tamaño de la memoria vectorizada, ni siquiera hay que recompilarlo. Dado que el procesador provee el valor máximo de la longitud del vector `mv1`, el código en la Figura 8.3 no cambia aunque el procesador incremente su memoria vectorial de 1024 bytes hasta, digamos, 4096 bytes, o la reduce a 256 bytes.



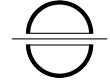
Simplicidad



Rendimiento

A diferencia de SIMD, donde el ISA impone el hardware requerido—y cambiar el ISA significa cambiar el compilador—el ISA RV32V permite a los diseñadores de procesadores elegir los recursos para paralelismo de datos sin afectar al programador o al compilador. Uno podría argumentar que SIMD viola el fundamento de diseño de ISAs del Capítulo 1 de separar la arquitectura de la implementación.

Creemos que el gran contraste entre costo-energía-rendimiento, complejidad y facilidad de programación entre el enfoque vectorial modular de RV32V y las arquitecturas incrementalistas SIMD de ARM-32, MIPS-32 y x86-32 podría ser el argumento más persuasivo para RISC-V.



Aislamiento de Arq e Impl



Elegancia

8.11 Para Aprender Más

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/>.

Notas

¹Paralelismo a nivel de datos: En inglés, *data-level parallelism*.

²Tipado dinámico de registros: En inglés, *dynamic register typing*.

³Cambio de contexto: En inglés, *context switch*.

⁴Transferencia de datos por zancadas: En inglés, *Strided data transfer*.

⁵Registros predicados vectoriales: En inglés, *vector predicate registers*.

```

# a0 es n, a2 apunta a x[0], a3 apunta a y[0], $w13 es a
00000000 <daxpy>:
  0: 2405fffe li      a1,-2
  4: 00852824 and     a1,a0,a1 # a1 = floor(n/2)*2 (mask bit 0)
  8: 000540c0 sll     t0,a1,0x3 # t0 = byte address de a1
  c: 00e81821 addu    v1,a3,t0 # v1 = &y[a1]
 10: 10e30009 beq     a3,v1,38 # si y==&y[a1] ir a Fringe (t0==0, n es 0|1)
 14: 00c01025 move    v0,a2 # (hueco del retardo) v0 = &x[0]
 18: 78786899 splati.d $w2,$w13[0] # w2 = llenar registro SIMD con copias de a
Loop:
 1c: 78003823 ld.d     $w0,0(a3) # w0 = 2 elementos de y
 20: 24e70010 addiu   a3,a3,16 # inc puntero C a y por 2 números de Pt.Fl.
 24: 78001063 ld.d     $w1,0(v0) # w1 = 2 elementos de x
 28: 24420010 addiu   v0,v0,16 # inc puntero C a x por 2 números de Pt.Fl.
2c: 7922081b fmaddd $w0,$w1,$w2 # w0 = w0 + w1 * w2
 30: 1467fffa bne     v1,a3,1c # si (fin de y != ptr a y) ir a Loop
 34: 7bfe3827 st.d     $w0,-16(a3) # (hueco del ret.) guardar 2 elementos de y
Fringe:
 38: 10a40005 beq     a1,a0,50 # si (n es par) ir a Done
 3c: 00c83021 addu    a2,a2,t0 # (hueco del retardo) 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 si n impar)
4c: f46d0000 sdc1    $f13,0(v1) # y[n-1] = f13 (almacenar resultado impar)
Done:
 50: 03e00008 jr      ra      # retornar
 54: 00000000 nop

```

Figura 8.5: Código MIPS-32 MSA para DAXPY de la Figura 5.7. El *overhead* de SIMD es evidente cuando se compara este código con el código RV32V de la Figura 8.3. La primera parte del código MIPS MSA (direcciones 0 a 18) duplica la variable escalar *a* en un registro SIMD y se asegura de que *n* sea al menos 2 antes de entrar al ciclo principal. La tercera parte del código MIPS MSA (direcciones 38 a 4c) manejan el caso marginal cuando *n* no es múltiplo de 2. Dicho código es innecesario en RV32V dado que el registro de longitud de vector *v1* y la instrucción *setv1* le permite al ciclo trabajar con cualquier valor de *n*, par o impar.

```

# eax es i, n es esi, a es xmm1, ebx apunta a x[0], ecx apunta a y[0]
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,0xffffffff # 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          # si n < 4, ir a Fringe
 25: 31 d2          xor  edx,edx     # edx = 0
Loop:
 27: c5 fd 28 04 d3  vmovapd ymm0,[ebx+edx*8] # cargar 4 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 # guardar en 4 elementos de y
 37: 83 c2 04          add  edx,0x4
 3a: 39 c2          cmp  edx,eax     # comparar con n
 3c: 72 e9          jb  27          # repetir loop si < n
Fringe:
 3e: 39 c6          cmp  esi,eax     # ¿quedan elementos marginales?
 40: 76 17          jbe  59          # si (n mod 4) == 0 ir a Done
FringeLoop:
 42: c5 fb 10 04 c3  vmovsd xmm0,[ebx+eax*8] # cargar 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 # guardar en elemento de y
 52: 83 c0 01          add  eax,0x1     # incrementar cuenta Fringe
 55: 39 c6          cmp  esi,eax     # comparar cuentas Loop y Fringe
 57: 75 e9          jne  42 <daxpy+0x42> # repetir FringeLoop si != 0
Done:
 59: 5b          pop  ebx         # epílogo de la función
 5a: 5e          pop  esi
 5b: c3          ret

```

Figura 8.6: Código x86-32 AVX2 para DAXPY de la Figura 5.7. La instrucción SSE `vmovsd` en la dirección a carga a en la mitad del registro `xmm1` de 128-bits. La instrucción SSE `vmovddup` en la dirección 14 duplica a a ambas mitades de `xmm1` para la computación SIMD posterior. La instrucción AVX `vinsertf128` en la dirección 1d hace cuatro copias de a en `ymm2` duplicando las dos copias de a en `xmm1`. Las tres instrucciones AVX en las direcciones 42 a 4d (`vmovsd`, `vfmadd213sd`, `vmovsd`) manejan el caso cuando $\text{mod}(n,4) \neq 0$. Ejecutan el cómputo de DAXPY un elemento a la vez, repitiendo el ciclo hasta que la función haya ejecutado exactamente n operaciones de multiplicación-suma. Nuevamente, dicho código es innecesario para RV32V porque el registro de longitud de vector `v1` y la instrucción `setv1` hacen que el ciclo funcione para cualquier valor de n.

RV64: Instrucciones de Memoria de 64 bits

C. Gordon Bell (1934-) fue uno de los principales arquitectos de dos de las arquitecturas de minicomputadoras más populares de su época: la PDP-11 de Digital Equipment Corporation (direcciones de 16 bits), la cual fue anunciada en 1970, y su sucesor siete años después, la VAX-11 (Virtual Address eXtension) con direcciones de 32 bits de Digital Equipment Corporation.



Solamente hay un error que se puede cometer en diseño de computadoras que es difícil reponerse—no tener suficientes bits para el manejo y direccionamiento de la memoria.

—C. Gordon Bell, 1976

9.1 Introducción

Las Figuras 9.1 a 9.4 dan una representación gráfica de las versiones RV64G de las instrucciones RV32G. Estas figuras ilustran el pequeño incremento en la cantidad de instrucciones para cambiar a un ISA de 64 bits en RISC-V. Los ISAs típicamente solo agregan un par de versiones *word*, *doubleword* o *long* de las instrucciones de 32 bits y expanden los registros, incluyendo al PC, a 64 bits. Por eso, sub en RV64I resta dos números de 64 bits en vez de dos números de 32 bits como en RV32I. RV64 se parece, pero es en realidad un ISA distinto a RV32; agrega un par de instrucciones y las instrucciones base funcionan un tanto distinto.

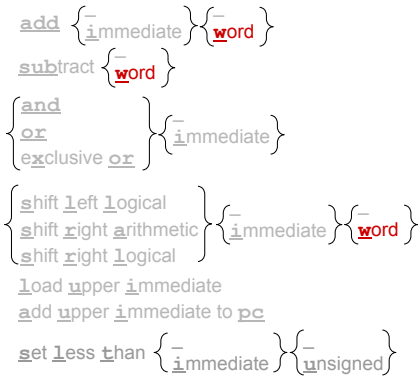
Por ejemplo, Ordenamiento por Inserción para RV64I en la Figura 9.8 se parece mucho al código para RV32I en la Figura 2.8 en la página 29 del Capítulo 2. Tiene la misma cantidad de instrucciones y ocupa lo mismo en bytes. El único cambio es que las instrucciones *load* y *store word* se vuelven *load* y *store doubleword*, y el incremento en memoria va de 4 para *words* (4 bytes) a 8 para *doublewords* (8 bytes). La Figura 9.5 muestra los opcodes de las instrucciones RV64GC en las Figuras 9.1 a 9.4.

A pesar de que RV64I usa direcciones de 64 bits y el tamaño de dato predeterminado es de 64 bits, *words* de 32 bits son tipos de datos válidos en programas. Por esto, RV64I debe soportar *words* así como RV32I debe soportar *bytes* y *halfwords*. Específicamente, dado que los registros ahora son de 64 bits, RV64I agrega versiones de 32 bits para la suma y la resta: *addw*, *addiw*, *subw*. Estas truncan el resultado a 32 bits, hacen extensión de signo y escriben el resultado al registro destino. RV64I también incluye versiones *word* para las instrucciones de corrimiento para que den el resultado de 32 bits en lugar del de 64 bits: *sllw*, *slliw*, *srlw*, *srliw*, *sraw*, *sraiw*. Para las transferencias de datos de 64 bits, usa *load* y *store doubleword*: *ld*, *sd*. Finalmente, así como hay versiones sin signo de *load byte* y *load halfword* en RV32I, RV64I debe tener una versión sin signo de *load word*: *lww*.

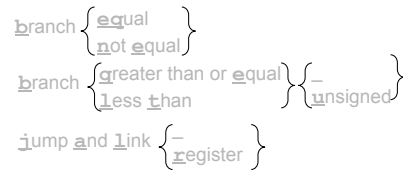
Por razones similares, RV64M debe agregar versiones *word* para multiplicación, división y residuo: *mulw*, *divw*, *divuw*, *remw*, *remuw*. Para permitir al programador sincronizar tanto *words* como *doublewords*, RV64A agrega versiones *doubleword* para todas sus 11 instrucciones.

RV64I

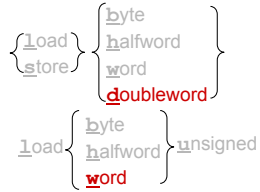
Computación de Enteros



Transferencia de Control



Loads y Stores



Instrucciones Misceláneas



Figura 9.1: Diagrama de las instrucciones RV64I. Las letras subrayadas se concatenan de izquierda a derecha para formar la instrucción RV64I. La parte atenuada son las instrucciones RV64I antiguas extendidas a registros de 64 bits y la parte oscura (rojo) son las nuevas instrucciones de RV64I.

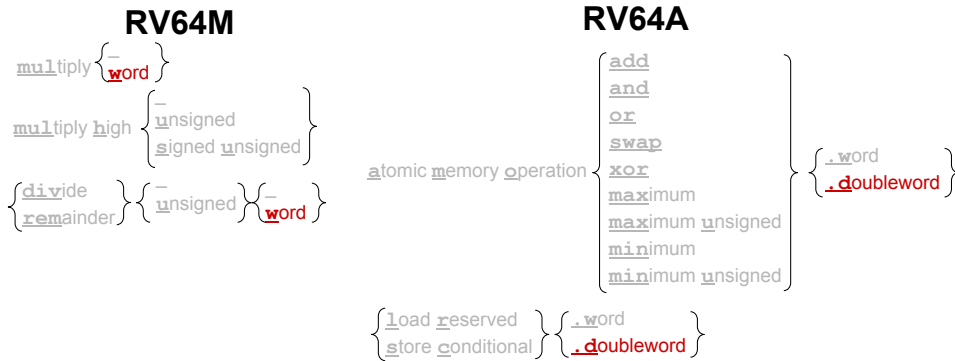


Figura 9.2: Diagrama de las instrucciones RV64M y RV64A.

RV64F y RV64D

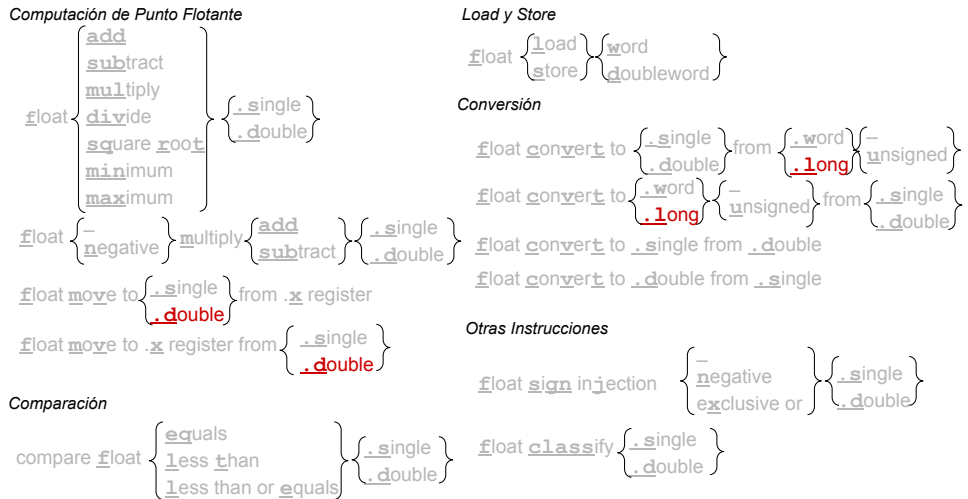


Figura 9.3: Diagrama de las instrucciones RV64F y RV64D.

RV64C

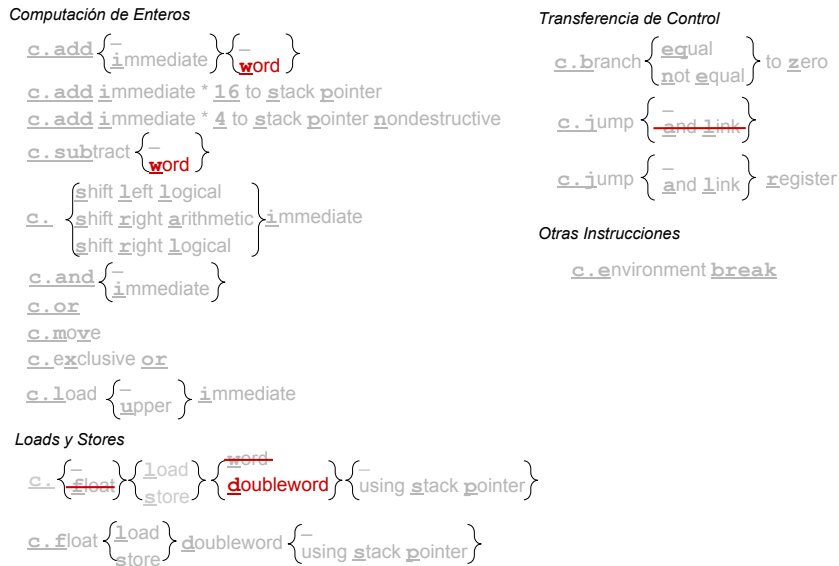


Figura 9.4: Diagrama de las instrucciones RV64C.

31	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]			rs1	110	rd	0000011		I	l	w		
imm[11:0]			rs1	011	rd	0000011		I	l	d		
imm[11:5]		rs2	rs1	011	imm[4:0]	0100011		S	s	d		
000000		shamt	rs1	001	rd	0010011		I	s	lli		
000000		shamt	rs1	101	rd	0010011		I	s	rli		
010000		shamt	rs1	101	rd	0010011		I	s	rai		
imm[11:0]			rs1	000	rd	0011011		I	a	ddi		
0000000		shamt	rs1	001	rd	0011011		I	s	lliw		
0000000		shamt	rs1	101	rd	0011011		I	s	rliw		
0100000		shamt	rs1	101	rd	0011011		I	s	raiw		
0000000		rs2	rs1	000	rd	0111011		R	a	ddw		
0100000		rs2	rs1	000	rd	0111011		R	s	ubw		
0000000		rs2	rs1	001	rd	0111011		R	s	llw		
0000000		rs2	rs1	101	rd	0111011		R	s	rlw		
0100000		rs2	rs1	101	rd	0111011		R	s	raw		

Extensión Estándar RV64M (agregado a RV32M)

0000001	rs2	rs1	000	rd	0111011		R	m	ulw		
0000001	rs2	rs1	100	rd	0111011		R	d	ivw		
0000001	rs2	rs1	101	rd	0111011		R	d	ivuw		
0000001	rs2	rs1	110	rd	0111011		R	r	emw		
0000001	rs2	rs1	111	rd	0111011		R	r	emuw		

Extensión Estándar RV64A (agregado a RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111		R	l	r.d	
00011	aq	rl	rs2	rs1	011	rd	0101111		R	s	c.d	
00001	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	swap.d
00000	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	oadd.d
00100	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	oxor.d
01100	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	oand.d
01000	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	oor.d
10000	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	omin.d
10100	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	omax.d
11000	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	ominu.d
11100	aq	rl	rs2	rs1	011	rd	0101111		R	a	m	omaxu.d

Extensión Estándar RV64F (agregado a RV32F)

1100000	00010	rs1	rm	rd	1010011		R	f	c	v	t.l.s	
1100000	00011	rs1	rm	rd	1010011		R	f	c	v	t.lu.s	
1101000	00010	rs1	rm	rd	1010011		R	f	c	v	t.s.l	
1101000	00011	rs1	rm	rd	1010011		R	f	c	v	t.s.lu	

Extensión Estándar RV64D (agregado a RV32D)

1100001	00010	rs1	rm	rd	1010011		R	f	c	v	t.l.d	
1100001	00011	rs1	rm	rd	1010011		R	f	c	v	t.lu.d	
1110001	00000	rs1	000	rd	1010011		R	f	m	v.x.d		
1101001	00010	rs1	rm	rd	1010011		R	f	c	v	t.d.l	
1101001	00011	rs1	rm	rd	1010011		R	f	c	v	t.d.lu	
1111001	00000	rs1	000	rd	1010011		R	f	m	v.d.x		

Figura 9.5: Mapa de los opcodes de las instrucciones base y extensiones opcionales de RV64. Muestra la estructura de la instrucción, opcodes, tipo de formato y nombres (La Tabla 19.2 de [Waterman and Asanović 2017] es la base para esta figura).

RV64F y RV64D agregan doublewords enteros a las instrucciones de conversión, llamándolas *longs* para prevenir confusión con datos de punto flotante de precisión doble: `fcvt.l.s`, `fcvt.l.d`, `fcvt.lu.s`, `fcvt.lu.d`, `fcvt.s.l`, `fcvt.s.lu`, `fcvt.d.l`, `fcvt.d.lu`. Como los registros enteros `x` ahora son de 64 bits, pueden almacenar datos de punto flotante de precisión doble, así que RV64D agrega dos moves de punto flotante: `fmv.x.w` y `fmv.w.x`.

La única excepción para la relación de superset entre RV64 y RV32 son las instrucciones comprimidas. RV64C reemplazó un par de instrucciones RV32C, dado que otras instrucciones reducían más el código para direcciones de 64 bits. RV64C descarta `jump and link` comprimido (`c.jal`) y las instrucciones `load` y `store word` para enteros y punto flotante (`c.lw`, `c.sw`, `c.lwsp`, `c.swsp`, `c.flw`, `c.fsw`, `c.flwsp` y `c.fswsp`). En su lugar, RV64C agrega las instrucciones más populares para sumar y restar words (`c.addw`, `c.addiw`, `c.subw`) y las instrucciones `load` y `store doubleword` (`c.ld`, `c.sd`, `c.ldsp`, `c.sdsp`).



■ **Elaboración:** *Los ABIs RV64 son `lp64`, `lp64f` y `lp64d`.*

`lp64` significa que los tipos de datos `long` y `pointer` son de 64 bits; `int` sigue siendo de 32 bits. Los sufijos `f` y `d` indican cómo se pasan los argumentos de punto flotante, lo cual es igual para RV32 (ver Capítulo 3).

■ **Elaboración:** *No hay un diagrama de instrucciones para RV64V*

porque es exactamente igual a RV32V debido al tipado dinámico de registros. La única diferencia es que los tipos dinámicos de registro `X64` y `X64U` en la Figura 8.2 de la página 79 están disponibles en RV64V pero no en RV32V.

9.2 Comparación con otros ISAs de 64 bits usando Ordenamiento por Inserción

Como Gordon Bell dijo al principio de este capítulo, el defecto fatal de una arquitectura es agotar los bits de direcciones. Conforme los programas empujaron el límite del espacio de memoria de 32 bits, los arquitectos comenzaron a hacer versiones de 64 bits de sus ISAs [Mashey 2009].

La primera fue MIPS en 1991. Extendió todos sus registros y el program counter de 32 a 64 bits y agregó nuevas versiones de 64 bits de las instrucciones MIPS-32. Todas las instrucciones de ensamblador de MIPS-64 comienzan con la letra “d”, tales como `daddu` o `dsll` (ver Figura 9.10). Los programadores pueden mezclar instrucciones de MIPS-32 y MIPS-64 en el mismo programa. MIPS-64 descartó el hueco de retardo de `load` de MIPS-32 (el pipeline se detiene si hay alguna dependencia de lectura luego de escritura).

Una década después, era hora de que `x86-32` tuviera un sucesor. Cuando los arquitectos incrementaron el tamaño de las direcciones, aprovecharon para hacerle unas mejoras a `x86-64`:

- Aumentaron la cantidad de registros enteros de 8 a 16 (`r8–r15`);
- Aumentaron la cantidad de registros SIMD de 8 a 16 (`xmm8–xmm15`); y
- Agregaron direccionamiento de datos relativo a PC para mejorar el soporte de código independiente de posición.

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

Figura 9.6: Número de instrucciones y tamaño del código de Ordenamiento por Inserción para los cuatro ISAs. ARM Thumb-2 y microMIPS son ISAs que usan direcciones de 32 bits, así que no están disponibles para ARM-64 y MIPS-64.

Estas mejoras suavizaron algunos bordes ásperos de x86-32.

Se puede apreciar los beneficios al comparar la versión x86-32 de Ordenamiento por Inserción en la Figura 2.11, página 32, Capítulo 2 con la versión x86-64 en la Figura 9.11. El nuevo ISA mantiene todas las variables en registros en lugar de almacenar algunas en memoria, lo cual reduce las instrucciones de 20 a 15. A pesar de tener menos instrucciones, el tamaño del código es de hecho un byte más grande: 46 versus 45. La razón es porque para encajar los nuevos opcodes y habilitar más registros, x86-64 agregó un byte de prefijo para identificar las nuevas instrucciones. La longitud promedio de instrucciones de x86-64 incrementó comparado con x86-32.

ARM se enfrentó al mismo problema una década después. En lugar de evolucionar el ISA para que soportara direcciones de 64 bits como lo hizo x86-64, aprovecharon esta oportunidad para inventar un ISA nuevo. El comenzar desde cero les permitió cambiar las cosas raras de ARM-32 y generar un ISA moderno:

- Incrementar la cantidad de registros enteros de 15 a 31;
- Quitar el PC del conjunto de registros;
- Proveer un registro que está alambrado a cero para la mayoría de las instrucciones (r31);
- A diferencia de ARM-32, todos los modos de direccionamiento de ARM-64 funcionan con todos los tamaños y tipos de datos;
- ARM-64 eliminó las instrucciones de load y store múltiple de ARM-32; y
- ARM-64 omitió la opción de ejecución condicional de instrucciones de ARM-32.

Aún comparte algunas de las debilidades de ARM-32: códigos de condición para branches, los campos para indicar el registro origen y destino cambian en el formato de instrucciones, instrucciones de movimiento condicionales, modos de direccionamiento complejos, indicadores de rendimiento inconsistentes y únicamente instrucciones de 32 bits. ARM-64 no puede cambiar al ISA Thumb-2, dado que Thumb-2 solo funciona con direcciones de 32 bits.

A diferencia de RISC-V, ARM tomó un enfoque maximalista al diseño del ISA. Definitivamente un mejor ISA que ARM-32, pero también es más grande. Por ejemplo, tiene más de 1000 instrucciones y el manual de ARM-64 es de 3185 páginas [ARM 2015]. Es más, sigue creciendo. Ha habido tres expansiones de ARM-64 desde que fue anunciado hace unos años.

El código de ARM-64 para Ordenamiento por Inserción en la Figura 9.9 se parece más al código de RV64I o x86-64 que al código de ARM-32. Por ejemplo, con 31 registros, no hay necesidad de guardar y restaurar registros del stack. Y dado que el PC ya no es uno de los registros, ARM-64 usa una instrucción especial de retorno.

La Figura 9.6 es una tabla que resume el número de instrucciones y cantidad de bytes en Ordenamiento por Inserción para los ISAs. Las Figuras 9.8 a 9.11 muestran el código compi-



Rendimiento



Programabilidad

Intel no inventó el ISA x86-64. En la migración a direcciones de 64 bits, Intel inventó un nuevo ISA llamado Itanium que era incompatible con x86-32. AMD, su mayor competidor en procesadores x86-32 no tenía acceso al ISA de Itanium, por lo que AMD inventó una versión de 64 bits de x86-32 llamada AMD64. Itanium eventualmente fracasó, por lo que Intel se vio obligado a adoptar el ISA AMD64 como el sucesor de 64 bits de x86-32, el cual llamamos x86-64 [Kerner and Padgett 2007].

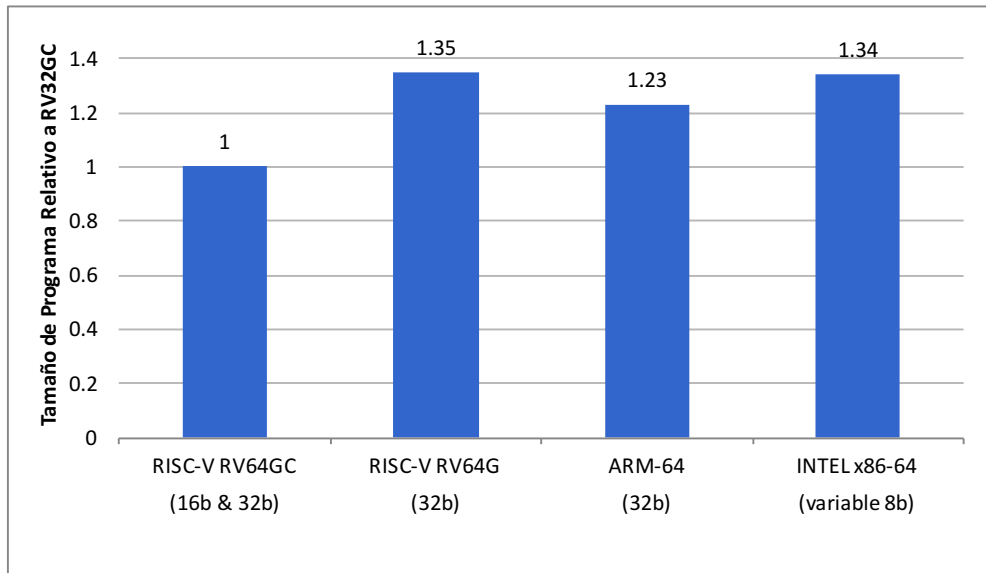


Figura 9.7: Tamaños del programa relativos para RV64G, ARM-64 y x86-64 versus RV64GC. Esta comparación mide programas mucho mayores que los de la Figura 9.6. Esta gráfica es el equivalente en direcciones de 64 bits a la gráfica para ISAs de 32 bits en la Figura 1.5, página 9, Capítulo 2. El tamaño del código de RV32C es casi igual al de RV64C; es 1% más pequeño. No hay opción de Thumb-2 para ARM-64, por lo que la base de otros ISAs de 64 bits exceden significativamente el tamaño del código de RV64GC. Los programas medidos son la prueba de rendimiento SPEC CPU2006 usando compiladores GCC [Waterman 2016].

lado para RV64I, ARM-64, MIPS-64 y x86-64. Frases entre paréntesis en los comentarios de estos cuatro programas identifican las diferencias entre las versiones RV32I en el Capítulo 2 y estas versiones RV64I.

MIPS-64 necesita la mayor cantidad de instrucciones, principalmente por las instrucciones `nop` de los huecos de branch retardado. RV64I usa menos gracias a las instrucciones `compare-and-branch` y por no usar branches retardados. A pesar de que ARM-64 y x86-64 usa dos instrucciones de comparación que son innecesarias en RV64I, sus modos de direccionamiento escalado evitan las instrucciones aritméticas usadas en RV64I, dándoles la menor cantidad de instrucciones. Sin embargo, RV64I+RV64C tienen un tamaño de código mucho menor, como lo indica la siguiente sección.

■ **Elaboración:** *ARM-64, MIPS-64 y x86-64 no son los nombres oficiales.*

Los nombres oficiales son: ARMv8 es a lo que llamamos ARM-64, MIPS-IV es MIPS-64 y AMD64 es x86-64 (ver la nota al margen arriba para conocer la historia de x86-64).

9.3 Tamaño del Programa

La Figura 9.7 compara tamaños de código promedio relativos para RV64, ARM-64 y x86-64. Compare esta figura con la Figura 1.5 en la página 9, Capítulo 1. Primero, el código de RV32GC es casi idéntico en tamaño a RV64GC; solo es 1% más pequeño. Dicha cercanía también aplica para RV32I y RV64I. A pesar de que el código de ARM-64 es 8% más pequeño que el de ARM-32, no hay versión de 64 bits de Thumb-2, por lo que todas las instrucciones permanecen de 32 bits. Por lo tanto, el código de ARM-64 es un 25% *más grande* que el de ARM Thumb-2. El código para x86-64 es 7% más grande que el de x86-32 dado que se agregó opcodes de prefijo para que x86-64 acomodara nuevas operaciones y el conjunto extendido de registros. RV64GC gana, dado que el código de ARM-64 es 23% más grande que RV64GC y el código de x86-64 es 34% más grande que RV64GC. Esa diferencia es lo suficientemente grande para ya sea mejorar el rendimiento por tener menos cache misses de instrucciones, o reducir el costo al permitir caches de instrucciones más pequeños y aún proveer tasas de miss satisfactorias.

9.4 Observaciones Finales

Uno de los problemas de ser pionero es que uno siempre comete errores, y yo nunca, nunca quiero ser un pionero. Siempre es mejor llegar de segundo, donde puedes ver los errores cometidos por los pioneros.

—Seymour Cray, arquitecto de la primera supercomputadora, 1976

Quedarse sin bits de direcciones es el talón de Aquiles en arquitectura de computadoras. Muchas arquitecturas han muerto por una herida de esas. ARM-32 y Thumb-2 siguen siendo arquitecturas de 32 bits, así que no sirven para programas grandes. Algunos ISAs como MIPS-64 y x86-64 sobrevivieron la transición, pero x86-64 no es un modelo de diseño de ISA y el futuro de MIPS-64 es incierto en este momento. ARM-64 es un nuevo ISA grande, y el tiempo dirá qué tan exitoso será.

RISC-V se benefició diseñando conjuntamente las arquitecturas de 32 y 64 bits, mientras que ISAs anteriores tuvieron que diseñarlas secuencialmente. Naturalmente, la transición entre 32 y 64 bits es más fácil para escritores de compiladores y programadores de RISC-V; el ISA RV64I tiene virtualmente todas las instrucciones RV32I. De hecho, es por eso que podemos listar RV32GCV y RV64GCV en solo dos páginas de la Tarjeta de Referencia. Aún más importante, el diseño simultáneo significó que la arquitectura de 64 bits no tuvo que empotrarse en un espacio de opcodes de 32 bits saturado. RV64I tiene suficiente espacio para extensiones opcionales, particularmente RV64C, haciéndolo el líder en tamaño del código.

Vemos la arquitectura de 64 bits como mayor evidencia del diseño sólido de RISC-V, evidentemente más fácil de lograr si se comienza 20 años después, así se puede usar las buenas ideas de los pioneros así como aprender de sus errores.



