

aux[2]



เดวิด แพตเตอร์
แอนดรูว์ วอเตอร์แมน
แปลโดย นฤมล รัตนคุณ
และ พอล เซอร์

คู่มือ

RISC-V

สถาปัตยกรรมแบบเปิด

สถาปัตยกรรมคอมพิวเตอร์ / โปรแกรมภาษาแอสเซมบลี

ค่าใช้จ่าย



ความเรียบง่าย



ประสิทธิภาพ



สง่างาม



มีที่ว่างเหลือไว้

สำหรับส่วนขยายเพิ่ม



ขนาดโปรแกรม



การแยกสถาปัตยกรรม

กับการนำไปใช้งาน



ง่ายในการเรียนรู้

การเขียนโปรแกรม

RISC-V เป็นบทนำและการอ้างอิงที่กระชับสำหรับโปรแกรมเมอร์ระบบฝังตัว นักเรียน และผู้สนใจ

ยกรรแบบเปิดที่ทันสมัยและเป็นที่ยอมรับ RISC-V โดยครอบคลุมตั้งแต่ ไมโครคอนโทรลเลอร์แบบฝังตัว 32

บิตที่สุด ไปจนถึงคอมพิวเตอร์ระบบคลาวด์ 64 บิตที่เร็วที่สุด ดังที่แสดงไว้ข้างบน ข้อความดังกล่าวแสดงให้เห็น

RISC-V ปฏิบัติตามแนวคิดที่ดีของสถาปัตยกรรมในอดีตอย่างไร โดยหลีกเลี่ยงข้อผิดพลาด

ของหนังสือ:

แนะนำชุดคำสั่ง RISC-V จำนวน 100 หน้า

ประกอบด้วย 75 รูปภาพ

อาร์ดอ้างอิง RISC-V แบบ 2 หน้า ที่สรุปคำ

แนะนำทั้งหมด

นิยามศัพท์คำสั่ง 50 หน้า ที่อธิบายทุกคำสั่ง

รายละเอียด

ทั้ง 10 บท ได้แนะนำแต่ละองค์ประกอบของชุดคำสั่ง RISC-V แบบแยกส่วน—โดยจะรวบรวมโค้ดที่ต่างกัน

ถึง RISC-V เทียบกับสถาปัตยกรรม ARM, Intel และ MIPS รุ่นเก่า—แต่ผู้อ่านสามารถเริ่มเขียนโปรแกรม

จากบทที่ 2

ผมชอบ RISC-V และหนังสือเล่มนี้ เนื่องจากมีความสง่างาม—สั้น ตรงประเด็น และครบถ้วน"—C. Gordon Bell



ด แพตเตอร์สัน (Google และมหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์)

แอนดรูว์ วอเตอร์แมน (SiFive) เป็น 2 ใน 4 สถาปนิกของ RISC-V

ISBN 978-0-9992491-0-9



9 780999 249109

คำนิยม หนังสือคู่มือ RISC-V

หนังสือที่ทันสมัยและเหมาะสมกับยุคสมัยเล่มนี้ ได้อธิบายอย่างกระชับรัดกุมคำสั่ง RISC-V ที่เรียบง่าย ฟรี และเปิดกว้าง ซึ่งกำลังได้รับการยอมรับอย่างรวดเร็วในอุตสาหกรรมคอมพิวเตอร์ที่หลากหลาย นอกจากนี้ หนังสือเล่มนี้ยังมีข้อมูลเชิงลึกมากมายเกี่ยวกับสถาปัตยกรรมคอมพิวเตอร์ รวมถึงการอธิบายการตัดสินใจออกแบบเฉพาะที่เราทำในการสร้าง RISC-V ด้วย ฉันสามารถจินตนาการได้ว่าหนังสือเล่มนี้จะกลายเป็นคู่มืออ้างอิงที่ใช้อย่างแพร่หลายสำหรับผู้ปฏิบัติ RISC-V หลายคน

—ศาสตราจารย์ Krste Asanović, มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ หนึ่งในผู้สถาปนาของ RISC-V

ฉันชอบ RISC-V และหนังสือเล่มนี้ เพราะมันสง่างาม—กระชับ ตรงประเด็น และครบถ้วน คำอธิบายประกอบของหนังสือให้ความรู้ทางประวัติศาสตร์ที่ดี แรงจูงใจ และการวิจารณ์สถาปัตยกรรม

—C. Gordon Bell, Microsoft และผู้ออกแบบสถาปัตยกรรมชุดคำสั่ง Digital PDP-11 และ VAX-11

หนังสือเล่มเล็กที่จับต้องได้เล่มนี้ สรุปลักษณะสำคัญของคำสั่งทั้งหมดของสถาปัตยกรรมชุดคำสั่ง RISC-V ได้อย่างง่ายดาย เป็นคู่มืออ้างอิงที่สมบูรณ์แบบสำหรับทั้งนักศึกษาและผู้ปฏิบัติงาน

—ศาสตราจารย์ Randy Katz, มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ หนึ่งในผู้ประดิษฐ์ระบบจัดเก็บข้อมูล RAID

RISC-V เป็นตัวเลือกที่ดี สำหรับนักเรียนในการเรียนรู้เกี่ยวกับสถาปัตยกรรมชุดคำสั่ง และการเขียนโปรแกรมระดับแอสเซมบลี ซึ่งเป็นพื้นฐานสำหรับงานในภาษาระดับสูงในภายหลัง หนังสือที่เขียนชัดเจนเล่มนี้ ให้การแนะนำที่ดีเกี่ยวกับ RISC-V เสริมด้วยความคิดเห็นเชิงลึกเกี่ยวกับประวัติศาสตร์การวิวัฒนาการของมันและการเปรียบเทียบกับสถาปัตยกรรมที่คุ้นเคยอื่นๆ โดยอาศัยประสบการณ์ในอดีตกับสถาปัตยกรรมอื่นๆ ผู้ออกแบบ RISC-V สามารถหลีกเลี่ยงคุณสมบัติที่ไม่จำเป็นและมักไม่ปกติได้ ส่งผลให้การสอนที่ง่ายดาย แม้จะเรียบง่าย แต่ก็ยังมีพลังมากพอสำหรับการใช้งานจริงในวงกว้างมานานแล้ว ฉันเคยสอนวิชาแรกเกี่ยวกับการเขียนโปรแกรมแอสเซมบลี และถ้าฉันสอนอยู่ตอนนี้ ฉันยินดีจะใช้หนังสือเล่มนี้

—John Mashey หนึ่งในผู้ออกแบบสถาปัตยกรรมชุดคำสั่ง MIPS

หนังสือเล่มนี้เล่าเรื่องที่ RISC-V ทำได้และเหตุผลที่ผู้ออกแบบเลือกมอบคุณความสามารถเหล่านั้นให้กับมัน สิ่งที่น่าสนใจยิ่งกว่านั้น ผู้เขียนเล่าว่าเหตุใด RISC-V จึงละเว้นสิ่งที่พบในเครื่องคอมพิวเตอร์รุ่นก่อนๆ เหตุผลเหล่านั้นน่าสนใจ อย่างน้อยก็เท่ากับความสามารถและการละเว้นของ RISC-V

—Ivan Sutherland ผู้ได้รับรางวัล Turing Award ได้รับการยกย่องให้เป็นบิดาแห่งคอมพิวเตอร์กราฟิก

RISC-V จะเปลี่ยนแปลงโลก และหนังสือเล่มนี้จะช่วยให้คุณเป็นส่วนหนึ่งของการเปลี่ยนแปลงนั้น

—ศาสตราจารย์ Michael B. Taylor มหาวิทยาลัยวอชิงตัน

หนังสือเล่มนี้ จะเป็นข้อมูลอ้างอิงที่มีค่า สำหรับทุกคนที่ทำงานกับ ISA ของ RISC-V รหัสดำเนินการถูกนำเสนอในรูปแบบที่เป็นประโยชน์หลายรูปแบบสำหรับการอ้างอิงด่วน ทำให้การเขียนโค้ดแอสเซมบลีและการตีความง่ายขึ้น นอกจากนี้ คำอธิบายและตัวอย่างเกี่ยวกับวิธีใช้ ISA ยังทำให้งานของโปรแกรมเมอร์ง่ายขึ้นอีก การเปรียบเทียบกับ ISA อื่น ๆ นั้นน่าสนใจ และแสดงให้เห็นว่าผู้สร้าง RISC-V ตัดสินใจออกแบบอย่างไร

—Megan Wachs, คุชกู๋บัณฑิต, วิศวกร SiFive

Base Integer Instructions: RV32I and RV64I					RV Privileged Instructions						
Category	Name	Fmt	RV32I Base		+RV64I		Category	Name	Fmt	RV mnemonic	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2	Trap	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	Interrupt	Wait for Interrupt	R	WFI	
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt		MMU Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2	
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2	Examples of the 60 RV Pseudoinstructions				
Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt	Branch = 0 (BEQ rs,x0,imm)		J	BEQZ rs,imm		
Arithmetic	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	Optional Compressed (16-bit) Instruction Extension: RV32C						
	Add Upper Imm to PC	U	AUIPC	rd,imm	Category	Name	Fmt	RVC		RISC-V equivalent	
Logical	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	CL	C.FLW	rd',rs1',imm	LW	rd',rs1',imm*8
	OR Immediate	I	ORI	rd,rs1,imm		Float Load Word SP	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.FLDSP	rd,imm	FLD	rd,sp,imm*16		
Compare	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	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	BLT	rs1,rs2,imm	Arithmetic	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	
Synch	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	
Environment	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	
					Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm	
					Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm	
Control Status Register (CSR)					Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
Read/Write	I	CSRRW	rd,csr,rs1	Shift Right Ari. Imm.		CI	C.SRAI	rd,imm	SRAI	rd,rd,imm	
Read & Set Bit	I	CSRRS	rd,csr,rs1	Shift Right Log. Imm.		CI	C.SRLI	rd,imm	SRLI	rd,rd,imm	
Read & Clear Bit	I	CSRRC	rd,csr,rs1	Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm	
Read/Write Imm	I	CSRRIW	rd,csr,imm		Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm	
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm	Jump	Jump	CJ	C.J	imm	JAL	x0,imm	
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm		Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0	
					Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
						Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
					System Env. BREAK	CI	C.EBREAK		EBREAK		
					Optional Compressed Extension: RV64C						
					All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:						
Loads	Load Byte	I	LB	rd,rs1,imm	ADD Word (C.ADDW)		Load Doubleword (C.LD)				
	Load Halfword	I	LH	rd,rs1,imm	ADD Imm. Word (C.ADDIW)		Load Doubleword SP (C.LDSP)				
	Load Byte Unsigned	I	LBU	rd,rs1,imm	SUBtract Word (C.SUBW)		Store Doubleword (C.SD)				
	Load Half Unsigned	I	LHU	rd,rs1,imm			Store Doubleword SP (C.SDSP)				
Stores	Load Word	I	LW	rd,rs1,imm							
	Store Byte	S	SB	rs1,rs2,imm							
	Store Halfword	S	SH	rs1,rs2,imm							
	Store Word	S	SW	rs1,rs2,imm	SD	rs1,rs2,imm					
32-bit Instruction Formats					16-bit (RVC) Instruction Formats						

32-bit Instruction Formats														16-bit (RVC) Instruction Formats																		
	31	27	26	25	24	20	19	15	14	12	11	7	6	0		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
R	funct7					rs2		rs1		funct3		rd		opcode		CR	funct4				rd/rs1				rs2				op			
I	imm[11:0]							rs1		funct3		rd		opcode		CI	funct3		imm		rd/rs1				imm				op			
S	imm[11:5]					rs2		rs1		funct3		imm[4:0]		opcode		CSS	funct3		imm				rs2				op					
B	imm[12:10:5]					rs2		rs1		funct3		imm[4:1:11]		opcode		CIW	funct3		imm				rd'				op					
U	imm[31:12]											rd		opcode		CL	funct3		imm		rs1'		imm		rd'		op					
J	imm[20:10:11:19:12]											rd		opcode		CS	funct3		imm		rs1'		imm		rs2'		op					
																CB	funct3		offset		rs1'		offset		op							
																CJ	funct3		jump target				op									

RISC-V Integer Base (RV32I/64I), privileged, and optional compressed instruction extension (RV32C/64C). Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 bits wide in RV64I (x0=0). RV64I adds 12 instructions for 64-bit data.