Tamaño del Programa



Rendimiento



Costo

MIPS tiene su tercer dueño. Imagination Technologies, quien compró el ISA de MIPS en el 2012 por \$100M, vendió su división de MIPS a Tallwood Venture Capital en el 2017 por \$65M.



Programabilidad



Espacio para Crecer



Tamaño del Programa



Elegancia

■ Elaboración: RV128

RV128 comenzó como una broma interna entre arquitectos de RISC-V, simplemente para demostrar que un ISA con direcciones de 128 bits era posible. Sin embargo, computadoras a escala de bodega podrán pronto tener más de 2^{64} bytes de almacenamiento basado en semiconductores (DRAM y Memoria Flash), al cual probablemente los programadores quisieran acceder como direcciones de memoria. También hay propuestas en usar direcciones de 128 bits para mejorar la seguridad [Woodruff et al. 2014]. El manual de RISC-V especifica un ISA de 128 bits llamado RV128G [Waterman and Asanović 2017]. Las instrucciones adicionales son básicamente las mismas que las que se usaron para ir de RV32 a RV64, ilustradas en las Figuras 9.1 a 9.4. Todos los registros también crecen a 128 bits, y las nuevas instrucciones RV128 especifican ya sea versiones de 128 bits de algunas instrucciones (usando la Q en su nombre por quadword) o versiones de 64 bits de otras (usando D en su nombre por doubleword).

9.5 Para Aprender Más

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 instrucciones, 76 bytes, o 52 bytes con RV64C)
# a1 es n, a3 apunta a a[0], a4 es i, a5 es j, a6 es x
0: 00850693 addi a3,a0,8 # (8 vs 4) a3 apunta a a[i]
4: 00100713 li a4,1 # i = 1
Outer Loop:
8: 00b76463 bltu a4,a1,10 # si i < n, saltar a Continue Outer loop
Exit Outer Loop:
c: 00008067 ret # retornar de la función
Continue Outer Loop:
10: 0006b803 ld a6,0(a3) # (ld vs lw) x = a[i]
14: 00068613 mv a2,a3 # a2 apunta a a[j]
18: 00070793 mv a5,a4 # j = i
Inner Loop:
1c: ff863883 ld a7,-8(a2) # (ld vs lw, 8 vs 4) a7 = a[j-1]
20: 01185a63 ble a7,a6,34 # si a[j-1] <= a[i], saltar a Exit Inner Loop
24: 01163023 sd a7,0(a2) # (sd vs sw) a[j] = a[j-1]
28: fff78793 addi a5,a5,-1 # j--
2c: ff860613 addi a2,a2,-8 # (8 vs 4) decrementar a2 para apuntar a a[j]
30: fe0796e3 bnez a5,1c # si j != 0, saltar a Inner Loop
Exit Inner Loop:
34: 00379793 slli a5,a5,0x3 # (8 vs 4) multiplicar a5 por 8
38: 00f507b3 add a5,a0,a5 # a5 es ahora el byte address 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 # incrementar a3 para apuntar a a[i]
48: fc1ff06f j 8 # saltar a Outer Loop

```

Figura 9.8: Código RV64I para Ordenamiento por Inserción en la Figura 2.5. El programa en ensamblador para RV64I es muy similar al programa para RV32I en la Figura 2.8, página 29, Capítulo 2. Mostramos las diferencias entre paréntesis en los comentarios. El tamaño de los datos es de 8 bytes en lugar de 4, por lo que tres instrucciones cambian la constante de 4 a 8. Este ancho adicional también cambia dos load words (*lw*) a load doublewords (*ld*) y dos store words (*sw*) a store doublewords (*sd*).

```

# ARM-64 (16 instrucciones, 64 bytes)
# x0 apunta a a[0], x1 es n, x2 es j, x3 es i, x4 es x
  0: d2800023  mov  x3, #0x1          # i = 1
Outer Loop:
  4: eb01007f  cmp  x3, x1          # comparar i vs n
  8: 54000043  b.cc 10             # si i < n, saltar a Continue Outer loop
Exit Outer Loop:
  c: d65f03c0  ret                # retornar de la función
Continue Outer Loop:
 10: f8637804  ldr  x4, [x0, x3, lsl #3] # (x4 ca r4) vs x = a[i]
 14: aa0303e2  mov  x2, x3         # (x2 vs r2) j = i
Inner Loop:
 18: 8b020c05  add  x5, x0, x2, lsl #3 # x5 apunta a a[j]
1c: f85f80a5  ldur x5, [x5, #-8]   # x5 = a[j]
20: eb0400bf  cmp  x5, x4         # comparar a[j-1] vs. x
24: 5400008d  b.le 34            # si a[j-1]<=a[i], saltar a Exit Inner Loop

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

```

Figura 9.9: Código de ARM-64 para Ordenamiento por Inserción en la Figura 2.5. El programa en ensamblador de ARM-64 es diferente al programa de ARM-32 en la Figura 2.11, página 32, Capítulo 2 dado que es un set de instrucciones nuevo. Los registros comienzan con una x en vez de una a. Los modos de direccionamiento de datos pueden hacer un corrimiento de 3 a los registros para escalar el índice a direcciones por bytes. Con 31 registros, no hay necesidad de guardar y restaurar registros del stack. Dado que el PC no es uno de los registros, utiliza una instrucción especial para retornar. De hecho, el código se parece más al de RV64I o x86-64 que al código de ARM-32.

```

# MIPS-64 (24 instrucciones, 96 bytes)
# a1 es n, a3 apunta a a[0], v0 es j, v1 es i, t0 es x
0: 64860008 daddiu a2,a0,8 # (daddiu vs addiu, 8 vs 4) a2 apunta a a[i]
4: 24030001 li    v1,1    # i = 1
Outer Loop:
8: 0065102b sltu   v0,v1,a1 # poner en 1 cuando i < n
c: 14400003 bnez   v0,1c    # si i < n, saltar a Continue Outer Loop
10: 00c03825 move  a3,a2    # a3 apunta a a[j] (hueco lleno)
14: 03e00008 jr    ra      # retornar de la función
18: 00000000 nop                    # hueco de retardo de branch vacío
Continue Outer Loop:
1c: dcc80000 ld    a4,0(a2) # (ld vs lw) x = a[i]
20: 00601025 move  v0,v1    # j = i
Inner Loop:
24: dce9fff8 ld    a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
28: 0109502a slt   a6,a4,a5 # (sin hueco de retardo de load) set a[i] < a[j-1]
2c: 11400005 beqz  a6,44    # si a[j-1] <= a[i], saltar a Exit Inner Loop
30: 00000000 nop                    # hueco de retardo de branch vacío
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    # si j != 0, saltar a Inner Loop (próximo hueco lleno)
40: 64e7fff8 daddiu a3,a3,-8 # (daddiu vs addiu, 8 vs 4) decr puntero a2 a a[j]
Exit Inner Loop:
44: 000210f8 dsll  v0,v0,0x3 # (dsll vs sll)
48: 0082102d daddu  v0,a0,v0 # (daddu vs addu) v0 ahora byte address 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      # saltar a Outer Loop (próximo hueco de retardo lleno)
58: 64c60008 daddiu a2,a2,8  # (daddiu vs addiu, 8 vs 4) incr puntero a2 a a[i]
5c: 00000000 nop                    # Innecesario(?)

```

Figura 9.10: Código de MIPS-64 para Ordenamiento por Inserción en la Figura 2.5. El programa en ensamblador de MIPS-64 tiene varias diferencias con respecto al programa de MIPS-32 en la Figura 2.10, página 31, Capítulo 2. Primero, la mayoría de instrucciones de 64 bits anteponen una “d” a sus nombres: daddiu, daddu, dsll. Como en la Figura 9.8, tres instrucciones cambian la constante de 4 a 8 dado que el tamaño de los datos creció de 4 a 8 bytes. Al igual que RV64I, el ancho adicional cambia dos load words (lw) a load doublewords (ld) y dos store words (sw) a store doublewords (sd). Finalmente, MIPS-64 no tiene el hueco de retardo de load de MIPS-32; el pipeline se detiene luego de una dependencia leer luego de escribir.

```

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

```

Figura 9.11: Código x86-64 para Ordenamiento por Inserción en la Figura 2.5. El programa en ensamblador de x86-64 es bastante distinto al programa de x86-32 en la Figura 2.11, página 32, Capítulo 2. Primero, a diferencia de RV64I, los registros más anchos tienen nombres distintos `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`. Segundo, dado que x86-64 agregó 8 registros nuevos, ahora hay suficientes para mantener todas las variables en registros en vez de memoria. Tercero, las instrucciones de x86-64 son más largas que las de x86-32 debido a que debieron anteponer 8 ó 16 bits para encajar las nuevas instrucciones en el espacio de opcodes. Por ejemplo, incrementar o decrementar un registro (`inc`, `dec`) ocupa 1 byte en x86-32 pero 3 bytes en x86-64. Por lo tanto, aunque tenga menos instrucciones, el tamaño del código de x86-64 para Ordenamiento por Inserción es casi idéntico al de x86-32: 45 bytes vs. 46 bytes.

Arquitectura Privilegiada RV32/64

Edsger W. Dijkstra

(1930–2002) recibió el Premio Turing en 1972 por contribuciones fundamentales en el desarrollo de lenguajes de programación.



La simpleza es un prerrequisito para la fiabilidad.

—Edsger W. Dijkstra

10.1 Introducción

Hasta ahora el libro se ha enfocado en el soporte de RISC-V para computación de propósito general: todas las instrucciones que hemos introducido están disponibles en el *modo usuario* (en inglés, *user mode*), donde usualmente corre el código de aplicaciones. Este capítulo introduce dos nuevos modos de *privilegio*: *modo máquina* (en inglés *machine mode*), que ejecuta el código más fiable, y *modo supervisor* (en inglés, *supervisor mode*), que provee soporte para sistemas operativos como Linux, FreeBSD y Windows. Ambos modos nuevos son más privilegiados que el modo usuario, de ahí el título de este capítulo. Los modos más privilegiados generalmente tienen acceso a todas las características de los modos menos privilegiados, y agregan funcionalidad adicional no disponible en los modos menos privilegiados, tales como la habilidad de manejar interrupciones y controlar I/O. Los procesadores típicamente pasan la mayor parte de su tiempo de ejecución en su modo menos privilegiado; interrupciones y excepciones transfieren el control a modos más privilegiados.

Sistemas operativos y entornos en tiempo de ejecución para sistemas embebidos emplean las características de estos nuevos modos para responder a eventos externos, como la llegada de paquetes de red; para soportar *multitasking* y protección entre tareas; y para abstraer y virtualizar características de hardware. Dada la amplitud de estos temas, una guía exhaustiva del programador sería un libro adicional completo; en lugar de eso, ese capítulo busca exponer las características más importantes de RISC-V. Programadores desinteresados en sistemas operativos y entornos en tiempo de ejecución para sistemas embebidos pueden omitir

Instrucciones Privilegiadas RV32/64

```

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

Figura 10.1: Diagrama de las instrucciones privilegiadas 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: Estructura de instrucciones privilegiadas RISC-V, opcodes, tipo de formato y nombre (La Tabla 6.1 de [Waterman and Asanović 2017] es la base de esta figura).

u hojear este capítulo.

La Figura 10.1 es una representación gráfica de las instrucciones privilegiadas RISC-V, y la Figura 10.2 lista los opcodes de estas instrucciones. Como se puede ver, la arquitectura privilegiada agrega muy pocas instrucciones; en su lugar, varios CSRs (Control and Status Registers: Registros de Control y Estado) nuevos exponen la funcionalidad adicional.

Este capítulo describe las arquitecturas privilegiadas RV32 y RV64 juntas. Algunos conceptos difieren solo en el tamaño de un registro entero, así que para mantener las descripciones concisas, introducimos el término XLEN para referirnos al ancho de un registro entero en bits. XLEN es 32 para RV32 o 64 para RV64.



Simplicidad

10.2 Modo Máquina para Sistemas Embebidos Simples

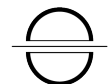
El modo máquina, abreviado como modo M, es el modo más privilegiado en el que un Hart (Hardware thread: Hilo de ejecución en hardware) de RISC-V puede ser ejecutado. Los harts que corren en modo M tienen acceso completo a la memoria, I/O, y funcionalidades del sistema de bajo nivel necesarias para arrancar y configurar el sistema. Como tal, es el único modo de privilegio que implementan todos los procesadores RISC-V estándar; en efecto, los microcontroladores simples RISC-V *solo* soportan el modo M. Tales sistemas son el enfoque de esta sección.

La característica más importante del modo máquina es la habilidad de interceptar y manejar *excepciones*: eventos inusuales en tiempo de ejecución. RISC-V clasifica las excepciones en dos categorías. Las *excepciones sincrónicas* ocurren como resultado de ejecución de instrucciones, como cuando se intenta acceder a direcciones inválidas de memoria o ejecutar una instrucción con un opcode inválido. Las *interrupciones* son eventos externos asíncronos al flujo de instrucciones, como un clic del botón del *mouse*. Las excepciones en RISC-V son *precisas*: todas las instrucciones previas a la excepción son ejecutadas completamente, y ninguna de las instrucciones subsiguientes aparentan haber iniciado su ejecución. La Figura 10.3 lista las causas estándar de excepciones.

Cinco tipos de excepciones sincrónicas pueden ocurrir durante la ejecución en modo M:

- Las *excepciones de falla de acceso* ocurren cuando una dirección física de memoria no soporta el tipo de acceso—por ejemplo, intentar escribir en ROM.
- Las *excepciones de breakpoint* ocurren al ejecutar una instrucción `ebreak`, o cuando una dirección o dato coincide con un *debug trigger*.
- Las *excepciones de llamadas al entorno* ocurren al ejecutar una instrucción `ecall`.
- Las *excepciones de instrucción ilegal* son el resultado de decodificar un opcode inválido.

Hart es una contracción de hardware thread. Usamos el término para distinguirlos de threads de software, que son conocidos por la mayoría de los programadores. Los threads de software son multiplexados en tiempo sobre harts. La mayoría de procesadores solo tienen un hart.



Aislamiento de Arq e Impl

Interrupción / Excepción mcause[XLEN-1]	Código de Excepción mcause[XLEN-2:0]	Descripción
1	1	Interrupción de software de Supervisor
1	3	Interrupción de software de Máquina
1	5	Interrupción de temporizador de Supervisor
1	7	Interrupción de temporizador de Máquina
1	9	Interrupción externa de Supervisor
1	11	Interrupción externa de Máquina
0	0	Dirección de instrucción desalineada
0	1	Fallo de acceso en instrucción
0	2	Instrucción ilegal
0	3	Breakpoint
0	4	Dirección de Load desalineada
0	5	Fallo de acceso en Load
0	6	Dirección de Store desalineada
0	7	Fallo de acceso en Store
0	8	Llamada al entorno desde modo U
0	9	Llamada al entorno desde modo S
0	11	Llamada al entorno desde modo M
0	12	Fallo de página en instrucción
0	13	Fallo de página en Load
0	15	Fallo de página en Store

Figura 10.3: Causas de excepciones e interrupciones en RISC-V. El bit más significativo de mcause es puesto en 1 para interrupciones o en 0 para excepciones síncronas, y los bits menos significativos identifican la interrupción o excepción. Interrupciones de supervisor y excepciones de fallo de página son posibles solo cuando el modo supervisor está implementado (ver Sección 10.5) (La Tabla 3.6 of [Waterman and Asanović 2017] es la base de esta figura).

- *Las excepciones de dirección de instrucción desalineada* ocurren cuando la dirección efectiva no es divisible por el tamaño de acceso—por ejemplo, `amoadd.w` con una dirección de `0x12`.

Si recuerda, el Capítulo 2 afirma que loads y stores desalineados son permitidos, podría preguntarse por qué se lista excepciones de loads y stores desalineados en la Figura 10.3. Hay dos razones para esto. Primero, las operaciones atómicas de memoria del Capítulo 6 requieren direcciones alineadas naturalmente. Segundo, algunos fabricantes eligen omitir soporte en hardware para loads y stores regulares desalineados, porque es una función difícil de implementar y no es frecuentemente utilizada. Procesadores sin este hardware dependen de un manejador de excepciones para tomar el control y emular loads y stores desalineados por software, usando una secuencia de loads y stores alineados más pequeños. El código de aplicación no es más inteligente: los accesos a memoria desalineados operan como se espera, aunque lentamente, mientras el hardware permanece simple. Alternativamente, los procesadores de mayor rendimiento pueden implementar loads y stores desalineados en hardware. Esta flexibilidad de implementación se debe a la decisión de RISC-V de permitir loads y stores desalineados usando los opcodes regulares de load y store, siguiendo la pauta del Capítulo 1 de aislar la arquitectura de la implementación.

Hay tres fuentes estándar de interrupciones: de software, de temporizador y externas. Las interrupciones de software son disparadas al escribir a un registro mapeado en memoria y son generalmente usadas por un hart para interrumpir a otro hart, un mecanismo que otras arquitecturas llaman una *interrupción interprocesador*. Las interrupciones de temporizador son levantadas cuando el comparador de tiempo de un hart, un registro mapeado en memoria llamado `mtimecmp`, iguala o excede al contador de tiempo real `mtime`. Las interrupciones externas son levantadas por un controlador de interrupciones a nivel de plataforma, al cual la mayoría de los dispositivos externos están conectados. Ya que diferentes plataformas de hardware tienen diferentes mapas de memoria y demandan características divergentes de sus controladores de interrupciones, los mecanismos para levantar y borrar estas interrupciones difieren de plataforma en plataforma. Lo que *sí* es constante entre todos los sistemas RISC-V es cómo las excepciones son manejadas y las interrupciones son enmascaradas, el tema de la próxima sección.

10.3 Manejo de Excepciones en Modo Máquina

Ocho registros de control y estado (CSRs) son integrales para el manejo de excepciones en modo máquina:

- `mstatus`, *Estado de Máquina*, contiene el habilitador global de interrupciones, junto con un sinnúmero de otros estados, como muestra la Figura 10.4.
- `mip`, *Interrupciones de Máquina Pendientes*, lista las interrupciones actualmente pendientes (Figura 10.5).
- `mie`, *Habilitador de Interrupciones de Máquina*, lista cuáles interrupciones puede tomar el procesador y cuáles debe ignorar (Figura 10.5).
- `mcause`, *Causa de Excepción de Máquina*, indica cuál excepción ocurrió (Figura 10.6).
- `mtvec`, *Vector de Interrupciones de Máquina*, contiene la dirección a la cual salta el procesador cuando ocurre una excepción (Figura 10.7).

Excepciones de dirección de instrucción desalineada no pueden ocurrir con la extensión C porque nunca sería posible saltar a una dirección impar: los branches y JAL inmediatos siempre son pares, y JALR enmascara a cero el bit menos significativo de su dirección efectiva. Sin la extensión C, esta excepción ocurre cuando se salta a una dirección igual a $2 \bmod 4$.



Aislamiento de Arq e Impl

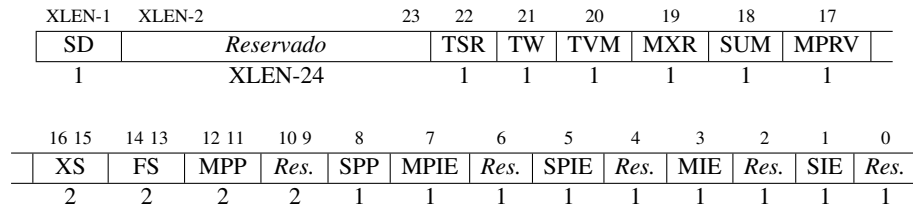


Figura 10.4: El CSR `mstatus`. Los únicos campos presentes en procesadores simples sólo con modo Máquina y sin las extensiones F ni V son el habilitador global de interrupciones, MIE, y MPIE, el cual después de una excepción contiene el valor anterior de MIE. XLEN es 32 para RV32, o 64 para RV64. La Figura 3.7 de [Waterman and Asanović 2017] es la base de esta figura; ver Sección 3.1 de ese documento para una descripción de los otros campos.

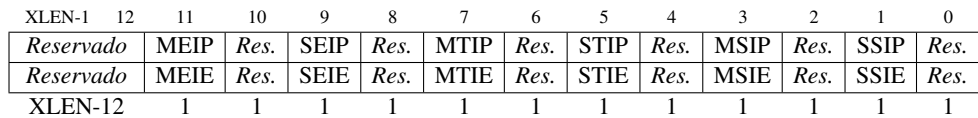


Figura 10.5: CSRs de interrupciones de máquina. Son registros de XLEN-bits de lectura y escritura que contienen los bits de interrupciones pendientes (`mip`) y habilitadores de interrupciones (`mie`). Solo es posible escribir en los bits correspondientes a las interrupciones del menor privilegio (SSIP), interrupciones de temporizador (STIP) e interrupciones externas (SEIP) en `mip` por medio de esta dirección CSR; el resto de los bits son de solo lectura.

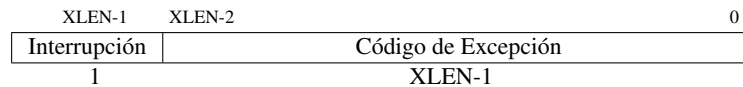


Figura 10.6: CSRs de causa para máquina y supervisor (`mcause` y `scause`). Cuando se atiende una excepción, se escribe al CSR un código que indica el evento que causó la excepción. El bit de Interrupción se pone en uno si la excepción fue causada por una interrupción. El campo de Código de Excepción contiene un código que identifica la última excepción. La Figura 10.3 mapea los valores de los códigos a las razones de las excepciones.

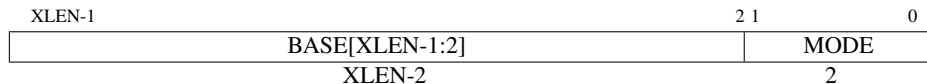


Figura 10.7: CSRs de direcciones base de vectores de excepciones para máquina y supervisor (`mtvec` y `stvec`). Son registros de XLEN-bits de lectura y escritura que contienen la configuración de vectores de excepciones, que consiste de una dirección base del vector (BASE) y un modo del vector (MODE). El valor del campo BASE siempre debe estar alineado en una frontera de 4 bytes. MODE = 0 significa que todas las excepciones escriben BASE en el PC. MODE = 1 escribe ($BASE + (4 \times causa)$) en el PC en interrupciones asíncronas.

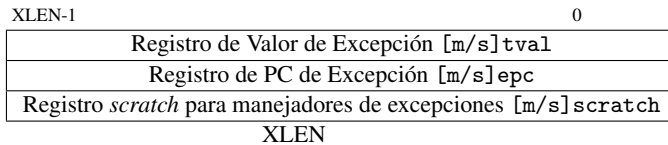


Figura 10.8: CSRs asociados con excepciones e interrupciones. Los registros de Valor de Excepción (*mtval* y *stval*) contienen información adicional útil tal como la dirección defectuosa de una instrucción ilegal. Los PCs de Excepción (*mepc* y *sepc*) apuntan a la instrucción defectuosa. Los registros *scratch* (*mscratch* y *sscratch*) dan a los manejadores de excepciones un registro libre para su uso.

Codificación	Nombre	Abreviación
00	Usuario	U
01	Supervisor	S
11	Máquina	M

Figura 10.9: Niveles de privilegio de RISC-V y su codificación.

- *mtval*, *Valor de Excepción de Máquina*, contiene información adicional de excepciones: la dirección defectuosa para excepciones de direcciones, la instrucción misma para excepciones de instrucción ilegal, y cero para otras excepciones (Figura 10.8).
- *mepc*, *PC de Excepción de Máquina*, apunta a la instrucción donde ocurrió la excepción (Figura 10.8).
- *mscratch*, *Scratch de Máquina*, contiene una palabra de datos para almacenamiento temporal de manejadores de excepciones (Figura 10.8).

Cuando se ejecuta en modo M, las interrupciones solo se toman si el bit de habilitación de interrupciones global, *mstatus.MIE*, es 1. Además, cada interrupción tiene su propio bit de habilitación en el CSR *mie*. Las posiciones de los bits en *mie* corresponden a los códigos de interrupción en la Figura 10.3: por ejemplo, *mie[7]* corresponde a la interrupción de temporizador en modo M. El CSR *mip* tiene la misma estructura e indica cuáles interrupciones se encuentran pendientes. Juntando los tres CSRs, una interrupción de temporizador de máquina puede tomarse si *mstatus.MIE*=1, *mie[7]*=1 y *mip[7]*=1.

Cuando un hart toma una excepción, el hardware atómicamente pasa por varias transiciones de estado:

- El PC de la instrucción que causó la excepción es preservado en *mepc*, y *mtvec* es escrito al PC (Para excepciones síncronas, *mepc* apunta a la instrucción que causó la excepción; para interrupciones, apunta a donde la ejecución debe reanudarse después de que la interrupción sea manejada).
- *mcause* recibe la causa de la excepción, como se codifica en la Figura 10.3, y *mtval* recibe la dirección defectuosa o alguna otra palabra de información específica de la excepción.
- Las interrupciones son deshabilitadas escribiendo *MIE*=0 en el CSR *mstatus*, y el valor previo de *MIE* es preservado en *MPIE*.

RISC-V también soporta interrupciones vectorizadas, en donde el procesador salta a una dirección específica de la interrupción, en lugar de saltar a una única rutina. Este direccionamiento elimina la necesidad de leer y decodificar *mcause*, agilizando el manejo de interrupciones. Escribir 1 a *mtval[0]* habilita esta funcionalidad; la causa *x* para una interrupción entonces escribe (*mtval*-1+4*x*) al PC, en lugar del usual *mtval*.

- El modo de privilegio pre-excepción es preservado en el campo MPP de `mstatus`, y el modo de privilegio es cambiado a M. La Figura 10.9 muestra la codificación del campo MPP (Si el procesador solo implementa el modo M, este paso es efectivamente omitido).

Para evitar sobrescribir el contenido de los registros enteros, el prólogo de un manejador de interrupción usualmente comienza intercambiando un registro entero (digamos, `a0`) con el CSR `mscratch`. Usualmente, el software habrá preparado `mscratch` para contener un puntero a un espacio adicional en memoria, que el manejador emplea para guardar tantos registros como su cuerpo use. Después de la ejecución del cuerpo del manejador, su epílogo restaura los registros que guardó en memoria, luego intercambia nuevamente `a0` con `mscratch`, restaurando ambos registros a sus valores pre-excepción. Finalmente, el manejador retorna con `mret`, una instrucción única al modo M. `mret` escribe `mepc` al PC, restaura la configuración previa de habilitación de interrupciones copiando el campo MPIE de `mstatus` a MIE, y escribe el modo de privilegio al valor en el campo MPP de `mstatus`, esencialmente revirtiendo las acciones descritas en el párrafo anterior.

La Figura 10.10 muestra el código en ensamblador RISC-V para un manejador básico de interrupción de temporizador siguiendo este patrón. Simplemente incrementa el comparador de tiempo y luego retorna a la tarea anterior, mientras que un manejador de interrupción de temporizador más realista podría invocar un calendarizador para cambiar de tarea. No es *preemptive*¹, así que mantiene las interrupciones deshabilitadas a lo largo del manejador. ¡Aparte de esas salvedades, es un ejemplo completo de un manejador de interrupciones RISC-V en una sola página!

En ocasiones es deseable tomar una interrupción de mayor prioridad mientras se procesa una excepción de menor prioridad. Por desgracia, solo hay una copia de los CSRs `mepc`, `mcause`, `mtval` y `mstatus`; tomar una segunda interrupción destruiría los valores antiguos en estos registros, causando una pérdida de información si no se contara con alguna ayuda adicional del software. Un manejador de interrupciones *preemptive* puede guardar estos registros a un stack en memoria antes de habilitar interrupciones, luego, justo antes de salir, deshabilitar interrupciones y restaurar los registros del stack.

Además de la instrucción `mret` que presentamos arriba, el modo M solo provee una instrucción más: `wfi` (Esperar Interrupción²). `wfi` informa al procesador que no hay trabajo útil por hacer, así que debe entrar en un modo de bajo consumo hasta que cualquier interrupción habilitada se vuelva pendiente, i.e., $(mie \& mip) \neq 0$. Los procesadores RISC-V implementan esta instrucción en una variedad de maneras, incluyendo detener el reloj hasta que una interrupción pase a pendiente; algunos simplemente la ejecutan como un `nop`. Por eso, `wfi` es típicamente empleado dentro de un ciclo.

■ **Elaboración:** *wfi funciona sin importar si las interrupciones se encuentran globalmente habilitadas o no.*

Si `wfi` es ejecutado con interrupciones globalmente habilitadas (`mstatus.MIE=1`), y luego una interrupción habilitada pasa a pendiente, el el procesador salta al manejador de la excepción. Si, por el otro lado, `wfi` es ejecutado con interrupciones globalmente deshabilitadas, y luego una interrupción habilitada pasa a pendiente, el procesador continúa ejecutando el código que sigue a `wfi`. Este código típicamente examina el CSR `mip` para decidir qué hacer en adelante. Esta estrategia puede reducir la latencia de interrupciones en contraste con saltar al manejador de la excepción, porque no hay necesidad de guardar ni restaurar registros enteros.



Simplicidad




```

# guardar registros
csrrw a0, mscratch, a0 # guardar a0; a0 = &almacenamiento temp
sw a1, 0(a0)           # guardar a1
sw a2, 4(a0)           # guardar a2
sw a3, 8(a0)           # guardar a3
sw a4, 12(a0)          # guardar a4

# decodificar la causa de la interrupción
csrr a1, mcause        # leer la causa de la excepción
bgez a1, exception     # branch si no es una interrupción
andi a1, a1, 0x3f      # aislar la causa de la interrupción
li a2, 7                # a2 = causa: interrupción de temporizador
bne a1, a2, otherInt   # branch si no es una interrupción de temporizador

# manejar la interrupción de temporizador incrementando el comparador de tiempo
la a1, mtimemp         # a1 = &comparador de tiempo
lw a2, 0(a1)           # cargar los 32 bits más bajos del comparador
lw a3, 4(a1)           # cargar los 32 bits más altos del comparador
addi a4, a2, 1000      # incrementar los bits bajos por 1000 ciclos
sltu a2, a4, a2        # generar acarreo de salida
add a3, a3, a2         # incrementar bits altos
sw a3, 4(a1)           # guardar los 32 bits altos
sw a4, 0(a1)           # guardar los 32 bits bajos

# restaurar registros y retornar
lw a4, 12(a0)          # restaurar a4
lw a3, 4(a0)           # restaurar a3
lw a2, 4(a0)           # restaurar a2
lw a1, 0(a0)           # restaurar a1
csrrw a0, mscratch, a0 # restaurar a0; mscratch = &almacenamiento temp
mret                   # retornar del manejador

```

Figura 10.10: Código RISC-V para un manejador simple de interrupción de temporizador. El código asume que las interrupciones han sido globalmente habilitadas poniendo en `1 mstatus.MIE`; que las interrupciones de temporizador han sido habilitadas poniendo en `1 mie[7]`; que al CSR `mtvec` se ha escrito la dirección de este manejador; y que al CSR `mscratch` se ha escrito la dirección de un *buffer* que contiene 16 bytes de almacenamiento temporal para guardar los registros. El prólogo guarda cinco registros, preservando `a0` en `mscratch` y `a1–a4` en memoria. Luego decodifica la causa de la excepción examinando `mcause`: interrupción si `mcause < 0`, o excepción síncrona si `mcause ≥ 0`. Si es una interrupción, revisa que los bits más bajos de `mcause` sean igual a 7, indicando una interrupción de temporizador en modo M. Si es una interrupción de temporizador, suma 1000 ciclos al comparador de tiempo, para que la próxima interrupción de temporizador ocurra alrededor de 1000 ciclos del temporizador en el futuro. Finalmente, el epílogo restaura los registros `a0–a4` y `mscratch`, luego retorna al punto de donde vino usando `mret`.

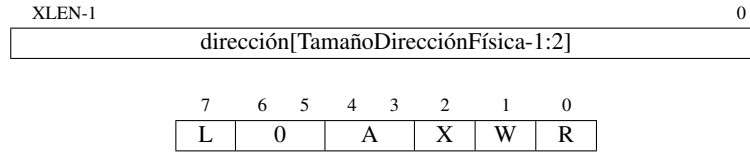


Figura 10.11: Un registro de dirección y configuración de PMP. El registro de dirección es corrido a la derecha por 2, y si las direcciones físicas son menores que $XLEN-2$ bits de ancho, los bits más altos son ceros. Los campos R, W y X otorgan permisos de lectura, escritura y ejecución. El campo A establece el modo PMP, y el campo L bloquea el registro PMP y sus registros de dirección correspondientes.

10.4 Modo Usuario y Aislamiento de Procesos en Sistemas Embebidos

A pesar de que el modo Máquina es suficiente para sistemas embebidos simples, solo es recomendable cuando todo el código es confiable, dado que el modo M tiene acceso ilimitado a la plataforma de hardware. Más frecuentemente, no es práctico confiar en todo el código de la aplicación, porque no se conoce de antemano o es muy vasto para saber si es correcto. Por eso, RISC-V provee mecanismos para proteger al sistema del código no confiable, y para proteger procesos no confiables entre ellos.

Debe ser prohibido para el código no confiable el ejecutar instrucciones privilegiadas, como `mret`, y acceder a CSRs privilegiados, como `mstatus`, ya que éstos permitirían al programa tomar el control del sistema. Esta restricción se logra fácilmente: un modo adicional de privilegio, *modo Usuario* (modo U), niega el acceso a estas funciones, generando una excepción de instrucción ilegal cuando se intente usar una instrucción o CSR de modo M. Por lo demás, el modo U y el modo M se comportan muy similarmente. El software de modo M puede entrar al modo U poniendo `mstatus.MPP` en U (el cual, como muestra la Figura 10.9, está codificado como 0), luego ejecutando una instrucción `mret`. Si una excepción ocurre en modo U, el control es devuelto al modo M.

También es necesario restringir el acceso del código no confiable a únicamente su propia memoria. Los procesadores que implementan modos M y U tienen una funcionalidad llamada PMP (Physical Memory Protection: Protección Física de Memoria), la cual permite al modo M especificar a cuáles direcciones de memoria puede acceder el modo U. PMP consiste de varios registros de dirección (usualmente de ocho a dieciséis) y sus registros de configuración correspondientes, los cuales otorgan o niegan permisos de lectura, escritura y ejecución. Cuando un procesador en modo U intenta hacer *fetch* de una instrucción, o ejecuta un *load* o *store*, la dirección es comparada contra todos los registros de dirección PMP. Si la dirección es mayor o igual que la dirección PMP i , pero menor que la dirección PMP $i+1$, entonces el registro de configuración de PMP $i+1$ decide si ese acceso puede proceder; de lo contrario, levanta una excepción de acceso.

La Figura 10.11 muestra la estructura de un registro de dirección y configuración de PMP. Ambos son CSRs, con nombres para los registros de direcciones de `pmpaddr0` a `pmpaddrN`, donde $N+1$ es el número de PMPs implementados. Los registros de direcciones son corridos a la derecha dos bits porque los PMPs tienen una granularidad de cuatro bytes. Los registros de configuración están densamente empaquetados en los CSRs para acelerar el cambio de contexto, como muestra la Figura 10.12. La configuración de un PMP consiste de los bits R, W y X, los cuales, cuando se encuentran en 1 permiten *loads*, *stores* y *fetches*, respectiva-

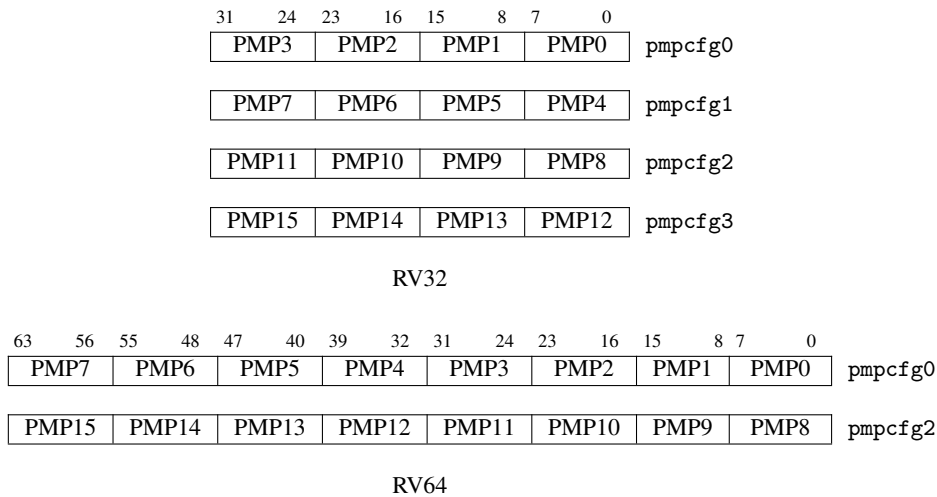


Figura 10.12: La estructura de las configuraciones de PMP en los CSRs `pmpcfg`. Para RV32 (arriba), los dieciséis registros de configuración están empaquetados en cuatro CSRs. Para RV64 (abajo), están empaquetados en los dos CSRs enumerados con números pares.

mente, y un campo de modo, *A*, el cual deshabilita este PMP cuando se encuentra en 0 o lo habilita cuando se encuentra en 1. La configuración de PMP también soporta otros modos y puede ser bloqueada. Estas características son descritas en [Waterman and Asanović 2017].

10.5 Modo Supervisor para Sistemas Operativos Modernos

El esquema de PMP descrito en la sección anterior es atractivo para sistemas embebidos porque provee protección de memoria a un costo relativamente bajo, pero tiene varios inconvenientes que limitan su uso en computación de propósito general. Dado que PMP solo soporta una cantidad fija de regiones de memoria, no escala a aplicaciones complejas. Y como estas regiones deben ser contiguas en memoria física, el sistema puede sufrir de fragmentación de memoria. Finalmente, PMP no soporta *paging* a almacenamiento secundario eficientemente.

Procesadores RISC-V más sofisticados tratan estos problemas de la misma manera que casi todas las arquitecturas de propósito general: usando memoria virtual basada en páginas. Esta funcionalidad forma el núcleo del *modo supervisor* (modo S), un modo de privilegio opcional diseñado para soportar sistemas operativos modernos similares a Unix, tales como Linux, FreeBSD y Windows. El modo S es más privilegiado que el modo U, pero menos privilegiado que el modo M. Al igual que en el modo U, el software del modo S no puede usar CSRs ni instrucciones del modo M, y está sujeto a restricciones de PMP. Esta sección cubre interrupciones y excepciones del modo S, y la siguiente sección detalla el sistema de memoria virtual del modo S.

Por defecto, todas las excepciones, sin importar el modo de privilegio, transfieren el control al manejador de excepción de modo M. La mayoría de las excepciones en un sistema Unix, sin embargo, deben invocar al sistema operativo, que corre en modo S. El manejador de excepciones de modo M puede re-encaminar las excepciones al modo S, pero este código

La fragmentación ocurre cuando hay memoria disponible, pero no en pedazos contiguos lo suficientemente grandes para ser útiles.

¿Por qué no delegar las interrupciones incondicionalmente al modo S? Una razón es la virtualización: si el modo M quiere virtualizar un dispositivo para el modo S, sus interrupciones deben ir al modo M, no al modo S.

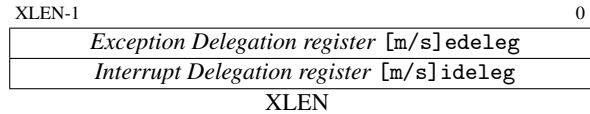


Figura 10.13: Los CSRs de delegación. CSRs de delegación de excepciones e interrupciones de máquina y supervisor (medeleg, sedeleg, mideleg, sideleg). Habilitan la delegación a un manejador de excepción de menor privilegio, donde el índice de la posición del bit habilita la excepción o interrupción correspondiente en el registro [m/s]ip.

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

Figura 10.14: CSRs de interrupción de supervisor. Son registros de XLEN bits de lectura y escritura que contienen las interrupciones pendientes (sip) y los bits de habilitación de interrupciones (sie).

adicional reduciría la velocidad del manejo de la mayoría de excepciones. Por eso, RISC-V provee un mecanismo de *delegación de excepciones*, por el cual las interrupciones y excepciones síncronas pueden ser delegadas al modo S selectivamente, evitando software de modo M por completo.

El CSR mideleg (*Delegación de Interrupciones de Máquina*³) controla cuales interrupciones son delegadas al modo S (Figura 10.13). Al igual que mip y mie, cada bit en mideleg corresponde al código de excepción del mismo número en la Figura 10.3. Por ejemplo, mideleg[5] corresponde a la interrupción de temporizador de modo S; si está en 1, las interrupciones de temporizador de modo S transferirán el control al manejador de excepciones del modo S, en lugar del manejador de excepciones del modo M.

Cualquier interrupción delegada al modo S puede ser enmascarada por software en modo S. Los CSRs sie (*Habilitador de Interrupciones de Supervisor*⁴) y sip (*Interrupciones de Supervisor Pendientes*⁵) son CSRs de modo S y subconjuntos de los CSRs mie y mip (Figura 10.14). Tienen la misma estructura que sus contrapartes de modo M, pero solo en los bits correspondientes a interrupciones que han sido delegadas en mideleg es posible leer y escribir con sie y sip. Los bits correspondientes a interrupciones que no han sido delegadas siempre son cero.

El modo M también puede delegar excepciones síncronas al modo S usando el CSR medeleg (*Delegación de Excepciones de Máquina*⁶) (Figura 10.13). El mecanismo es análogo a la delegación de interrupciones, pero los bits en medeleg corresponden a los códigos de excepciones síncronas en la Figura 10.3. Por ejemplo, poner en 1 medeleg[15] delegará fallo de página en store al modo S.

Nótese que las excepciones nunca transfieren el control a un modo menos privilegiado, sin importar la configuración de delegación. Una excepción que ocurre en modo M siempre es manejada en modo M. Una excepción que ocurre en modo S puede ser manejada ya sea por el modo M o por el modo S, dependiendo de la configuración de delegación, pero nunca por el modo U.

El modo S tiene varios CSRs de manejo de excepciones, scause, stvec, sepc, stval,

El modo S no controla directamente las interrupciones del temporizador y software sino que usa la instrucción ecall para solicitar al modo M para configurar temporizadores o enviar interrupciones interprocesador en su nombre. Esta convención de software es parte de la *Interfaz Binaria de Supervisor*.

XLEN-1		XLEN-2										20	19	18	17
SD		Reservado										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: El CSR `sstatus`. `sstatus` es un subconjunto de `mstatus` (Figura 10.4), de ahí su estructura similar. SIE y SPIE contienen los habilitadores de interrupción actual y pre-excepción, análogo a MIE y MPIE en `mstatus`. XLEN es 32 para RV32, o 64 para RV64. La Figura 4.2 de [Waterman and Asanović 2017] es la base de esta figura; ver Sección 4.1 de ese documento para una descripción de los otros campos.

`sscratch` y `sstatus`, que llevan a cabo la misma función que sus contrapartes de modo M descritas en la Sección 10.2 (Figuras 10.7 a 10.8). La Figura 10.15 muestra la estructura del registro `sstatus`. La instrucción de retorno de excepción de supervisor, `sret`, se comporta del mismo modo que `mret`, pero actúa sobre los CSRs de manejo de excepciones de modo S en lugar de los de modo M.

La acción de tomar una excepción también es muy similar al modo M. Si un hart toma una excepción y ésta es delegada al modo S, el hardware atómicamente pasa por varias transiciones de estados similares, usando CSRs de modo S en lugar de los de modo M:

- El PC de la instrucción que causó la excepción es preservado en `sepc`, y `stvec` es escrito al PC.
- `scause` recibe la causa de la excepción, como se codifica en la Figura 10.3, y `stval` recibe la dirección defectuosa o alguna otra palabra de información específica de la excepción.
- Las interrupciones son deshabilitadas escribiendo SIE=0 en el CSR `sstatus`, y el valor previo de SIE es preservado en SPIE.
- El modo de privilegio pre-excepción es preservado en el campo SPP de `sstatus`, y el modo de privilegio es cambiado a S.

10.6 Memoria Virtual Basada en Páginas

El modo S provee un sistema convencional de memoria virtual que divide la memoria en *páginas* de tamaño fijo con los propósitos de traducción de direcciones y protección de memoria. Cuando la paginación está habilitada, la mayoría de las direcciones (incluyendo las direcciones efectivas de `load`, `store` y el PC) son *direcciones virtuales* que deben ser traducidas a *direcciones físicas* para tener acceso a la memoria física. Las direcciones virtuales son traducidas a direcciones físicas por medio del recorrido de un árbol (*radix tree*) alto conocido como la *tabla de páginas*⁷. Un nodo hoja en la tabla de páginas indica si la dirección virtual es mapeada a una página física, y si lo es, qué modos de privilegio y tipos de acceso tienen permiso de acceder a la página. El acceso a una página no mapeada o que otorga permisos insuficientes resulta en un *fallo de página*.



Simplicidad

Las páginas de 4 KiB han sido populares por cinco décadas empezando con la IBM 360 modelo 67. Atlas, la primera computadora con paginación, tenía páginas de 3 KiB (tenía palabras de 6 bytes). Nos parece destacable que, después de medio siglo de crecimiento exponencial en rendimiento computacional y capacidad de memoria, el tamaño de las páginas permanezca virtualmente inalterado.

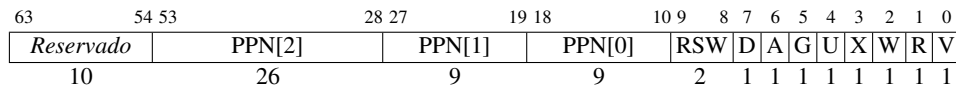


Figura 10.17: Una entrada de la tabla de páginas (PTE) de RV64 Sv39.

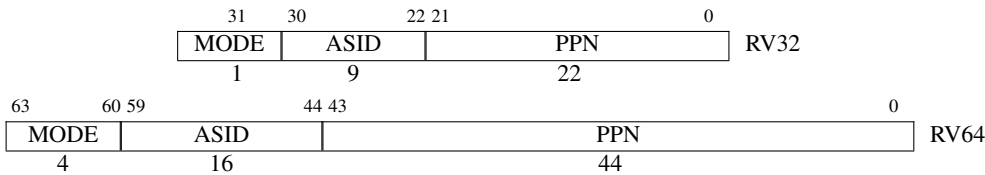


Figura 10.18: El CSR `satp`. Las Figuras 4.11 y 4.12 de [Waterman and Asanović 2017] son las bases para esta figura.

página, el *radix* del árbol correspondientemente baja a 2^9 . El árbol tiene tres niveles. El espacio de direcciones de 512 GiB de Sv39 está dividido en 2^9 *gigapáginas*, cada 1 GiB. Cada gigapágina está subdividida en 2^9 megapáginas, las cuales en Sv39 son ligeramente más pequeñas que en Sv32: 2 MiB. Cada megapágina está subdividida en 2^9 páginas base de 4 KiB.

La Figura 10.17 muestra la estructura de una PTE de Sv39. Es idéntica a una PTE de Sv32, excepto que el campo PPN ha sido extendido a 44 bits para soportar direcciones físicas de 56 bits, o 2^{26} GiB de espacio físico de direcciones.

■ **Elaboración: Bits de direcciones no usados**

Dado que las direcciones virtuales de Sv39 son más angostas que un registro entero de RV64, usted podría preguntarse qué pasa con los 25 bits restantes. Sv39 ordena que los bits de dirección 63–39 sean copias del bit 38. Así, las direcciones virtuales válidas son `0000_0000_0000_0000hex–0000_003f_ffff_ffffhex` y `ffff_ffc0_0000_0000hex–ffff_ffff_ffff_ffffhex`. La brecha entre estos dos rangos es, por supuesto, 2^{25} veces más grande que el tamaño de los dos rangos combinados, aparentemente desperdiciando el 99.999997% de los valores que un registro de 64 bits puede representar. ¿Por qué no hacer un mejor uso de esos 25 bits adicionales? La respuesta es que, cuando los programas crezcan lo suficiente para requerir más que 512 GiB de espacio virtual de direcciones, los arquitectos querrán incrementar el espacio de direcciones sin eliminar la compatibilidad retroactiva. Si permitiéramos que los programas almacenaran datos adicionales en los 25 bits más altos, sería imposible reclamar esos bits posteriormente para almacenar direcciones más grandes. Permitir almacenamiento de datos en bits de direcciones no usados no es solo un error doloroso, sino uno que ha recurrido muchas veces en la historia de la computación.

RV32		
Valor	Nombre	Descripción
0	Bare	Sin traducción o protección.
1	Sv32	Direccionamiento virtual de 32 bits basado en página.
RV64		
Valor	Nombre	Descripción
0	Bare	Sin traducción o protección.
8	Sv39	Direccionamiento virtual de 39 bits basado en página.
9	Sv48	Direccionamiento virtual de 48 bits basado en página.

Figura 10.19: La codificación del campo MODE en el CSR *satp*. La Tabla 4.3 de [Waterman and Asanović 2017] es la base para esta figura.

Un CSR de modo S, *satp* (Traducción y Protección de Direcciones de Supervisor⁸), controla el sistema de paginación. Como muestra la Figura 10.18, *satp* tiene tres campos. El campo MODE habilita paginación y selecciona la profundidad de la tabla de páginas; La Figura 10.19 muestra su codificación. El campo ASID (Identificador de Espacio de Direcciones⁹) es opcional y puede ser usado para reducir el costo de cambios de contexto. Finalmente, el campo PPN contiene la dirección física de la tabla de páginas raíz, dividido entre el tamaño de página 4 KiB. Típicamente, el software de modo M escribirá cero a *satp* antes de ingresar al modo S por primera vez, deshabilitando paginación, luego el software de modo S lo escribirá nuevamente después de configurar las tablas de páginas.

Cuando la paginación está habilitada en el registro *satp*, las direcciones virtuales de los modos S y U son traducidas a direcciones físicas por un recorrido de la tabla de páginas, iniciando en la raíz. La Figura 10.20 ilustra este proceso:

1. *satp*.PPN da la dirección base de la tabla de páginas del primer nivel, y VA[31:22] da el índice del primer nivel, de modo que el procesador lee la PTE ubicada en la dirección ($\text{satp.PPN} \times 4096 + \text{VA}[31:22] \times 4$).
2. Esa PTE contiene la dirección base de la tabla de páginas del segundo nivel y VA[21:12] da el índice del segundo nivel, de modo que el procesador lee la PTE hoja ubicada en $(\text{PTE.PPN} \times 4096 + \text{VA}[21:12] \times 4)$.
3. El campo PPN de la PTE hoja y el *offset de página* (los doce bits menos significativos de la dirección virtual original) forman el resultado final: la dirección física es $(\text{PTEhoja.PPN} \times 4096 + \text{VA}[11:0])$.

Luego el procesador lleva a cabo el acceso a memoria física. El proceso de traducción es casi el mismo para Sv39 que para Sv32, pero con PTEs más grandes y un nivel más de indirectación. La Figura 10.27, al final de este capítulo, da una descripción completa del algoritmo de recorrido de la tabla de páginas, detallando las condiciones que causan excepciones y el caso especial de traducciones de superpáginas.

Eso es casi todo respecto al sistema de paginación de RISC-V, excepto por un detalle. ¡Si todos los fetches de instrucciones, loads y stores resultaran en varios accesos a la tabla de páginas, entonces la paginación reduciría el rendimiento sustancialmente! Todos los procesadores modernos reducen este *overhead* con un cache de traducciones de direcciones usualmente llamado TLB (Translation Lookaside Buffer: Cache de Traducciones). Para reducir



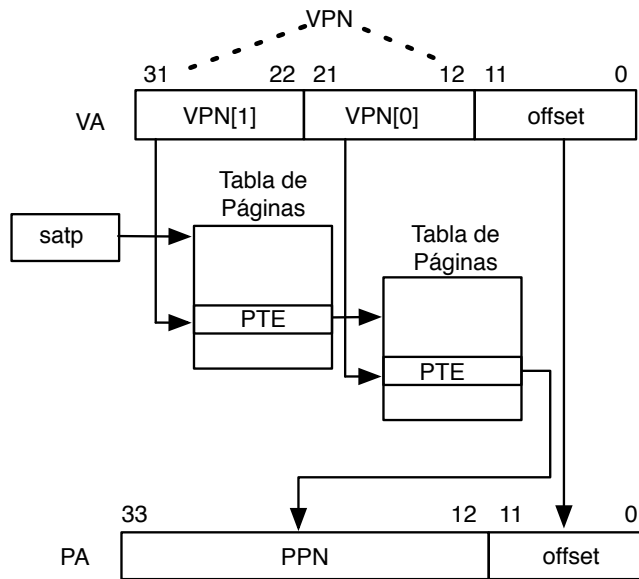


Figura 10.20: Diagrama del proceso de traducción de direcciones Sv32.

el costo de este cache, la mayoría de procesadores no lo mantienen coherente con la tabla de páginas automáticamente—si el sistema operativo modifica la tabla de páginas, el cache caduca. El modo S agrega una instrucción más para resolver este problema: `sfence.vma` informa al procesador que el software pudo haber modificado las tablas de páginas, así que el procesador puede desalojar los caches de traducciones correspondientes. Recibe dos argumentos opcionales, los cuales reducen el alcance del desalojo de caches: `rs1` indica a cuál dirección virtual corresponde la traducción que ha sido modificada en la tabla de páginas, y `rs2` da el identificador del espacio de direcciones del proceso cuya tabla de páginas ha sido modificada. Si `x0` es pasado a ambos, el cache de traducciones completo es desalojado.

■ **Elaboración: Coherencia de caches de traducciones de direcciones en multiprocesadores**

`sfence.vma` solo afecta el hardware de traducción de direcciones para el hart que ejecutó la instrucción. Cuando un hart modifica una tabla de páginas que otro hart está usando, el primer hart debe usar una interrupción interprocesador para informar al segundo hart que debe ejecutar una instrucción `sfence.vma`. Es común referirse a este procedimiento como un *TLB shutdown*.

10.7 CSRs de Identificación y Rendimiento

Los CSRs restantes identifican características del procesador o ayudan a medir el rendimiento. Los CSRs de identidad son:

- El CSR de ISA de Máquina `misa` da el ancho de la dirección del procesador (32, 64 ó 128 bits) e identifica cuáles extensiones de instrucciones están incluidas (Figura 10.21).
- El CSR de ID de Proveedor `mvendorid` provee el ID de fabricante JEDEC del proveedor del procesador (Figura 10.22).
- El CSR de ID de Arquitectura de máquina `marchid` da la microarquitectura base. Combinar `mvendorid` con `marchid` identifica de manera única la microarquitectura implementada (Figura 10.23).
- El CSR de ID de Implementación de máquina `mimpid` da la versión de la *implementación* de la microarquitectura base en `marchid` (Figura 10.23).
- El CSR de ID de Hart `mhartid` da el ID entero del hart que se encuentra en ejecución (Figura 10.23).

Aquí están los CSRs de medición:

- El CSR de Tiempo de Máquina `mtime` es un contador de tiempo real de 64 bits (Figura 10.24).
- El CSR de Comparación de Tiempo de Máquina `mtimecmp` causa una interrupción cuando `mtime` iguala o excede su valor (Figura 10.24).
- Los CSRs de habilitación de contadores de máquina y supervisor de 32 bits (`mcounteren` y `scounteren`) controlan la disponibilidad de los CSRs monitores de rendimiento de hardware al siguiente nivel menos privilegiado (Figura 10.25).
- Los 32 CSRs monitores de rendimiento de hardware (`mcycle`, `minstret`, `mhpcounter3`, ..., `mhpcounter31`) cuentan ciclos de reloj, instrucciones retiradas, y luego hasta 29 eventos seleccionados por el programador usando los CSRs `mhpmevent3`, ..., `mhpmevent31` (Figura 10.26).



Figura 10.21: El CSR de ISA de Máquina *mis*a reporta el ISA soportado. El campo MXL (XLEN de Máquina) codifica el ancho del ISA base nativo de enteros: 1 es 32 bits, 2 es 64, y 3 es 128. El campo Extensiones codifica la presencia de las extensiones estándar, con un único bit por letra del alfabeto (el bit 0 codifica la presencia de la extensión “A”, el bit 1 codifica la presencia de la extensión “B”, hasta llegar al bit 25 que codifica “Z”).

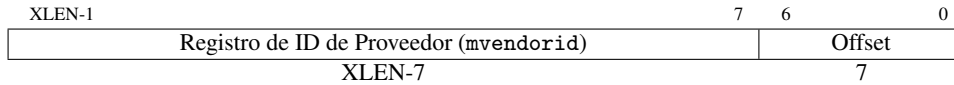


Figura 10.22: El CSR *mvendid* provee el ID de fabricante JEDEC del procesador.

10.8 Observaciones Finales

Estudio tras estudio muestran que los diseñadores excelentes producen estructuras que son más rápidas, pequeñas, simples, claras y producen más con menos esfuerzo. Las diferencias entre los excelentes y la media se aproxima a un orden de magnitud.

—Fred Brooks, Jr., 1986.

Brooks es un ganador del Premio Turing y un arquitecto de la familia de computadoras IBM System/360, la cual demostró la importancia de distinguir la arquitectura de la implementación. Descendientes de esa arquitectura de 1964 se siguen vendiendo en la actualidad.

La modularidad de las arquitecturas privilegiadas RISC-V suple las necesidades de una variedad de sistemas. El minimalista modo Máquina soporta aplicaciones embebidas básicas a un bajo costo. El modo adicional Usuario y la Protección de Memoria Física juntos habilitan *multitasking* en sistemas embebidos más sofisticados. Finalmente, el modo Supervisor y memoria virtual basada en páginas proveen la flexibilidad necesaria para alojar sistemas operativos modernos.



Simplicidad



Programabilidad

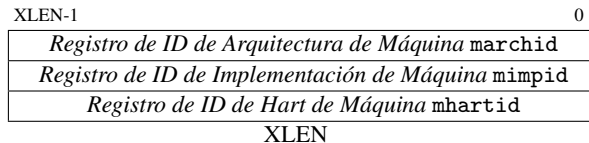


Figura 10.23: Los CSRs de identificación de Máquina (*marchid*, *mimpid*, *mhartid*) identifican la microarquitectura e implementación del procesador y el número del hart actualmente ejecutado.

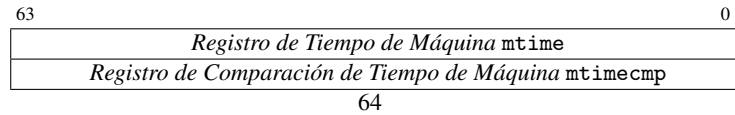


Figura 10.24: Los CSRs de Tiempo de Máquina (*mtime* y *mtimecmp*) miden tiempo y causan una interrupción cuando $mtime \geq mtimecmp$.

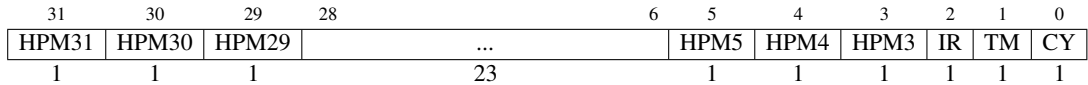


Figura 10.25: Los registros de habilitación de contadores *mcounteren* y *scounteren* controlan la disponibilidad de los contadores de monitoreo de rendimiento de hardware al siguiente modo menos privilegiado.

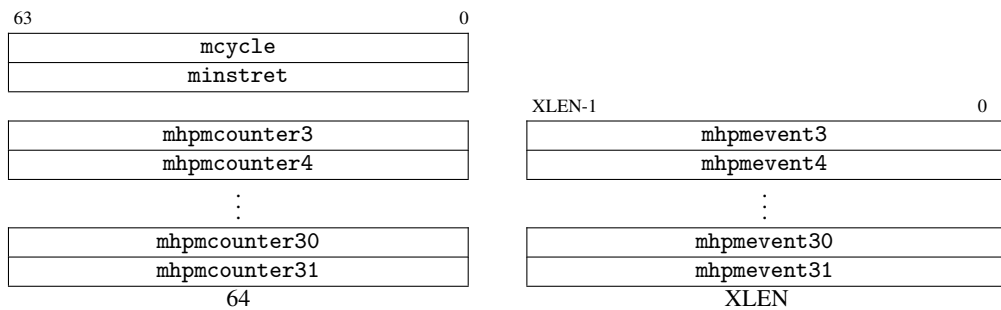


Figura 10.26: Los CSRs de monitoreo de rendimiento de hardware (*mcycle*, *minstret*, *mhpmcounter3*, ..., *mhpmcounter31*) y los eventos que ellos cuentan *mhpmevent3*, ..., *mhpmevent31*. Solo para RV32, lecturas de los CSRs *mcycle*, *minstret* y *mhpmcounter_n* retornan los 32 bits más bajos, mientras que lecturas de los CSRs *mcycle_h*, *minstreth* y *mhpmcounter_{nh}* retornan los bits 63–32 del contador correspondiente.

1. Asignar $a = \text{satp.ppn} \times \text{PAGESIZE}$, y asignar $i = \text{LEVELS} - 1$.
2. Asignar pte el valor de la PTE en la dirección $a + va.vpn[i] \times \text{PTESIZE}$.
3. Si $pte.v = 0$, o si $pte.r = 0$ y $pte.w = 1$, detener y levantar una excepción de fallo de página.
4. De lo contrario, la PTE es válida. Si $pte.r = 1$ o $pte.x = 1$, ir al paso 5. De lo contrario, esta PTE es un puntero al siguiente nivel de la tabla de páginas. Asignar $i = i - 1$. Si $i < 0$, detener y levantar una excepción de fallo de página. De lo contrario, asignar $a = pte.ppn \times \text{PAGESIZE}$ e ir al paso 2.
5. Se ha encontrado una PTE hoja. Determinar si el acceso a memoria solicitado es permitido por los bits $pte.r$, $pte.w$, $pte.x$, y $pte.u$, dado el modo de privilegio actual y el valor de los campos SUM y MXR del registro `mstatus`. Si no, detener y levantar una excepción de fallo de página.
6. Si $i > 0$ y $pa.ppn[i - 1 : 0] \neq 0$, esta es una superpágina desalineada; detener y levantar una excepción de fallo de página.
7. Si $pte.a = 0$, o si el acceso a memoria es un store y $pte.d = 0$, entonces ya sea:
 - Levantar una excepción de fallo de página, o:
 - Poner $pte.a$ en 1 y, si el acceso a memoria es un store, también poner $pte.d$ en 1.
8. La traducción fue exitosa. La dirección física traducida está dada de la siguiente manera:
 - $pa.pgoff = va.pgoff$.
 - Si $i > 0$, entonces esta es una traducción de superpágina y $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
 - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

Figura 10.27: El algoritmo completo para traducción de direcciones virtuales a físicas. va es la dirección virtual de entrada y pa es la dirección física de salida. La constante `PAGESIZE` es 2^{12} . Para Sv32, `LEVELS=2` y `PTESIZE=4`, mientras que para Sv39, `LEVELS=3` y `PTESIZE=8`. La Sección 4.3.2 de [Waterman and Asanović 2017] es la base para esta figura.

10.9 Para Aprender Más

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/>.

Notas

¹Preemptive: Con derecho preferente. En el contexto de interrupciones, se refiere a la habilidad de interrumpir un manejador que se encuentra en ejecución.

²`wfi`: Wait For Interrupt.

³`mdeleg`: Machine Interrupt Delegation.

⁴`sie`: Supervisor Interrupt Enable.

⁵`sip`: Supervisor Interrupt Pending.

⁶`medeleg`: Machine Exception Delegation.

⁷Tabla de Páginas: En inglés, *Page Table*.

⁸`satp`: Supervisor Address Translation and Protection.

⁹ASID: *Address Space Identifier*

Futuras Extensiones RISC-V Opcionales

Alan Perlis (1922–1990) fue el primer galardonado al Premio Turing (1966), otorgado por su influencia en lenguajes de programación y compiladores avanzados. En 1958 ayudó en el diseño de ALGOL, el cual ha influenciado virtualmente a todos los lenguajes de programación imperativos incluyendo C y Java.