Optional Multiply-Divide Instruction Extension: RVM						Optional Vector Extension: RVV						
Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M		RV32/64V in Figure 8.1. All names really start with “vector”. Using regex notation, { _ F } means integer and FP versions: e.g., V{ _ F }ADD is VADD and VFADD. Similarly, VREM{ _ U } represents VREM and VREMU, the signed and Unsigned versions. For data transfer instructions, { 8-64 } means 4 widths: e.g., VL{ 8-64 } is VL8, VL16, VL32, and VL64.					
Multiply	MULTiply	R	MUL	rd, rs1, rs2	MULW	rd, rs1, rs2						
	MULTiply High	R	MULH	rd, rs1, rs2								
	MULTiply High Sign/Uns	R	MULHSU	rd, rs1, rs2								
	MULTiply High Uns	R	MULHU	rd, rs1, rs2								
Divide	DIVide	R	DIV	rd, rs1, rs2	DIVW	rd, rs1, rs2						
	DIVide Unsigned	R	DIVU	rd, rs1, rs2								
Remainder	REMAinder	R	REM	rd, rs1, rs2	REMW	rd, rs1, rs2						
	REMAinder Unsigned	R	REMU	rd, rs1, rs2	REMUW	rd, rs1, rs2	Name Fmt RV32V/RV64V					
Optional Atomic Instruction Extension: RVA							SET Vector Len.					
Category	Name	Fmt	RV32A (Atomic)		+RV64A		SET VL Imm.	R	VSETVLI	rd, rs1		
Load	Load Reserved	R	LR.W	rd, rs1	LR.D	rd, rs1	SET Imm. VL	R	VSETIVL	rd, rs1		
Store	Store Conditional	R	SC.W	rd, rs1, rs2	SC.D	rd, rs1, rs2	Load	I	VL{ 8-64 }	rd, rs1, imm		
Swap	SWAP	R	AMOSWAP.W	rd, rs1, rs2	AMOSWAP.D	rd, rs1, rs2	Load Strided	R	VLS{ 8-64 }	rd, rs1, rs2		
Add	ADD	R	AMOADD.W	rd, rs1, rs2	AMOADD.D	rd, rs1, rs2	Load Ord. index	R	VLOX{ 8-64 }	rd, rs1, rs2		
Logical	XOR	R	AMOXOR.W	rd, rs1, rs2	AMOXOR.D	rd, rs1, rs2	Load Uno. index	R	VLUX{ 8-64 }	rd, rs1, rs2		
	AND	R	AMOAND.W	rd, rs1, rs2	AMOAND.D	rd, rs1, rs2	Store	S	VS{ 8-64 }	rd, rs1, imm		
	OR	R	AMoor.W	rd, rs1, rs2	AMoor.D	rd, rs1, rs2		Store Strided	R	VSS{ 8-64 }	rd, rs1, rs2	
Min/Max	MINimum	R	AMOMIN.W	rd, rs1, rs2	AMOMIN.D	rd, rs1, rs2	Store Ord. index	R	VSOX{ 8-64 }	rd, rs1, rs2		
	MAXimum	R	AMOMAX.W	rd, rs1, rs2	AMOMAX.D	rd, rs1, rs2	Store Uno. index	R	VSUX{ 8-64 }	rd, rs1, rs2		
	MINimum Unsigned	R	AMOMINU.W	rd, rs1, rs2	AMOMINU.D	rd, rs1, rs2	Mask =	R	VM{ S F }EQ	rd, rs1, rs2		
	MAXimum Unsigned	R	AMOMAXU.W	rd, rs1, rs2	AMOMAXU.D	rd, rs1, rs2	Mask ≠	R	VM{ S F }NE	rd, rs1, rs2		
3 Optional Floating-Point Instruction Extensions: Zfh, RVF, & RVD							Mask <	R	VM{ S F }LT{ _ U }	rd, rs1, rs2		
Category	Name	Fmt	RV32{ H F D } (HP, SP, DP)		+RV64{ H F D }		Mask ≤	R	VM{ S F }LE{ _ U }	rd, rs1, rs2		
Move	Move from Integer	R	FMV.{ H S }.X	rd, rs1	FMV.D.X	rd, rs1	Mask >	R	VM{ S F }GT{ _ U }	rd, rs1, rs2		
	Move to Integer	R	FMV.X.{ H S }	rd, rs1	FMV.X.D	rd, rs1	Mask ≥	R	VM{ S F }GE{ _ U }	rd, rs1, rs2		
Convert	ConVerT from Int	R	FCVT.{ H S D }.W	rd, rs1	FCVT.{ H S D }.L	rd, rs1	Shift Left Log.	R	SLL	rd, srl, rs2		
	ConVerT from Int Unsigned	R	FCVT.{ H S D }.WU	rd, rs1	FCVT.{ H S D }.LU	rd, rs1		Shift Right Log.	R	SRL	rd, srl, rs2	
	ConVerT to Int	R	FCVT.W.{ H S D }	rd, rs1	FCVT.L.{ H S D }	rd, rs1			Shift Right Arith.	R	SRA	rd, srl, rs2
	ConVerT to Int Unsigned	R	FCVT.WU.{ H S D }	rd, rs1	FCVT.LU.{ H S D }	rd, rs1		XOR		R	VXOR	rd, srl, rs2
Load	Load	I	FL{ H, W, D }	rd, rs1, imm	Calling Convention			AND	R	VAND	rd, srl, rs2	
Store	Store	S	FS{ H, W, D }	rs1, rs2, imm	Register	ABI name	Saver	OR	R	VOR	rd, srl, rs2	
Arithmetic	ADD	R	FADD.{ H S D }	rd, rs1, rs2	x0	zero	---	ADD	R	V{ _ F }ADD	rd, rs1, rs2	
	SUBtract	R	FSUB.{ H S D }	rd, rs1, rs2	x1	ra	Caller	SUBtract	R	V{ _ F }SUB	rd, rs1, rs2	
	MULTiply	R	FMUL.{ H S D }	rd, rs1, rs2	x2	sp	Callee	MULTiply	R	V{ _ F }MUL	rd, rs1, rs2	
	DIVide	R	FDIV.{ H S D }	rd, rs1, rs2	x3	gp	---	DIVide	R	V{ _ F }DIV{ _ U }	rd, rs1, rs2	
	Square Root	R	FSQRT.{ H S D }	rd, rs1	x4	tp	---	Square Root	R	V{ _ F }SQRT	rd, rs1, rs2	
Mul-Add	MULTiply-ADD	R	FMADD.{ H S D }	rd, rs1/2/3	x5-7	t0-2	Caller	MULTiply-ADD	R	V{ _ F }MADD	rd, rs1, rs2	
	MULTiply-SUBtract	R	FMSUB.{ H S D }	rd, rs1/2/3	x8	s0/fp	Callee	MULTiply-SUB	R	V{ _ F }MSUB	rd, rs1, rs2	
	Negative Multiply-SUBtract	R	FNMSUB.{ H S D }	rd, rs1/2/3	x9	s1	Callee	Neg. MUL-ADD	R	V{ _ F }NMSUB	rd, rs1, rs2	
	Negative Multiply-ADD	R	FNMADD.{ H S D }	rd, rs1/2/3	x10-11	a0-1	Caller	Neg. MUL-SUB	R	V{ _ F }NMADD	rd, rs1, rs2	
Sign Inject	SIGN source	R	FSGNJ.{ H S D }	rd, rs1, rs2	x12-17	a2-a7	Caller	Fl. Pt. Sign inject	R	VFSJ	rd, srl, rs2	
	Negative SIGN source	R	FSGNJN.{ H S D }	rd, rs1, rs2	x18-27	s2-s11	Callee	FSJ Negative	R	VFSJN	rd, srl, rs2	
	Xor SIGN source	R	FSGNJX.{ H S D }	rd, rs1, rs2	x28-31	t3-t6	Caller	FSJ eXclusive or	R	VFSJX	rd, srl, rs2	
Min/Max	MINimum	R	FMIN.{ H S D }	rd, rs1, rs2	f0-7	ft0-7	Caller	MINimum	R	V{ _ F }MIN{ _ U }	rd, rs1, rs2	
	MAXimum	R	FMAX.{ H S D }	rd, rs1, rs2	f8-9	fs0-1	Callee	MAXimum	R	V{ _ F }MAX{ _ U }	rd, rs1, rs2	
Compare	compare Float =	R	FEQ.{ H S D }	rd, rs1, rs2	f10-11	fa0-1	Caller	MULTiply High	R	VMULH{ _ U }	rd, rs1, rs2	
	compare Float <	R	FLT.{ H S D }	rd, rs1, rs2	f12-17	fa2-7	Caller	REMAinder	R	VREM{ _ U }	rd, rs1, rs2	
	compare Float ≤	R	FLE.{ H S D }	rd, rs1, rs2	f18-27	fs2-11	Callee	Reverse SUB	R	V{ _ F }RSUB	rd, rs1, rs2	
Categorize	CLASSify type	R	FCLASS.{ H S D }	rd, rs1	f28-31	ft8-11	Caller	Reverse DIV	R	V{ _ F }RDIV	rd, rs1, rs2	
Configure	Read Status	R	FRCSR	rd	zero	Hardwired zero	Mask XOR	R	VMXOR	rd, srl, rs2		
	Read Rounding Mode	R	FRRM	rd	ra	Return address	Mask AND	R	VMAND	rd, srl, rs2		
	Read Flags	R	FRFLAGS	rd	sp	Stack pointer	Mask OR	R	VMOR	rd, srl, rs2		
	Swap Status Reg	R	FSCSR	rd, rs1	gp	Global pointer	Mask XNOR	R	VMXNOR	rd, srl, rs2		
	Swap Rounding Mode	R	FSRM	rd, rs1	tp	Thread pointer	Mask NAND	R	VMNAND	rd, srl, rs2		
	Swap Flags	R	FSFLAGS	rd, rs1	t0-6,ft0-11	Temporaries	Mask NOR	R	VMNOR	rd, srl, rs2		
	Swap Rounding Mode Imm	I	FSRMI	rd, imm	s0-11,fs0-11	Saved registers	Mask AND NOT	R	VMANDNOT	rd, srl, rs2		
	Swap Flags Imm	I	FSFLAGSI	rd, imm	a0-7,fa0-7	Function args	Mask OR NOT	R	VMORNOT	rd, srl, rs2		

Calling convention and 6 optional extensions: 8 RV32M (Multiply-divide) instructions; 11 RV32A (Atomic); 49 RV32V (Vector); and 34 floating-point instructions each for 16-, 32-, and 64-bit data (Zfh, RV32F, RV32D). For multiply-add operations, “rs1/2/3” represents “rs1, rs2, rs3”. Using regex notation, FADD.{F|D} is both FADD.F and FADD.D. RV32F and RV32 add floating-point registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. RV32V adds vector registers v0-v31, vector predicate registers vp0-vp7, and a vector length register vl. RV64 adds a few more instructions: RVM adds 4, RVA 11, Zfh 6, RVF 6, RVD 6, and RVV 0.

คู่มือ RISC-V:
สถาปัตยกรรมแบบเปิด
ฉบับปรับปรุงที่ 2, 2.0.0

เดวิด แพตเตอร์สัน และ แอนดรูว์ วอเตอร์แมน

แปลโดย พอล เซอร์แมน และ นฤมล รัตนคุณากร

3 สิงหาคม 2569

ลิขสิทธิ์ พ.ศ. ๒๕๖๕ Strawberry Canyon จำกัด สงวนลิขสิทธิ์

ห้ามทำซ้ำส่วนใดส่วนหนึ่งของหนังสือเล่มนี้หรือวัสดุที่เกี่ยวข้องในรูปแบบใดๆ โดยไม่ได้รับความยินยอมเป็นลายลักษณ์อักษรจากผู้อั้ลิขสิทธิ์

พิมพ์ครั้งที่: 2.0.0

พื้นหลังของหน้าปกเป็นภาพของ **โมนา ลิซ่า** เป็นภาพเหมือนของ Lisa Gherardini วาดโดย Leonardo da Vinci ระหว่างปี พ.ศ. 2040 ถึง 2049 ซึ่งกษัตริย์แห่งฝรั่งเศสได้ทรงซื้อภาพนี้จาก เลโอนาร์โด เมื่อประมาณ ปี พ.ศ. 2073 และจัดแสดงที่พิพิธภัณฑลั้ฟูร์ในปารีส ตั้งแต่ปี พ.ศ. 2340 โมนา ลิซ่าถือเป็นงาน ศิลปะที่รู้จักกันดีที่สุดในโลก โมนา ลิซ่าเป็นตัวแทนของความสง่างาม ซึ่งเราเชื่อว่าเป็นคุณสมบัติของ RISC-V

หนังสือ เล่ม นี้ จัด ทำ ด้วย $\text{\LaTeX}$ สคริปต์ และ มาโคร ที่ ใช้ ใน ฉบับ ภาษา ไทย มี ให้ ใช้ งาน ที่ riscvthai.org

Arthur Klepchukov ออกแบบหน้าปกและกราฟิกสำหรับฉบับภาษาอังกฤษ หน้าปกฉบับภาษาไทย ออกแบบโดย พอล เซอร์แมน และ นฤมล รัตนคุณากร

ภาษาไทยเรียงพิมพ์ด้วยฟอนต์ TH Sarabun New ออกแบบโดย สุพจน์ เจริมลาภ ภาษาอังกฤษและคณิตศาสตร์ใช้ฟอนต์ TeX Gyre Pagella โค้ดโปรแกรมใช้ฟอนต์ TeX Gyre Cursor ทุกฟอนต์เผยแพร่ภายใต้ใบอนุญาตแบบเปิด

ข้อมูลรายการบรรณานุกรม

ข้อมูลทางบรรณานุกรมของสำนักหอสมุดแห่งชาติ

(รอข้อมูล CIP จากสำนักหอสมุดแห่งชาติ และ ISBN จากสำนักพิมพ์)

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

Sherman, Paul | Ratanakunakorn, Naruemon

Title: คู่มือ RISC-V: สถาปัตยกรรมแบบเปิด

(The RISC-V Reader: An Open Architecture Atlas)

David Patterson and Andrew Waterman;

แปลโดย พอล เซอร์แมน และ นฤมล รัตนคุณากร

Description: ฉบับภาษาไทย, รุ่นปรับปรุงที่ 2 |

เผยแพร่ฟรีที่ riscvthai.org |

Includes bibliographical references and index.