Los tontos ignoran la complejidad. Los pragmáticos la sufren. Algunos pueden evitarla. Los genios la eliminan.

—Alan Perlis, 1982

La Fundación RISC-V desarrollará por lo menos ocho extensiones opcionales.

11.1 Extensión Estándar “B” para Manipulación de Bits

La extensión B ofrece manipulación de bits, incluyendo inserción, extracción y prueba de campos de bits¹; rotaciones; *funnel shifts*; permutaciones de bits y bytes; conteo de ceros a la izquierda y a la derecha; y conteo de bits en 1.

11.2 Extensión Estándar “E” para Embebidos

Para reducir el costo de núcleos de gama baja, tiene 16 registros menos. RV32E es la razón por la cual los registros *saved* y *temporary* están divididos entre los registros 0-15 y 16-31 (Figura 3.2).

11.3 Extensión de la Arquitectura Privilegiada “H” para Soporte de Hypervisor

La extensión H a la arquitectura privilegiada agrega un nuevo modo *hypervisor* y un segundo nivel de traducción de direcciones basado en páginas para mejorar la eficiencia de correr múltiples sistemas operativos en la misma máquina.

11.4 Extensión Estándar “J” para Lenguajes Traducidos Dinámicamente

Muchos lenguajes populares son usualmente implementados por traducción dinámica, incluyendo Java y Javascript. Esos lenguajes pueden beneficiarse de soporte adicional del ISA para revisiones dinámicas y *garbage collection*. (La letra J es para abreviar compilador *Just-In-Time*).



11.5 Extensión Estándar “L” para Punto Flotante Decimal

La extensión L está dirigida al soporte de aritmética de punto flotante decimal como se define en el estándar IEEE 754-2008. El problema con números binarios es que no pueden representar algunas fracciones decimales comunes, tales como 0.1. La motivación para RV32L es que la base numérica de la computación puede ser idéntica a la de la entrada y salida.

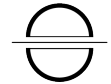


11.6 Extensión Estándar “N” para Interrupciones a Nivel de User

La extensión N permite a interrupciones y excepciones que ocurren en programas de nivel de usuario transferir el control directamente a un manejador a nivel de usuario sin invocar un ambiente de ejecución externo. Las interrupciones de nivel de usuario están principalmente destinadas al soporte de sistemas embebidos seguros únicamente con los modos M y U presentes (Capítulo 10). Sin embargo, también pueden soportar manejadores a nivel de usuario en sistemas que corren sistemas operativos como Unix. Cuando se utiliza en un ambiente Unix, el manejo de señales convencionales probablemente prevalecerá, pero las interrupciones a nivel de usuario pueden ser usadas como un bloque básico para extensiones futuras que generan eventos a nivel de usuario tales como barreras de recolección de basura, overflow de enteros y excepciones de punto flotante.



Simplicidad



Aislamiento de Arq e Impl

11.7 Extensión Estándar “P” para Instrucciones Packed-SIMD

La extensión P subdivide los registros arquitecturales existentes para proveer computación paralela-en-datos en tipos de datos pequeños. Los diseños *Packed-SIMD* representan un punto razonable de diseño al reutilizar recursos de un datapath ancho existente. Sin embargo, si es necesario dedicar una cantidad significativa de recursos adicionales a la ejecución paralela-en-datos, el Capítulo 8 muestra que diseños para arquitecturas vectorizadas son una mejor opción, y los arquitectos deberían usar la extensión RVV.



Rendimiento

11.8 Extensión Estándar “Q” para Punto Flotante de Precisión Cuádruple

La extensión Q agrega instrucciones de punto flotante binario de precisión cuádruple de 128 bits apegadas al estándar IEEE 754-2008 de aritmética. Los registros de punto flotante son ahora extendidos para almacenar un valor de punto flotante de precisión simple, doble o cuádruple. La extensión de punto flotante binario de precisión cuádruple requiere RV64IFD.

11.9 Observaciones Finales

Simplifica, simplifica.

—Henry David Thoreau, un escritor eminente del siglo XIX, 1854



Esperamos que abordar la expansión de RISC-V por medio de un comité abierto basado en estándares signifique que la retroalimentación y el debate ocurran *antes* de que las instrucciones sean finalizadas y no después, cuando sea demasiado tarde para cambiarlas. En el caso ideal, unos pocos miembros implementarán la propuesta antes de que sea ratificada, lo cual es mucho más fácil con *FPGAs*. Proponer extensiones de instrucciones por medio de los comités de la Fundación RISC-V Foundation también representará una cantidad considerable de trabajo, lo cual hará que evolucione lentamente, a diferencia de lo que le pasó a x86-32 (ver Figura 1.2 en la página 4, Capítulo 1). No olvidemos que todo lo mencionado en este capítulo será opcional, sin importar cuántas extensiones sean adoptadas.

Es nuestra esperanza que RISC-V pueda evolucionar con las exigencias tecnológicas, al mismo tiempo que mantenga su reputación como un ISA simple y eficiente. Si lo consigue, RISC-V será un importante avance con respecto a los ISAs incrementales del pasado.



A

Listados de Instrucciones RISC-V

Coco Chanel (1883-1971) Fundadora de la marca de moda Chanel, su búsqueda de simpleza costosa moldeó la moda del siglo XX.



La simpleza es la pauta de toda elegancia real.

—Coco Chanel, 1923

Este apéndice lista todas las instrucciones para RV32/64I, todas las extensiones cubiertas en este libro excepto RVV (RVM, RVA, RVF, RVD y RVC) y todas las pseudoinstrucciones. Cada elemento tiene el nombre de la instrucción, operandos, una definición a nivel de transferencia de registros, tipo de formato de la instrucción, descripción en Español, versiones comprimidas (si las hay) y una figura que muestra la estructura con opcodes. Creemos que usted tiene todo lo que necesita para comprender todas las instrucciones en estos resúmenes compactos. Sin embargo, si desea aun más detalle, puede referirse a las especificaciones oficiales de RISC-V [Waterman and Asanović 2017].

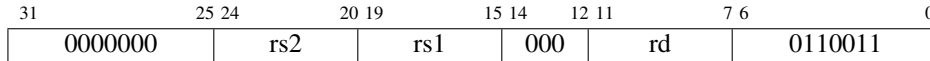
Para ayudar a los lectores a encontrar la instrucción deseada en este apéndice, el encabezado de la página de la izquierda (par) contiene la primera instrucción del inicio de esa página y el encabezado de la página de la derecha (impar) contiene la última instrucción del fin de esa página. El formato es similar a los encabezados de diccionarios, el cual ayuda en la búsqueda de la página en donde se encuentra su palabra. Por ejemplo, el encabezado de la próxima página *par* muestra **AMOADD.W**, la primera instrucción en la página, y el encabezado de la siguiente página *impar* muestra **AMOMINU.D**, la última instrucción en esa página. Estas son las dos páginas en donde se encontrará cualquiera de las siguientes 10 instrucciones: amoadd.w, amoand.d, amoand.w, amomax.d, amomax.w, amomaxu.d, amomaxu.w, amomin.d, amomin.w y amominu.d.

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

Add. Tipo R, RV32I y RV64I.

Suma el registro $x[rs2]$ al registro $x[rs1]$ y escribe el resultado en $x[rd]$. Overflow aritmético ignorado.

Formas comprimidas: **c.add** rd, rs2; **c.mv** rd, rs2

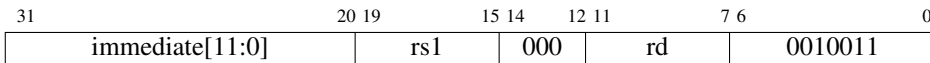


addi rd, rs1, immediate $x[rd] = x[rs1] + \text{sext}(\text{immediate})$

Add Immediate. Tipo I, RV32I y RV64I.

Suma el *inmediato* sign-extended al registro $x[rs1]$ y escribe el resultado en $x[rd]$. Overflow aritmético ignorado.

Formas comprimidas: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

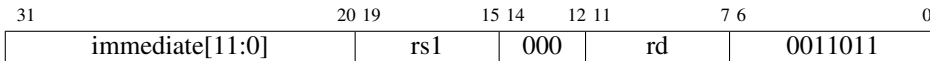


addiw rd, rs1, immediate $x[rd] = \text{sext}((x[rs1] + \text{sext}(\text{immediate})) [31:0])$

Add Word Immediate. Tipo I, solo RV64I.

Suma el *inmediato* sign-extended a $x[rs1]$, trunca el resultado a 32 bits, y escribe el resultado sign-extended en $x[rd]$. Overflow aritmético ignorado.

Forma comprimida: **c.addiw** rd, imm

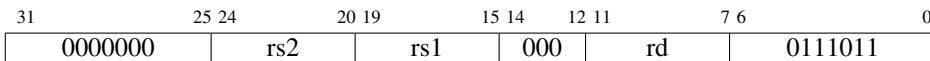


addw rd, rs1, rs2 $x[rd] = \text{sext}((x[rs1] + x[rs2]) [31:0])$

Add Word. Tipo R, solo RV64I.

Suma el registro $x[rs2]$ al registro $x[rs1]$, trunca el resultado a 32 bits, y escribe el resultado sign-extended en $x[rd]$. Overflow aritmético ignorado.

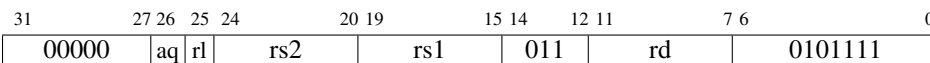
Forma comprimida: **c.addw** rd, rs2



amoadd.d rd, rs2, (rs1) $x[rd] = \text{AM064}(M[x[rs1]] + x[rs2])$

Atomic Memory Operation: Add Doubleword. Tipo R, solo RV64A.

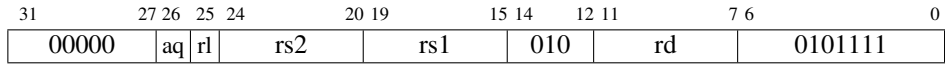
Atómicamente, siendo t el valor del doubleword en memoria en la dirección $x[rs1]$, asignar $t + x[rs2]$ a ese doubleword en memoria. Escribir t a $x[rd]$.



amoadd.w rd, rs2, (rs1) $x[rd] = \text{AMO32}(M[x[rs1]] + x[rs2])$

Atomic Memory Operation: Add Word. Tipo R, RV32A y RV64A.

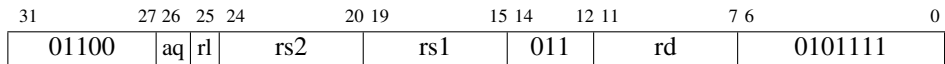
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar $t + x[rs2]$ a ese word en memoria. Escribir la extensión de signo de t a $x[rd]$.



amoand.d rd, rs2, (rs1) $x[rd] = \text{AMO64}(M[x[rs1]] \& x[rs2])$

Atomic Memory Operation: AND Doubleword. Tipo R, solo RV64A.

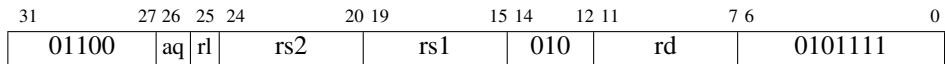
Atómicamente, siendo t el valor del doubleword en memoria en la dirección $x[rs1]$, asignar el AND a nivel de bits de t y $x[rs2]$ a ese doubleword en memoria. Escribir t a $x[rd]$.



amoand.w rd, rs2, (rs1) $x[rd] = \text{AMO32}(M[x[rs1]] \& x[rs2])$

Atomic Memory Operation: AND Word. Tipo R, RV32A y RV64A.

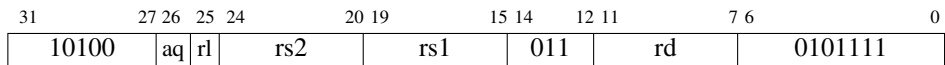
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el AND a nivel de bits de t y $x[rs2]$ a ese word en memoria. Escribir la extensión de signo de t a $x[rd]$.



amomax.d rd, rs2, (rs1) $x[rd] = \text{AMO64}(M[x[rs1]] \text{ MAX } x[rs2])$

Atomic Memory Operation: Maximum Doubleword. Tipo R, solo RV64A.

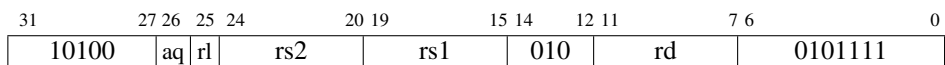
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más grande entre t y $x[rs2]$ a ese doubleword en memoria, usando una comparación de complemento a dos. Escribir t a $x[rd]$.



amomax.w rd, rs2, (rs1) $x[rd] = \text{AMO32}(M[x[rs1]] \text{ MAX } x[rs2])$

Atomic Memory Operation: Maximum Word. Tipo R, RV32A y RV64A.

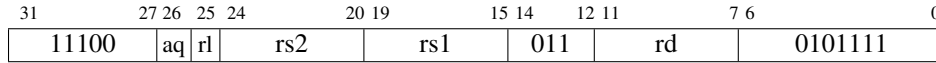
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más grande entre t y $x[rs2]$ a ese word en memoria, usando una comparación de complemento a dos. Escribir la extensión de signo de t a $x[rd]$.



amomaxu.d rd, rs2, (rs1) $x[rd] = AMO64(M[x[rs1]] \text{ MAXU } x[rs2])$

Atomic Memory Operation: Maximum Doubleword, Unsigned. Tipo R, solo RV64A.

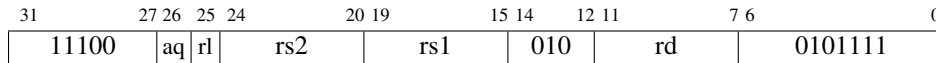
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más grande entre t y $x[rs2]$ a ese word en memoria, usando una comparación sin signo. Escribir t a $x[rd]$.



amomaxu.w rd, rs2, (rs1) $x[rd] = AMO32(M[x[rs1]] \text{ MAXU } x[rs2])$

Atomic Memory Operation: Maximum Word, Unsigned. Tipo R, RV32A y RV64A.

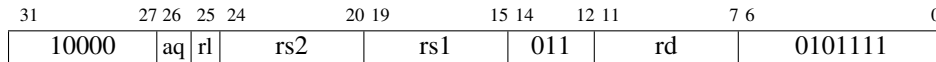
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más grande entre t y $x[rs2]$ a ese word en memoria, usando una comparación sin signo. Escribir la extensión de signo de t a $x[rd]$.



amomin.d rd, rs2, (rs1) $x[rd] = AMO64(M[x[rs1]] \text{ MIN } x[rs2])$

Atomic Memory Operation: Minimum Doubleword. Tipo R, solo RV64A.

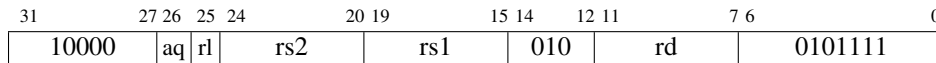
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más pequeño entre t y $x[rs2]$ a ese doubleword en memoria, usando una comparación de complemento a dos. Escribir t a $x[rd]$.



amomin.w rd, rs2, (rs1) $x[rd] = AMO32(M[x[rs1]] \text{ MIN } x[rs2])$

Atomic Memory Operation: Minimum Word. Tipo R, RV32A y RV64A.

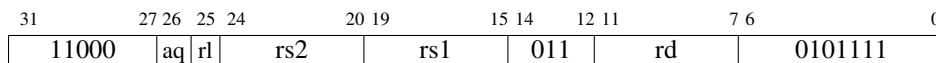
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más pequeño entre t y $x[rs2]$ a ese word en memoria, usando una comparación de complemento a dos. Escribir la extensión de signo de t a $x[rd]$.



amominu.d rd, rs2, (rs1) $x[rd] = AMO64(M[x[rs1]] \text{ MINU } x[rs2])$

Atomic Memory Operation: Minimum Doubleword, Unsigned. Tipo R, solo RV64A.

Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más pequeño entre t y $x[rs2]$ a ese word en memoria, usando una comparación sin signo. Escribir t a $x[rd]$.



amominu.w rd, rs2, (rs1) $x[rd] = \text{AM032}(M[x[rs1]] \text{ MINU } x[rs2])$

Atomic Memory Operation: Minimum Word, Unsigned. Tipo R, RV32A y RV64A.

Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el más pequeño entre t y $x[rs2]$ a ese word en memoria, usando una comparación sin signo. Escribir la extensión de signo de t a $x[rd]$.

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) $x[rd] = \text{AM064}(M[x[rs1]] \mid x[rs2])$

Atomic Memory Operation: OR Doubleword. Tipo R, solo RV64A.

Atómicamente, siendo t el valor del doubleword en memoria en la dirección $x[rs1]$, asignar el OR a nivel de bits de t y $x[rs2]$ a ese doubleword en memoria. Escribir t a $x[rd]$.

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) $x[rd] = \text{AM032}(M[x[rs1]] \mid x[rs2])$

Atomic Memory Operation: OR Word. Tipo R, RV32A y RV64A.

Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el OR a nivel de bits de t y $x[rs2]$ a ese word en memoria. Escribir la extensión de signo de t a $x[rd]$.

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) $x[rd] = \text{AM064}(M[x[rs1]] \text{ SWAP } x[rs2])$

Atomic Memory Operation: Swap Doubleword. Tipo R, solo RV64A.

Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar $x[rs2]$ a ese doubleword en memoria. Escribir t a $x[rd]$.

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) $x[rd] = \text{AM032}(M[x[rs1]] \text{ SWAP } x[rs2])$

Atomic Memory Operation: Swap Word. Tipo R, RV32A y RV64A.

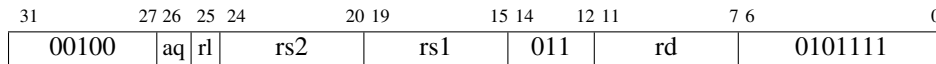
Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar $x[rs2]$ a ese word en memoria. Escribir la extensión de signo de t a $x[rd]$.

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) $x[rd] = \text{AM064}(M[x[rs1]] \wedge x[rs2])$

Atomic Memory Operation: XOR Doubleword. Tipo R, solo RV64A.

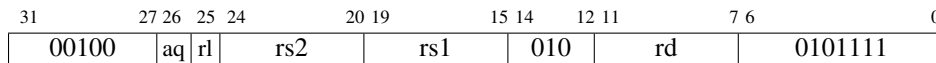
Atómicamente, siendo t el valor del doubleword en memoria en la dirección $x[rs1]$, asignar el XOR a nivel de bits de t y $x[rs2]$ a ese doubleword en memoria. Escribir t a $x[rd]$.



amoxor.w rd, rs2, (rs1) $x[rd] = \text{AM032}(M[x[rs1]] \wedge x[rs2])$

Atomic Memory Operation: XOR Word. Tipo R, RV32A y RV64A.

Atómicamente, siendo t el valor del word en memoria en la dirección $x[rs1]$, asignar el XOR a nivel de bits de t y $x[rs2]$ a ese word en memoria. Escribir la extensión de signo de t a $x[rd]$.

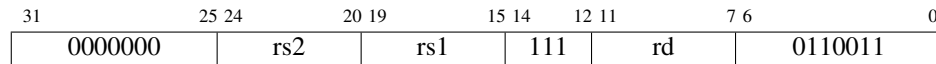


and rd, rs1, rs2 $x[rd] = x[rs1] \& x[rs2]$

AND. Tipo R, RV32I y RV64I.

Calcula el AND a nivel de bits de los registros $x[rs1]$ y $x[rs2]$ y escribe el resultado en $x[rd]$.

Forma comprimida: **c.and** rd, rs2

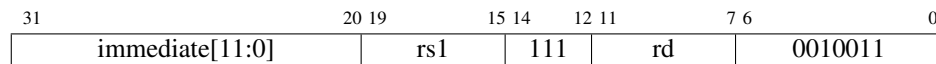


andi rd, rs1, immediate $x[rd] = x[rs1] \& \text{sext}(\text{immediate})$

AND Immediate. Tipo I, RV32I y RV64I.

Calcula el AND a nivel de bits del *inmediato* sign-extended y el registro $x[rs1]$ y escribe el resultado en $x[rd]$.

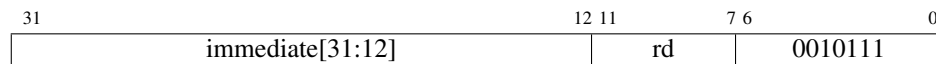
Forma comprimida: **c.andi** rd, imm



auipc rd, immediate $x[rd] = pc + \text{sext}(\text{immediate}[31:12] \ll 12)$

Add Upper Immediate to PC. Tipo U, RV32I y RV64I.

Suma el *inmediato* sign-extended de 20 bits, corrido a la izquierda por 12 bits, al pc , y escribe el resultado en $x[rd]$.



beq rs1, rs2, offset if (rs1 == rs2) pc += sext(offset)

Branch if Equal. Tipo B, RV32I y RV64I.

Si el registro x[rs1] es igual al registro x[rs2], asignar al pc su valor actual más el *offset* sign-extended.

Forma comprimida: **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. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **beq** rs1, x0, offset.

bge rs1, rs2, offset if (rs1 \geq_s rs2) pc += sext(offset)

Branch if Greater Than or Equal. Tipo B, RV32I y RV64I.

Si el registro x[rs1] es por lo menos x[rs2], tratando los valores como números de complemento a dos, asignar al pc su valor actual más el *offset* sign-extended.

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. Tipo B, RV32I y RV64I.

Si el registro x[rs1] es por lo menos x[rs2], tratando los valores como números sin signo, asignar al pc su valor actual más el *offset* sign-extended.

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. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **bge** rs1, x0, offset.

bgt rs1, rs2, offset if (rs1 $>_s$ rs2) pc += sext(offset)

Branch if Greater Than. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **blt** rs2, rs1, offset.

bgtu rs1, rs2, offset if (rs1 $>_u$ rs2) pc += sext(offset)

Branch if Greater Than, Unsigned. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **bltu** rs2, rs1, offset.

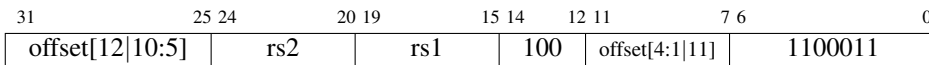
bgtz rs2, offset if (rs2 >_s 0) pc += sext(offset)
Branch if Greater Than Zero. Pseudoinstrucción, RV32I y RV64I.
 Se extiende a **blt** x0, rs2, offset.

ble rs1, rs2, offset if (rs1 ≤_s rs2) pc += sext(offset)
Branch if Less Than or Equal. Pseudoinstrucción, RV32I y RV64I.
 Se extiende a **bge** rs2, rs1, offset.

bleu rs1, rs2, offset if (rs1 ≤_u rs2) pc += sext(offset)
Branch if Less Than or Equal, Unsigned. Pseudoinstrucción, RV32I y RV64I.
 Se extiende a **bgeu** rs2, rs1, offset.

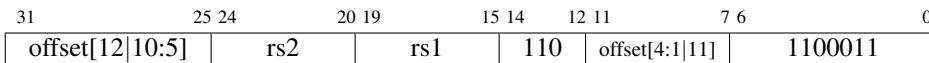
blez rs2, offset if (rs2 ≤_s 0) pc += sext(offset)
Branch if Less Than or Equal to Zero. Pseudoinstrucción, RV32I y RV64I.
 Se extiende a **bge** x0, rs2, offset.

blt rs1, rs2, offset if (rs1 <_s rs2) pc += sext(offset)
Branch if Less Than. Tipo B, RV32I y RV64I.
 Si el registro x[rs1] es menor que x[rs2], tratando los valores como números de complemento a dos, asignar al pc su valor actual más el *offset* sign-extended.



bltz rs1, offset if (rs1 <_s 0) pc += sext(offset)
Branch if Less Than Zero. Pseudoinstrucción, RV32I y RV64I.
 Se extiende a **blt** rs1, x0, offset.

bltu rs1, rs2, offset if (rs1 <_u rs2) pc += sext(offset)
Branch if Less Than, Unsigned. Tipo B, RV32I y RV64I.
 Si el registro x[rs1] es menor que x[rs2], tratando los valores como números sin signo, asignar al pc su valor actual más el *offset* sign-extended.

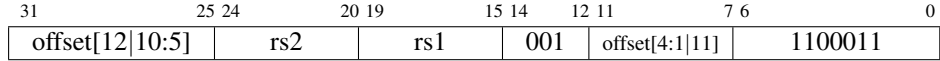


bne rs1, rs2, offset if (rs1 \neq rs2) pc += sext(offset)

Branch if Not Equal. Tipo B, RV32I y RV64I.

Si el registro x[rs1] no es igual al registro x[rs2], asignar al pc su valor actual más el *offset* sign-extended.

Forma comprimida: **c.bnez** rs1, offset



bnez rs1, offset if (rs1 \neq 0) pc += sext(offset)

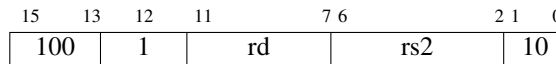
Branch if Not Equal to Zero. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **bne** rs1, x0, offset.

c.add rd, rs2 x[rd] = x[rd] + x[rs2]

Add. RV32IC y RV64IC.

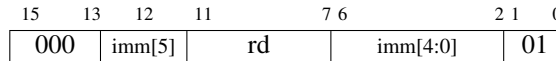
Se extiende a **add** rd, rd, rs2. Inválido cuando rd=x0 o rs2=x0.



c.addi rd, imm x[rd] = x[rd] + sext(imm)

Add Immediate. RV32IC y RV64IC.

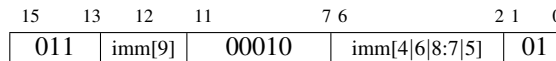
Se extiende a **addi** rd, rd, imm.



c.addi16sp imm x[2] = x[2] + sext(imm)

Add Immediate, Scaled y 16, to Stack Pointer. RV32IC y RV64IC.

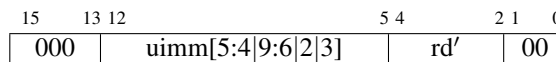
Se extiende a **addi** x2, x2, imm. Inválido cuando imm=0.



c.addi4spn rd', uimm x[8+rd'] = x[2] + uimm

Add Immediate, Scaled by 4, to Stack Pointer; Nondestructive. RV32IC y RV64IC.

Se extiende a **addi** rd, x2, uimm, donde rd=8+rd'. Inválido cuando uimm=0.



c.addiw rd, imm $x[\text{rd}] = \text{sext}((x[\text{rd}] + \text{sext}(\text{imm})) [31:0])$

Add Word Immediate. Solo RV64IC.

Se extiende a **addiw** rd, rd, imm. Inválido cuando rd=x0.

15	13	12	11	7	6	2	1	0
001	imm[5]		rd	imm[4:0]		01		

c.and rd', rs2' $x[8+\text{rd}'] = x[8+\text{rd}'] \& x[8+\text{rs2}']$

AND. RV32IC y RV64IC.

Se extiende a **and** rd, rd, rs2, donde rd=8+rd' y rs2=8+rs2'.

15	10	9	7	6	5	4	2	1	0
100011			rd'	11		rs2'	01		

c.addw rd', rs2' $x[8+\text{rd}'] = \text{sext}((x[8+\text{rd}'] + x[8+\text{rs2}']) [31:0])$

Add Word. Solo RV64IC.

Se extiende a **addw** rd, rd, rs2, donde rd=8+rd' y rs2=8+rs2'.

15	10	9	7	6	5	4	2	1	0
100111			rd'	01	rs2'	01			

c.andi rd', imm $x[8+\text{rd}'] = x[8+\text{rd}'] \& \text{sext}(\text{imm})$

AND Immediate. RV32IC y RV64IC.

Se extiende a **andi** rd, rd, imm, donde 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 $\text{if } (x[8+\text{rs1}'] == 0) \text{ pc} += \text{sext}(\text{offset})$

Branch if Equal to Zero. RV32IC y RV64IC.

Se extiende a **beq** rs1, x0, offset, donde 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 $\text{if } (x[8+\text{rs1}'] \neq 0) \text{ pc} += \text{sext}(\text{offset})$

Branch if Not Equal to Zero. RV32IC y RV64IC.

Se extiende a **bne** rs1, x0, offset, donde 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. RV31IC y RV64IC.Se extiende a **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 y RV64DC.Se extiende a **fld** rd, uimm(rs1), donde $rd=8+rd'$ y $rs1=8+rs1'$.

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 y RV64DC.Se extiende a **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.* Solo RV32FC.Se extiende a **flw** rd, uimm(rs1), donde $rd=8+rd'$ y $rs1=8+rs1'$.

15	13	12	10	9	7	6	5	4	2	1	0
011	uimm[5:3]	rs1'	uimm[2 6]	rd'	00						

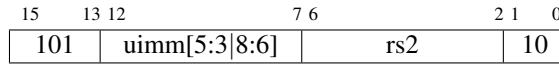
c.flwsp rd, uimm(x2) $f[rd] = M[x[2] + uimm][31:0]$ *Floating-point Load Word, Stack-Pointer Relative.* Solo RV32FC.Se extiende a **flw** rd, uimm(x2).

15	13	12	11	7	6	2	1	0
011	uimm[5]	rd	uimm[4:2 7:6]	10				

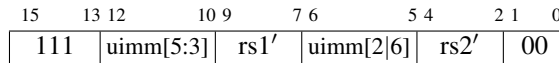
c.fsd rs2', uimm(rs1') $M[x[8+rs1'] + uimm][63:0] = f[8+rs2']$ *Floating-point Store Doubleword.* RV32DC y RV64DC.Se extiende a **fsd** rs2, uimm(rs1), donde $rs2=8+rs2'$ y $rs1=8+rs1'$.

15	13	12	10	9	7	6	5	4	2	1	0
101	uimm[5:3]	rs1'	uimm[7:6]	rs2'	00						

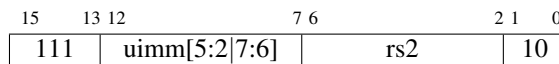
c.fsdsp $rs2, uimm(x2) \quad M[x[2] + uimm][63:0] = f[rs2]$
Floating-point Store Doubleword, Stack-Pointer Relative. RV32DC y RV64DC.
 Se extiende a **fsd** $rs2, uimm(x2)$.



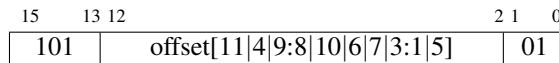
c.fsw $rs2', uimm(rs1') \quad M[x[8+rs1'] + uimm][31:0] = f[8+rs2']$
Floating-point Store Word. Solo RV32FC.
 Se extiende a **fsd** $rs2, uimm(rs1)$, donde $rs2=8+rs2'$ y $rs1=8+rs1'$.



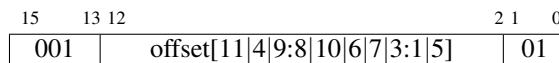
c.fswsp $rs2, uimm(x2) \quad M[x[2] + uimm][31:0] = f[rs2]$
Floating-point Store Word, Stack-Pointer Relative. Solo RV32FC.
 Se extiende a **fsd** $rs2, uimm(x2)$.



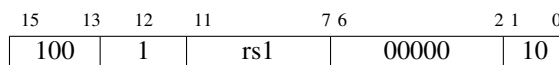
C.j $offset \quad pc += sext(offset)$
Jump. RV32IC y RV64IC.
 Se extiende a **jal** $x0, offset$.



c.jal $offset \quad x[1] = pc+2; pc += sext(offset)$
Jump and Link. Solo RV32IC.
 Se extiende a **jal** $x1, offset$.



c.jalr $rs1 \quad t = pc+2; pc = x[rs1]; x[1] = t$
Jump and Link Register. RV32IC y RV64IC.
 Se extiende a **jalr** $x1, 0(rs1)$. Inválido cuando $rs1=x0$.



c.jr rs1

pc = x[rs1]

Jump Register. RV32IC y RV64IC.Se extiende a **jalr** x0, 0(rs1). Inválido cuando 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. Solo RV64IC.Se extiende a **ld** rd, uimm(rs1), donde rd=8+rd' y 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. Solo RV64IC.Se extiende a **ld** rd, uimm(x2). Inválido cuando 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 y RV64IC.Se extiende a **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 y RV64IC.Se extiende a **lui** rd, imm. Inválido cuando 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 y RV64IC.Se extiende a **lw** rd, uimm(rs1), donde rd=8+rd' y 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) $x[rd] = \text{sext}(M[x[2] + \text{uimm}][31:0])$

Load Word, Stack-Pointer Relative. RV32IC y RV64IC.

Se extiende a **lw** rd, uimm(x2). Inválido cuando 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 y RV64IC.

Se extiende a **add** rd, x0, rs2. Inválido cuando rs2=x0.

15	13	12	11	7	6	2	1	0
100	0	rd	rs2			10		

c.or rd', rs2' $x[8+rd'] = x[8+rd'] \mid x[8+rs2']$

OR. RV32IC y RV64IC.

Se extiende a **or** rd, rd, rs2, donde rd=8+rd' y rs2=8+rs2'.

15	10	9	7	6	5	4	2	1	0
100011	rd'	10	rs2'			01			

c.sd rs2', uimm(rs1') $M[x[8+rs1'] + \text{uimm}][63:0] = x[8+rs2']$

Store Doubleword. Solo RV64IC.

Se extiende a **sd** rs2, uimm(rs1), donde rs2=8+rs2' y 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) $M[x[2] + \text{uimm}][63:0] = x[rs2]$

Store Doubleword, Stack-Pointer Relative. Solo RV64IC.

Se extiende a **sd** rs2, uimm(x2).

15	13	12	7	6	2	1	0
111	uimm[5:3]	8:6	rs2			10	

c.slli rd, uimm $x[rd] = x[rd] \ll \text{uimm}$

Shift Left Logical Immediate. RV32IC y RV64IC.

Se extiende a **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 $x[8+rd'] = x[8+rd'] \gg_s uimm$
Shift Right Arithmetic Immediate. RV32IC y RV64IC.
 Se extiende a **srai** rd, rd, uimm, donde $rd=8+rd'$.

15	13	12	11	10 9	7 6	2 1	0
100	uimm[5]	01	rd'	uimm[4:0]	01		

c.srli rd', uimm $x[8+rd'] = x[8+rd'] \gg_u uimm$
Shift Right Logical Immediate. RV32IC y RV64IC.
 Se extiende a **srli** rd, rd, uimm, donde $rd=8+rd'$.

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 y RV64IC.
 Se extiende a **sub** rd, rd, rs2, donde $rd=8+rd'$ y $rs2=8+rs2'$.

15	10 9	7 6	5 4	2 1	0
100011	rd'	00	rs2'	01	

c.subw rd', rs2' $x[8+rd'] = \text{sext}((x[8+rd'] - x[8+rs2'])[31:0])$
Subtract Word. Solo RV64IC.
 Se extiende a **subw** rd, rd, rs2, donde $rd=8+rd'$ y $rs2=8+rs2'$.

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 y RV64IC.
 Se extiende a **sw** rs2, uimm(rs1), donde $rs2=8+rs2'$ y $rs1=8+rs1'$.

15	13 12	10 9	7 6	5 4	2 1	0
110	uimm[5:3]	rs1'	uimm[2:6]	rs2'	00	

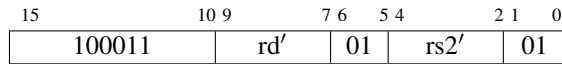
C.SWSP rs2, uimm(x2) $M[x[2] + uimm][31:0] = x[rs2]$
Store Word, Stack-Pointer Relative. RV32IC y RV64IC.
 Se extiende a **sw** rs2, uimm(x2).

15	13 12	7 6	2 1	0
110	uimm[5:2 7:6]	rs2	10	

C.XOR rd', rs2' $x[8+rd'] = x[8+rd'] \wedge x[8+rs2']$

Exclusive-OR. RV32IC y RV64IC.

Se extiende a **xor** rd, rd, rs2, donde $rd=8+rd'$ y $rs2=8+rs2'$.



call rd, symbol $x[rd] = pc+8; pc = \&symbol$

Call. Pseudoinstrucción, RV32I y RV64I.

Escribe la dirección de la siguiente instrucción ($pc+8$) en $x[rd]$, luego asigna *symbol* al *pc*.

Se extiende a **auipc** rd, offsetHi luego **jalr** rd, offsetLo(rd). Si *rd* es omitido, se asume x1.

csrr rd, csr $x[rd] = CSRs[csr]$

Control and Status Register Read. Pseudoinstrucción, RV32I y RV64I.

Copia el *control and status register csr* en $x[rd]$. Se extiende a **csrrs** rd, csr, x0.

csrc csr, rs1 $CSRs[csr] \&= \sim x[rs1]$

Control and Status Register Clear. Pseudoinstrucción, RV32I y RV64I.

Para cada bit en 1 en $x[rs1]$, poner en 0 el bit correspondiente en el *control and status register csr*. Se extiende a **csrrc** x0, csr, rs1.

csrci csr, zimm[4:0] $CSRs[csr] \&= \sim zimm$

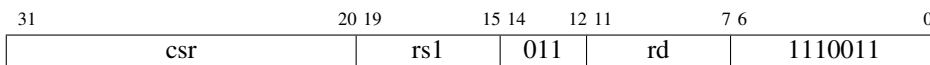
Control and Status Register Clear Immediate. Pseudoinstrucción, RV32I y RV64I.

Para cada bit en 1 en el inmediato de cinco bits zero-extended, poner en 0 el bit correspondiente en el *control and status register csr*. Se extiende a **csrrci** x0, csr, zimm.

csrrc rd, csr, rs1 $t = CSRs[csr]; CSRs[csr] = t \& \sim x[rs1]; x[rd] = t$

Control and Status Register Read and Clear. Tipo I, RV32I y RV64I.

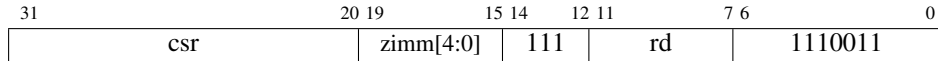
Sea *t* el valor del *control and status register csr*. Escribir el AND a nivel de bits de *t* y el complemento a uno de $x[rs1]$ en el *csr*, luego escribir *t* en $x[rd]$.



csrrci rd, csr, zimm[4:0] $t = \text{CSR}s[\text{csr}]; \text{CSR}s[\text{csr}] = t \& \sim \text{zimm}; x[\text{rd}] = t$

Control and Status Register Read and Clear Immediate. Tipo I, RV32I y RV64I.

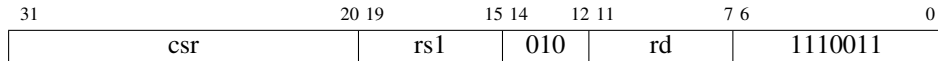
Sea t el valor del *control and status register csr*. Escribir el AND a nivel de bits de t y el complemento a uno del inmediato de cinco bits zero-extended zimm en el *csr*, luego escribir t en $x[\text{rd}]$ (Los bits del 5 en adelante en el *csr* no son modificados).



CSRrs rd, csr, rs1 $t = \text{CSR}s[\text{csr}]; \text{CSR}s[\text{csr}] = t | x[\text{rs1}]; x[\text{rd}] = t$

Control and Status Register Read and Set. Tipo I, RV32I y RV64I.

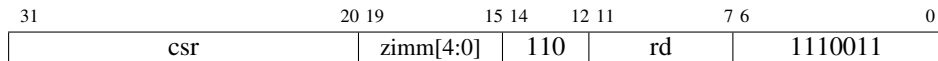
Sea t el valor del *control and status register csr*. Escribir el OR a nivel de bits de t y $x[\text{rs1}]$ en el *csr*, luego escribir t en $x[\text{rd}]$.



CSRrsi rd, csr, zimm[4:0] $t = \text{CSR}s[\text{csr}]; \text{CSR}s[\text{csr}] = t | \text{zimm}; x[\text{rd}] = t$

Control and Status Register Read and Set Immediate. Tipo I, RV32I y RV64I.

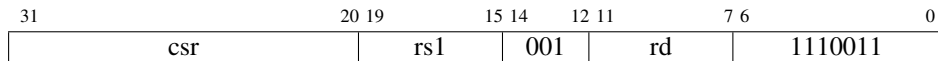
Sea t el valor del *control and status register csr*. Escribir el OR a nivel de bits de t y el inmediato de cinco bits zero-extended zimm en el *csr*, luego escribir t en $x[\text{rd}]$ (Los bits del 5 en adelante en el *csr* no son modificados).



CSRrw rd, csr, rs1 $t = \text{CSR}s[\text{csr}]; \text{CSR}s[\text{csr}] = x[\text{rs1}]; x[\text{rd}] = t$

Control and Status Register Read and Write. Tipo I, RV32I y RV64I.

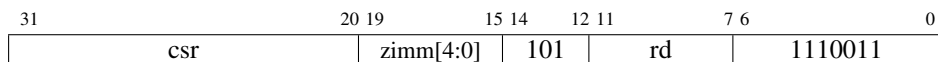
Sea t el valor del *control and status register csr*. Copiar $x[\text{rs1}]$ en el *csr*, luego escribir t en $x[\text{rd}]$.



CSRrwi rd, csr, zimm[4:0] $x[\text{rd}] = \text{CSR}s[\text{csr}]; \text{CSR}s[\text{csr}] = \text{zimm}$

Control and Status Register Read and Write Immediate. Tipo I, RV32I y RV64I.

Copia el *control and status register csr* en $x[\text{rd}]$, luego escribe el inmediato de cinco bits zero-extended zimm en el *csr*.



CSRS csr, rs1 CSRs[csr] |= x[rs1]

Control and Status Register Set. Pseudoinstrucción, RV32I y RV64I.

Para cada bit en 1 en $x[rs1]$, poner en 1 el bit correspondiente en el *control and status register csr*. Se extiende a **csrrs** $\times 0$, csr, rs1.

csrsi csr, zimm[4:0] CSRs[csr] |= zimm

Control and Status Register Set Immediate. Pseudoinstrucción, RV32I y RV64I.

Para cada bit en 1 en el inmediato de cinco bits zero-extended, poner en 1 el bit correspondiente en el *control and status register csr*. Se extiende a **csrrsi** $\times 0$, csr, zimm.

CSRW csr, rs1 CSRs[csr] = x[rs1]

Control and Status Register Write. Pseudoinstrucción, RV32I y RV64I.

Copia $x[rs1]$ en el *control and status register csr*. Se extiende a **csrrw** $\times 0$, csr, rs1.

csrwi csr, zimm[4:0] CSRs[csr] = zimm

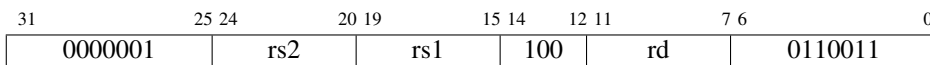
Control and Status Register Write Immediate. Pseudoinstrucción, RV32I y RV64I.

Copia el inmediato de cinco bits zero-extended en el *control and status register csr*. Se extiende a **csrrwi** $\times 0$, csr, zimm.

div rd, rs1, rs2 $x[rd] = x[rs1] \div_s x[rs2]$

Divide. Tipo R, RV32M y RV64M.

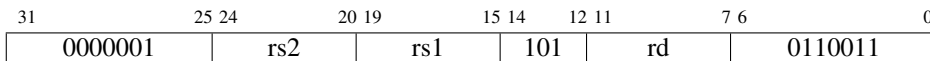
Divide $x[rs1]$ entre $x[rs2]$, redondeando hacia cero, tratando los valores como números de complemento a 2, y escribe el cociente en $x[rd]$.



divu rd, rs1, rs2 $x[rd] = x[rs1] \div_u x[rs2]$

Divide, Unsigned. Tipo R, RV32M y RV64M.

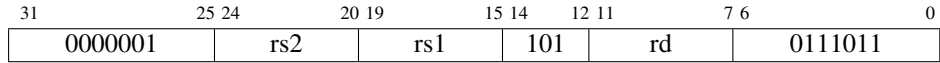
Divide $x[rs1]$ entre $x[rs2]$, redondeando hacia cero, tratando los valores como números sin signo, y escribe el cociente en $x[rd]$.



divuw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \div_u x[rs2][31:0])$

Divide Word, Unsigned. Tipo R, solo RV64M.

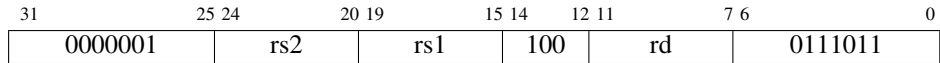
Divide los 32 bits más bajos de $x[rs1]$ entre los 32 bits más bajos de $x[rs2]$, redondeando hacia cero, tratando los valores como números sin signo, y escribe el cociente de 32 bits sign-extended en $x[rd]$.



divw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \div_s x[rs2][31:0])$

Divide Word. Tipo R, solo RV64M.

Divide los 32 bits más bajos de $x[rs1]$ entre los 32 bits más bajos de $x[rs2]$, redondeando hacia cero, tratando los valores como números de complemento a 2, y escribe el cociente de 32 bits sign-extended en $x[rd]$.

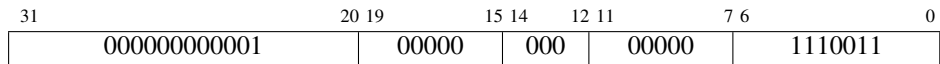


ebreak

RaiseException(Breakpoint)

Environment Breakpoint. Tipo I, RV32I y RV64I.

Hace una petición del debugger levantando una excepción de *Breakpoint*.

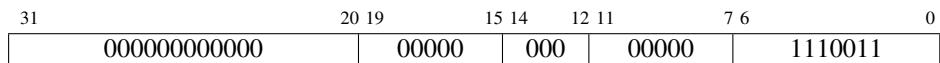


ecall

RaiseException(EnvironmentCall)

Environment Call. Tipo I, RV32I y RV64I.

Hace una petición del ambiente de ejecución levantando una excepción de *Environment Call*.



fabs.d rd, rs1

$f[rd] = |f[rs1]|$

Floating-point Absolute Value. Pseudoinstrucción, RV32D y RV64D.

Escribe en $f[rd]$ el valor absoluto del número de punto flotante de precisión doble en $f[rs1]$.

Se extiende a **fsgnjx.d** rd, rs1, rs1.

fabs.s rd, rs1

$f[rd] = |f[rs1]|$

Floating-point Absolute Value. Pseudoinstrucción, RV32F y RV64F.

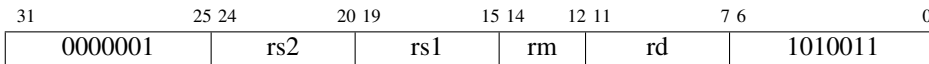
Escribe en $f[rd]$ el valor absoluto del número de punto flotante de precisión simple en $f[rs1]$.

Se extiende a **fsgnjx.s** rd, rs1, rs1.

fadd.d rd, rs1, rs2 $f[rd] = f[rs1] + f[rs2]$

Floating-point Add, Double-Precision. Tipo R, RV32D y RV64D.

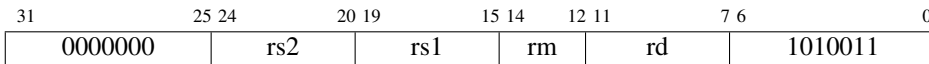
Suma los números de punto flotante de precisión doble en los registros $f[rs1]$ y $f[rs2]$ y escribe la suma redondeada de precisión doble en $f[rd]$.



fadd.s rd, rs1, rs2 $f[rd] = f[rs1] + f[rs2]$

Floating-point Add, Single-Precision. Tipo R, RV32F y RV64F.

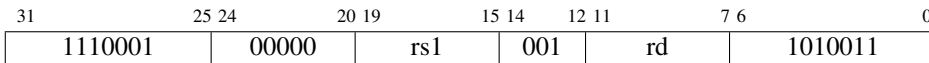
Suma los números de punto flotante de precisión simple en los registros $f[rs1]$ y $f[rs2]$ y escribe la suma redondeada de precisión simple en $f[rd]$.



fclass.d rd, rs1, rs2 $x[rd] = \text{classify}_d(f[rs1])$

Floating-point Classify, Double-Precision. Tipo R, RV32D y RV64D.

Escribe en $x[rd]$ una máscara que indica la clase del número de punto flotante de precisión doble en $f[rs1]$. Para interpretar el valor escrito en $x[rd]$, ver la descripción de **fclass.s**.

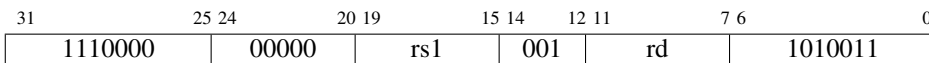


fclass.s rd, rs1, rs2 $x[rd] = \text{classify}_s(f[rs1])$

Floating-point Classify, Single-Precision. Tipo R, RV32F y RV64F.

Escribe en $x[rd]$ una máscara que indica la clase del número de punto flotante de precisión simple en $f[rs1]$. Exactamente un bit en $x[rd]$ es puesto en 1, de acuerdo a la siguiente tabla:

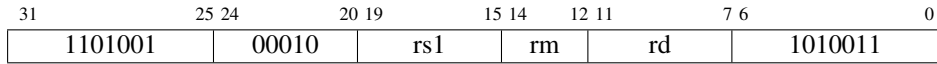
Bit en $x[rd]$	Significado
0	$f[rs1]$ es $-\infty$.
1	$f[rs1]$ es un número negativo normal.
2	$f[rs1]$ es un número negativo subnormal.
3	$f[rs1]$ es -0 .
4	$f[rs1]$ es $+0$.
5	$f[rs1]$ es un número positivo subnormal.
6	$f[rs1]$ es un número positivo normal.
7	$f[rs1]$ es $+\infty$.
8	$f[rs1]$ es un NaN señalizado.
9	$f[rs1]$ es un NaN simple.



fcvt.d.l rd, rs1, rs2 $f[rd] = f64_{s64}(x[rs1])$

Floating-point Convert to Double from Long. Tipo R, solo RV64D.

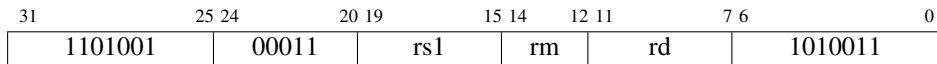
Convierte el entero de 64 bits de complemento a dos en $x[rs1]$ en un número de punto flotante de precisión doble y lo escribe en $f[rd]$.



fcvt.d.lu rd, rs1, rs2 $f[rd] = f64_{u64}(x[rs1])$

Floating-point Convert to Double from Unsigned Long. Tipo R, solo RV64D.

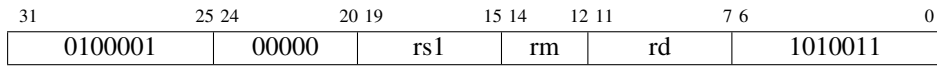
Convierte el entero de 64 bits sin signo en $x[rs1]$ en un número de punto flotante de precisión doble y lo escribe en $f[rd]$.



fcvt.d.s rd, rs1, rs2 $f[rd] = f64_{f32}(f[rs1])$

Floating-point Convert to Double from Single. Tipo R, RV32D y RV64D.

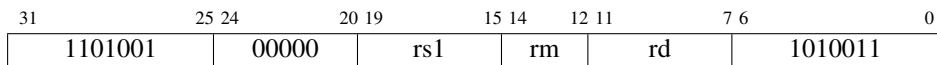
Convierte el número de punto flotante de precisión simple en $f[rs1]$ en un número de punto flotante de precisión doble y lo escribe en $f[rd]$.



fcvt.d.w rd, rs1, rs2 $f[rd] = f64_{s32}(x[rs1])$

Floating-point Convert to Double from Word. Tipo R, RV32D y RV64D.

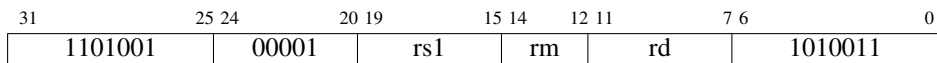
Convierte el entero de 32 bits de complemento a dos en $x[rs1]$ en un número de punto flotante de precisión doble y lo escribe en $f[rd]$.



fcvt.d.wu rd, rs1, rs2 $f[rd] = f64_{u32}(x[rs1])$

Floating-point Convert to Double from Unsigned Word. Tipo R, RV32D y RV64D.

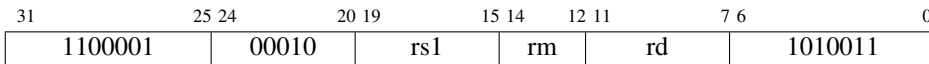
Convierte el entero de 32 bits sin signo en $x[rs1]$ en un número de punto flotante de precisión doble y lo escribe en $f[rd]$.



fcvt.l.d rd, rs1, rs2 $x[rd] = s64_{f64}(f[rs1])$

Floating-point Convert to Long from Double. Tipo R, solo RV64D.

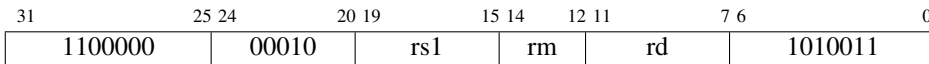
Convierte el número de punto flotante de precisión doble en el registro $f[rs1]$ en un entero de 64 bits de complemento a dos y lo escribe en $x[rd]$.



fcvt.l.s rd, rs1, rs2 $x[rd] = s64_{f32}(f[rs1])$

Floating-point Convert to Long from Single. Tipo R, solo RV64F.

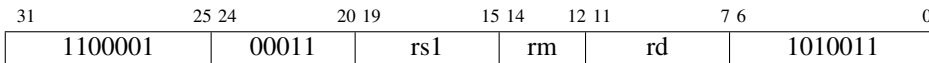
Convierte el número de punto flotante de precisión simple en el registro $f[rs1]$ en un entero de 64 bits de complemento a dos y lo escribe en $x[rd]$.



fcvt.lu.d rd, rs1, rs2 $x[rd] = u64_{f64}(f[rs1])$

Floating-point Convert to Unsigned Long from Double. Tipo R, solo RV64D.

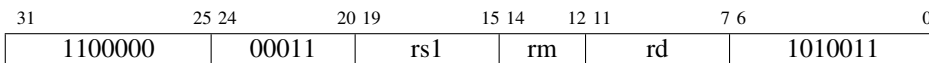
Convierte el número de punto flotante de precisión doble en el registro $f[rs1]$ en un entero de 64 bits sin signo y lo escribe en $x[rd]$.



fcvt.lu.s rd, rs1, rs2 $x[rd] = u64_{f32}(f[rs1])$

Floating-point Convert to Unsigned Long from Single. Tipo R, solo RV64F.

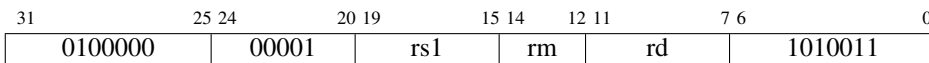
Convierte el número de punto flotante de precisión simple en el registro $f[rs1]$ en un entero de 64 bits sin signo y lo escribe en $x[rd]$.



fcvt.s.d rd, rs1, rs2 $f[rd] = f32_{f64}(f[rs1])$

Floating-point Convert to Single from Double. Tipo R, RV32D y RV64D.

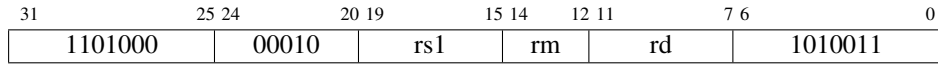
Convierte el número de punto flotante de precisión doble en $f[rs1]$ en un número de punto flotante de precisión simple y lo escribe en $f[rd]$.



fcvt.s.l rd, rs1, rs2 $f[rd] = f32_{s64}(x[rs1])$

Floating-point Convert to Single from Long. Tipo R, solo RV64F.

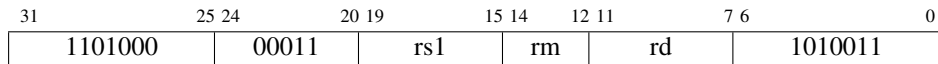
Convierte el entero de 64 bits de complemento a dos en $x[rs1]$ en un número de punto flotante de precisión simple y lo escribe en $f[rd]$.



fcvt.s.lu rd, rs1, rs2 $f[rd] = f32_{u64}(x[rs1])$

Floating-point Convert to Single from Unsigned Long. Tipo R, solo RV64F.

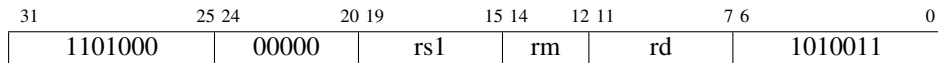
Convierte el entero de 64 bits sin signo en $x[rs1]$ en un número de punto flotante de precisión simple y lo escribe en $f[rd]$.



fcvt.s.w rd, rs1, rs2 $f[rd] = f32_{s32}(x[rs1])$

Floating-point Convert to Single from Word. Tipo R, RV32F y RV64F.

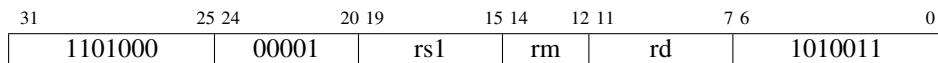
Convierte el entero de 32 bits de complemento a dos en $x[rs1]$ en un número de punto flotante de precisión simple y lo escribe en $f[rd]$.



fcvt.s.wu rd, rs1, rs2 $f[rd] = f32_{u32}(x[rs1])$

Floating-point Convert to Single from Unsigned Word. Tipo R, RV32F y RV64F.

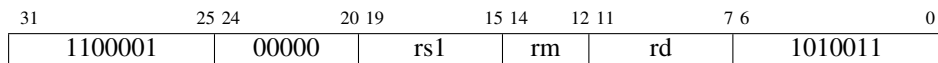
Convierte el entero de 32 bits sin signo en $x[rs1]$ en un número de punto flotante de precisión simple y lo escribe en $f[rd]$.



fcvt.w.d rd, rs1, rs2 $x[rd] = sext(s32_{f64}(f[rs1]))$

Floating-point Convert to Word from Double. Tipo R, RV32D y RV64D.

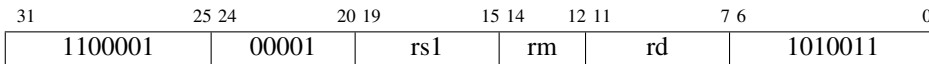
Convierte el número de punto flotante de precisión doble en el registro $f[rs1]$ en un entero de 32 bits de complemento a dos y escribe el resultado sign-extended en $x[rd]$.



fcvt.wu.d rd, rs1, rs2 $x[rd] = \text{sext}(u32_{f64}(f[rs1]))$

Floating-point Convert to Unsigned Word from Double. Tipo R, RV32D y RV64D.

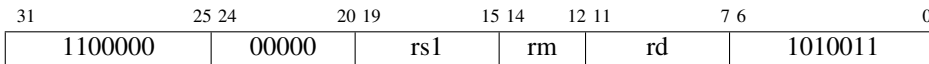
Convierte el número de punto flotante de precisión doble en el registro $f[rs1]$ en un entero de 32 bits sin signo y escribe el resultado sign-extended en $x[rd]$.



fcvt.w.s rd, rs1, rs2 $x[rd] = \text{sext}(s32_{f32}(f[rs1]))$

Floating-point Convert to Word from Single. Tipo R, RV32F y RV64F.

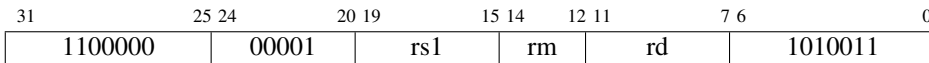
Convierte el número de punto flotante de precisión simple en el registro $f[rs1]$ en un entero de 32 bits de complemento a dos y escribe el resultado sign-extended en $x[rd]$.



fcvt.wu.s rd, rs1, rs2 $x[rd] = \text{sext}(u32_{f32}(f[rs1]))$

Floating-point Convert to Unsigned Word from Single. Tipo R, RV32F y RV64F.

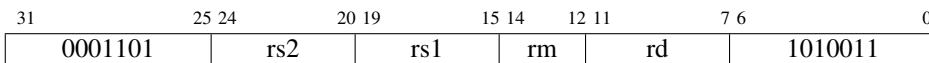
Convierte el número de punto flotante de precisión simple en el registro $f[rs1]$ en un entero de 32 bits sin signo y escribe el resultado sign-extended en $x[rd]$.



fdiv.d rd, rs1, rs2 $f[rd] = f[rs1] \div f[rs2]$

Floating-point Divide, Double-Precision. Tipo R, RV32D y RV64D.

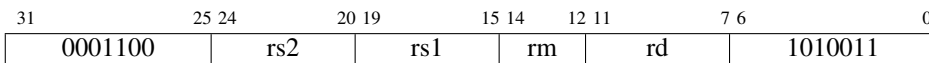
Divide el número de punto flotante de precisión doble en el registro $f[rs1]$ entre $f[rs2]$ y escribe el cociente redondeado de precisión doble en $f[rd]$.



fdiv.s rd, rs1, rs2 $f[rd] = f[rs1] \div f[rs2]$

Floating-point Divide, Single-Precision. Tipo R, RV32F y RV64F.

Divide el número de punto flotante de precisión simple en el registro $f[rs1]$ entre $f[rs2]$ y escribe el cociente redondeado de precisión simple en $f[rd]$.



fence pred, succ

Fence(pred, succ)

Fence Memory and I/O. Tipo I, RV32I y RV64I.

Vuelve los accesos previos a memoria e I/O en el conjunto *predecessor* observables a otros threads y dispositivos antes de que los accesos subsiguientes a memoria e I/O en el conjunto *successor* sean observables. Los bits 3, 2, 1, y 0 en estos conjuntos corresponden a *device input*, *device output*, *memory reads*, y *memory writes*, respectivamente. La instrucción **fence** r,rw, por ejemplo, ordena lecturas más antiguas con lecturas y escrituras más recientes, y se codifica con *pred*=0010 y *succ*=0011. Si los argumentos son omitidos, se asume un **fence** iorw, iorw completo.

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. Tipo I, RV32I y RV64I.

Vuelve *stores* a memoria de instrucciones observable a *instruction fetches* subsiguientes.

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.* Tipo R, RV32D y RV64D.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión doble en $f[rs1]$ es igual al número en $f[rs2]$, y 0 si no.

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.* Tipo R, RV32F y RV64F.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión simple en $f[rs1]$ es igual al número en $f[rs2]$, y 0 si no.

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.* Tipo I, RV32D y RV64D.

Carga un número de punto flotante de precisión doble de la dirección de memoria $x[rs1] + sign-extend(offset)$ y lo escribe en $f[rd]$.

Formas comprimidas: **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 $x[rd] = f[rs1] \leq f[rs2]$

Floating-point Less Than or Equal, Double-Precision. Tipo R, RV32D y RV64D.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión doble en $f[rs1]$ es menor o igual que el número en $f[rs2]$, y 0 si no.

31	25 24	20 19	15 14	12 11	7 6	0
1010001	rs2	rs1	000	rd	1010011	

fle.s rd, rs1, rs2 $x[rd] = f[rs1] \leq f[rs2]$

Floating-point Less Than or Equal, Single-Precision. Tipo R, RV32F y RV64F.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión simple en $f[rs1]$ es menor o igual que el número en $f[rs2]$, y 0 si no.

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. Tipo R, RV32D y RV64D.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión doble en $f[rs1]$ es menor que el número en $f[rs2]$, y 0 si no.

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. Tipo R, RV32F y RV64F.

Escribe 1 en $x[rd]$ si el número de punto flotante de precisión simple en $f[rs1]$ es menor que el número en $f[rs2]$, y 0 si no.

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. Tipo I, RV32F y RV64F.

Carga un número de punto flotante de precisión simple de la dirección de memoria $x[rs1] + sign-extend(offset)$ y lo escribe en $f[rd]$.

Formas comprimidas: **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 $f[rd] = f[rs1] \times f[rs2] + f[rs3]$

Floating-point Fused Multiply-Add, Double-Precision. Tipo R4, RV32D y RV64D.