Original ISBN: 978-0-9992491-1-6

Subjects: สถาปัตยกรรมคอมพิวเตอร์ | RISC microprocessors. |

ภาษาแอสเซมบลี (Assembly languages)

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

คำอุทิศ

เดวิด แพตเตอร์สัน อุทิศหนังสือเล่มนี้ให้กับพ่อ-แม่
ของเขา

—ถึงพ่อของฉัน เดวิด ผู้ซึ่งฉันได้รับความคิด
สร้างสรรค์ ความสามารถด้านกีฬา และความกล้า
หาญในการต่อสู้เพื่อสิ่งที่ถูกต้อง และ

—ถึงแม่ของฉัน ลูซี่ ผู้ซึ่งฉันได้รับสติปัญญา ความ
มองโลกในแง่ดี และอุปนิสัยใจคอของฉันมาจากท่าน

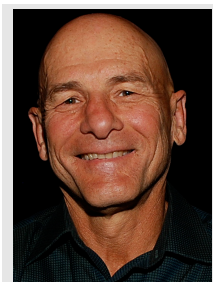
ขอบคุณที่เป็นแบบอย่างที่ดีเยี่ยม ซึ่งสอนฉันถึงความ
หมายของการเป็นคู่ครอง พ่อแม่ และปู่ย่าตายายที่ดี



แอนดรูว์ วอเตอร์แมน อุทิศหนังสือเล่มนี้ให้กับพ่อ-แม่
ของเขา จอห์น และเอลิซาเบธ ผู้ซึ่งให้การสนับสนุน
อย่างมาก แม้ว่าจะอยู่ห่างออกไปหลายพันไมล์



เกี่ยวกับผู้เขียน



เดวิด แพตเตอร์สัน เกษียณในปี 2016 หลังจากทำงานเป็นศาสตราจารย์ด้านวิทยาการคอมพิวเตอร์ที่มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ นาน 40 ปี จากนั้นเขาเข้าร่วมกับ Google Brain ในฐานะวิศวกรอาวุโส เขายังดำรงตำแหน่งรองประธานคณะกรรมการของ RISC-V International และผู้อำนวยการห้องปฏิบัติการโอเพนซอร์ส RISC-V International ในอดีต เขาเคยเป็นประธานของภาควิชาวิทยาการคอมพิวเตอร์ที่เบิร์กลีย์ และได้รับเลือกให้เป็นประธานของสมาคมวิจัยคอมพิวเตอร์ (CRA) และประธานของสมาคมเพื่อการคำนวณ (ACM) ในช่วงทศวรรษ 1980 เขานำโครงการ Reduced Instruction Set Computer (RISC) สู่รุ่น ซึ่งเป็นแรงบันดาลใจให้ตั้งชื่อ RISC รุ่นล่าสุดของ เบิร์กลีย์ถูกตั้งชื่อไว้ว่า “RISC Five” ร่วมกับแอนดรูว์ วอเตอร์แมน เขาเป็นหนึ่งในสี่สถาปนิกของ RISC-V

นอกเหนือจาก RISC แล้ว งานวิจัยที่โด่งดังที่สุดของเขาคือ Redundant Arrays of Inexpensive Disks (RAID) ซึ่งนำไปสู่เอกสารวิชาการจำนวนมาก หนังสือ 7 เล่ม และรางวัลกว่า 40 รางวัล รวมถึงการได้รับเลือกเป็นสมาชิกของ National Academy of Engineering, National Academy of Sciences และ Silicon Valley Engineering Hall of Fame เขาได้รับรางวัลบริการดีเด่นจาก ACM, CRA และ SIGARCH รวมถึงรางวัลด้านการศึกษารางวัล Distinguished Teaching Award จาก UC Berkeley, รางวัล Karlstrom Outstanding Educator Award จาก ACM, รางวัล Mulligan Education Medal จาก IEEE และรางวัลความเป็นเลิศด้านตำราเรียน 2 รางวัลจาก Text and Academic Authors Association รางวัลล่าสุดของเขาคือรางวัล Tapia Achievement Award สำหรับการศึกษาวิทยาศาสตร์ การมีส่วนร่วมต่อสังคม และการส่งเสริมความหลากหลายในวงการคอมพิวเตอร์, รางวัล Frontiers of Knowledge Award จากมูลนิธิ BBVA และรางวัล A.M. Turing Award จาก ACM ซึ่งสองรางวัลหลังนี้เขาได้รับร่วมกับจอห์น เฮนเนสซี

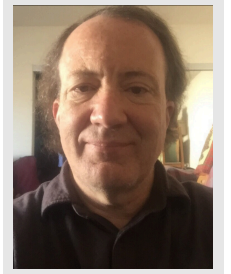
เขาสำเร็จการศึกษาทุกปริญญาจาก UCLA ซึ่งมอบรางวัลศิษย์เก่าดีเด่นด้านวิศวกรรมให้เขา เขาเติบโตในแคลิฟอร์เนียตอนใต้ และเพื่อความสนุก เขาชอบเล่นบอดีชีร์ฟ ซิกกีรยาน เล่นฟุตบอลกับลูกชาย และเดินป่ากับภรรยา ซึ่งทั้งคู่เป็นรักแรกสมัยมัธยม และในปี 2022 พวกเขาเฉลิมฉลองครบรอบแต่งงาน 55 ปี



แอนดรูว์ วอเตอร์แมน ดำรงตำแหน่งหัวหน้าวิศวกรและผู้ร่วมก่อตั้ง SiFive ซึ่ง ก่อตั้งโดยผู้สร้างสถาปัตยกรรม RISC-V เพื่อพัฒนาชิปแบบกำหนดเองต้นทุนต่ำโดยใช้ RISC-V เขาสำเร็จการศึกษาระดับปริญญาเอกด้านวิทยาการคอมพิวเตอร์จากมหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ ซึ่งในช่วงนั้น เขารู้สึกเหน็ดเหนื่อยกับความไม่แน่นอนของสถาปัตยกรรมชุดคำสั่งที่มีอยู่ จึงร่วมออกแบบ RISC-V ISA และไมโครโปรเซสเซอร์ RISC-V รุ่นแรก เขาเป็นหนึ่งในผู้มีส่วนร่วมหลักในเครื่องมือสร้างชิป Rocket ที่ใช้ RISC-V แบบโอเพนซอร์ส ภาษาการสร้างฮาร์ดแวร์ Chisel และการพอร์ต RISC-V สำหรับเคอร์เนลระบบปฏิบัติการ Linux รวมถึงคอมไพเลอร์ GNU C และ C Library นอกจากนี้ เขายังได้รับปริญญาโทจาก UC Berkeley ซึ่งเป็นพื้นฐานสำหรับส่วนขยาย RVC ของ RISC-V และปริญญาตรีด้านวิศวกรรมจากมหาวิทยาลัยดุก

เกี่ยวกับผู้แปล

พอล เซอร์แมน เป็นวิศวกรคอมพิวเตอร์ผู้ทำงานในซิลิคอนวัลลีย์ ซึ่งมีความห่วงใยต่อปัญหาของประชาชนทั่วโลก เขาได้มีบทบาทสำคัญในการพัฒนาอุตสาหกรรมจัดเก็บข้อมูลในช่วง 25 ปีที่ผ่านมา โดยทำงานร่วมกับบริษัทชั้นนำอย่างเวสเทิร์น ดิจิทัลและซีเกต เขาเผยแพร่บทความจำนวนมากในชุมชนผู้ใช้สถาบัน SAS เกี่ยวกับปัญหาทางสถิติ ร่วมแต่งบทความและหนังสือหลายเล่มเกี่ยวกับเศรษฐศาสตร์ก้าวหน้า และได้รับสิทธิบัตรสหรัฐอเมริกา 4 ฉบับ สำหรับนวัตกรรมด้านการทดสอบและการผลิตอุปกรณ์จัดเก็บข้อมูล เขาเป็นผู้ที่หลงใหลใน MRAM และได้แสดงให้เห็นว่า RISC-V ใช้งานง่ายเหมือน PI ในงานประชุม Flash Memory Summit เมื่อเดือนสิงหาคม 2022 เขาสำเร็จการศึกษาระดับปริญญาโท สาขาฟิสิกส์จาก มหาวิทยาลัยแคลิฟอร์เนีย เออร์ไวน์ และปริญญาตรีจากมหาวิทยาลัยแคลิฟอร์เนีย ซานตาครูซ



นฤมล รัตนคุณากร เป็นนักวิทยาศาสตร์คอมพิวเตอร์ที่หลงใหลในความเรียบง่ายและสง่างาม เธอมีประสบการณ์หลายปีในอุตสาหกรรมนี้ เธอสำเร็จการศึกษาระดับปริญญาตรีจากมหาวิทยาลัยขอนแก่น



สารบัญย่อ

การ์ดอ้างอิง RISC-V	5
สารบัญรูปภาพ	xvi
คำนำ	xviii
1 ทำไมต้อง RISC-V ?	2
2 RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V	18
3 ภาษาแอสเซมบลี RISC-V	38
4 RV32M: การคูณและการหาร	52
5 RV32FD: จำนวนลอยตัวแบบความแม่นยำเดียวและสองเท่า	56
6 RV32A: คำสั่งแบบอะตอมมิก	70
7 RV32C: คำสั่งแบบบีบอัด	74
8 RV32V: เวกเตอร์	82
9 RV32B: การจัดการบิต	122
10 RV64: ชุดคำสั่งแอดเดรสแบบ 64 บิต	126
11 RV32/64 สถาปัตยกรรมสิทธิพิเศษ	142
12 ส่วนขยายเพิ่มเติม RISC-V ในอนาคต	166
A รายการคำสั่ง RISC-V	168
B การแปลงคำสั่ง RISC-V	228
C การแปลภาษาไทย	236
ดัชนี	274

สารบัญ

การอ้างอิง RISC-V	5
สารบัญรูปภาพ	xvi
คำนำ	xviii
1 ทำไมต้อง RISC-V ?	2
1.1 บทนำ	2
1.2 ISA แบบโมดูลาร์ เทียบกับ ISA แบบเพิ่มขยาย	3
1.3 พื้นฐานการออกแบบ ISA	5
1.4 ภาพรวมของหนังสือเล่มนี้	13
1.5 ข้อสรุปท้ายบท	15
1.6 เรียนรู้เพิ่มเติม	16
2 RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V	18
2.1 บทนำ	18
2.2 รูปแบบคำสั่ง RV32I	18
2.3 รีจิสเตอร์ RV32I	22
2.4 การคำนวณจำนวนเต็มใน RV32I	24
2.5 การโหลดและการจัดเก็บข้อมูลใน RV32I	26
2.6 การกระโดดแบบมีเงื่อนไขใน RV32I	27
2.7 การกระโดดแบบไม่มีเงื่อนไขใน RV32I	28
2.8 คำสั่งเบ็ดเตล็ดใน RV32I	29
2.9 เปรียบเทียบ RV32I, ARM-32, MIPS-32, และ x86-32	30

2.10	ข้อสังเกตสรุป	31
2.11	เรียนรู้เพิ่มเติม	33
3	ภาษาแอสเซมบลี RISC-V	38
3.1	บทนำ	38
3.2	ข้อตกลงในการเรียกฟังก์ชัน	38
3.3	แอสเซมบลี	41
3.4	ตัวเชื่อมโยง (Linker)	46
3.5	การเชื่อมโยงแบบคงที่เทียบกับแบบไดนามิก	49
3.6	โหลดเดอร์ (Loader)	50
3.7	ข้อสังเกตสรุป	50
3.8	เรียนรู้เพิ่มเติม	50
4	RV32M: การคูณและการหาร	52
4.1	บทนำ	52
4.2	ข้อสังเกตสรุป	54
4.3	เรียนรู้เพิ่มเติม	55
5	RV32FD: จำนวนลอยตัวแบบความแม่นยำเดียวและสองเท่า	56
5.1	บทนำ	56
5.2	รีจิสเตอร์จำนวนลอยตัว	56
5.3	การโหลด การจัดเก็บ และการคำนวณจำนวนลอยตัว	58
5.4	การย้ายข้อมูลและการแปลงชนิดของจำนวนลอยตัว	60
5.5	คำสั่งจำนวนลอยตัวสำหรับไลบรารีคณิตศาสตร์	60
5.6	ส่วนขยายมาตรฐาน “Q” สำหรับ จำนวน ลอยตัว ความ แม่นยำ สี่ เท่า (Quad-Precision Floating-Point)	62
5.7	การเปรียบเทียบ RV32FD, ARM-32, MIPS-32 และ x86-32 ด้วย DAXPY	62
5.8	ข้อสังเกตสรุป	63
5.9	เรียนรู้เพิ่มเติม	64
6	RV32A: คำสั่งแบบอะตอมมิก	70
6.1	บทนำ	70

6.2	ข้อสังเกต สรุป	73
6.3	เรียนรู้เพิ่มเติม	73
7	RV32C: คำสั่งแบบบีบอัด	74
7.1	บทนำ	74
7.2	การเปรียบเทียบ RV32GC, Thumb-2, microMIPS และ x86-32	76
7.3	ข้อสังเกตสรุป	77
7.4	เรียนรู้เพิ่มเติม	77
8	RV32V: เวกเตอร์	82
8.1	บทนำ	82
8.2	คำสั่งคำนวณเวกเตอร์	86
8.3	รีจิสเตอร์เวกเตอร์และการกำหนดชนิดข้อมูลแบบไดนามิก	86
8.4	คำสั่งโหลดและเก็บข้อมูลแบบเวกเตอร์	88
8.5	ความขนานระหว่างการประมวลผลแบบเวกเตอร์	91
8.6	การดำเนินการเวกเตอร์แบบมีเงื่อนไข	91
8.7	การควบคุมความยาวเวกเตอร์	93
8.8	ตัวอย่างเวกเตอร์ การคำนวณ DAXPY ใน RV32V	93
8.9	คำสั่งขยายขนาดและลดขนาด	95
8.10	Vector Register Move, Integer Extend, and Convert	98
8.11	Vector Instructions for Libraries	99
8.12	การเปรียบเทียบ RV64V กับ ARmv8 SVE และ x86-64 AVX SIMD . . .	108
8.13	ข้อสังเกตสรุป	114
8.14	เรียนรู้เพิ่มเติม	116
9	RV32B: การจัดการบิต	122
9.1	บทนำ	122
9.2	Zba: การคำนวณที่อยู่	123
9.3	Zbb: การจัดการบิตทั่วไป	123
9.4	Zbc: การคูณแบบไม่มีแครี่	124
9.5	Zbs: การจัดการบิตเดียว	124
9.6	ข้อสังเกตสรุป	124

9.7	เรียนรู้เพิ่มเติม	124
10	RV64: ชุดคำสั่งแอดเดรสแบบ 64 บิต	126
10.1	บทนำ	126
10.2	การเปรียบเทียบกับ ISA 64 บิตอื่นโดยใช้การเรียงลำดับแบบแทรก	132
10.3	ขนาดโปรแกรม	134
10.4	ข้อสรุปท้ายบท	135
10.5	เรียนรู้เพิ่มเติม	136
11	RV32/64 สถาปัตยกรรมสิทธิพิเศษ	142
11.1	บทนำ	142
11.2	โหมดเครื่องสำหรับระบบฝังตัวอย่างง่าย	143
11.3	การจัดการข้อยกเว้นในโหมดเครื่อง	146
11.4	โหมดผู้ดูแลและการแยกกระบวนการในระบบฝังตัว	151
11.5	โหมดผู้ดูแลและระบบสำหรับระบบปฏิบัติการสมัยใหม่	153
11.6	หน่วยความจำเสมือนแบบแบ่งเป็นเพจ	156
11.7	CSR ระบุคุณสมบัติและประสิทธิภาพ	160
11.8	ข้อสังเกตสรุป	161
11.9	เรียนรู้เพิ่มเติม	162
12	ส่วนขยายเพิ่มเติม RISC-V ในอนาคต	166
12.1	ส่วนขยายมาตรฐาน “B” สำหรับการจัดการบิต	166
12.2	ส่วนขยายมาตรฐาน “J” สำหรับภาษาที่แปลแบบไดนามิก	166
12.3	ส่วนขยายมาตรฐาน “P” สำหรับคำสั่ง Packed-SIMD	166
12.4	ส่วนขยายมาตรฐาน “Zk” สำหรับการเข้ารหัสลับแบบสเกลาร์	166
12.5	ข้อสังเกตสรุป	167
A	รายการคำสั่ง RISC-V	168
B	การแปลงคำสั่ง RISC-V	228
B.1	บทนำ	228
B.2	การเปรียบเทียบ RV32I, ARM-32 และ x86-32 โดยใช้ Tree Sum	231
B.3	บทสรุป	231

C การแปลภาษาไทย	236
C.1 บทนำ	236
C.2 RISC-V in a Nutshell	236
C.3 RISC-V ง่ายเหมือน Pi	237
C.4 ทำไมต้องใช้ รหัสดำเนินการ (Opcode)?	240
C.5 ส่วนเติมเต็มสอง	242
C.6 การขยายบิตเครื่องหมาย	244
C.7 การจัดเรียงลำดับไบต์	245
C.8 รายการคำศัพท์ย่อ	248
C.9 ความสอดคล้องของการแปล	252
C.10 สรุปท้ายบท	273
ดัชนี	274

สารบัญรูปภาพ

1.1	สมาชิกขององค์กรของ RISC-V International	3
1.2	การเติบโตของชุดคำสั่ง x86 ตลอดช่วงเวลา	4
1.3	คำอธิบายของคำสั่ง x86-32 aaa (<i>ASCII Adjust after Addition</i>) . . .	5
1.4	เวเฟอร์ขนาดเวเฟอร์ขนาดเส้นผ่านศูนย์กลาง 8 นิ้วที่ประกอบด้วยชิป RISC-V ที่ออกแบบโดย SiFive	7
1.5	การเปรียบเทียบขนาดโปรแกรมสัมพัทธ์ของ RV32G, ARM-32, x86-32, RV32C และ Thumb-2	11
1.6	จำนวนหน้าและจำนวนคำของคู่มือ ISA	15
2.1	แผนภาพของคำสั่ง RV32I	19
2.2	รูปแบบคำสั่ง RISC-V	19
2.3	แผนผังรหัสดำเนินการของ RV32I	20
2.4	รีจิสเตอร์ของ RV32I	23
2.5	การเรียงลำดับแบบแทรก (Insertion Sort) ในภาษา C	29
2.6	จำนวนคำสั่ง และขนาดโค้ดสำหรับการเรียงลำดับแบบแทรก สำหรับแต่ละ ISA	30
2.7	บทเรียนที่สถาปนิกของ RISC-V ได้เรียนรู้จากความผิดพลาดของชุดคำสั่งในอดีต	32
2.8	โค้ด RV32I สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5.	34
2.9	โค้ด ARM-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5	35
2.10	โค้ด MIPS-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5	36
2.11	โค้ด x86-32 สำหรับอัลกอริทึมการเรียงลำดับแบบแทรก ใน 2.5 แสดงดังนี้	37
3.1	ขั้นตอนการแปลจากโค้ดภาษา C ไปเป็นโปรแกรมที่สามารถทำงานได้ . .	39
3.2	ชื่อย่อแบบแอสเซมบลีสำหรับรีจิสเตอร์จำนวนเต็มและรีจิสเตอร์จุดลอยตัวของ RISC-V	40
3.3	คำสั่งเทียม ของ RISC-V จำนวน 32 คำสั่ง ที่ใช้รีจิสเตอร์ x0, รีจิสเตอร์ศูนย์	42
3.4	คำสั่งเทียม 28 คำสั่ง ของ RISC-V ที่ไม่ขึ้นกับรีจิสเตอร์ x0 ,รีจิสเตอร์ศูนย์	43

3.5	โปรแกรม Hello World ภาษาซี (hello.c)	44
3.6	โปรแกรม Hello World ภาษาแอสเซมบลีใน RISC-V (hello.s)	44
3.7	โปรแกรม Hello World ในระดับภาษาเครื่องใน RISC-V (hello.o) . .	45
3.8	โปรแกรม Hello World ภาษาเครื่องใน RISC-V หลังการเชื่อมโยง	45
3.9	คำสั่งแอสเซมเบลอร์ทั่วไปของ RISC-V	46
3.10	การจัดสรรหน่วยความจำของโปรแกรมและข้อมูล RV32I	47
4.1	แผนภาพของคำสั่ง RV32M	52
4.2	แผนผังรหัสดำเนินการของ RV32M แสดงรายละเอียดของโครงสร้างคำ สั่ง รหัสดำเนินการ ประเภทของรูปแบบคำสั่ง และชื่อคำสั่ง	53
4.3	โค้ด RV32M สำหรับการหารด้วยค่าคงที่ที่ใช้การคูณ	54
5.1	แผนภาพของคำสั่ง Zfh, RV32F และ RV32D	57
5.2	รีจิสเตอร์จำนวนลอยตัวของ Zfh, RV32F, และ RV32D	59
5.3	รีจิสเตอร์ควบคุมและสถานะของจำนวนลอยตัว	60
5.4	คำสั่งแปลงข้อมูลของ Zfh, RV32F และ RV32D	61
5.5	โปรแกรม DAXPY ในภาษา C ที่เน้นการคำนวณจำนวนจำนวนลอยตัว . .	63
5.6	จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับสำหรับ ISA ทั้งสี่แบบ . .	63
5.7	โค้ด RV32D สำหรับ DAXPY แสดงในรูปที่ 5.5	65
5.8	โค้ด ARM-32 สำหรับ DAXPY แสดงในรูปที่ 5.5	65
5.9	โค้ด MIPS-32 สำหรับ DAXPY แสดงในรูป 5.5	66
5.10	โค้ด x86-32 สำหรับ DAXPY แสดงในรูป 5.5	66
5.11	แผนผังรหัสดำเนินการของ RV32F แสดง โครงสร้างคำสั่ง, รหัสดำเนินการ การ, ประเภทของรูปแบบ และชื่อคำสั่ง	67
5.12	แผนผังรหัสดำเนินการ ของ RV32D แสดง โครงสร้างคำสั่ง, รหัสดำเนินการ การ, ประเภทของรูปแบบ และชื่อคำสั่ง	68
5.13	แผนผังรหัสดำเนินการ Zfh แสดงรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภท ของรูปแบบ และชื่อคำสั่ง	69
6.1	ไดอะแกรมของคำสั่ง RV32A	71
6.2	แผนผังรหัสดำเนินการ RV32A แสดงโครงสร้างคำสั่ง, รหัสดำเนินการ, รูป แบบของคำสั่ง และชื่อคำสั่ง	71
6.3	ตัวอย่างการชิงโครโนซ์ มี 2 แบบ	73
7.1	แผนภาพของคำสั่ง RV32C	75
7.2	คำสั่งและขนาดของโค้ดสำหรับ Insertion Sort และ DAXPY ใน ISA แบบ บีบอัด	76
7.3	โค้ด RV32C สำหรับการเรียงลำดับการแทรก ที่รูป 2.5	78

7.4	โค้ด RV32DC สำหรับ DAXPY ที่รูป 2.5	78
7.5	แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 01) แสดงโครงสร้าง op- codes รูปแบบคำสั่งและชื่อคำสั่ง	79
7.6	แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 00) แสดงโครงสร้าง op- codes รูปแบบคำสั่งและชื่อคำสั่ง	79
7.7	แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 10) แสดงรายการ โครงสร้าง opcodes รูปแบบและชื่อ	80
7.8	รูปแบบคำสั่งแบบบีบอัด RVC 16 บิต	80
8.1	แผนภาพของคำสั่งเวกเตอร์ RV32V ที่ได้รับความนิยมมากที่สุด	84
8.2	รีจิสเตอร์ควบคุมชนิดเวกเตอร์ของ RV32V, vtype	88
8.3	ชื่อและความหมายของตัวชี้ความยาวและความกว้างของเวกเตอร์	89
8.4	โค้ด RV32V สำหรับ DAXPY ในรูปที่ 5.5	94
8.5	แผนภาพของคำสั่งคำนวณและลอจิกคอลแบบขยายขนาดและลดขนาด ซึ่ง ไม่แสดงในรูปที่ 8.1	97
8.6	แผนภาพของคำสั่งย้าย การขยายจำนวนเต็ม และการแปลงของ RV32V	98
8.7	แสดงแผนภาพของคำสั่งรัดกั้น, คำสั่งดำเนินการกับบิตของรีจิสเตอร์มา สก์, คำสั่งประมวลผลสัญญาณดิจิทัล, คำสั่งจัดการองค์ประกอบ และคำ สั่งเลขคณิตมัลติเวิร์ดที่รองรับตัวทด (carry)	100
8.8	แผนภาพของคำสั่งรับส่งข้อมูลแบบสมบูร์ณ ซึ่งไม่ได้แสดงไว้ใน 8.1	106
8.9	โค้ด x86-64 AVX2 สำหรับ DAXPY ในรูปที่ 5.5	108
8.10	โค้ด x86-64 AVX-512 สำหรับ DAXPY ในรูปที่ 5.5	110
8.11	โค้ด ARMv8 SVE สำหรับ DAXPY ในรูปที่ 5.5	110
8.12	จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับ SIMD และ ISA แบบเวก เตอร์	112
8.13	แผนผังรหัสดำเนินการ ของ RV32V สำหรับการโหลดและจัดเก็บ	117
8.14	แผนผังรหัสดำเนินการ ของ RV32V สำหรับการดำเนินการทางเลขคณิต และการดำเนินการเชิงตรรกะบางส่วน	118
8.15	แผนผังรหัสดำเนินการ RV32V สำหรับการดำเนินการเลขคณิตบางส่วน	119
8.16	รหัสดำเนินการ RV32V สำหรับการดำเนินการจำนวนลอยตัวบางส่วน	120
8.17	รหัสดำเนินการ ของ RV32V สำหรับการดำเนินการของจำนวนลอยตัว และการดำเนินการกับมาสก์	121
9.1	แผนผังคำสั่ง RV32B	122
9.2	แผนผังรหัสดำเนินการของ RV32B	125
10.1	แผนภาพคำสั่ง RV64I	127

10.2	แผนภาพของคำสั่ง RV64M และ RV64A	127
10.3	แผนภาพของคำสั่ง Zfh, RV64F, และ RV64D	128
10.4	แผนภาพของคำสั่ง RV64C	128
10.5	แผนผังภาพรหัสดำเนินการ ของ RV64 ครอบคลุมทั้ง คำสั่งพื้นฐานและ ส่วนขยายแบบเลือกได้	129
10.6	แผนผังรหัสดำเนินการของการจัดการบิตของ	131
10.7	แผนผังของคำสั่ง RV64B	131
10.8	จำนวนคำสั่งและขนาดโค้ดของการเรียงลำดับแบบแทรก สำหรับ ISA ทั้ง สี่แบบ	133
10.9	ขนาดโปรแกรมสัมพันธ์ของ RV64G, ARM-64 และ x86-64 เมื่อเทียบกับ RV64GC	134
10.10	โค้ด RV64I สำหรับการเรียงลำดับการแทรก ในรูป 2.5	137
10.11	โค้ด ARM-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5	138
10.12	โค้ด MIPS-64 สำหรับการเรียงลำดับการแทรกในรูปที่ 2.5	139
10.13	โค้ด x86-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5	140
11.1	ไดอะแกรมของคำสั่งสิทธิพิเศษของ RISC-V	142
11.2	รูปแบบคำสั่งสิทธิพิเศษของ RISC-V, รหัสดำเนินการ, ประเภทรูปแบบคำ สั่ง และชื่อคำสั่ง	143
11.3	สาเหตุของข้อยกเว้นและอินเทอร์รัปต์ใน RISC-V	144
11.4	รีจิสเตอร์ CSR mstatus	146
11.5	รีจิสเตอร์ CSRs สำหรับอินเทอร์รัปต์ในระดับเครื่อง	146
11.6	รีจิสเตอร์ CSR สำหรับสาเหตุของโหมดเครื่อง และโหมดผู้ดูแลระบบ (mcause และ scause)	147
11.7	รีจิสเตอร์ CSR สำหรับที่อยู่ฐานของเวกเตอร์ดักจับในระดับโหมดเครื่อง และโหมดผู้ดูแลระบบ (mtvec และ stvec)	147
11.8	รีจิสเตอร์ CSR ที่เกี่ยวข้องกับข้อยกเว้นและอินเทอร์รัปต์	147
11.9	ระดับสิทธิพิเศษของ RISC-V และการเข้ารหัส	148
11.10	โค้ด RISC-V สำหรับตัวจัดการอินเทอร์รัปต์ของตัวจับเวลาแบบง่าย . . .	150
11.11	รีจิสเตอร์ที่อยู่และรีจิสเตอร์กำหนดค่าของ PMP	152
11.12	โครงสร้างของการกำหนดค่า PMP ภายในรีจิสเตอร์ pmpcfg	153
11.13	รีจิสเตอร์มอบหมายข้อยกเว้น	154
11.14	รีจิสเตอร์ CSR สำหรับ อินเทอร์รัปต์ในโหมดผู้ดูแลระบบ	154
11.15	sstatus CSR	155
11.16	รายการตารางเพจ RV32 Sv32 (PTE)	156
11.17	รายการตารางเพจ RV64 Sv39 (PTE)	158
11.18	รีจิสเตอร์ satp	158

11.19	การเข้ารหัสของฟิลด์ MODE ในรีจิสเตอร์ควบคุมและสถานะ satp . . .	159
11.20	ไดอะแกรมของกระบวนการแปลงที่อยู่ Sv32	159
11.21	CSR misa รายงาน ISA ที่รองรับ	162
11.22	CSR mvendorid ให้รหัสผู้ผลิต JEDEC ของคอร์	162
11.23	CSR ระบุเครื่องจักร (marchid, mimpid, mhartid)	162
11.24	CSR เวลาเครื่องจักร (mtime and mtimecmp)	163
11.25	รีจิสเตอร์ควบคุมตัวนับ mcounteren และ scounteren	163
11.26	CSR ตัวนับประสิทธิภาพฮาร์ดแวร์	163
11.27	ขั้นตอนการแปลงที่อยู่เสมือนเป็นที่อยู่กายภาพ	164
B.1	คำสั่งการเข้าถึงหน่วยความจำ RV32I แปลงเป็น ARM-32 และ x86-32 .	228
B.2	คำสั่งทางคณิตศาสตร์ของ RV32I ที่แปลงเป็น ARM-32 และ x86-32 . .	229
B.3	คำสั่งการควบคุม RV32I แปลงเป็น ARM-32 และ x86-32	230
B.4	ฟังก์ชัน C ที่รวมค่าต่างๆ ในไบนารีทรี โดยใช้การสำรวจแบบตามลำดับ (in-order traversal)	232
B.5	โค้ด RV32I สำหรับการสำรวจทรีแบบตามลำดับ (in-order)	233
B.6	โค้ด ARM-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order)	233
B.7	โค้ด x86-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order)	234
C.1	โครงสร้างโค้ดโปรแกรมแอสเซมบลี	241

คำนำ

ยินดีต้อนรับ!