Multiplica los números de punto flotante de precisión doble en $f[rs1]$ y $f[rs2]$, suma el producto sin redondear al número de punto flotante de precisión doble en $f[rs3]$, y escribe el resultado redondeado de precisión doble en $f[rd]$.

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 $f[rd] = f[rs1] \times f[rs2] + f[rs3]$

Floating-point Fused Multiply-Add, Single-Precision. Tipo R4, RV32F y RV64F.

Multiplica los números de punto flotante de precisión simple en $f[rs1]$ y $f[rs2]$, suma el producto sin redondear al número de punto flotante de precisión simple en $f[rs3]$, y escribe el resultado redondeado de precisión simple en $f[rd]$.

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. Tipo R, RV32D y RV64D.

Copia el mayor de los números de punto flotante de precisión doble de los registros $f[rs1]$ y $f[rs2]$ a $f[rd]$.

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. Tipo R, RV32F y RV64F.

Copia el mayor de los números de punto flotante de precisión simple de los registros $f[rs1]$ y $f[rs2]$ a $f[rd]$.

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. Tipo R, RV32D y RV64D.

Copia el menor de los números de punto flotante de precisión doble de los registros $f[rs1]$ y $f[rs2]$ a $f[rd]$.

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. Tipo R, RV32F y RV64F.

Copia el menor de los números de punto flotante de precisión simple de los registros $f[rs1]$ y $f[rs2]$ a $f[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0010100	rs2	rs1	000	rd	1010011	

fmsub.d rd, rs1, rs2, rs3 $f[rd] = f[rs1] \times f[rs2] - f[rs3]$

Floating-point Fused Multiply-Subtract, Double-Precision. Tipo R4, RV32D y RV64D.

Multiplica los números de punto flotante de precisión doble en $f[rs1]$ y $f[rs2]$, resta el número de punto flotante de precisión doble en $f[rs3]$ del producto sin redondear, y escribe el resultado redondeado de precisión doble en $f[rd]$.

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 $f[rd] = f[rs1] \times f[rs2] - f[rs3]$

Floating-point Fused Multiply-Subtract, Single-Precision. Tipo R4, RV32F y RV64F.

Multiplica los números de punto flotante de precisión simple en $f[rs1]$ y $f[rs2]$, resta el número de punto flotante de precisión simple en $f[rs3]$ del producto sin redondear, y escribe el resultado redondeado de precisión simple en $f[rd]$.

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 $f[rd] = f[rs1] \times f[rs2]$

Floating-point Multiply, Double-Precision. Tipo R, RV32D y RV64D.

Multiplica los números de punto flotante de precisión doble en los registros $f[rs1]$ y $f[rs2]$ y escribe el producto redondeado de precisión doble en $f[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0001001	rs2	rs1	rm	rd	1010011	

fmul.s rd, rs1, rs2 $f[rd] = f[rs1] \times f[rs2]$

Floating-point Multiply, Single-Precision. Tipo R, RV32F y RV64F.

Multiplica los números de punto flotante de precisión simple en los registros $f[rs1]$ y $f[rs2]$ y escribe el producto redondeado de precisión simple en $f[rd]$.

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. Pseudoinstrucción, RV32D y RV64D.

Copia el número de punto flotante de precisión doble en $f[rs1]$ a $f[rd]$. Se extiende a **fsgnj.d** rd, rs1, rs1.

fmv.d.x rd, rs1, rs2 $f[rd] = x[rs1][63:0]$

Floating-point Move Doubleword from Integer. Tipo R, solo RV64D.

Copia el número de punto flotante de precisión doble en el registro $x[rs1]$ a $f[rd]$.

<small>31</small>	<small>25 24</small>	<small>20 19</small>	<small>15 14</small>	<small>12 11</small>	<small>7 6</small>	<small>0</small>
1111001	00000	rs1	000	rd	1010011	

fmv.s rd, rs1 $f[rd] = f[rs1]$

Floating-point Move. Pseudoinstrucción, RV32F y RV64F.

Copia el número de punto flotante de precisión simple en $f[rs1]$ a $f[rd]$. Se extiende a **fsgnj.s** rd, rs1, rs1.

fmv.w.x rd, rs1, rs2 $f[rd] = x[rs1][31:0]$

Floating-point Move Word from Integer. Tipo R, RV32F y RV64F.

Copia el número de punto flotante de precisión simple en el registro $x[rs1]$ a $f[rd]$.

<small>31</small>	<small>25 24</small>	<small>20 19</small>	<small>15 14</small>	<small>12 11</small>	<small>7 6</small>	<small>0</small>
1111000	00000	rs1	000	rd	1010011	

fmv.x.d rd, rs1, rs2 $x[rd] = f[rs1][63:0]$

Floating-point Move Doubleword to Integer. Tipo R, solo RV64D.

Copia el número de punto flotante de precisión doble en el registro $f[rs1]$ a $x[rd]$.

<small>31</small>	<small>25 24</small>	<small>20 19</small>	<small>15 14</small>	<small>12 11</small>	<small>7 6</small>	<small>0</small>
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. Tipo R, RV32F y RV64F.

Copia el número de punto flotante de precisión simple en el registro $f[rs1]$ a $x[rd]$, extendiendo en signo el resultado para RV64F.

<small>31</small>	<small>25 24</small>	<small>20 19</small>	<small>15 14</small>	<small>12 11</small>	<small>7 6</small>	<small>0</small>
1110000	00000	rs1	000	rd	1010011	

fneg.d rd, rs1 $f[rd] = -f[rs1]$

Floating-point Negate. Pseudoinstrucción, RV32D y RV64D.

Escribe el opuesto del número de punto flotante de precisión doble en $f[rs1]$ a $f[rd]$. Se extiende a **fsgnj.d** rd, rs1, rs1.

fneg.s rd, rs1 $f[rd] = -f[rs1]$

Floating-point Negate. Pseudoinstrucción, RV32F y RV64F.

Escribe el opuesto del número de punto flotante de precisión simple en $f[rs1]$ a $f[rd]$. Se extiende a **fsgnj.s** rd, rs1, rs1.

fnmadd.d rd, rs1, rs2, rs3 $f[rd] = -f[rs1] \times f[rs2] - f[rs3]$

Floating-point Fused Negative Multiply-Add, Double-Precision. Tipo R4, RV32D y RV64D. Multiplica los números de punto flotante de precisión doble en $f[rs1]$ y $f[rs2]$, niega el resultado, resta el número de punto flotante de precisión doble en $f[rs3]$ del producto sin redondear, y escribe el resultado redondeado de precisión doble en $f[rd]$.

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 $f[rd] = -f[rs1] \times f[rs2] - f[rs3]$

Floating-point Fused Negative Multiply-Add, Single-Precision. Tipo R4, RV32F y RV64F.

Multiplica los números de punto flotante de precisión simple en $f[rs1]$ y $f[rs2]$, niega el resultado, resta el número de punto flotante de precisión simple en $f[rs3]$ del producto sin redondear, y escribe el resultado redondeado de precisión simple en $f[rd]$.

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 $f[rd] = -f[rs1] \times f[rs2] + f[rs3]$

Floating-point Fused Negative Multiply-Subtract, Double-Precision. Tipo R4, RV32D y RV64D.

Multiplica los números de punto flotante de precisión doble en $f[rs1]$ y $f[rs2]$, niega el resultado, suma el producto sin redondear al número de punto flotante de precisión doble en $f[rs3]$, y escribe el resultado redondeado de precisión doble en $f[rd]$.

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 $f[rd] = -f[rs1] \times f[rs2] + f[rs3]$
Floating-point Fused Negative Multiply-Subtract, Single-Precision. Tipo R4, RV32F y RV64F.

Multiplica los números de punto flotante de precisión simple en $f[rs1]$ y $f[rs2]$, niega el resultado, suma el producto sin redondear al número de punto flotante de precisión simple en $f[rs3]$, y escribe el resultado redondeado de precisión simple en $f[rd]$.

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. Pseudoinstrucción, RV32F y RV64F.
 Copia el *floating-point control and status register* a $x[rd]$. Se extiende a **csrrs** rd, fcsr, x0.

frflags rd $x[rd] = CSRs[fflags]$
Floating-point Read Exception Flags. Pseudoinstrucción, RV32F y RV64F.
 Copia las banderas de excepción de punto flotante a $x[rd]$. Se extiende a **csrrs** rd, fflags, x0.

frrm rd $x[rd] = CSRs[frm]$
Floating-point Read Rounding Mode. Pseudoinstrucción, RV32F y RV64F.
 Copia el modo de redondeo de punto flotante a $x[rd]$. Se extiende a **csrrs** rd, frm, x0.

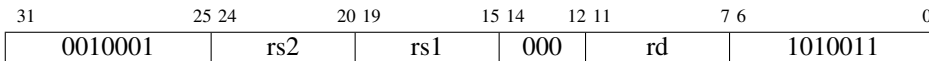
fscsr rd, rs1 $t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t$
Floating-point Swap Control and Status Register. Pseudoinstrucción, RV32F y RV64F.
 Copia $x[rs1]$ al *floating-point control and status register*, luego copia el valor previo del *floating-point control and status register* a $x[rd]$. Se extiende a **csrrw** rd, fcsr, rs1. Si *rd* es omitido, se asume x0.

fsd rs2, offset(rs1) $M[x[rs1] + sext(offset)] = f[rs2][63:0]$
Floating-point Store Doubleword. Tipo S, RV32D y RV64D.
 Almacena el número de punto flotante de precisión doble del registro $f[rs2]$ a memoria en la dirección $x[rs1] + sign-extend(offset)$.
 Formas comprimidas: **c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)

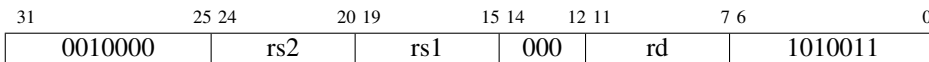
31	25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100111	

fsflags rd, rs1 $t = \text{CSRs}[\text{fflags}]; \text{CSRs}[\text{fflags}] = x[\text{rs1}]; x[\text{rd}] = t$
Floating-point Swap Exception Flags. Pseudoinstrucción, RV32F y RV64F.
 Copia $x[\text{rs1}]$ al *floating-point exception flags register*, luego copia el valor previo del *floating-point exception flags register* a $x[\text{rd}]$. Se extiende a **csrrw** rd, fflags, rs1. Si *rd* es omitido, se asume x0.

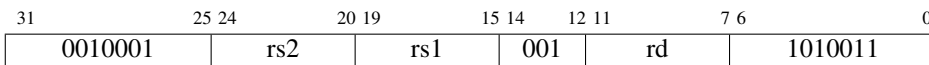
fsgnj.d rd, rs1, rs2 $f[\text{rd}] = \{f[\text{rs2}][63], f[\text{rs1}][62:0]\}$
Floating-point Sign Inject, Double-Precision. Tipo R, RV32D y RV64D.
 Construye un nuevo número de punto flotante de precisión doble del exponente y significando de $f[\text{rs1}]$, tomando el signo de $f[\text{rs2}]$, y lo escribe en $f[\text{rd}]$.



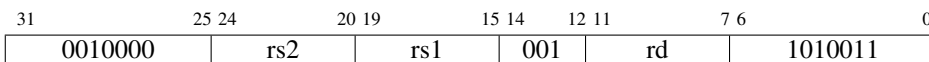
fsgnj.s rd, rs1, rs2 $f[\text{rd}] = \{f[\text{rs2}][31], f[\text{rs1}][30:0]\}$
Floating-point Sign Inject, Single-Precision. Tipo R, RV32F y RV64F.
 Construye un nuevo número de punto flotante de precisión simple del exponente y significando de $f[\text{rs1}]$, tomando el signo de $f[\text{rs2}]$, y lo escribe en $f[\text{rd}]$.



fsgnjn.d rd, rs1, rs2 $f[\text{rd}] = \{\sim f[\text{rs2}][63], f[\text{rs1}][62:0]\}$
Floating-point Sign Inject-Negate, Double-Precision. Tipo R, RV32D y RV64D.
 Construye un nuevo número de punto flotante de precisión doble del exponente y significando de $f[\text{rs1}]$, tomando el signo opuesto de $f[\text{rs2}]$, y lo escribe en $f[\text{rd}]$.



fsgnjn.s rd, rs1, rs2 $f[\text{rd}] = \{\sim f[\text{rs2}][31], f[\text{rs1}][30:0]\}$
Floating-point Sign Inject-Negate, Single-Precision. Tipo R, RV32F y RV64F.
 Construye un nuevo número de punto flotante de precisión simple del exponente y significando de $f[\text{rs1}]$, tomando el signo opuesto de $f[\text{rs2}]$, y lo escribe en $f[\text{rd}]$.



fsgnjx.d rd, rs1, rs2 $f[rd] = \{f[rs1][63] \wedge f[rs2][63], f[rs1][62:0]\}$
Floating-point Sign Inject-XOR, Double-Precision. Tipo R, RV32D y RV64D.

Construye un nuevo número de punto flotante de precisión doble del exponente y significando de $f[rs1]$, tomando el signo del XOR de los signos de $f[rs1]$ y $f[rs2]$, y lo escribe en $f[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0010001	rs2	rs1	010	rd	1010011	

fsgnjx.s rd, rs1, rs2 $f[rd] = \{f[rs1][31] \wedge f[rs2][31], f[rs1][30:0]\}$
Floating-point Sign Inject-XOR, Single-Precision. Tipo R, RV32F y RV64F.

Construye un nuevo número de punto flotante de precisión simple del exponente y significando de $f[rs1]$, tomando el signo del XOR de los signos de $f[rs1]$ y $f[rs2]$, y lo escribe en $f[rd]$.

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. Tipo R, RV32D y RV64D.

Calcula la raíz cuadrada del número de punto flotante de precisión doble en el registro $f[rs1]$ y escribe el resultado redondeado de precisión doble en $f[rd]$.

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. Tipo R, RV32F y RV64F.

Calcula la raíz cuadrada del número de punto flotante de precisión simple en el registro $f[rs1]$ y escribe el resultado redondeado de precisión simple en $f[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0101100	00000	rs1	rm	rd	1010011	

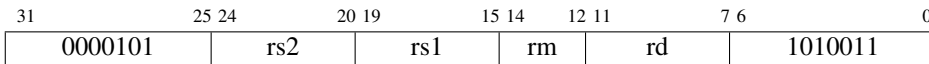
fsrm rd, rs1 $t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t$
Floating-point Swap Rounding Mode. Pseudoinstrucción, RV32F y RV64F.

Copia $x[rs1]$ al *floating-point rounding mode register*, luego copia el valor previo del *floating-point rounding mode register* a $x[rd]$. Se extiende a **csrrw** rd, frm, rs1. Si *rd* es omitido, se asume $x0$.

fsub.d rd, rs1, rs2 $f[rd] = f[rs1] - f[rs2]$

Floating-point Subtract, Double-Precision. Tipo R, RV32D y RV64D.

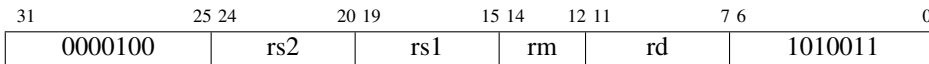
Resta el número de punto flotante de precisión doble en el registro $f[rs2]$ de $f[rs1]$ y escribe la diferencia de precisión doble en $f[rd]$.



fsub.s rd, rs1, rs2 $f[rd] = f[rs1] - f[rs2]$

Floating-point Subtract, Single-Precision. Tipo R, RV32F y RV64F.

Resta el número de punto flotante de precisión simple en el registro $f[rs2]$ de $f[rs1]$ y escribe la diferencia de precisión simple en $f[rd]$.

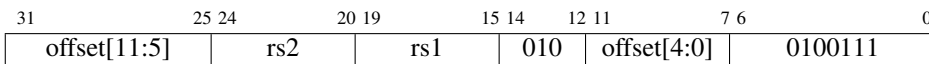


fsw rs2, offset(rs1) $M[x[rs1] + sext(offset)] = f[rs2][31:0]$

Floating-point Store Word. Tipo S, RV32F y RV64F.

Almacena el número de punto flotante de precisión simple del registro $f[rs2]$ a memoria en la dirección $x[rs1] + sign-extend(offset)$.

Formas comprimidas: **c.fswsp** rs2, offset; **c.fsw** rs2, offset(rs1)



j offset $pc += sext(offset)$

Jump. Pseudoinstrucción, RV32I y RV64I.

Escribe al pc su valor actual más el $offset$ extendido en signo. Se extiende a **jal** $x0$, $offset$.

jal rd, offset $x[rd] = pc+4; pc += sext(offset)$

Jump and Link. Tipo J, RV32I y RV64I.

Escribe la dirección de la siguiente instrucción ($pc+4$) en $x[rd]$, luego asigna al pc su valor actual más el $offset$ extendido en signo. Si rd es omitido, se asume $x1$.

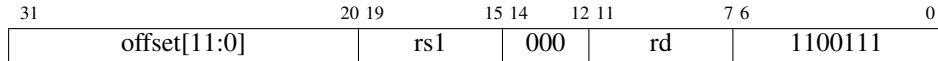
Formas comprimidas: **c.j** offset; **c.jal** offset



jalr rd, offset(rs1) $t = pc + 4; pc = (x[rs1] + sext(offset)) \& \sim 1; x[rd] = t$
Jump and Link Register. Tipo I, RV32I y RV64I.

Escribe $x[rs1] + sign-extend(offset)$ al *pc*, enmascarando a cero el bit menos significativo de la dirección calculada, luego escribe el valor previo del *pc*+4 en $x[rd]$. Si *rd* es omitido, se asume *x1*.

Formas comprimidas: **c.jr** rs1; **c.jalr** rs1



jr rs1 $pc = x[rs1]$

Jump Register. Pseudoinstrucción, RV32I y RV64I.

Escribe $x[rs1]$ al *pc*. Se extiende a **jalr** *x0*, 0(rs1).

la rd, symbol $x[rd] = \&symbol$

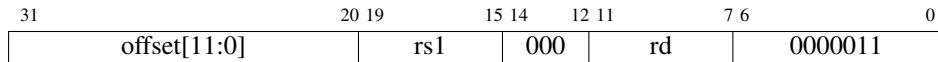
Load Address. Pseudoinstrucción, RV32I y RV64I.

Carga la dirección de *symbol* en $x[rd]$. Cuando se ensambla *position-independent code*, se extiende a un load del *Global Offset Table*: para RV32I, **auipc** rd, offsetHi luego **lw** rd, offsetLo(rd); para RV64I, **auipc** rd, offsetHi luego **ld** rd, offsetLo(rd). En cualquier otro caso, se extiende a **auipc** rd, offsetHi luego **addi** rd, rd, offsetLo.

lb rd, offset(rs1) $x[rd] = sext(M[x[rs1] + sext(offset)][7:0])$

Load Byte. Tipo I, RV32I y RV64I.

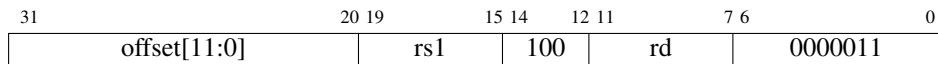
Carga un byte de memoria en la dirección $x[rs1] + sign-extend(offset)$ y lo escribe en $x[rd]$, extendiendo en signo el resultado.



lbu rd, offset(rs1) $x[rd] = M[x[rs1] + sext(offset)][7:0]$

Load Byte, Unsigned. Tipo I, RV32I y RV64I.

Carga un byte de memoria en la dirección $x[rs1] + sign-extend(offset)$ y lo escribe en $x[rd]$, extendiendo con cero el resultado.

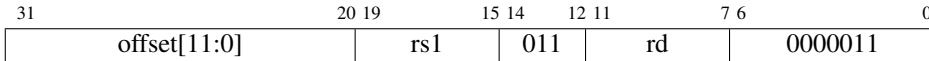


ld rd, offset(rs1) $x[rd] = M[x[rs1] + sext(offset)][63:0]$

Load Doubleword. Tipo I, solo RV64I.

Carga ocho bytes de memoria en la dirección $x[rs1] + sign-extend(offset)$ y los escribe en $x[rd]$.

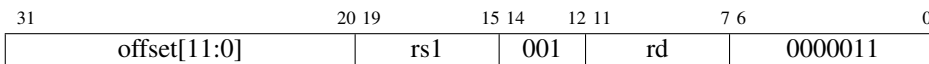
Formas comprimidas: **c.ldsp** rd, offset; **c.ld** rd, offset(rs1)



lh rd, offset(rs1) $x[rd] = sext(M[x[rs1] + sext(offset)][15:0])$

Load Halfword. Tipo I, RV32I y RV64I.

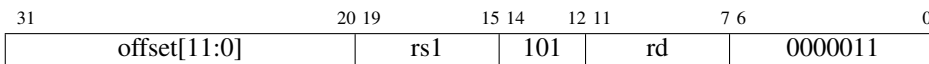
Carga dos bytes de memoria en la dirección $x[rs1] + sign-extend(offset)$ y los escribe en $x[rd]$, extendiendo en signo el resultado.



lhu rd, offset(rs1) $x[rd] = M[x[rs1] + sext(offset)][15:0]$

Load Halfword, Unsigned. Tipo I, RV32I y RV64I.

Carga dos bytes de memoria en la dirección $x[rs1] + sign-extend(offset)$ y los escribe en $x[rd]$, extendiendo con cero el resultado.



li rd, immediate $x[rd] = immediate$

Load Immediate. Pseudoinstrucción, RV32I y RV64I.

Carga una constante en $x[rd]$, usando tan pocas instrucciones como sea posible. Para RV32I, se extiende a **lui** y/o **addi**; para RV64I, es tan largo como **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

lla rd, symbol $x[rd] = \&symbol$

Load Local Address. Pseudoinstrucción, RV32I y RV64I.

Carga la dirección de *symbol* en $x[rd]$. Se extiende a **auipc** rd, offsetHi luego **addi** rd, rd, offsetLo.

lr.d rd, (rs1) $x[rd] = \text{LoadReserved64}(M[x[rs1]])$

Load-Reserved Doubleword. Tipo R, solo RV64A.

Carga los ocho bytes de memoria en la dirección $x[rs1]$, los escribe en $x[rd]$, y registra una reserva en ese doubleword de memoria.

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) $x[rd] = \text{LoadReserved32}(M[x[rs1]])$

Load-Reserved Word. Tipo R, RV32A y RV64A.

Carga los cuatro bytes de memoria en la dirección $x[rs1]$, los escribe en $x[rd]$, extendiendo en signo el resultado, y registra una reserva en ese word de memoria.

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) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][31:0])$

Load Word. Tipo I, RV32I y RV64I.

Carga cuatro bytes de memoria en la dirección $x[rs1] + \text{sign-extend}(\text{offset})$ y los escribe en $x[rd]$. Para RV64I, el resultado es extendido en signo.

Formas comprimidas: **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) $x[rd] = M[x[rs1] + \text{sext}(\text{offset})][31:0]$

Load Word, Unsigned. Tipo I, solo RV64I.

Carga cuatro bytes de memoria en la dirección $x[rs1] + \text{sign-extend}(\text{offset})$ y los escribe en $x[rd]$, extendiendo con cero el resultado.

31	20 19	15 14	12 11	7 6	0
offset[11:0]		rs1	110	rd	0000011

lui rd, immediate $x[rd] = \text{sext}(\text{immediate}[31:12] \ll 12)$

Load Upper Immediate. Tipo U, RV32I y RV64I.

Escribe el *inmediato* de 20 bits extendido en signo, corrido a la izquierda por 12 bits, en $x[rd]$, volviendo cero los 12 bits más bajos.

Forma comprimida: **c.lui** rd, imm

31	12 11	7 6	0
immediate[31:12]		rd	0110111

mret

ExceptionReturn(Machine)

Machine-mode Exception Return. Tipo R, arquitecturas privilegiadas RV32I y RV64I.

Retorna de un manejador de excepción en modo máquina. Escribe CSRs[mepc] al *pc*, CSRs[mstatus].MPP al modo de privilegio, CSRs[mstatus].MPIE a CSRs[mstatus].MIE, y 1 a CSRs[mstatus].MPIE; y, si el modo usuario es soportado, escribe 0 a CSRs[mstatus].MPP.

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.* Tipo R, RV32M y RV64M.

Multiplica $x[rs1]$ por $x[rs2]$ y escribe el producto en $x[rd]$. Overflow aritmético ignorado.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	000	rd	0110011	

mulh rd, rs1, rs2 $x[rd] = (x[rs1] \times_s x[rs2]) \gg_s \text{XLEN}$ *Multiply High.* Tipo R, RV32M y RV64M.

Multiplica $x[rs1]$ por $x[rs2]$, tratando los valores como números de complemento a dos, y escribe la mitad alta del producto a $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	001	rd	0110011	

mulhsu rd, rs1, rs2 $x[rd] = (x[rs1] \times_u x[rs2]) \gg_s \text{XLEN}$ *Multiply High Signed-Unsigned.* Tipo R, RV32M y RV64M.

Multiplica $x[rs1]$ por $x[rs2]$, tratando a $x[rs1]$ como un número de complemento a dos, y a $x[rs2]$ como un número sin signo, y escribe la mitad alta del producto a $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	010	rd	0110011	

mulhu rd, rs1, rs2 $x[rd] = (x[rs1] \times_u x[rs2]) \gg_u \text{XLEN}$ *Multiply High Unsigned.* Tipo R, RV32M y RV64M.

Multiplica $x[rs1]$ por $x[rs2]$, tratando los valores como números sin signo, y escribe la mitad alta del producto a $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	011	rd	0110011	

mulw rd, rs1, rs2 $x[rd] = \text{sext}((x[rs1] \times x[rs2])[31:0])$

Multiply Word. Tipo R, solo RV64M.

Multiplifica $x[rs1]$ por $x[rs2]$, trunca el producto a 32 bits, y escribe el resultado sign-extended en $x[rd]$. Overflow aritmético 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. Pseudoinstrucción, RV32I y RV64I.

Copia el registro $x[rs1]$ a $x[rd]$. Se extiende a **addi** rd, rs1, 0.

neg rd, rs2 $x[rd] = -x[rs2]$

Negate. Pseudoinstrucción, RV32I y RV64I.

Escribe el complemento a dos de $x[rs2]$ a $x[rd]$. Se extiende a **sub** rd, x0, rs2.

negw rd, rs2 $x[rd] = \text{sext}((-x[rs2])[31:0])$

Negate Word. Pseudoinstrucción, Solo RV64I.

Calcula el complemento a dos de $x[rs2]$, trunca el resultado a 32 bits, y escribe el resultado sign-extended en $x[rd]$. Se extiende a **subw** rd, x0, rs2.

nop *Nothing*

No operation. Pseudoinstrucción, RV32I y RV64I.

Solo incrementa el *pc* a la siguiente instrucción. Se extiende a **addi** x0, x0, 0.

not rd, rs1 $x[rd] = \sim x[rs1]$

NOT. Pseudoinstrucción, RV32I y RV64I.

Escribe el complemento a uno de $x[rs1]$ a $x[rd]$. Se extiende a **xori** rd, rs1, -1.

or rd, rs1, rs2 $x[rd] = x[rs1] | x[rs2]$

OR. Tipo R, RV32I y RV64I.

Calcula el OR inclusivo a nivel de bits de los registros $x[rs1]$ y $x[rs2]$ y escribe el resultado en $x[rd]$.

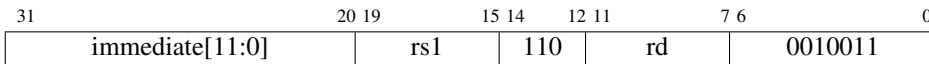
Forma comprimida: **c.or** rd, rs2

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	110	rd		0110011

ori rd, rs1, immediate $x[rd] = x[rs1] \mid \text{sext}(\text{immediate})$

OR Immediate. Tipo I, RV32I y RV64I.

Calcula el OR inclusivo a nivel de bits del *inmediato* sign-extended y el registro $x[rs1]$ y escribe el resultado en $x[rd]$.



rdcycle rd $x[rd] = \text{CSRs}[\text{cycle}]$

Read Cycle Counter. Pseudoinstrucción, RV32I y RV64I.

Escribe el número de ciclos que han pasado a $x[rd]$. Se extiende a **csrrs** rd, cycle, x0.

rdcycleh rd $x[rd] = \text{CSRs}[\text{cycleh}]$

Read Cycle Counter High. Pseudoinstrucción, Solo RV32I.

Escribe el número de ciclos que han pasado, corrido a la derecha por 32 bits, a $x[rd]$. Se extiende a **csrrs** rd, cycleh, x0.

rdinstret rd $x[rd] = \text{CSRs}[\text{instret}]$

Read Instructions-Retired Counter. Pseudoinstrucción, RV32I y RV64I.

Escribe el número de instrucciones que han sido retiradas a $x[rd]$. Se extiende a **csrrs** rd, instret, x0.

rdinstreth rd $x[rd] = \text{CSRs}[\text{instreth}]$

Read Instructions-Retired Counter High. Pseudoinstrucción, Solo RV32I.

Escribe el número de instrucciones que han sido retiradas, corrido a la derecha por 32 bits, a $x[rd]$. Se extiende a **csrrs** rd, instreth, x0.

rdtime rd $x[rd] = \text{CSRs}[\text{time}]$

Read Time. Pseudoinstrucción, RV32I y RV64I.

Escribe el tiempo actual a $x[rd]$. La frecuencia del temporizador depende de la plataforma. Se extiende a **csrrs** rd, time, x0.

rdtimeh rd $x[rd] = \text{CSRs}[\text{timeh}]$

Read Time High. Pseudoinstrucción, Solo RV32I.

Escribe el tiempo actual, corrido a la derecha por 32 bits, a $x[rd]$. La frecuencia del temporizador depende de la plataforma. Se extiende a **csrrs** rd, timeh, x0.

rem rd, rs1, rs2 $x[rd] = x[rs1] \%_s x[rs2]$

Remainder. Tipo R, RV32M y RV64M.

Divide $x[rs1]$ entre $x[rs2]$, redondeando hacia cero, tratando los valores como números de complemento a 2, y escribe el residuo en $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	110	rd	0110011	

remu rd, rs1, rs2 $x[rd] = x[rs1] \%_u x[rs2]$

Remainder, Unsigned. Tipo R, RV32M y RV64M.

Divide $x[rs1]$ entre $x[rs2]$, redondeando hacia cero, tratando los valores como números sin signo, y escribe el residuo en $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	111	rd	0110011	

remuw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \%_u x[rs2][31:0])$

Remainder Word, Unsigned. Tipo R, solo RV64M.

Divide los 32 bits más bajos de $x[rs1]$ entre los 32 bits más bajos de $x[rs2]$, redondeando hacia cero, tratando los valores como números sin signo, y escribe el residuo de 32 bits, sign-extended, en $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	111	rd	0111011	

remw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \%_s x[rs2][31:0])$

Remainder Word. Tipo R, solo RV64M.

Divide los 32 bits más bajos de $x[rs1]$ entre los 32 bits más bajos de $x[rs2]$, redondeando hacia cero, tratando los valores como números de complemento a dos, y escribe el residuo de 32 bits, sign-extended, en $x[rd]$.

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	110	rd	0111011	

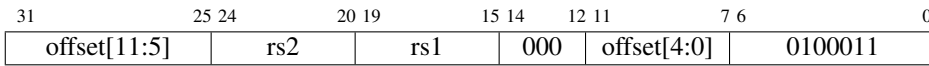
ret

pc = x[1]

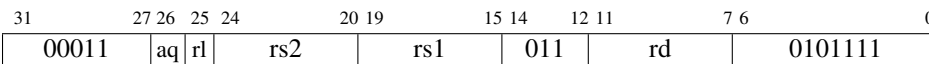
Return. Pseudoinstrucción, RV32I y RV64I.

Retorna de una subrutina. Se extiende a **jalr** x0, 0(x1).