RISC-V ได้กลายเป็นปรากฏการณ์ที่เติบโตอย่างรวดเร็วในด้านความนิยม ตั้งแต่เปิดตัวในปี 2010 เราเชื่อว่าคู่มือสำหรับโปรแกรมเมอร์ที่กระชับจะช่วยจุดประกายการเติบโตและกระตุ้นให้ผู้อ่านเข้าใจว่าทำไม RISC-V จึงเป็นชุดคำสั่งที่น่าสนใจ และแตกต่างจากสถาปัตยกรรมชุดคำสั่ง (ISA) แบบดั้งเดิมในอดีตอย่างไร

หนังสือสำหรับสถาปัตยกรรมชุดคำสั่งอื่นๆ สร้างแรงบันดาลใจให้เรา แม้ว่าเราจะหวังว่าความเรียบง่ายของ RISC-V จะหมายถึงการเขียนหนังสือที่สั้นกว่าหนังสืออื่นๆ อย่าง *See MIPS Run* ที่มีความยาวกว่า 500 หน้า ด้วยความยาวรวมเพียงหนึ่งในสาม ของหนังสือเหล่านั้น อย่างน้อยในแง่ความเรียบง่ายเราก็ประสบความสำเร็จแล้ว ในความเป็นจริง มี 10 บทที่แนะนำแต่ละส่วนประกอบของชุดคำสั่ง RISC-V แบบโมดูลาร์ มีจำนวน 170 หน้า—แม้จะเฉลี่ยเกือบหนึ่งรูปต่อหน้า (รวม 121รูป)—ซึ่งทำให้อ่านได้รวดเร็ว

หลังจากอธิบายหลักการออกแบบชุดคำสั่งแล้ว เราได้แสดงให้เห็นว่าสถาปนิกของ RISC-V เรียนรู้จากชุดคำสั่งในช่วง 40 ปีที่ผ่านมา ด้วยการนำแนวคิดดี ๆ มาใช้และหลีกเลี่ยงข้อผิดพลาดในอดีต การประเมินสถาปัตยกรรมชุดคำสั่ง (ISA) นั้นขึ้นอยู่กับสิ่งที่ถูกนำออกไม่น้อยไปกว่าสิ่งที่ถูกรวมไว้

จากนั้นเราแนะนำแต่ละส่วนประกอบของสถาปัตยกรรมแบบโมดูลาร์นี้ ผ่านลำดับของบทต่างๆ โดยทุกบทจะมีโปรแกรมในภาษาแอสเซมบลีของ RISC-V ที่แสดงการใช้คำสั่งที่แนะนำในบทนั้น ซึ่งช่วยให้โปรแกรมเมอร์ภาษาแอสเซมบลีเรียนรู้โค้ด RISC-V ได้ง่ายขึ้น นอกจากนี้ เรายังแสดงโปรแกรมที่เทียบเท่าใน ARM, MIPS และ x86 เพื่อเน้นย้ำถึงความเรียบง่ายและประโยชน์ด้านต้นทุน พลังงาน และประสิทธิภาพของ RISC-V

เพื่อให้หนังสือเล่มนี้อ่านสนุกขึ้น เราได้แทรกแถบข้างเกือบ 80 ข้อความไว้ที่ขอบหน้าพร้อมคำอธิบายที่น่าสนใจเกี่ยวกับเนื้อหา และยังมีภาพราว 125 ภาพที่ขอบหน้าเพื่อเน้นตัวอย่างการออกแบบ ISA ที่ดี (ขอบหน้าของเราใช้งานได้อย่างคุ้มค่า!) สำหรับผู้อ่านที่ทุ่มเท เราจะเพิ่มส่วนขยายความราว 35 ส่วนตลอดทั้งเล่ม ซึ่งผู้อ่านสามารถเจาะลึกในหัวข้อที่สนใจได้ ส่วนเหล่านี้ไม่จำเป็นต่อการทำความเข้าใจเนื้อหาอื่นในหนังสือ ดังนั้นหากไม่น่าสนใจก็สามารถข้ามไปได้ สำหรับผู้ที่หลงใหลในสถาปัตยกรรมคอมพิวเตอร์ เราได้

อ้างอิงบทความและหนังสือ 37 เล่มที่อาจขยายขอบเขตมุมมองของคุณ เราเองก็ได้เรียนรู้มากมายจากการอ่านงานเหล่านี้เพื่อเขียนหนังสือเล่มนี้!

ทำไมถึงมีคำพูดอ้างอิงมากมายขนาดนั้น?

เราคิดว่าคำพูดอ้างอิงเหล่านี้ทำให้การอ่านหนังสือสนุกยิ่งขึ้น จึงได้สอดแทรก 34 คำพูดไว้ตลอดทั้งเล่ม คำพูดอ้างอิงเหล่านี้ยังเป็นกลไกที่มีประสิทธิภาพในการถ่ายทอดความรู้จากผู้ใหญ่ผู้มีประสบการณ์ไปสู่มือใหม่ และช่วยกำหนดมาตรฐานทางวัฒนธรรมสำหรับการออกแบบ ISA ที่ดีด้วย เรายังต้องการให้ผู้อ่านได้รับความรู้ทางประวัติศาสตร์ของสาขานี้ด้วย ซึ่งนั่นคือเหตุผลที่เรานำเสนอคำพูดจากนักวิทยาศาสตร์คอมพิวเตอร์และวิศวกรชื่อดังตลอดทั้งเล่ม

การแนะนำและการอ้างอิง

เราตั้งใจให้หนังสือเล่มนี้ทำหน้าที่ทั้งเป็นการแนะนำและเป็นแหล่งอ้างอิงสำหรับ RISC-V สำหรับนักศึกษาและโปรแกรมเมอร์ระบบฝังตัวที่สนใจเขียนโค้ด RISC-V หนังสือเล่มนี้สมมติว่าผู้อ่านเคยเห็นชุดคำสั่งอย่างน้อยหนึ่งชุดมาก่อน หากยังไม่เคยมีพื้นฐาน อาจลองอ่านหนังสือแนะนำสถาปัตยกรรมที่เกี่ยวข้องของเรา ซึ่งใช้ RISC-V เป็นพื้นฐาน: *Computer Organization and Design RISC-V Edition, Second Edition: The Hardware Software Interface*

เอกสารอ้างอิงที่กระชับในหนังสือเล่มนี้ประกอบด้วย:

- **บัตร อ้างอิง**—เอกสาร สรุป หนึ่ง หน้า (สอง ด้าน) ครอบคลุม ทั้ง RV32GCV และ RV64GCV ซึ่งรวมชุดคำสั่งพื้นฐานและส่วนขยายที่กำหนดทั้งหมด: RVI, RVM, RVA, RVF, RVD, RVC และแกนหลักของ RVV
- **แผนภาพคำสั่ง**—คำอธิบายกราฟิกครึ่งหน้าสำหรับแต่ละส่วนขยายคำสั่ง ซึ่งเป็นรูปแรกของแต่ละบท แสดงรายชื่อเต็มของคำสั่ง RISC-V ทั้งหมด ในรูปแบบที่ทำให้เห็นความแตกต่างของแต่ละคำสั่งได้อย่างชัดเจน ดูภาพประกอบที่ 2.1, 4.1, 5.1, 6.1, 7.1, 8.1, 10.1, 10.2, 10.3, และ 10.4
- **ตารางรหัสคำสั่ง (Opcodes Maps)**—ตารางเหล่านี้แสดงโครงสร้างคำสั่ง รหัสดำเนินการ (opcode) ประเภทรูปแบบคำสั่ง และชื่อย่อของคำสั่งสำหรับแต่ละส่วนขยายคำสั่ง โดยใช้พื้นที่เพียงบางส่วนของหน้า ดูภาพประกอบที่ 2.3, 3.3, 3.4, 4.2, 5.11, 5.12, 6.2, 7.6, 7.5, 7.7, 10.5, และ 11.1 (แผนภาพคำสั่งและตารางรหัสคำสั่งเป็นแรงบันดาลใจให้ใช้คำว่า atlas ในชื่อรองของหนังสือ)
- **รายการคำสั่ง**—ภาคผนวก A เป็นคำอธิบายอย่างละเอียดของทุกคำสั่งและคำสั่งเทียม (pseudoinstruction) ของ RISC-V รวมถึงชื่อการดำเนินการและตัวดำเนินการ คำอธิบายเป็นภาษาไทย คำจำกัดภาษาการถ่ายโอนเรจิสเตอร์ ส่วนขยายของ RISC-V ที่

เกี่ยวข้อง ชื่อเต็มของคำสั่ง รูปแบบคำสั่ง แผนภาพของคำสั่งที่แสดงรหัสดำเนินการ และอ้างอิงถึงคำสั่งแบบบีบอัด ทั้งหมดนี้รวมอยู่ในพื้นที่ไม่ถึง 70 หน้า

- **การแปลงคำสั่ง**—ภาคผนวก B ช่วยโปรแกรมเมอร์ภาษาแอสเซมบลีที่มีประสบการณ์ โดยให้ตารางที่แสดงคำสั่ง ARM-32 หรือ x86-32 ที่เทียบเท่ากับ คำสั่ง RV32I รวมถึงแสดงผลลัพธ์ของคอมไพเลอร์ C สำหรับโปรแกรมการค้นหาแบบทริกอย่างง่าย สำหรับสถาปัตยกรรมชุดคำสั่งทั้งสามนี้ และอธิบายความแตกต่าง เล็กน้อยระหว่างสถาปัตยกรรมเหล่านี้ พร้อมด้วยคำแนะนำเกี่ยวกับวิธีการแปลโค้ด จากสถาปัตยกรรมเก่าไปเป็น RISC-V ซึ่งง่ายกว่าที่คิด
- **การแปลภาษา**—ผู้อ่านที่ใช้ภาษาอื่นจะพบว่าคำศัพท์เทียบเคียงคำศัพท์ยากในภาคผนวก C เป็นแหล่งอ้างอิงที่สะดวกขณะเรียนรู้ RISC-V ด้วยหนังสือเล่มนี้
- **ดัชนี**—ช่วยให้ผู้อ่านค้นหาหน้าที่อธิบายคำอธิบายคำสั่ง คำจำกัดความ หรือแผนภาพได้ทั้งจากชื่อเต็มหรือชื่อย่อ จัดเรียงเหมือนพจนานุกรม

ข้อผิดพลาดและเนื้อหาเพิ่มเติม

เราตั้งใจที่จะรวบรวมข้อผิดพลาดเหล่านั้นเข้าด้วยกันและเผยแพร่การอัปเดตปีละสองสามครั้ง เว็บไซต์ของหนังสือจะแสดงเวอร์ชันล่าสุดของหนังสือ พร้อมคำอธิบายสั้นๆ เกี่ยวกับการเปลี่ยนแปลงจากเวอร์ชันก่อนหน้า คุณสามารถตรวจสอบข้อผิดพลาดเก่าๆ และรายงานรายการใหม่ ได้ที่เว็บไซต์ของหนังสือ (www.riscbook.com) เราขออภัยล่วงหน้าสำหรับปัญหาที่พบในฉบับนี้ และเราหวังว่าจะได้รับความคิดเห็นของคุณ เพื่อช่วยปรับปรุงเนื้อหาของหนังสือเล่มนี้ให้ดียิ่งขึ้น

ประวัติของฉบับพิมพ์ครั้งแรก

ที่การประชุมเชิงปฏิบัติการ RISC-V ครั้งที่ 6 ซึ่งจัดขึ้นระหว่างวันที่ 8–11 พฤษภาคม 2560 ที่เซี่ยงไฮ้ เราเห็นความจำเป็นสำหรับหนังสือเล่มนี้ และเริ่มเขียนไม่กี่สัปดาห์ต่อมา ด้วยประสบการณ์การเขียนหนังสือที่มากกว่าของแพตเตอร์สัน แผนคือให้เขาเขียนเนื้อหาส่วนใหญ่ของบทต่างๆ เราร่วมกันวางโครงสร้างและเป็นผู้ตรวจทานแรกสำหรับบทของกันและกัน โดยแพตเตอร์สันเป็นผู้เขียนบทที่ 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, การ์ดอ้างอิง, และคำนำ ส่วนวอเตอร์แมนเขียนบทที่ 11, ภาคผนวก A—ซึ่งเป็นส่วนที่ยาวที่สุดของหนังสือ—ภาคผนวก B, และเขียนโค้ดโปรแกรมทั้งหมดในหนังสือ วอเตอร์แมน ยังรักษารูปแบบ LaTeX ที่ได้รับจาก Armando Fox ซึ่งช่วยให้เราผลิตหนังสือเล่มนี้ได้