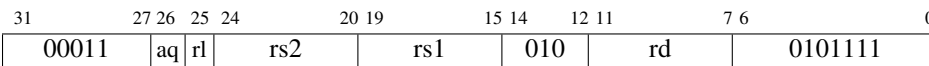
sb $rs2, \text{offset}(rs1) \quad M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][7:0]$
Store Byte. Tipo S, RV32I y RV64I.
 Almacena el byte menos significativo del registro $x[rs2]$ a memoria en la dirección $x[rs1] + \text{sign-extend}(\text{offset})$.



sc.d $rd, rs2, (rs1) \quad x[rd] = \text{StoreConditional64}(M[x[rs1]], x[rs2])$
Store-Conditional Doubleword. Tipo R, solo RV64A.
 Almacena los ocho bytes del registro $x[rs2]$ a memoria en la dirección $x[rs1]$, siempre y cuando haya una reserva de load en esa dirección de memoria. Escribe 0 en $x[rd]$ si tuvo éxito, o un código de error distinto de cero en caso contrario.

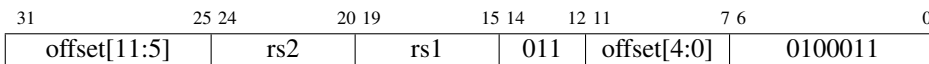


SC.W $rd, rs2, (rs1) \quad x[rd] = \text{StoreConditional32}(M[x[rs1]], x[rs2])$
Store-Conditional Word. Tipo R, RV32A y RV64A.
 Almacena los cuatro bytes del registro $x[rs2]$ a memoria en la dirección $x[rs1]$, siempre y cuando haya una reserva de load en esa dirección de memoria. Escribe 0 en $x[rd]$ si tuvo éxito, o un código de error distinto de cero en caso contrario.



sd $rs2, \text{offset}(rs1) \quad M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][63:0]$
Store Doubleword. Tipo S, solo RV64I.
 Almacena los ocho bytes del registro $x[rs2]$ a memoria en la dirección $x[rs1] + \text{sign-extend}(\text{offset})$.

Formas comprimidas: **c.sdsp** $rs2, \text{offset}$; **c.sd** $rs2, \text{offset}(rs1)$



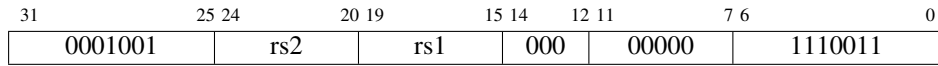
seqz $rd, rs1 \quad x[rd] = (x[rs1] == 0)$
Set if Equal to Zero. Pseudoinstrucción, RV32I y RV64I.
 Escribe 1 en $x[rd]$ si $x[rs1]$ es igual a 0, ó 0 si no. Se extiende a **sltiu** $rd, rs1, 1$.

sext.w $rd, rs1 \quad x[rd] = \text{sext}(x[rs1][31:0])$
Sign-extend Word. Pseudoinstrucción, Solo RV64I.
 Lee los 32 bits más bajos de $x[rs1]$, los extiende en signo, y escribe el resultado en $x[rd]$. Se extiende a **addiw** $rd, rs1, 0$.

sfence.vma rs1, rs2 Fence(Store, AddressTranslation)

Fence Virtual Memory. Tipo R, arquitecturas privilegiadas RV32I y RV64I.

Ordena los *stores* previos a las tablas de páginas con traducciones de direcciones virtuales subsiguientes. Cuando $rs2=0$, las traducciones para todos los *address spaces* son afectadas; de lo contrario, solo las traducciones para el *address space* identificado por $x[rs2]$ son ordenadas. Cuando $rs1=0$, las traducciones para todas las direcciones virtuales en los *address spaces* seleccionados son ordenadas; de lo contrario, solo las traducciones para la página que contiene la dirección virtual $x[rs1]$ en los *address spaces* seleccionados son ordenadas.



sgtz rd, rs2 $x[rd] = (x[rs2] >_s 0)$

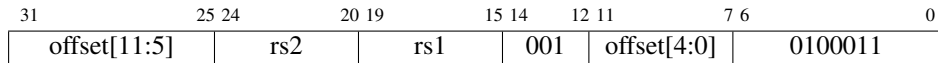
Set if Greater Than to Zero. Pseudoinstrucción, RV32I y RV64I.

Escribe 1 en $x[rd]$ si $x[rs2]$ es mayor que 0, ó 0 si no. Se extiende a **slt** rd, x0, rs2.

sh rs2, offset(rs1) $M[x[rs1] + sext(offset)] = x[rs2][15:0]$

Store Halfword. Tipo S, RV32I y RV64I.

Almacena los dos bytes menos significativos del registro $x[rs2]$ a memoria en la dirección $x[rs1] + sign-extend(offset)$.

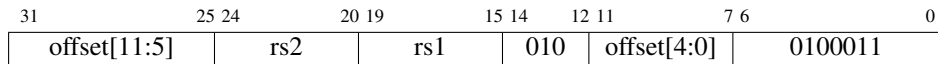


SW rs2, offset(rs1) $M[x[rs1] + sext(offset)] = x[rs2][31:0]$

Store Word. Tipo S, RV32I y RV64I.

Almacena los cuatro bytes menos significativos del registro $x[rs2]$ a memoria en la dirección $x[rs1] + sign-extend(offset)$.

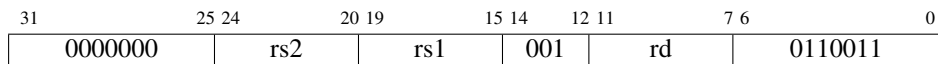
Formas comprimidas: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)



sll rd, rs1, rs2 $x[rd] = x[rs1] \ll x[rs2]$

Shift Left Logical. Tipo R, RV32I y RV64I.

Corre el registro $x[rs1]$ a la izquierda por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ (o seis bits para RV64I) forman la cantidad a correr; los bits altos son ignorados.

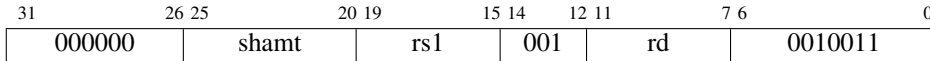


slli rd, rs1, shamt $x[rd] = x[rs1] \ll \text{shamt}$

Shift Left Logical Immediate. Tipo I, RV32I y RV64I.

Corre el registro $x[rs1]$ a la izquierda por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado es escrito en $x[rd]$. Para RV32I, la instrucción solo es legal cuando *shamt*[5]=0.

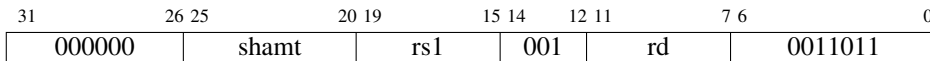
Forma comprimida: **c.slli** rd, shamt



slliw rd, rs1, shamt $x[rd] = \text{sext}((x[rs1] \ll \text{shamt}) [31:0])$

Shift Left Logical Word Immediate. Tipo I, solo RV64I.

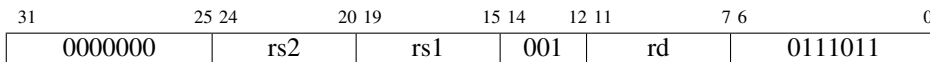
Corre $x[rs1]$ a la izquierda por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, el resultado es truncado a 32 bits, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. La instrucción solo es legal cuando *shamt*[5]=0.



sllw rd, rs1, rs2 $x[rd] = \text{sext}((x[rs1] \ll x[rs2] [4:0]) [31:0])$

Shift Left Logical Word. Tipo R, solo RV64I.

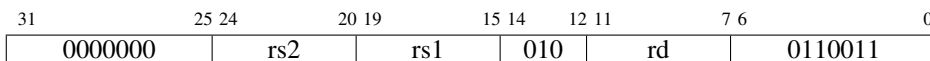
Corre los 32 bits más bajos de $x[rs1]$ a la izquierda por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ forman la cantidad a correr; los bits altos son ignorados.



slt rd, rs1, rs2 $x[rd] = x[rs1] <_s x[rs2]$

Set if Less Than. Tipo R, RV32I y RV64I.

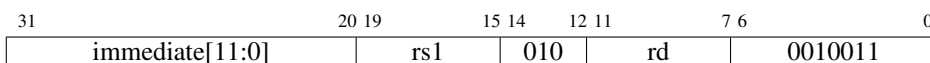
Compara $x[rs1]$ con $x[rs2]$ como números de complemento a dos, y escribe 1 en $x[rd]$ si $x[rs1]$ es menor, ó 0 si no.



slti rd, rs1, immediate $x[rd] = x[rs1] <_s \text{sext}(\text{immediate})$

Set if Less Than Immediate. Tipo I, RV32I y RV64I.

Compara $x[rs1]$ con el *inmediato* sign-extended como números de complemento a dos, y escribe 1 en $x[rd]$ si $x[rs1]$ es menor, ó 0 si no.



sltiu rd, rs1, immediate $x[rd] = x[rs1] <_u \text{sext}(\text{immediate})$

Set if Less Than Immediate, Unsigned. Tipo I, RV32I y RV64I.

Compara $x[rs1]$ con el *immediato* sign-extended como números sin signo, y escribe 1 en $x[rd]$ si $x[rs1]$ es menor, ó 0 si no.

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. Tipo R, RV32I y RV64I.

Compara $x[rs1]$ con $x[rs2]$ como números sin signo, y escribe 1 en $x[rd]$ si $x[rs1]$ es menor, ó 0 si no.

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. Pseudoinstrucción, RV32I y RV64I.

Escribe 1 en $x[rd]$ si $x[rs1]$ es menor que cero, ó 0 si no. Se extiende a **slt** rd, rs1, x0.

snez rd, rs2 $x[rd] = (x[rs2] \neq 0)$

Set if Not Equal to Zero. Pseudoinstrucción, RV32I y RV64I.

Escribe 0 en $x[rd]$ si $x[rs2]$ es igual a 0, ó 1 si no. Se extiende a **sltu** rd, x0, rs2.

sra rd, rs1, rs2 $x[rd] = x[rs1] >>_s x[rs2]$

Shift Right Arithmetic. Tipo R, RV32I y RV64I.

Corre el registro $x[rs1]$ a la derecha por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por copias del bit más significativo de $x[rs1]$, y el resultado es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ (o seis bits para RV64I) forman la cantidad a correr; los bits altos son ignorados.

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	101	rd	0110011	

srai rd, rs1, shamt $x[rd] = x[rs1] >>_s \text{shamt}$

Shift Right Arithmetic Immediate. Tipo I, RV32I y RV64I.

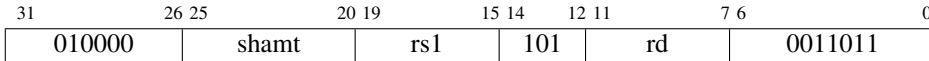
Corre el registro $x[rs1]$ a la derecha por *shamt* posiciones de bits. Los bits liberados son reemplazados por copias del bit más significativo de $x[rs1]$, y el resultado es escrito en $x[rd]$. Para RV32I, la instrucción solo es legal cuando *shamt*[5]=0.

Forma comprimida: **c.srai** rd, shamt

31	26 25	20 19	15 14	12 11	7 6	0
010000	shamt	rs1	101	rd	0010011	

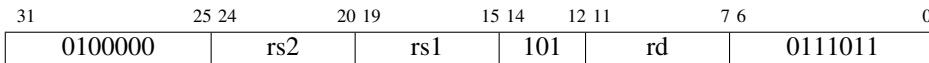
sraiw rd, rs1, shamt $x[rd] = sext(x[rs1][31:0] \gg_s shamt)$
Shift Right Arithmetic Word Immediate. Tipo I, solo RV64I.

Corre los 32 bits más bajos de $x[rs1]$ a la derecha por *shamt* posiciones de bits. Los bits liberados son reemplazados por copias de $x[rs1][31]$, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. La instrucción solo es legal cuando *shamt*[5]=0.



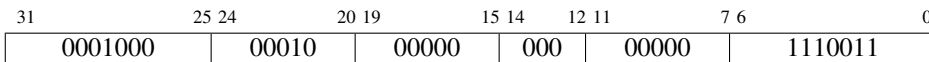
sraw rd, rs1, rs2 $x[rd] = sext(x[rs1][31:0] \gg_s x[rs2][4:0])$
Shift Right Arithmetic Word. Tipo R, solo RV64I.

Corre los 32 bits más bajos de $x[rs1]$ a la derecha por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por $x[rs1][31]$, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ forman la cantidad a correr; los bits altos son ignorados.



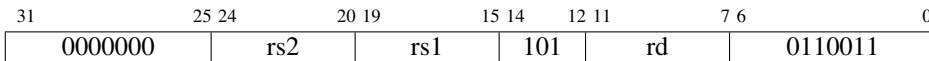
sret ExceptionReturn(Supervisor)
Supervisor-mode Exception Return. Tipo R, arquitecturas privilegiadas RV32I y RV64I.

Retorna de un manejador de excepción en modo supervisor. Escribe $CSRs[sepc]$ al *pc*, $CSRs[sstatus].SPP$ al modo de privilegio, $CSRs[sstatus].SPIE$ a $CSRs[sstatus].SIE$, 1 a $CSRs[sstatus].SPIE$, y 0 a $CSRs[sstatus].SPP$.



srl rd, rs1, rs2 $x[rd] = x[rs1] \gg_u x[rs2]$
Shift Right Logical. Tipo R, RV32I y RV64I.

Corre el registro $x[rs1]$ a la derecha por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por ceros ceros, y el resultado es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ (o seis bits para RV64I) forman la cantidad a correr; los bits altos son ignorados.



srl rd, rs1, shamt $x[rd] = x[rs1] \gg_u shamt$

Shift Right Logical Immediate. Tipo I, RV32I y RV64I.

Corre el registro $x[rs1]$ a la derecha por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado es escrito en $x[rd]$. Para RV32I, la instrucción solo es legal cuando *shamt*[5]=0.

Forma comprimida: **c.srl** rd, shamt

31	26 25	20 19	15 14	12 11	7 6	0	
000000	shamt	rs1	101	rd	0010011		

srlw rd, rs1, shamt $x[rd] = sext(x[rs1][31:0] \gg_u shamt)$

Shift Right Logical Word Immediate. Tipo I, solo RV64I.

Corre los 32 bits más bajos de $x[rs1]$ a la derecha por *shamt* posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. La instrucción solo es legal cuando *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 $x[rd] = sext(x[rs1][31:0] \gg_u x[rs2][4:0])$

Shift Right Logical Word. Tipo R, solo RV64I.

Corre los 32 bits más bajos de $x[rs1]$ a la derecha por $x[rs2]$ posiciones de bits. Los bits liberados son reemplazados por ceros, y el resultado de 32 bits, sign-extended, es escrito en $x[rd]$. Los cinco bits menos significativos de $x[rs2]$ forman la cantidad a correr; los bits altos son 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. Tipo R, RV32I y RV64I.

Subtracts register $x[rs2]$ from register $x[rs1]$ y escribe el resultado en $x[rd]$. Overflow aritmético ignorado.

Forma comprimida: **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 $x[rd] = \text{sext}((x[rs1] - x[rs2])[31:0])$

Subtract Word. Tipo R, solo RV64I.

Resta el registro $x[rs2]$ del registro $x[rs1]$, trunca el resultado a 32 bits, y escribe el resultado de 32 bits, sign-extended, en $x[rd]$. Overflow aritmético ignorado.

Forma comprimida: **c.subw** rd, rs2

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	000	rd	0111011	

tail symbol $pc = \&symbol; ; \text{clobber } x[6]$

Tail call. Pseudoinstrucción, RV32I y RV64I.

Escribe *symbol* en el *pc*, sobrescribiendo $x[6]$ en el proceso. Se extiende a **auipc** $x6$, **offsetHi** luego **jalr** $x0$, **offsetLo**($x6$).

wfi $\text{while (noInterruptsPending) idle}$

Wait for Interrupt. Tipo R, arquitecturas privilegiadas RV32I y RV64I.

Desocupa al procesador para ahorrar energía si ninguna de las interrupciones habilitadas se encuentra pendiente.

31	25 24	20 19	15 14	12 11	7 6	0
0001000	00101	00000	000	00000	1110011	

XOR rd, rs1, rs2 $x[rd] = x[rs1] \wedge x[rs2]$

Exclusive-OR. Tipo R, RV32I y RV64I.

Calcula el OR exclusivo a nivel de bits de los registros $x[rs1]$ y $x[rs2]$ y escribe el resultado en $x[rd]$.

Forma comprimida: **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 $x[rd] = x[rs1] \wedge \text{sext}(\text{immediate})$

Exclusive-OR Immediate. Tipo I, RV32I y RV64I.

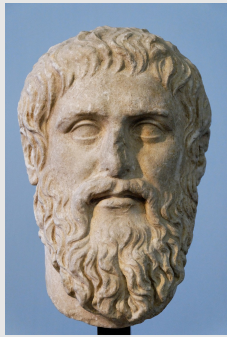
Calcula el OR exclusivo a nivel de bits del *immediato* sign-extended y el registro $x[rs1]$ y escribe el resultado en $x[rd]$.

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	100	rd	0010011	

B

Transliteración de RISC-V

Platón (428–348 AEC) fue un filósofo clásico Griego que sentó las bases de la matemática, filosofía y ciencia Occidental.



Simplicidad

La belleza del estilo, armonía, gracia y buen ritmo dependen de la simpleza.

— Platón, *La República*.

B.1 Introducción

Este apéndice incluye tablas que transliteran instrucciones y expresiones comunes en RV32I a código equivalente en ARM-32 y x86-32. Nuestra meta al escribir este apéndice es asistir a programadores no conocedores de RISC-V que se sienten cómodos con ARM-32 o x86-32 para ayudarles a comprender RISC-V y para ayudarles a traducir sus ISAs antiguos a código básico RISC-V. El apéndice concluye con una rutina en C que recorre un árbol binario con código en lenguaje ensamblador comentado para los tres ISAs. Ordenamos las instrucciones de las tres implementaciones lo más parecido posible para aclarar su correspondencia.

Las instrucciones de transferencia de datos en la Figura B.1 muestran la similitud entre los *loads* y *stores* de RV32I y ARM-32 para el modo de direccionamiento más popular. Dada la orientación memoria-registro del ISA x86 en lugar de la orientación load-store de los ISAs RV32I y ARM-32, x86 transfiere datos empleando instrucciones *move*.

Además de las instrucciones estándar de aritmética de enteros, lógica y corrimientos, la Figura B.2 muestra cómo algunas operaciones comunes se implementan en cada ISA. Por ejemplo, escribir cero a un registro emplea la pseudoinstrucción *li* en RV32I, una instrucción *move immediate* en ARM-32, y una operación OR-exclusivo de un registro con él mismo

Descripción	RV32I	ARM-32	x86-32
Load word	lw t0, 4(t1)	ldr r0, [r1, #4]	mov eax, [edi+4]
Load halfword unsigned	lh t0, 4(t1)	ldrsh r0, [r1, #4]	movsx eax,WORD PTR[edi+4]
Load halfword	lhu t0, 4(t1)	ldrh r0, [r1, #4]	movzx eax,WORD PTR[edi+4]
Load byte	lb t0, 4(t1)	ldrshb r0, [r1, #4]	movsx eax,BYTE PTR[edi+4]
Load byte unsigned	lbu t0, 4(t1)	ldrb r0, [r1, #4]	movzx eax,BYTE PTR[edi+4]
Store byte	sb t0, 4(t1)	strb r0, [r1, #4]	mov [edi+4], al
Store halfword	sh t0, 4(t1)	strh r0, [r1, #4]	mov [edi+4], ax
Store word	sw t0, 4(t1)	str r0, [r1, #4]	mov [edi+4], eax

Figura B.1: Instrucciones RV32I de acceso a memoria transliteradas a ARM-32 y x86-32.

Descripción	RV32I	ARM-32	x86-32
Cargar cero a registro	li t0, 0	mov r0, #0	xor eax, eax
Mover registro	mv t0, t1	mov r0, r1	mov eax, edi
Complementar registro	not t0, t1	mvn r0, r1	not eax, edi
Negar registro	neg t0, t1	rsb r0, r1, #0	mov eax, edi neg eax
Cargar constante grande	lui t0, 0xABCDE addi t0, t0, 0x123	movw r0, #0xE123 movt r0, #0xABCD	mov eax, 0xABCDE123
Mover PC a registro	auipc t0, 0	ldr r0, [pc, #-8]	call lf 1: pop eax
Suma	add t0, t1, t2	add r0, r1, r2	lea eax, [edi+esi]
Suma (inm.)	addi t0, t0, 1	add r0, r0, #1	add eax, 1
Resta	sub t0, t0, t1	sub r0, r0, r1	sub eax, edi
Asignar a reg. (reg=0)	sltiu t0, t1, 1	rsbs r0, r1, #1 movcc r0, #0	xor eax, eax test edx, edx sete al
Asignar a reg. (reg≠0)	sltu t0, x0, t1	adds r0, r1, #0 movne r0, #1	xor eax, eax test edx, edx setne al
OR a nivel de bits	or t0, t0, t1	orr r0, r0, r1	or eax, edi
AND a nivel de bits	and t0, t0, t1	and r0, r0, r1	and eax, edi
XOR a nivel de bits	xor t0, t0, t1	eor r0, r0, r1	xor eax, edi
OR a nivel de bits (inm.)	ori t0, t0, 1	orr r0, r0, #1	or eax, 1
AND a nivel de bits (inm.)	andi t0, t0, 1	and r0, r0, #1	and eax, 1
XOR a nivel de bits (inm.)	xori t0, t0, 1	eor r0, r0, #1	xor eax, 1
Shift left	sll t0, t0, t1	lsl r0, r0, r1	sal eax, cl
Shift right lógico	srl t0, t0, t1	lsr r0, r0, r1	shr eax, cl
Shift right aritmético	sra t0, t0, t1	asr r0, r0, r1	sar eax, cl
Shift left (inm.)	slli t0, t0, 1	lsl r0, r0, #1	sal eax, 1
Shift right lógico (inm.)	srlt0, t0, 1	lsr r0, r0, #1	shr eax, 1
Shift right aritmético. (inm.)	srait0, t0, 1	asr r0, r0, #1	sar eax, 1

Figura B.2: Instrucciones aritméticas RV32I transliteradas a ARM-32 y x86-32. El formato de instrucciones x86-32 de dos operandos normalmente requiere más instrucciones que el formato de tres operandos de ARM-32 y RV32I.

Descripción	RV32I	ARM-32	x86-32
Branch si =	beq t0, t1, foo	cmp r0, r1 beq foo	cmp eax, esi je foo
Branch si \neq	bne t0, t1, foo	cmp r0, r1 bne foo	cmp eax, esi jne foo
Branch si <	blt t0, t1, foo	cmp r0, r1 blt foo	cmp eax, esi jl foo
Branch si \geq_s	bge t0, t1, foo	cmp r0, r1 bge foo	cmp eax, esi jge foo
Branch si $<_u$	bltu t0, t1, foo	cmp r0, r1 bcc foo	cmp eax, esi jb foo
Branch si \geq_u	bgeu t0, t1, foo	cmp r0, r1 bcs foo	cmp eax, esi jnb foo
Branch si =0	beqz t0, foo	cmp r0, #0 beq foo	test eax, eax je foo
Branch si $\neq 0$	bnez t0, foo	cmp r0, #0 bne foo	test eax, eax jne foo
<i>Jump o tail call</i> directo	jal x0, foo	b foo	jmp foo
Llamada a subrutina	jal ra, foo	bl foo	call foo
Retorno de subrutina	jalr x0, 0(ra)	bx lr	ret
Llamada indirecta	jalr ra, 0(t0)	blx r0	call eax
<i>Jump o tail call</i> indirecto	jalr x0, 0(t0)	bx r0	jmp eax

Figura B.3: Instrucciones de control de flujo de ejecución RV32I transliteradas a ARM-32 y x86-32. La instrucción *compare-and-branch* de RV32I ocupa la mitad de las instrucciones de los branches basados en *códigos de condición*² de ARM-32 y x86-32.

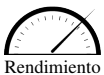
en x86-32. La limitación de dos operandos de las instrucciones x86-32 significa más instrucciones en algunos casos, pero el formato de instrucción de longitud variable le permite cargar una constante grande en una sola instrucción. Las instrucciones convencionales de suma, resta, lógica y corrimientos—que son responsables de la mayor parte de las instrucciones ejecutadas—tienen un mapeo uno-a-uno entre ISAs.

La Figura B.3 lista las instrucciones de branches y llamadas condicionales e incondicionales. Implementar branches condicionales por medio de *códigos de condición* requiere dos instrucciones para ARM-32 y x86-32 mientras que RV32I sólo necesita una. Como se ilustra en el Capítulo 2, en las Figuras 2.5 a 2.11, a pesar de su enfoque minimalista al diseño del set de instrucciones, los branches *compare-and-execute* de RISC-V reducen el número de instrucciones en Ordenamiento por Inserción tanto como los modos de direccionamiento más sofisticados y las instrucciones push y pop de ARM-32 y x86-32.

B.2 Comparando RV32I, ARM-32 y x86-32 empleando Suma de Árboles

La Figura B.4 es nuestro programa de ejemplo en C que usamos para comparar lado-a-lado los tres ISAs en las Figuras B.5 a B.7.

Suma los valores en un árbol binario, usando un recorrido en-orden del árbol. Los árboles son una estructura de datos fundamental, y a pesar de que esta operación de árboles pueda parecer excesivamente simple, la elegimos porque muestra tanto recursión como iteración en pocas instrucciones de lenguaje ensamblador. La rutina recurre para calcular la suma del



subárbol de la izquierda, pero usa iteración para calcular la suma del subárbol de la derecha, lo cual reduce el consumo de memoria³ y la cantidad de instrucciones. Un compilador optimizante puede transformar código completamente recursivo a esta versión; mostramos la iteración explícitamente para mayor claridad.

Las principales diferencias en tamaño de los tres programas en lenguaje ensamblador están en la entrada y salida de funciones. RISC-V utiliza cuatro instrucciones para guardar y restaurar tres registros en el stack y para ajustar el *stack pointer*. x86-32 guarda y restaura solo dos registros en el stack porque puede hacer operaciones aritméticas en operandos de memoria sin tener que cargarlos todos a registros. También los guarda y restaura usando instrucciones push y pop, que ajustan el *stack pointer* implícitamente en lugar de hacerlo explícitamente como en RISC-V. ARM-32 puede guardar tres registros más el *link register* con la dirección de retorno en el stack en una única instrucción push, y restaurarlos con una única instrucción pop.

RISC-V ejecuta el ciclo principal en siete instrucciones en lugar de ocho para otros ISAs porque, como muestra la Figura B.3, puede hacer *compare-and-branch* en una sola instrucción mientras que esa operación ocupa dos instrucciones para ARM-32 y x86-32. El resto de las instrucciones en el ciclo se mapean uno-a uno entre RV32I y ARM-32, como ilustran las Figuras B.1 y B.2. Una diferencia es que las instrucciones *call* y *ret* de x86 implícitamente hacen push y pop de la dirección de retorno en el stack, mientras que los otros ISAs lo hacen explícitamente en sus prólogos y epílogos (por medio de guardar y restaurar *ra* para RV32I, o hacer *push* de *lr* y *pop* al *pc* para ARM-32). También, dado que la convención de llamadas de x86-32 pasa argumentos en el stack, el código x86-32 tiene una instrucción push y una pop en el ciclo que otros ISAs pueden evitar. La transferencia de datos adicional reduce el rendimiento.



B.3 Conclusión

A pesar de tener filosofías de ISA ampliamente distintas, los programas resultantes son muy parecidos, haciendo muy sencilla la traducción de versiones del programa de arquitecturas antiguas a RISC-V. Tener 32 registros para RISC-V versus 16 para ARM-32 y 8 para x86-32 simplifica la traducción a RISC-V, lo cual sería mucho más difícil en el sentido opuesto. Primero hay que ajustar los prólogos y epílogos de funciones, luego cambiar los branches condicionales de una orientación basada en *códigos de condición* a instrucciones *compare-and-branch*, y finalmente reemplazar todos los nombres de registros e instrucciones por sus equivalentes RISC-V. Pueden restar algunos ajustes adicionales, tales como manejar constantes y direcciones largas en el ISA de longitud variable x86-32 o agregar instrucciones RISC-V para lograr los modos de direccionamiento sofisticados si son empleados en transferencias de datos, pero se llegaría muy cerca luego se seguir solo estos tres pasos.



Notas

¹Campos de Bits: En inglés, *Bit Fields*. Estructuras de datos para acceso a bits individuales.

²Códigos de Condición: En inglés, *Condition Codes*.

³Consumo de Memoria: En inglés, *Memory Footprint*.

```

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: Una rutina en C que suma los valores en un árbol binario, empleando un recorrido en-orden.

```

addi sp,sp,-16 # Reservar stack frame
sw  s1,4(sp) # Preservar s1
sw  s0,8(sp) # Preservar s0
sw  ra,12(sp) # Preservar ra
li  s1,0 # sum = 0
beqz a0,.L1 # Omitir ciclo si node == 0
mv  s0,a0 # s0 = node
.L3:
lw  a0,0(s0) # a0 = node->left
jal tree_sum # Recurrir; resultado en a0
lw  a5,8(s0) # a5 = node->value
lw  s0,4(s0) # node = node->right
add s1,a0,s1 # sum += a0
add s1,s1,a5 # sum += a5
bnez s0,.L3 # Iterar si node != 0
.L1:
mv  a0,s1 # Retornar sum in a0
lw  s1,4(sp) # Restaurar s1
lw  s0,8(sp) # Restaurar s0
lw  ra,12(sp) # Restaurar ra
addi sp,sp,16 # Liberar stack frame
ret # Retornar

```

Figura B.5: Código RV32I para recorrido en-orden del árbol. El ciclo principal es más corto que las versiones para los otros dos ISAs debido a la instrucción *compare-and-branch* bnez.

```

push {r4, r5, r6, lr} # Preservar regs
mov r5, #0           # sum = 0
subs r4, r0, #0     # r4 = node; ¿Es node == 0?
beq .L1             # Omitir ciclo si lo es
.L3:
ldr r0, [r4]        # r0 = node->left
bl tree_sum         # Recurrir; resultado en r0
ldr r3, [r4, #8]    # r3 = node->value
ldr r4, [r4, #4]    # r4 = node->right
add r5, r0, r5      # sum += r0
add r5, r5, r3      # sum += r3
cmp r4, #0          # ¿Es node == 0?
bne .L3             # Iterar si no lo es
.L1:
mov r0, r5          # Retornar sum en r0
pop {r4, r5, r6, pc} # Restaurar regs y retornar

```

Figura B.6: Código ARM-32 para recorrido en-orden del árbol. Las instrucciones push y pop *multiword* reducen el tamaño del código para ARM-32 versus los otros ISAs.