เราเสนอฉบับเบต้าของหนังสือให้กับนักศึกษา UC Berkeley 800 คน ในภาคเรียนฤดูใบไม้ร่วง 2560 ผู้อ่านพบเพียงข้อผิดพลาดด้านการพิมพ์และปัญหา LaTeX เพียงเล็กน้อย ซึ่งเราแก้ไขสำหรับฉบับพิมพ์ครั้งแรก นอกจากนี้ เรายังปรับปรุงไอคอนที่ขอบหน้าให้ง่ายขึ้น และแก้ไขภาพประกอบบางภาพที่ดูไม่ดีเท่าที่หวังเมื่อพิมพ์ลงกระดาษ

ที่สำคัญ ฉบับพิมพ์ครั้งแรกได้ขยายบท 11 ให้รวมรีจิสเตอร์ควบคุมและสถานะกว่า 60 ตัว และเพิ่มภาคผนวก B เพื่อช่วยโปรแกรมเมอร์ที่สนใจแปลงโปรแกรมภาษาแอสเซมบลีจาก ISA รุ่นเก่าเป็น RISC-V

ฉบับแรกเผยแพร่ทันเวลาสำหรับการประชุมเชิงปฏิบัติการ RISC-V ครั้งที่ 7 ซึ่งจัดขึ้นในซิลิคอนแวลลีย์ ระหว่างวันที่ 28–30 พฤศจิกายน 2560

RISC-V เป็นผลพลอยได้จาก โครงการวิจัยของ Berkeley¹ ที่พัฒนาเทคโนโลยีเพื่อให้ง่ายต่อการสร้างฮาร์ดแวร์และซอฟต์แวร์แบบขนานง่ายขึ้น

ประกาศเกียรติคุณสำหรับฉบับแรก

เราขอขอบคุณ อาร์มันโด ฟอกซ์ สำหรับการใช้ระบบ LaTeX ของเขาและคำแนะนำในการเผยแพร่ด้วยตนเอง

คำขอบคุณอย่างยิ่งต่อผู้อ่านร่างต้นฉบับของหนังสือและให้คำแนะนำที่มีค่า: Krste Asanović, Nikhil Athreya, C. Gordon Bell, Elkim Fuentes, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor และ Megan Wachs

และสุดท้าย เราขอบคุณนักศึกษา UC Berkeley จำนวนมากสำหรับความช่วยเหลือในการแก้ไขข้อบกพร่องและความสนใจอย่างต่อเนื่องในเนื้อหา!!

เราขอขอบคุณนักแปลจำนวนมาก ที่ได้แปลเนื้อหาให้ให้บริการฟรี ในภาษาจีน ญี่ปุ่น จีนกลาง โปรตุเกส สเปน และไทย (ดู riscbook.com)

ประวัติและคำขอบคุณสำหรับฉบับพิมพ์ครั้งที่สอง

เมื่อส่วนขยายเวกเตอร์เสร็จสมบูรณ์ เราเห็นว่าถึงเวลาเริ่มทำงานฉบับพิมพ์ครั้งที่สองเนื่องจากบทเกี่ยวกับเวกเตอร์ในฉบับแรกเริ่มล้าสมัยอย่างมาก เราเริ่มติดต่อกันทางอีเมลในเดือนพฤษภาคม 2564 และเริ่มเขียนอย่างจริงจังในเดือนถัดมา นอกจากการปรับปรุงบทเวกเตอร์ (บทที่ 8) แล้ว เรายังตัดสินใจที่จะเพิ่มความแม่นยำครั้งเท่าให้กับบทจุดทศนิยม (บทที่ 5) และเขียนบทใหม่เกี่ยวกับส่วนขยายการจัดการบิต (บทที่ 9) และไฮเปอร์ไวเซอร์ รวมถึงปรับปรุงบัตรอ้างอิง, คำนำ, บทนำ, บท RV64, บททิศทางในอนาคต และภาคผนวก A (เวอร์เตอร์แมนเป็นผู้เขียนบทไฮเปอร์ไวเซอร์ ปรับปรุงภาคผนวก A และจัดหาตัวอย่างโปรแกรมทั้งหมด ส่วนแพตเตอร์สรับผิดชอบการปรับปรุงที่เหลือ)

เรายินดีที่การเปลี่ยนแปลงเหล่านี้เพิ่มความยาวของบทเพิ่มขึ้นเพียง 47 หน้า ในขณะที่ภาคผนวกเพิ่มขึ้น 42 หน้า ทำให้ฉบับพิมพ์ครั้งที่สองยังคงค่อนข้างบาง เราตัดสินใจให้ PDF ของเวอร์ชันภาษาอังกฤษของฉบับนี้เผยแพร่ฟรี เช่นเดียวกับรุ่นในภาษาอื่น ๆ (ดู riscbook.com)

เราขอขอบคุณผู้อ่านฉบับร่างของฉบับที่สอง สำหรับสำหรับข้อเสนอแนะที่มีค่า: Cliff Young และคนอื่น ๆ

เดวิด แพตเตอร์สัน และ แอนดรูว์ วอเตอร์แมน
๑๖ พฤศจิกายน พ.ศ. ๒๕๖๔
เบิร์กลีย์ แคลิฟอร์เนีย

หมายเหตุ

¹<http://parlab.eecs.berkeley.edu>

ทำไมต้อง RISC-V ?

เลโอนาร์โด ดา วินชี
(ค.ศ. 1452–1519) เป็น
สถาปนิกในยุคฟื้นฟู
ศิลปวิทยา วิศวกร ประติ
มากรและจิตรกรภาพโมนา
ลิซา



ความเรียบง่ายคือความซับซ้อนขั้นสูงสุด

—เลโอนาร์โด ดา วินชี

1.1 บทนำ

เป้าหมายของ RISC-V (อ่านว่า “RISC Five”) คือการเป็น *สถาปัตยกรรมชุดคำสั่ง* (Instruction Set Architecture: ISA) แบบสากล:

- ควรใช้งานได้กับโปรเซสเซอร์ทุกขนาด ตั้งแต่ตัวควบคุมระบบฝังตัวขนาดเล็กมาก ไปจนถึงคอมพิวเตอร์ประสิทธิภาพสูง
- ควรทำงานได้ดีกับซอฟต์แวร์สแต็กยอดนิยมและภาษาโปรแกรมที่หลากหลาย
- ควรรองรับเทคโนโลยีการสร้าง ฮาร์ดแวร์ ทุกประเภท ได้แก่ ชิปที่โปรแกรมใหม่ได้ (Field-Programmable Gate Arrays (FPGAs)), ชิปที่ออกแบบเฉพาะงาน (Application-Specific Integrated Circuits (ASICs)), ชิปแบบออกแบบเองทั้งหมด (full-custom chips) รวมถึงเทคโนโลยีอุปกรณ์ใหม่ๆ ในอนาคต
- ควรมีประสิทธิภาพดีกับสไตล์ของไมโครสถาปัตยกรรม (microarchitecture) ทุกแบบ ได้แก่ การควบคุมแบบไมโครโค้ดหรือฮาร์ดไวร์; ไปป์ไลน์แบบลำดับคำสั่ง (in-order), แยกส่วน (decoupled), นอกลำดับคำสั่ง (out-of-order); การออกคำสั่งแบบทีละคำสั่ง (single issue) หรือหลายคำสั่งพร้อมกัน (superscalar); และอื่นๆ
- ควรรองรับ การปรับแต่ง เฉพาะทาง อย่าง กว้างขวาง เพื่อเป็นฐานสำหรับ ตัวเร่งประสิทธิภาพที่กำหนดเอง ซึ่งมีความสำคัญเพิ่มขึ้นเมื่อกฎของมัวร์ค่อย ๆ ลดความสำคัญลง
- ต้องมีความเสถียร โดยที่ ISA พื้นฐานไม่ควรถูกเปลี่ยนแปลง และที่สำคัญกว่านั้น ISA ต้องไม่ถูกยกเลิก เช่นที่เคยเกิดขึ้นกับ ISA ที่เป็นแบบกรรมสิทธิ์ เช่น AMD Am29000, Digital Alpha, Digital VAX, Hewlett Packard PA-RISC, Intel i860, Intel i960, Motorola 88000 และ Zilog Z8000

>\$50B		>\$5B, <\$50B		>\$0.5B, <\$5B	
Alibaba	China	BAE Systems	UK	AMD	USA
Google	USA	MediaTek	Taiwan	Andes Technology	China
Huawei	China	Micron Tech.	USA	Cadence	USA
IBM	USA	Nvidia	USA	Integrated Device Tech.	USA
Samsung	Korea	NXN Semi.	Netherlands	Mellanox Technology	Israel
Sony	Japan	Qualcomm	USA	Seagate	USA
		Western Digital	USA	Xilinx	USA

รูปที่ 1.1: สมาชิกขององค์กรของ RISC-V International ณ เดือนสิงหาคม 2021 จัดอันดับตามยอดขายประจำปี บริษัทคอลัมน์ซ้าย: มียอดขายประจำปีเกิน \$US 50B คอลัมน์กลาง: บริษัทที่มีมียอดขายน้อยกว่า \$US 50B แต่มากกว่า \$US 5B และบริษัทในคอลัมน์ขวามีมียอดขายน้อยกว่า \$US 5B แต่มากกว่า \$US 0.5B มูลนิธิรวมถึงองค์กรอีก 300 แห่งที่มีสำนักงานใหญ่ใน 28 ประเทศ ดูข้อมูลเพิ่มเติมได้ที่ www.riscv.org

RISC-V ถือว่าแตกต่าง ไม่ใช่แค่เพราะเป็น ISA ที่ค่อนข้างใหม่—ถือกำเนิดในปี 2010 ในขณะที่ ISA อื่นๆ ส่วนใหญ่มาจากยุคตั้งแต่ทศวรรษ 1970 หรือ 1980—but ยังแตกต่างเพราะมันเป็น ISA แบบเปิด ต่างจากสถาปัตยกรรมก่อนหน้านี้เกือบทั้งหมด อนาคตของ RISC-V ไม่ได้ขึ้นอยู่กับชะตาหรือการตัดสินใจของบริษัทใดบริษัทหนึ่ง ซึ่งเคยทำให้ ISA หลายตัวต้องล้มเหลวไปเพราะบริษัทเจ้าของ แต่ RISC-V อยู่ภายใต้โครงการเปิดแบบไม่แสวงหากำไร เป้าหมายของ RISC-V International คือรักษาความเสถียรของ RISC-V พัฒนาแบบช้าๆ และรอบคอบ ตัดสินใจจากเหตุผลทางเทคนิคเท่านั้น และพยายามทำให้ RISC-V ได้รับความนิยมในโลกฮาร์ดแวร์เหมือนกับที่ Linux ได้รับความนิยมในโลกของระบบปฏิบัติการ การที่มีบริษัทใหญ่จำนวนมากเข้าร่วมแสดงถึงความสำเร็จชีวิตชีวา ในรูป 1.1 ซึ่งแสดงรายชื่อสมาชิกองค์กรที่ใหญ่ที่สุดของ RISC-V International

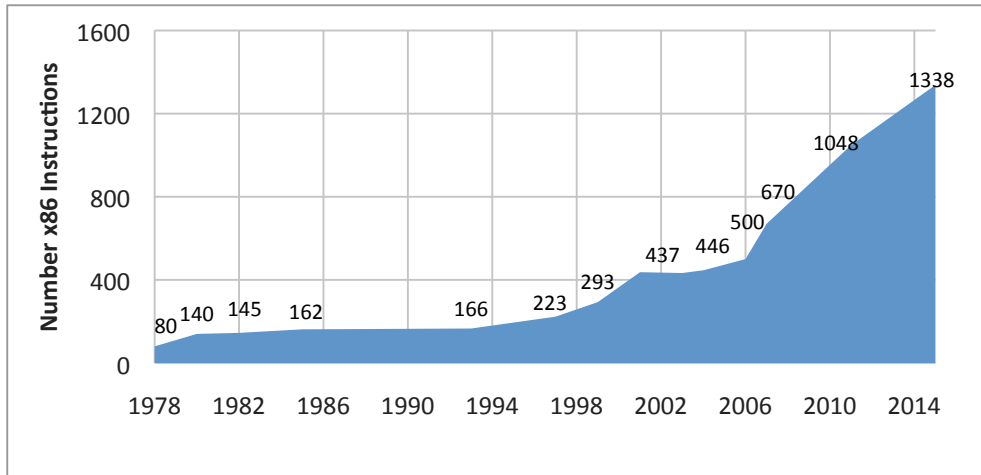
เราเพิ่มแถบด้านข้างในส่วนขอบ เพื่อเสนอความคิดเห็นที่น่าสนใจ ตัวอย่างเช่น RISC-V เดิมถูกพัฒนาขึ้น เพื่อให้ภายในงานวิจัยและการเรียนการสอนที่ UC Berkeley แต่ต่อมาได้กลายเป็น ISA แบบเปิด เนื่องจากมีคนนอกเริ่มนำไปใช้งานด้วยตัวเอง สถาบันของ RISC-V รับรู้ถึงความสนใจจากภายนอก เมื่อพวกเขาเริ่มได้รับ ข้อร้องเรียนเกี่ยวกับการเปลี่ยนแปลง ISA ในเอกสารการเรียนการสอน ซึ่งถูกเผยแพร่ไว้บนเว็บหลังจากที่ผู้พัฒนาเข้าใจถึงความต้องการนี้แล้ว พวกเขาจึงเริ่มพยายามทำให้ RISC-V กลายเป็นมาตรฐาน ISA แบบเปิด

1.2 ISA แบบโมดูลาร์ เทียบกับ ISA แบบเพิ่มขยาย

อินเทลเคยวางแผนอนาคตไว้กับโปรเซสเซอร์ระดับสูง แต่การพัฒนาจะต้องใช้เวลาอีกหลายปี เพื่อแข่งขันกับ Zilog อินเทลจึงพัฒนาโปรเซสเซอร์ชั่วคราวขึ้นมาชื่อ 8086 โดยถูกออกแบบให้เป็น โปรเซสเซอร์ชั่วคราวและไม่มีรุ่นต่อ แต่สิ่งที่เกิดขึ้นจริงคือ โปรเซสเซอร์ระดับสูงที่วางแผนไว้เข้าสู่ตลาดช้า และเมื่อมันออกมา ก็ช้าเกินไปที่จะแข่งขันได้ ดังนั้นสถาปัตยกรรมของ 8086 จึงยังคงอยู่—พัฒนาเป็นโปรเซสเซอร์ 32 บิต และต่อมากลายเป็น 64 บิต แม้ชื่อจะเปลี่ยนไป (80186, 80286, i386, i486, Pentium) แต่ชุดคำสั่งพื้นฐานยังคงอยู่

—สตีเฟน พี มอร์ส, สถาปนิกของ 8086 [Morse 2017]

แนวทางทั่วไปของการออกแบบสถาปัตยกรรมคอมพิวเตอร์คือ การใช้ ISA แบบเพิ่มขยาย ซึ่งโปรเซสเซอร์รุ่นใหม่ต้องรองรับทั้ง ส่วนขยายของชุดคำสั่งใหม่และชุดคำสั่งเก่าทั้งหมด เพื่อการรักษาความเข้ากันได้ย้อนหลังในระดับไบนารี เพื่อให้โปรแกรมแบบไบนารีที่สร้างเมื่อหลายสิบปีก่อน ยังสามารถรันได้บนโปรเซสเซอร์รุ่นใหม่ เมื่อรวมกับแรงจูงใจทางการตลาด ที่ต้องการเพิ่ม “คำสั่งใหม่” ทุกครั้งที่ออก CPU รุ่นใหม่ ทำให้ ISA มีขนาดใหญ่ขึ้นเรื่อยๆ ตามเวลา ตัวอย่างจากรูป 1.2 แสดงให้เห็นการเพิ่มขึ้นของจำนวนคำสั่งของ ISA ที่



รูปที่ 1.2: การเติบโตของชุดคำสั่ง x86 ตลอดช่วงเวลา ชุดคำสั่ง x86 เริ่มต้นในปี 1978 โดยมีเพียง 80 คำสั่ง แต่เพิ่มขึ้นถึง 16 เท่า กลายเป็น 1338 คำสั่งในปี 2015 และยังคงเพิ่มขึ้นต่อไป ที่น่าทึ่งคือ กราฟนี้ยังถือว่า “ประหลาด” เพราะบล็อกของ Intel ระบุว่า มีคำสั่งใหม่เพิ่มหนึ่งคำสั่ง *ทุกๆ สัปดาห์* ระหว่างปี 1978 ถึง 2015 เรานับคำสั่งในภาษาแอสเซมบลี ในขณะที่ Intel น่าจะนับคำสั่งในภาษาเครื่อง ตามที่อธิบายในบทที่ 8 เหตุผลที่คำสั่งเพิ่มขึ้นมากที่สำคัญมาจาก ISA ของ x86 ใช้คำสั่ง SIMD เพื่อการประมวลผลข้อมูลแบบขนาน

มีอิทธิพลอย่างมากในปัจจุบัน คือ 80x86 ซึ่งมีจุดเริ่มต้นตั้งแต่ปี 1978 แต่ตลอดช่วงเวลายาวนานนั้น ได้มีการเพิ่มคำสั่งใหม่เฉลี่ยประมาณ *สามคำสั่งต่อเดือน*

แนวทางนี้หมายความว่า การนำ x86-32 (ซึ่งเป็นชื่อที่ใช้เรียก x86 เวอร์ชัน 32 บิต) ไปใช้งานทุกครั้ง จะต้องรองรับ “ข้อผิดพลาดจากส่วนขยายในอดีต” ด้วย แม้ว่าสิ่งเหล่านั้นจะไม่มี ความหมาย หรือไม่จำเป็นอีกต่อไปแล้วก็ตาม ตัวอย่างเช่น รูป 1.3 อธิบายคำสั่ง `aaa` (ASCII Adjust after Addition) ของ x86 ซึ่งเป็นคำสั่งที่หมดความจำเป็นในการใช้งานไปนานแล้ว

หากเปรียบเทียบ ลองจินตนาการถึงร้านอาหารที่ขายอาหารแบบราคาคงที่ (fixed-price) เริ่มต้นจากมื้อเล็กๆ ที่มีแค่แฮมเบอร์เกอร์และมิลค์เชค ต่อมาเพิ่ม เฟรนช์ฟรายส์ ไอศกรีมชานเดย์ สลัด พาย ไวน์ พาสต้าเจ สเต็ก เปียร์ และอื่นๆ ต่อไปเรื่อยๆ จนกลายเป็นบุฟเฟ่ต์ขนาดใหญ่ แม้ภาพรวมจะดูไม่สมเหตุสมผล แต่ลูกค้าสามารถหาอาหารที่เคยกินในอดีตได้เสมอ ข้อเสียคือ ลูกค้าต้องจ่ายค่าอาหารที่แพงขึ้นเพื่อรองรับเมนูทั้งหมด ที่ขยายใหญ่ขึ้นในแต่ละมื้อ

นอกจากจะเป็น ISA ที่ใหม่และเปิดแล้ว RISC-V ยังมีความโดดเด่นตรงที่ แตกต่างจาก ISA เกือบทั้งหมดในอดีต คือมีลักษณะเป็นแบบ *โมดูลาร์* (modular) แกนหลักคือชุดคำสั่งพื้นฐานที่เรียกว่า *RV32I* ซึ่งสามารถรองรับการทำงานของซอฟต์แวร์ได้ครบถ้วน *RV32I* ถูกกำหนดให้คงที่และจะไม่เปลี่ยนแปลงอีก ทำให้นักพัฒนาคอมไพเลอร์ นักพัฒนา

```

AL เป็นรีจิสเตอร์ต้นทางและปลายทางค่าเริ่มต้น
If 4 บิตล่างของรีจิสเตอร์ AL มีค่า > 9
หรือ กำหนดให้แฟลกตัวทด AF = 1
Then
    บวก 6 เข้ากับ 4 บิตล่างของรีจิสเตอร์ AL และไม่ตรวจสอบ overflow
    เพิ่ม 1 ให้กับ 8 บิตบนของ AL
    กำหนดให้แฟลกการบวก CF = 1
    กำหนดให้แฟลกการบวกเพิ่มเติม AF = 1
Else
    CF = AF = 0
4 บิตบนของรีจิสเตอร์ AL = 0

```

รูปที่ 1.3: คำอธิบายของคำสั่ง x86-32 *aaa* (*ASCII Adjust after Addition*) คำสั่งนี้ใช้สำหรับการคำนวณแบบ Binary Coded Decimal (BCD) ซึ่งปัจจุบันได้กลายเป็นสิ่งที่ล้าสมัยไปแล้วในวงการเทคโนโลยีสารสนเทศ นอกจากนี้ x86 ยังมีคำสั่งที่เกี่ยวข้องอีกสามคำสั่ง ได้แก่ *aas* สำหรับการลบ *aam* สำหรับการคูณ และ *aad* สำหรับการหาร เนื่องจากแต่ละคำสั่งมีขนาดเพียง 1 ไบต์ คำสั่งทั้งสี่นี้จึงรวมกันใช้พื้นที่ถึง 1.6% (4/256) ของพื้นที่รหัสดำเนินการ(opcode) ซึ่งเป็นทรัพยากรที่มีจำกัดและมีค่ามาก

ระบบปฏิบัติการ และโปรแกรมเมอร์ภาษาแอสเซมบลี มีเป้าหมายที่เสถียร สำหรับการพัฒนา ความเป็นโมดูลาร์มาจากส่วนขยายมาตรฐานที่เลือกได้ ซึ่งฮาร์ดแวร์สามารถรวมหรือไม่รวมก็ได้ตามความต้องการของแอปพลิเคชัน ความเป็นโมดูลาร์นี้ทำให้สามารถสร้าง RISC-V ในมีขนาดที่เล็กมากและใช้พลังงานต่ำ ซึ่งเป็นสิ่งสำคัญอย่างยิ่งสำหรับงานระบบฝังตัว หากแจ้งให้คอมไพเลอร์ RISC-V ทราบว่าฮาร์ดแวร์รองรับส่วนขยาย (extensions) ไต่บ้าง คอมไพเลอร์ก็จะสามารถสร้างโค้ดที่เหมาะสมที่สุดสำหรับฮาร์ดแวร์นั้นได้ แนวปฏิบัติคือ จะใช้ตัวอักษรของส่วนขยายต่อท้ายชื่อ เพื่อระบุว่ามีส่วนขยายใดบ้าง ตัวอย่างเช่น RV32IMFD หมายถึงการเพิ่มส่วนขยายการคูณ (RV32M) จุดลอยตัวความแม่นยำเดียว (RV32F) และจุดลอยตัวความแม่นยำคู่ (RV32D) เข้าไปในชุดคำสั่งพื้นฐานที่จำเป็น (RV32I)

หากซอฟต์แวร์มีการใช้คำสั่งของ RISC-V ที่อยู่ในส่วนขยายเสริม แต่ฮาร์ดแวร์ไม่ได้อรองรับคำสั่งนั้น ฮาร์ดแวร์จะเกิดการขัดจังหวะพิเศษ (trap) และจะไปเรียกใช้ฟังก์ชันที่ทำงานเทียบเท่ากันในซอฟต์แวร์ ซึ่งอยู่ในไลบรารีมาตรฐาน

ย้อนกลับไปที่การเปรียบเทียบเดิม RISC-V เปรียบเสมือน “เมนูอาหาร” มากกว่าบุฟเฟต์ เซฟจำเป็นต้องปรุงเฉพาะสิ่งที่ลูกค้าต้องการเท่านั้น—ไม่ใช่ต้องเตรียมอาหารครบทุกอย่างในทุกมื้อ—และลูกค้าก็จ่ายเงินเฉพาะสิ่งที่สั่ง RISC-V จึงไม่มีความจำเป็นต้องเพิ่มคำสั่งเพียงเพื่อสร้างกระแสทางการตลาด RISC-V International เป็นผู้ตัดสินใจว่าจะเพิ่มตัวเลือกใหม่ลงในเมนูเมื่อใด และจะทำเช่นนั้นก็ต่อเมื่อมีเหตุผลทางเทคนิคที่ชัดเจน หลังจากผ่านการอภิปรายอย่างเปิดกว้างและยาวนาน โดยคณะผู้เชี่ยวชาญทั้งด้านฮาร์ดแวร์และซอฟต์แวร์ แม้ว่าจะมีตัวเลือกใหม่เพิ่มเข้ามาในเมนู สิ่งเหล่านั้นก็ยังคงเป็น “ตัวเลือก” ไม่ใช่ข้อบังคับ สำหรับการออกแบบในอนาคตทั้งหมด เหมือนกับ ISA แบบเพิ่มขยาย

1.3 พื้นฐานการออกแบบ ISA

ก่อนที่จะแนะนำ ISA ของ RISC-V การทำความเข้าใจหลักการพื้นฐานและการแลกเปลี่ยน

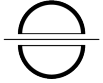




ความเรียบง่าย



ประสิทธิภาพ



แยกสถาปัตยกรรม กับนำไปใช้



ที่ว่างสำหรับส่วนขยาย



ขนาดโปรแกรม



การเขียนโปรแกรม

ที่สถาปนิกคอมพิวเตอร์ต้องพิจารณาในการออกแบบ ISA จะช่วยให้เข้าใจได้ดียิ่งขึ้น ด้านล่างนี้คือรายการของตัวชี้วัดทั้งเจ็ดข้อ พร้อมไอคอนที่จะแสดงไว้บริเวณขอบหน้า เพื่อเน้นให้เห็นว่าจุดใดที่ RISC-V ตรงกับตัวชี้วัดเหล่านี้ในบทต่อไป (ที่ปกหลังของหนังสือฉบับพิมพ์จะมีคำอธิบายสำหรับไอคอนเหล่านี้)

- ต้นทุน (ไอคอนเหรียญดอลลาร์สหรัฐ)
- ความเรียบง่าย (ไอคอนวงล้อ)
- ประสิทธิภาพ (ไอคอนมาตรวัดความเร็ว)
- การแยกสถาปัตยกรรมออกจากการนำไปใช้งาน (ไอคอนวงกลมที่แยกออกเป็นสองส่วน)
- ความสามารถในการขยายในอนาคต (ไอคอนหีบเพลง)
- ขนาดโปรแกรม (ไอคอนลูกศรสองตัวตรงข้ามบรรทัด)
- ความง่ายในการเขียนโปรแกรม / คอมไพล์ / ลิงก์ (“ไอคอน “ง่ายเหมือน A B C””)

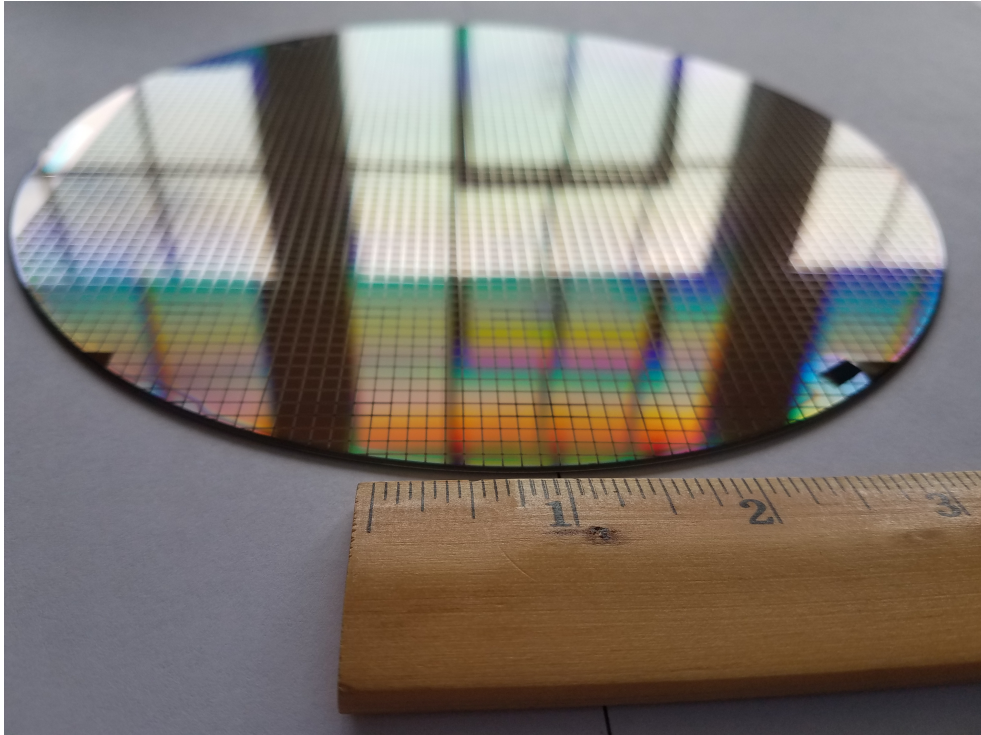
เพื่อให้เห็นภาพชัดเจนว่าเราหมายถึงอะไร ในส่วนนี้เราจะยกตัวอย่างการตัดสินใจบางอย่างจาก ISA รุ่นเก่า ซึ่งเมื่อมองย้อนกลับไปแล้วอาจไม่เหมาะสมนัก และในหลายกรณี RISC-V ได้ตัดสินใจได้ดีกว่าอย่างมาก