```

push esi           # Preservar esi
push ebx           # Preservar ebx
xor esi, esi      # sum = 0
mov ebx, [esp+12] # ebx = node
test ebx, ebx     # ¿Es node == 0?
je .L1            # Omitir ciclo si lo es
.L3:
push [ebx]        # Cargar node->left; push en stack
call tree_sum     # Recurrir; resultado en eax
pop edx           # Pop arg anterior y descartar
add esi, [ebx+8]  # sum += node->value
mov ebx, [ebx+4]  # node = node->right
add esi, eax      # sum += eax
test ebx, ebx     # ¿Es node == 0?
jne .L3           # Iterar si no lo es
.L1:
mov eax, esi      # Retornar sum en eax
pop ebx           # Restaurar ebx
pop esi           # Restaurar esi
ret               # Retornar

```

Figura B.7: Código x86-32 para recorrido en-orden del árbol. El ciclo principal tiene instrucciones push y pop no utilizadas en las versiones del programa para los otros ISAs, lo cual ocasiona un tráfico adicional de datos.



Acrónimos

ABI Application Binary Interface: Interfaz Binaria de Aplicaciones.

AMO Atomic Memory Operations: Operaciones de Memoria Atómicas.

ASICs Application-Specific Integrated Circuits: Circuitos integrados de propósito específico.

BCD Binary Coded Decimal: Decimal Codificado en Binario.

CPI Cycles Per Instruction: Ciclos Por Instrucción.

CSRs Control and Status Registers: Registros de Control y Estado.

ELF Executable and Linkable Format: Formato Ejecutable y *Linkeable*.

FPGAs Field-Programmable Gate Arrays: Arreglos de compuertas programables.

FPU Floating Point Units: Unidades de Punto Flotante.

Hart Hardware thread: Hilo de ejecución en hardware.

MFLOPS Mega floating-point operations per second: millones de operaciones de punto flotante por segundo.

MSA MIPS SIMD Architecture: Arquitectura SIMD de MIPS.

PIC Position Independent Code: Código independiente de su posición.

PMP Physical Memory Protection: Protección Física de Memoria.

PPN Physical Page Number: Número de Página Física.

PTE Page Table Element: Entrada de la Tabla de Páginas.

SIMD Single Instruction Multiple Data: Una Instrucción, Múltiples Datos.

TLB Translation Lookaside Buffer: Cache de Traducciones.

TLS Thread-Local Storage: Almacenamiento local del thread.

Índice

- ABI, *véase también* application binary interface
- ABI de RISC-V, *véase* Application Binary Interface de RISC-V, 44
- Add, 20, 129
 - immediate, 20, 129
 - immediate word, 90, 129
 - upper immediate to PC, 133
 - word, 90, 129
- add, 20, *véase también* c.add, 129
- Add upper immediate to PC, 22
- addi, *véase también* Add immediate, *véase también* c.addi16sp, *véase también* c.addi4spn, *véase también* c.addi, *véase también* c.li, 129
- addiw, *véase también* Add immediate word, *véase también* c.addiw, 129
- addw, *véase también* Add word, *véase también* c.addw, 129
- aislamiento de arquitectura e implementación, *véase también* arquitectura de set de instrucciones, principios de diseño, aislamiento de arquitectura e implementación
- ALGOL, 124
- Allen, Fran, 16
- AMD64, 96
- AMO, *véase también* Operación Atómica de Memoria
- amoadd.d, *véase también* Atomic Memory Operation Add Doubleword, 129
- amoadd.w, *véase también* Atomic Memory Operation Add Word, 130
- amoand.d, *véase también* Atomic Memory Operation And Doubleword, 130
- amoand.w, *véase también* Atomic Memory Operation And Word, 130
- amomax.d, *véase también* Atomic Memory Operation Maximum Doubleword, 130
- amomax.w, *véase también* Atomic Memory Operation Maximum Word, 130
- amomaxu.d, *véase también* Atomic Memory Operation Maximum Unsigned Doubleword, 131
- amomaxu.w, *véase también* Atomic Memory Operation Maximum Unsigned Word, 131
- amomin.d, *véase también* Atomic Memory Operation Minimum Doubleword, 131
- amomin.w, *véase también* Atomic Memory Operation Minimum Word, 131
- amominu.d, *véase también* Atomic Memory Operation Minimum Unsigned Doubleword, 131
- amominu.w, *véase también* Atomic Memory Operation Minimum Unsigned Word, 132
- amoor.d, *véase también* Atomic Memory Operation Or Doubleword, 132
- amoor.w, *véase también* Atomic Memory Operation Or Word, 132
- amoswap.d, *véase también* Atomic Memory Operation Swap Doubleword, 132
- amoswap.w, *véase también* Atomic Memory Operation Swap Word, 132
- amoxor.d, *véase también* Atomic Memory Operation Exclusive Or Doubleword, 133
- amoxor.w, *véase también* Atomic Memory Operation Exclusive Or Word, 133
- And, 20, 133
 - immediate, 20, 133
- and, *véase también* c.and, 133
- andi, 20, *véase también* And immediate, *véase también* c.andi, 133
- application binary interface, 20, 33, 35, 36, 52
- Application Binary Interface de RISC-V
 - ilp32, 44
 - ilp32d, 44
 - ilp32f, 44
 - lp64, 94
 - lp64d, 94
 - lp64f, 94
- Application Specific Integrated Circuits, 2
- ARM
 - Cortex-A5, 7
 - Cortex-A9, 8
 - DAXPY, 60
 - Load Multiple, 7, 8
 - manual de referencia de instrucciones
 - número de páginas, 12
 - número de registros, 10
 - Ordenamiento por Inserción, 25
 - Suma de Árboles, 178
 - tamaño del código, 9, 96
 - Thumb, 9
 - Thumb-2, 9
- ARMv8, 96
- arquitectura, 8
- arquitectura de set de instrucciones, *véase* arquitectura de set de instrucciones, 2
 - abierta, 3
 - elegancia, 13, 27, 46, 71, 87, 97, 126
 - errores del pasado, 27
 - incremental, 3
 - métricas de diseño, 6
 - aislamiento de arquitectura e implementación, 8, 27, 76, 87, 105, 125

- costo, **6**, 16, 19, 22, 23, 27, 50, 69, 97, 118
 espacio para crecer, **8**, 19, 27, 97
 facilidad de programar, compilar y *linkear*, **10**, 19, 20, 22, 23, 27, 59, 77, 78, 80, 81, 86, 94, 95, 97, 110, 121
 rendimiento, **7**, 16, 19, 27, 49, 52, 57, 60, 65, 76, 80–82, 84, 86, 95, 97
 simplicidad, **7**, 11, 12, 20, 22–25, 27, 60, 65, 68, 77, 86, 110, 115, 121, 125
 tamaño del programa, **9**, 27, 68, 71, 94, 97
 modularidad, **5**
 principios de diseño
 costo, 46
 espacio para crecer, 126
 facilidad de programar, compilar y *linkear*, 41, 46, 125
 rendimiento, 34, 46, 124, 125
 simplicidad, 37
 retrocompatibilidad binaria, **3**
 Arquitectura vectorizada, **77**
 cambio de contexto, 79
 codificación de tipos, 79
 tipado dinámico de registros, 78, 94
 arquitectura von Neumann, 12
 ASIC, *véase también* Application Specific Integrated Circuits
 auipc, *véase también* Add upper immediate to PC, 133
- Bell, C. Gordon, 50, 90
 beq, *véase también* Branch if equal, *véase también* c.beqz, 134
 beqz, 37, 134
 bge, *véase también* Branch if greater or equal, 134
 bgeu, *véase también* 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, *véase también* Branch if less than, 135
 bltu, *véase también* Branch if less than unsigned, 135
 bltz, 37, 135
 bne, *véase también* Branch if not equal, *véase también* c.bnez, 136
 bnez, 37, 136
 Branch
 if equal, 23, 134
 if greater or equal, 23, 134
 if greater or equal unsigned, 23, 134
 if less than, 23, 135
 if less than unsigned, 23, 135
 if not equal, 23, 136
 branch retardado, 8
 Brooks, Fred, 121
 Browning, Robert, 60
- c.add, *véase también* add, 136
 c.addi, *véase también* addi, 136
 c.addi16sp, *véase también* addi, 136
 c.addi4spn, *véase también* addi, 136
 c.addiw, 90, *véase también* addiw, 137
 c.addw, 90, *véase también* addw, 137
 c.and, *véase también* and, 137
 c.andi, *véase también* andi, 137
 c.beqz, *véase también* beq, 137
 c.bnez, *véase también* bne, 137
 c.ebreak, *véase también* ebreak, 138
 c.fld, *véase también* fld, 138
 c.fldsp, *véase también* fld, 138
 c.flw, *véase también* flw, 138
 c.flwsp, *véase también* flw, 138
 c.fsd, *véase también* fsd, 138
 c.fsdsp, *véase también* fsd, 139
 c.fsw, *véase también* fsw, 139
 c.fswsp, *véase también* fsw, 139
 c.j, *véase también* jal, 139
 c.jal, *véase también* jal, 139
 c.jalr, *véase también* jalr, 139
 c.jr, *véase también* jalr, 140
 c.ld, 90, *véase también* ld, 140
 c.ldsp, 90, *véase también* ld, 140
 c.li, *véase también* addi, 140
 c.lui, *véase también* lui, 140
 c.lw, *véase también* lw, 140
 c.lwsp, *véase también* lw, 141
 c.mv, *véase también* add, 141
 c.or, *véase también* or, 141
 c.sd, 90, *véase también* sd, 141
 c.sdsp, 90, *véase también* sd, 141
 c.slli, *véase también* slli, 141
 c.srai, *véase también* srai, 142
 c.srli, *véase también* srli, 142
 c.sub, *véase también* sub, 142
 c.subw, 90, *véase también* subw, 142
 c.sw, *véase también* sw, 142
 c.swsp, *véase también* sw, 142
 c.xor, *véase también* xor, 143
 código independiente de posición, 10, 24, 41, 95
 call, 37, 143
 Callee saved registers, 36
 Caller saved registers, 36
 cambio de contexto, 79
 Chanel, Coco, 128
 chip, *véase también* die, 7
 Circuitos integrados de propósito específico, *véase también* Application Specific Integrated Circuits
 comparativa CoreMark, 8
 Compiladores
 Premio Turing, 124
 Control and Status Register
 read and clear, 143
 read and clear immediate, 144
 read and set, 144
 read and set immediate, 144
 read and write, 144
 read and write immediate, 144
 Convenciones de llamadas, **34**
 costo, *véase también* arquitectura de set de instrucciones, principios de diseño, costo
 Cray, Seymour, 76, 97
 CSR, *véase* Control and Status Register
 csrc, 37, 143
 csrri, 37, 143
 csrr, 37, 143
 csrrc, *véase también* Control and Status Register read and clear, 143
 csrrci, *véase también* Control and Status Register read and clear immediate, 144
 csrrs, *véase también* Control and Status Register read and set, 144
 csrrsi, *véase también* Control and Status Register read and set immediate, 144
 csrrw, *véase también* Control and Status Register read and write, 144
 csrrwi, *véase también* 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
 DAXPY, **60**
 de Saint-Exupéry, Antoine, 52
 Diagrama de Instrucciones
 RV32C, **68**
 Diagrama de instrucciones
 Instrucciones privilegiadas, **105**
 RV32A, **64**
 RV32D, **52**
 RV32F, **52**
 RV32I, **16**
 RV32M, **48**
 RV64A, **90**
 RV64C, **90**
 RV64D, **90**

- RV64F, **90**
 RV64I, **90**
 RV64M, **90**
- die, *véase también* chip, 7
 rendimiento, 7
- Dirección virtual, 115
- directivas del ensamblador, 41, 41
 div, 145
- Divide, 49, 145
 unsigned, 49, 145
 unsigned word, 90, 146
 usando corrimiento a la derecha, 49
 word, 90, 146
- divu, *véase también* Divide unsigned, 145
- divuw, *véase también* Divide unsigned word, 146
- divw, *véase también* Divide word, 146
- ebreak, 146
- ecall, 146
- Efecto Lindy, 27
- Einstein, Albert, 64
- ELF, *véase también* executable and linkable format
 endianness, 23
- epílogo, *véase también* epílogo de función
 epílogo de función, 36
- espacio para crecer, *véase también* arquitectura de set de instrucciones, principios de diseño, espacio para crecer
- Esperar Interrupción, 110
- Estándar de punto flotante IEEE 754-2008, 52
- Excepción, 105
- Exclusive Or, 20, 175
 immediate, 20, 175
- executable and linkable format, 41
- fabs.d, 37, 146
- fabs.s, 37, 146
- facilidad de programar, compilar y *linkear*, *véase también* arquitectura de set de instrucciones, principios de diseño, facilidad de programar, compilar y *linkear*
- fadd.d, *véase también* Floating-point Add double-precision, 147
- fadd.s, *véase también* Floating-point Add single-precision, 147
- Fallo de página, 115
- fclass.d, *véase también* Floating-point Classify double-precision, 147
- fclass.s, *véase también* Floating-point Classify single-precision, 147
- fcvt.d.l, *véase también* Floating-point Convert double from long, 148
- fcvt.d.lu, *véase también* Floating-point Convert double from long unsigned, 148
- fcvt.d.s, *véase también* Floating-point Convert double from single, 148
- fcvt.d.w, *véase también* Floating-point Convert double from word, 148
- fcvt.d.wu, *véase también* Floating-point Convert double from word unsigned, 148
- fcvt.l.d, *véase también* Floating-point Convert long from double, 149
- fcvt.l.s, *véase también* Floating-point Convert long from single, 149
- fcvt.lu.d, *véase también* Floating-point Convert long unsigned from double, 149
- fcvt.lu.s, *véase también* Floating-point Convert long unsigned from single, 149
- fcvt.s.d, *véase también* Floating-point Convert single from double, 149
- fcvt.s.l, *véase también* Floating-point Convert single from long, 150
- fcvt.s.lu, *véase también* Floating-point Convert single from long unsigned, 150
- fcvt.s.w, 150
- fcvt.s.wu, *véase también* Floating-point Convert single from word unsigned, 150
- fcvt.w.d, *véase también* Floating-point Convert word from double, 150
- fcvt.w.s, *véase también* Floating-point Convert word from single, 151
- fcvt.wu.d, *véase también* Floating-point Convert word unsigned from double, 151
- fcvt.wu.s, *véase también* Floating-point Convert word unsigned from single, 151
- fdiv.d, *véase también* Floating-point Divide double-precision, 151
- fdiv.s, *véase también* Floating-point Divide single-precision, 151
- Fence
 Instruction Stream, 152
 Memoria Virtual, 119
 Memory and I/O, 152
 Virtual Memory, 170
- fence, 37, *véase también* Fence Memory and I/O, 152
- fence.i, *véase también* Fence Instruction Stream, 152
- feq.d, *véase también* Floating-point Equals double-precision, 152
- feq.s, *véase también* Floating-point Equals single-precision, 152
- Field-Programmable Gate Array, 2
- fld, *véase también* c.fldsp, *véase también* c.fld, *véase también* Floating-point load doubleword, 152
- fle.d, *véase también* Floating-point Less or Equals double-precision, *véase también* Floating-point Less Than double-precision, 153
- fle.s, *véase también* Floating-point Less or Equals single-precision, *véase también* Floating-point Less Than single-precision, 153
- flt.d, 153
- flt.s, 153
- flw, *véase también* c.flwsp, *véase también* c.flw, *véase también* Floating-point load word, 153
- fmadd.d, *véase también* Floating-point fused multiply-add double-precision, 154
- fmadd.s, *véase también* Floating-point fused multiply-add single-precision, 154
- fmax.d, *véase también* Floating-point maximum double-precision, *véase también* Floating-point maximum single-precision, 154
- fmax.s, 154
- fmin.d, *véase también* Floating-point minimum double-precision, 154
- fmin.s, *véase también* Floating-point minimum single-precision, 155
- fmsub.d, *véase también* Floating-point fused multiply-subtract double-precision, 155
- fmsub.s, *véase también* Floating-point fused multiply-subtract single-precision, 155
- fmul.d, *véase también* Floating-point Multiply double-precision, 155
- fmul.s, *véase también* Floating-point Multiply single-precision, 155, *véase también* Floating-point Subtract single-precision
- fmv.d, 37, 156
- fmv.d.x, *véase también* Floating-point move doubleword from integer, 156
- fmv.s, 37, 156

- fmv.w.x, *véase también* Floating-point move word from integer, 156
- fmv.x.d, *véase también* Floating-point move doubleword to integer, 156
- fmv.x.w, *véase también* Floating-point move word to integer, 156
- fneg.d, 37, 157
- fneg.s, 37, 157
- fnmadd.d, *véase también* Floating-point fused negative multiply-add double-precision, *véase también* Floating-point fused negative multiply-add single-precision, 157
- fnmadd.s, 157
- fnmsub.d, *véase también* Floating-point fused negative multiply-subtract double-precision, 157
- fnmsub.s, *véase también* Floating-point fused negative multiply-subtract single-precision, 158
- FPGA, *véase también* Field-Programmable Gate Array, 2
- frcsr, 37, 158
- frflags, 37, 158
- firm, 37, 158
- fscsr, 37, 158
- fsd, *véase también* c.fsdsp, *véase también* c.fsd, *véase también* Floating-point store doubleword, 158
- fsflags, 37, 159
- fsgnj.d, *véase también* Floating-point Sign-inject double-precision, 159
- fsgnj.s, *véase también* Floating-point Sign-inject single-precision, 159
- fsgnjn.d, *véase también* Floating-point Sign-inject negative double-precision, 159
- fsgnjn.s, *véase también* Floating-point Sign-inject negative single-precision, 159
- fsgnjx.d, *véase también* Floating-point Sign-inject XOR double-precision, 160
- fsgnjx.s, *véase también* Floating-point Sign-inject XOR single-precision, 160
- fsqrt.d, *véase también* Floating-point Square Root double-precision, 160
- fsqrt.s, *véase también* Floating-point Square Root single-precision, 160
- fsm, 37, 160
- fsub.d, *véase también* Floating-point Subtract double-precision, 161
- fsub.s, 161
- fsw, *véase también* c.fswsp, *véase también* c.fsw, *véase también* Floating-point store word, 161
- función hoja, 35
- Fundación RISC-V, 2
- Fused multiply-add, 57
- gather, 80
- Hart, 105
- hueco de retardo, 8
- Illiac IV, 85
- implementación, 8
- intercambio de registros con xor, 22
- Interrupción, 107
- ISA, *véase* arquitectura de set de instrucciones
- Itanium, 95
- j, 37, 161
- jal, 37, *véase también* c.jal, *véase también* c.j, *véase también* Jump and link, 161
- jalr, 37, *véase también* c.jalr, *véase también* c.jr, *véase también* Jump and link register, 162
- Johnson, Kelly, 46
- jr, 37, 162
- Jump and link, 25, 161
- register, 25, 162
- la, 162
- lb, *véase también* Load byte, 162
- lbu, *véase también* Load byte unsigned, 162
- ld, *véase también* c.ldsp, *véase también* c.ld, *véase también* Load doubleword, 163
- Lenguajes de programación
- Premio Turing, 124
- Ley de Moore, 2
- lh, *véase también* Load halfword, 163
- lhu, *véase también* Load halfword unsigned, 163
- li, 37, 163
- linking dinámico, 45
- linking estático, 45
- little-endian, 23
- lla, 163
- Load
- byte, 23, 162
- byte unsigned, 23, 162
- doubleword, 90, 163
- halfword, 23, 163
- halfword unsigned, 23, 163
- reserved
- doubleword, 90, 164
- word, 64, 164
- upper immediate, 22, 164
- word, 23, 164
- word unsigned, 23, 164
- Load upper immediate, 22
- lr.d, *véase también* Load reserved doubleword, 164
- lr.w, *véase también* Load reserved word, 164
- lui, *véase también* c.lui, *véase también* Load upper immediate, 164
- lw, *véase también* c.lwsp, *véase también* c.lw, *véase también* Load word, 164
- lwu, *véase también* Load word unsigned, 164
- métricas de diseño de ISAs, *véase también* arquitectura de set de instrucciones, métricas de diseño macrofusión, 7, 7, 70
- marchid, *véase* Registro de Control y Estado
- mcause, *véase* Registro de Control y Estado
- mcounteren, *véase* Registro de Control y Estado
- mcycle, *véase* Registro de Control y Estado
- Memoria virtual, 115
- mepc, *véase* Registro de Control y Estado
- mhartid, *véase* Registro de Control y Estado
- mhpcounteri, *véase* Registro de Control y Estado
- mhpmeventi, *véase* Registro de Control y Estado
- microMIPS, 60
- mie, *véase* Registro de Control y Estado
- mimpid, *véase* Registro de Control y Estado
- minstret, *véase* Registro de Control y Estado
- mip, *véase* Registro de Control y Estado
- MIPS
- assembler, 22
- branch retardado, 8, 24, 33, 61, 98
- DAXPY, 60
- load retardado, 23, 33, 98
- Ordenamiento por Inserción, 25
- MIPS MSA, 85
- MIPS-IV, 96
- misa, *véase* Registro de Control y Estado
- Modo de privilegio, 104
- Modo máquina, 105
- Modo usuario, 112

- Modo máquina, 105
 Modo usuario, 112
 mret, *véase también* Retorno de Excepción Machine, 165
 mscratch, *véase* Registro de Control y Estado
 mstatus, *véase* Registro de Control y Estado
 mtime, *véase* Registro de Control y Estado
 mtimecmp, *véase* Registro de Control y Estado
 mtval, *véase* Registro de Control y Estado
 mvec, *véase* Registro de Control y Estado
 mul, *véase también* Multiply, 165
 mulh, *véase también* Multiply high, 165
 mulhsu, *véase también* Multiply high signed-unsigned, 165
 mulhu, *véase también* Multiply high unsigned, 165
 Multiply, 49, 165
 high, 49, 165
 high signed-unsigned, 49, 165
 high unsigned, 49, 165
 multi-word, 50
 usando corrimiento a la izquierda, 49
 word, 90, 166
 mulw, *véase también* Multiply word, 166
 mv, 37, 166
 mvendovid, *véase* Registro de Control y Estado

 neg, 37, 166
 negw, 37, 166
 nop, 37, 166
 not, 37, 166

 Ockham, William de, 48
 Operación Atómica de Memoria
 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
 Or, 20, 166
 immediate, 20, 167
 or, *véase también* c.or, 166
 Ordenamiento por Inserción, 25
 ori, 20, *véase también* Or immediate, 167

 Página, 115
 paralelismo a nivel de datos, 76
 Pascal, Blaise, 71
 Pequeño es hermoso, 68
 Perlis, Alan, 124
 PIC *véase también* código independiente de posición 41
 Platón, 176
 prólogo, *véase también* prólogo de función
 prólogo de función, 36
 predicción de branches, 8, 20
 Premio Turing
 Allen, Fran, 16
 Brooks, Fred, 121
 Dijkstra, Edsger W., 104
 Perlis, Alan, 124
 Sutherland, Ivan, 34
 Wirth, Niklaus, 71
 procesadores con pipeline, 20
 procesadores fuera-de-orden, 19
 Programas en C
 DAXPY, 60
 Ordenamiento por Inserción, 25
 Suma de Árboles, 178
 propiedades de xor, 22
 Pseudoinstrucción, 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
 frmm, 37, 158
 fscsr, 37, 158
 fsflags, 37, 159
 fsm, 37, 160
 j, 37, 161
 jr, 37, 162
 la, 162
 li, 37, 163
 lla, 163
 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, 167
 ret, 37, 168
 seqz, 37, 169
 sext.w, 37, 169
 sgtz, 37, 170
 sltz, 37, 172
 snez, 37, 172
 tail, 37, 175
 Punto flotante, 52
 Add
 double-precision, 52, 147
 single-precision, 52, 147
 binary128, 61
 binary16, 61
 binary256, 61
 binary32, 61
 binary64, 61
 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, **61**
- decimal32, **61**
- decimal64, **61**
- Divide
 - double-precision, 52, 151
 - single-precision, 52, 151
- dynamic rounding mode, 53
- Equals
 - double-precision, 52, 152
 - single-precision, 52, 152
- Estándar de punto flotante IEEE 754-2008, 52
- Fused multiply-add
 - double-precision, 52, 154
 - single-precision, 52, 154
- fused multiply-add, 57
- 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, 157
 - single-precision, 52, 158
- half precision, **61**
- Less or Equals
 - double-precision, 52, 153
 - single-precision, 52, 153
- Less Than
 - double-precision, 52, 153
 - single-precision, 52, 153
- Load
 - doubleword, 52, 152
 - word, 52, 153
- Maximum
 - double-precision, 52, 154
 - single-precision, 52, 154
- Minimum
 - double-precision, 52, 154
 - single-precision, 52, 155
- Move
 - doubleword from integer, 52, 156
 - doubleword to integer, 52, 156
 - word from integer, 52, 156
 - word to integer, 52, 156
- Multiply
 - double-precision, 52, 155
 - single-precision, 52, 155
- octuple precision, **61**
- quadruple precision, **61**
- registro de control y estado, 52
- Sign-inject
 - double-precision, 52, 159
 - single-precision, 52, 159
- Sign-inject negative
 - double-precision, 52, 159
 - single-precision, 52, 159
- Sign-inject XOR
 - double-precision, 52, 160
 - single-precision, 52, 160
- sign-injection, 58
- Square root
 - double-precision, 52, 160
 - single-precision, 52, 160
- static rounding mode, 53
- Store
 - doubleword, 52, 158
 - word, 52, 161
- Subtract
 - double-precision, 52, 161
 - single-precision, 52, 161
- rdcycle, 37, 167
- rdcycleh, 37, 167
- rdinstret, 37, 167
- rdinstreth, 37, 167
- rdtime, 37, 167
- rdtimeh, 37, 167
- Registro de Control y Estado
 - marchid, 120, 121
 - mcause, 107, 107, 109, 110
 - mcounteren, 120, 121
 - mcycle, 120, 121
 - medeleg, 114, 114
 - mepc, 107, 109, 110
 - mhartid, 120, 121
 - mhpcounteri, 120, 121
 - mhpmeventi, 120, 121
 - mideleg, 114
 - mie, 107, 107, 109, 110
 - mimpid, 120, 121
 - minstret, 120, 121
 - mip, 107, 107, 109, 110
 - misa, 120, 121
 - mscratch, 107, 109, 110
 - mstatus, 107, 107, 109, 110, 112
- mtimecmp, 120, 121
- mtime, 120, 121
- mtval, 107, 109, 110
- mtvec, 107, 107, 109, 110
- mvendorid, 120, 121
- mpmccfg, 113
- satp, 117, 118
- scause, 107, 115
- scounteren, 120, 121
- 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
- read and clear immediate, 25
- read and set, 25
- read and set immediate, 25
- read and write, 25
- read and write immediate, 25
- registros
 - número de, 10
- relajación del linker, **45**
- rem, *véase también* Remainder, 168
- Remainder, 49, 168
 - unsigned, 49, 168
 - unsigned word, 90, 168
 - word, 90, 168
- remu, *véase también* Remainder unsigned, 168
- remuw, *véase también* Remainder unsigned word, 168
- remw, *véase también* Remainder word, 168
- rendimiento, *véase también* arquitectura de set de instrucciones, principios de diseño, rendimiento comparativa CoreMark, 8 ecuación de, 7
- ret, 37, 168
- Retorno de Excepción
 - Máquina, 110, 165
 - Supervisor, 115, 173
- retrocompatibilidad binaria, 3
- RISC-V
 - application binary interface, 20, 33, 35, 36, 52
 - BOOM, 8
 - Convenciones de llamadas, **36**
 - DAXPY, **60**
 - directivas del ensamblador, 41
 - epflogo de función, 36
 - esquema de nombramiento del set de instrucciones, **5**
 - Fundación, **2**

- lecciones aprendidas, **27**
 Linker, **41**
 Loader, **46**
 long, *94*
 macrofusión, *7*
 manual de referencia de instrucciones
 número de páginas, *12*
 modularidad, *5*
 número de registros, *10*
 Ordenamiento por Inserción, **25**
 prólogo de función, *36*
 Pseudoinstrucción, **37**
 pseudoinstrucciones, *11*
 región heap, **41**
 región stack, **41**
 región static, **41**
 región text, **41**
 registros temporales, *35*
 reserva de memoria, **41**
 Rocket, *7*
 RV128, *98*
 RV32A, **64**
 RV32C, *9, 11, 68*
 RV32D, **52**
 RV32F, **52**
 RV32G, *9, 11*
 RV32I, **16**
 RV32M, **48**
 RV32V, *11, 76*
 RV64A, **90**
 RV64C, **90, 96**
 RV64D, **90**
 RV64F, **90**
 RV64G, *11*
 RV64I, **90**
 RV64M, **90**
 saved registers, *35*
 Suma de Árboles, **178**
 tamaño del código, *9, 96*
 temporary registers, *35*
 RV128, *98*
 RV32C, *60*
 RV32V, *85*
- Santayana, George, *26*
 sb, *véase también* Store byte, *169*
 sc.d, *véase también* Store conditional doubleword, *169*
 sc.w, *véase también* Store conditional word, *169*
 scatter, *80*
 scause, *véase* Registro de Control y Estado
 Schumacher, E. F., *68*
 scouteren, *véase* Registro de Control y Estado
 sd, *véase también* c.sdsp, *véase también* c.sd, *véase también* Store doubleword, *169*
 sepc, *véase* Registro de Control y Estado
 seqz, *37, 169*
 Set less than, *22, 171*
 immediate, *22, 171*
 immediate unsigned, *22, 172*
 unsigned, *22, 172*
 sext.w, *37, 169*
 sfence.vma, *véase también* Fence Virtual Memory, *170*
 sgtz, *37, 170*
 sh, *véase también* 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, 172*
 right arithmetic immediate word, *90, 173*
 right arithmetic word, *90, 173*
 right logical, *20, 173*
 right logical immediate, *20, 174*
 right logical immediate word, *90, 174*
 right logical word, *90, 174*
 siete métricas de diseño de ISAs, *véase también* arquitectura de set de instrucciones, métricas de diseño
 SIMD, *véase también* Single Instruction Multiple Data *11*
 simplicidad, *véase también* arquitectura de set de instrucciones, principios de diseño, simplicidad
 Single Instruction Multiple Data, *3, 11, 76*
 sll, *véase también* Shift left logical, *170*
 slli, *véase también* c.slli, *véase también* Shift left logical immediate, *171*
 slliw, *véase también* Shift left logical immediate word, *171*
 sllw, *véase también* Shift left logical word, *171*
 slt, *véase también* Set less than, *171*
 slti, *véase también* Set less than immediate, *171*
 sltiu, *véase también* Set less than immediate unsigned, *172*
 sltu, *véase también* Set less than unsigned, *172*
 sltz, *37, 172*
 Smith, Jim, *86*
 snez, *37, 172*
 sra, *véase también* Shift right arithmetic, *172*
 srai, *véase también* c.srai, *véase también* Shift right arithmetic immediate, *172*
 sraiw, *véase también* Shift right arithmetic immediate word, *173*
 sraw, *véase también* Shift right arithmetic word, *173*
 sret, *véase también* Retorno de Excepción Supervisor, *173*
 srl, *véase también* Shift right logical, *173*
 srli, *véase también* c.srli, *véase también* Shift right logical immediate, *174*
 srliw, *véase también* Shift right logical immediate word, *174*
 srlw, *véase también* Shift right logical word, *174*
 sscratch, *véase* Registro de Control y Estado
 Store
 byte, *23, 169*
 conditional
 doubleword, *90, 169*
 word, *64, 169*
 doubleword, *90, 169*
 halfword, *23, 170*
 word, *23, 170*
 strip mining, *84*
 stval, *véase* Registro de Control y Estado
 stvec, *véase* Registro de Control y Estado
 sub, *véase también* Subtract, *20, véase también* c.sub, *véase también* Subtract, *174*
 Subtract, *20, 174*
 word, *90, 175*
 subw, *véase también* c.subw, *véase también* Subtract word, *175*
 Suma de Árboles, **178**
 superescalar, *2, 8, 70*
 Sutherland, Ivan, *34*
 sw, *véase también* c.swsp, *véase también* c.sw, *véase también* Store word, *170*
- Tabla de páginas, *115*
 tail, *37, 175*
 tamaño del programa, *véase también* arquitectura de set de instrucciones, principios de diseño, tamaño del programa
 Thoreau, Henry David, *126*
 Thumb-2, *60*
 tipado dinámico de registros, *78, 94*
 TLB, *119*
 TLB shutdown, *119*

- Translation Lookaside Buffer, *119*
- Vector
- gather, *80*
 - indexed load, *80*
 - indexed store, *80*
 - load indexado, *80*
 - load por zancadas, *80*
 - scatter, *80*
 - store indexado, *80*
 - store por zancadas, *80*
 - strided load, *80*
 - strided store, *80*
 - strip mining, *84*
 - vectorizable, *81, 86*
- vectorizable, *81, 86*
- von Neumann, John, *12*
- Wait for Interrupt, *175*
- wfi, *véase también* Wait for Interrupt, *175*
- William de Ockham, *48*
- Wirth, Niklaus, *71*
- x86
- código independiente de posición, *10*
 - crecimiento del ISA, *3*
 - DAXPY, **60**
 - instrucción aaa, *3*
 - instrucción aad, *3*
 - instrucción aam, *3*
 - instrucción aas, *3*
 - instrucción enter, *7*
 - manual de referencia de instrucciones
 - número de páginas, *12*
 - número de registros, *10*
 - Ordenamiento por Inserción, **25**
 - Suma de Árboles, **178**
 - tamaño del código, *9, 96*
 - x86-32 AVX2, *85*
 - x86-64
 - AMD64, **95**
 - XLEN, *105*
 - xor, *20, véase también* c.xor, *véase también* Exclusive Or, *175*
 - xori, *20, véase también* Exclusive Or immediate, *175*