ต้นทุน โปรเซสเซอร์ถูกสร้างขึ้นในรูปของ วงจรรวม (integrated circuits) ซึ่งมักเรียกกันทั่วไปว่า ชิพ (chips) หรือ ได (dies) ที่เรียกว่า “ได” เพราะเริ่มต้นจากการเป็นชิ้นส่วนของแผ่นเวเฟอร์ (wafer) ทรงกลมขนาดใหญ่ ซึ่งจะถูกตัด ออกเป็นชิ้นเล็กๆ หลายชิ้น รูปที่ 1.4 แสดงเวเฟอร์ของโปรเซสเซอร์ RISC-V ต้นทุนมีความสัมพันธ์อย่างมากกับ พื้นที่ของได (die area):

$$\text{cost} \approx f(\text{die area}^2)$$

ยิ่งไดมีขนาดเล็ก จำนวนไดต่อเวเฟอร์ยิ่งมากขึ้น และต้นทุนส่วนใหญ่ของได มาจากเวเฟอร์ที่ผ่านกระบวนการผลิตแล้ว สิ่งที่เราอาจเห็นไม่ชัดเจนคือ ยิ่งไดมีขนาดเล็กเท่าไร อัตราผลผลิต (yield) หรือสัดส่วนของชิ้นงานที่ผลิตแล้วใช้งานได้จริงจะยิ่งสูงขึ้น เหตุผลคือในกระบวนการผลิตซิลิคอนมักจะมีตำหนิเล็กๆ กระจายอยู่ทั่วทั้งเวเฟอร์ ดังนั้น ยิ่งไดมีขนาดเล็ก สัดส่วนของไดที่มีตำหนิก็ยิ่งน้อยลง อย่างไรก็ตาม หากทำให้ไดเล็กเกินไป จนเข้าใกล้ขีดจำกัดของเทคโนโลยีการผลิต จะทำให้การผลิตและการประกอบยากขึ้น เกิดข้อผิดพลาดมากขึ้น ส่งผลให้ผลผลิตลดลงและต้นทุนสูงขึ้น

สถาปนิก คอมพิวเตอร์ ต้องการ ให้ ISA มีความ เรียบง่าย เพื่อ ช่วย ลด ขนาด ของ โปรเซสเซอร์ที่นำไปใช้งาน ดังที่เราจะเห็นในบทถัดไป ISA ของ RISC-V เป็น ISA มีความง่ายมากกว่า ISA ของ ARM-32 เพื่อให้เห็นผลกระทบของความเรียบง่ายอย่างเป็นรูปธรรม ลองเปรียบเทียบโปรเซสเซอร์ RISC-V Rocket กับ ARM-32 Cortex-A5 โดยใช้เทคโนโลยีเดียวกัน (TSMC40GPLUS) และมีขนาดแคชเท่ากัน (16 KiB) ชิพ RISC-V มีขนาด 0.27 mm² เทียบกับ ARM-32 มีพื้นที่ประมาณ 0.53 mm² เมื่อพื้นที่เพิ่มขึ้นเกือบ



รูปที่ 1.4: เวเฟอร์ขนาดเวเฟอร์ขนาดเส้นผ่านศูนย์กลาง 8 นิ้วที่ประกอบด้วยชิป RISC-V ที่ออกแบบโดย SiFive เวเฟอร์นี้มีชิป RISC-V สองประเภท ที่ใช้สายการผลิตที่เก่ากว่าและมีขนาดใหญ่กว่า ชิป FE310 มีขนาด 2.65 มม×2.72 มม และชิปทดสอบของ SiFive มีขนาด 2.89 มม×2.72 มม เวเฟอร์นี้มีชิป FE310 จำนวน 1846 ชิป และชิปทดสอบ SiFive จำนวน 1866 ชิป รวมทั้งสิ้น 3712 ชิป

โปรเซสเซอร์ระดับสูง

สามารถเพิ่มประสิทธิภาพได้โดยการรวมคำสั่งที่เรียบง่ายเข้าด้วยกัน โดยไม่ทำให้โปรเซสเซอร์ระดับล่างทั้งหมดต้องแบกรับ ISA ที่มีขนาดใหญ่และซับซ้อนมากขึ้น เทคนิคนี้เรียกว่า มาโครฟิวชั่น (macrofusion) ซึ่งเป็นการรวมคำสั่ง “ระดับสูง” เข้าด้วยกัน



ความเรียบง่าย

โปรเซสเซอร์ที่เรียบง่าย มีประโยชน์สำหรับงานแบบระบบฝังตัว (embedded applications)

เนื่องจากสามารถคาดการณ์เวลาในการทำงานได้ง่ายกว่า โปรแกรมเมอร์ภาษาแอสเซมบลีของไมโครคอนโทรลเลอร์มักต้องการควบคุมเวลาที่แม่นยำ ดังนั้นพวกเขาจึงพึ่งพาโค้ดที่ใช้จำนวนรอบที่แน่นอน ซึ่งสามารถนับได้ด้วยตนเอง



ประสิทธิภาพ

ปัจจัยสุดท้ายนี้คือ ค่ากลับกันของอัตราสัญญาณนาฬิกา (clock rate) ดังนั้น หากมีอัตราสัญญาณนาฬิกาเท่ากับ 1 GHz จะหมายความว่าเวลาในหนึ่งรอบสัญญาณนาฬิกา (clock cycle) เท่ากับ 1 นาโนวินาที (1 ns) (1/10⁹)

เป็นสองเท่า ต้นทุนของ ARM-32 Cortex-A5 จึงสูงกว่าประมาณ 4 เท่า (2²) เมื่อเทียบกับ RISC-V Rocket แม้ลดขนาดไดลงเพียง 10% ก็สามารถลดต้นทุนได้ประมาณ 1.2 เท่า (1.1²)

ความเรียบง่าย เนื่องจากต้นทุนมีความไวต่อความซับซ้อน สถาปนิกจึงต้องการให้ ISA เรียบง่าย เพื่อช่วยลดพื้นที่ของได ความเรียบง่ายยังช่วยลดเวลาในการออกแบบชิป และเวลาในการตรวจสอบความถูกต้อง ซึ่งต้นทุนส่วนนี้อาจเป็นต้นทุนส่วนใหญ่ในการพัฒนาชิป ต้นทุนเหล่านี้ต้องถูกนำไปรวมเข้ากับต้นทุนของชิป โดยภาระต้นทุนจะขึ้นอยู่กับจำนวนชิปที่ผลิตและจำหน่าย นอกจากนี้ ความเรียบง่ายยังช่วยลดต้นทุนของเอกสารประกอบและความยากในการทำให้อุปกรณ์ใช้งานเข้าใจ ISA

ด้านล่างนี้คือตัวอย่างที่ชัดเจนของความซับซ้อนของ ISA จาก ARM-32 :

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

คำสั่งที่ย่อมาจาก Load Multiple, Increment-Address on Equal มันทำการโหลดข้อมูล 5 ครั้ง และเขียนค่าลงในรีจิสเตอร์ 6 ตัว แต่จะทำงานก็ต่อเมื่อรหัสเงื่อนไขแบบ EQ (Equal) ถูกตั้งค่าไว้เท่านั้น นอกจากนี้มันยังเขียนผลลัพธ์ไปยัง PC (Program Counter) ซึ่งหมายความว่ามันยังทำการกระโดดแบบมีเงื่อนไข (conditional branch) ไปพร้อมกันด้วย เรียกได้ว่าทำหลายอย่างในคำสั่งเดียว!

เป็นที่น่าแปลกใจที่คำสั่งที่เรียบง่ายมีแนวโน้มที่จะถูกใช้งานมากกว่าคำสั่งที่ซับซ้อน เช่น x86-32 มีคำสั่ง `enter` ซึ่งถูกออกแบบมาให้เป็นคำสั่งแรกๆ ที่ทำงาน เมื่อเข้าสู่ฟังก์ชันเพื่อสร้าง stack frame สำหรับฟังก์ชันนั้น (ดูบทที่ 3) แต่คอมพิวเตอร์ส่วนใหญ่กลับใช้เพียงแค่คำสั่งง่ายๆ ของ x86-32 สองคำสั่งต่อไปนี้แทน:

```
push ebp      # ดันตัวชี้เฟรมไปที่สแต็ก
mov  ebp, esp # คัดลอกตัวชี้สแต็กไปยังตัวชี้เฟรม
```

ประสิทธิภาพ ยกเว้นชิปขนาดเล็กสำหรับแอปพลิเคชันแบบฝังตัว สถาปนิกมักให้ความสำคัญกับประสิทธิภาพควบคู่กับต้นทุน ประสิทธิภาพสามารถแบ่งออกเป็นสามองค์ประกอบ:

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

แม้ว่า ISA ที่เรียบง่ายอาจต้องใช้จำนวนคำสั่งต่อโปรแกรมมากกว่า เมื่อเทียบกับ ISA ที่ซับซ้อน แต่มันสามารถชดเชยได้มากกว่า ด้วยการมีรอบนาฬิกาที่เร็วกว่า หรือมีจำนวนรอบต่อคำสั่ง (CPI) โดยเฉลี่ยที่น้อยลง

ตัวอย่างเช่น สำหรับการทดสอบ CoreMark [Gal-On and Levy 2012] (100,000 รอบการทำงาน) ประสิทธิภาพของ ARM-32 Cortex-A9 คือ

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

สำหรับโปรเซสเซอร์ RISC-V แบบ Berkeley Out-of-Order Machine BOOM (Berkeley Out-of-Order Machine) จะได้ว่า

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

ในกรณีนี้ โปรเซสเซอร์ ARM ไม่ได้ใช้จำนวนคำสั่งน้อยกว่า RISC-V ดังที่เราจะเห็นต่อไป คำสั่งที่เรียบง่ายมักเป็นคำสั่งที่ถูกใช้งานบ่อยที่สุด ดังนั้นความเรียบง่ายของ ISA สามารถให้ผลลัพธ์ที่ดีกว่าในทุกตัวชี้วัด สำหรับโปรแกรมนี้ โปรเซสเซอร์ RISC-V มีประสิทธิภาพดีกว่าเกือบ 10% ในแต่ละปัจจัยทั้งสาม ซึ่งส่งผลให้ได้เปรียบด้านประสิทธิภาพรวมเกือบ 30% หาก ISA ที่เรียบง่ายยังช่วยให้ชิปที่มีขนาดเล็กลงด้วย ก็จะทำให้มี อัตราส่วนประสิทธิภาพต่อต้นทุนที่ยอดเยี่ยม

การ แยก สถาปัตยกรรม ออกจาก การนำไป ใช้ งาน ความแตกต่างดั้งเดิมระหว่างสถาปัตยกรรมและการใช้งาน ซึ่งมีมาตั้งแต่ช่วงทศวรรษ 1960 คือ สถาปัตยกรรมคือสิ่งที่โปรแกรมเมอร์ภาษาเครื่องต้องรู้เพื่อเขียนโปรแกรมให้ทำงานได้ถูกต้อง แต่ไม่รวมถึงประสิทธิภาพของโปรแกรมนั้น ความท้าทายสำหรับสถาปนิกคือ เพิ่มคำสั่งใน ISA เพื่อช่วยด้านประสิทธิภาพหรือต้นทุนของการนำไปใช้งานหนึ่ง ในช่วงเวลานั้น แต่กลับสร้างภาระให้กับการนำไปใช้งานอื่นๆ ที่แตกต่างรวมถึงในอนาคต

สำหรับ ISA ของ MIPS-32 ตัวอย่างที่น่าเสียดายคือ แนวคิดของ *การกระโดดแบบหน่วงเวลา (delayed branch)* คำสั่งกระโดดแบบมีเงื่อนไข สร้างปัญหาในการประมวลผลแบบไปป์ไลน์ เพราะโปรเซสเซอร์ต้องการโหลดคำสั่งถัดไปล่วงหน้าเข้า pipeline แต่ไม่สามารถตัดสินใจได้ว่า จะใช้คำสั่งถัดไปตามลำดับ (ถ้าการกระโดดไม่เกิดขึ้น) หรือจะใช้คำสั่งที่อยู่เป้าหมายของการกระโดด (ถ้าการกระโดดเกิดขึ้น) สำหรับไมโครโปรเซสเซอร์รุ่นแรกของ MIPS ที่ใช้ไปป์ไลน์ 5 ขั้นตอน ความไม่แน่นอนนี้อาจทำให้เกิด การหยุดรอ (stall) หนึ่งรอบนาฬิกา MIPS-32 แก้ปัญหานี้โดย กำหนดให้การกระโดดมีผล หลังจากคำสั่งถัดไปอีกหนึ่งคำสั่ง ดังนั้น คำสั่งถัดไปจะถูก *ประมวลผลเสมอ* เรียกว่า ช่องหน่วงเวลา (delay slot) หน้าที่ของโปรแกรมเมอร์หรือคอมไพเลอร์คือ ต้องใส่คำสั่งที่มีประโยชน์ลงใน *ช่องหน่วงเวลา*

แต่น่าเสียดายที่ “วิธีแก้ปัญหานี้” นี้ ไม่ได้ช่วยโปรเซสเซอร์ MIPS-32 รุ่นหลังที่มีจำนวนขั้นตอนไปป์ไลน์ (pipeline stages) มากขึ้น (ซึ่งซึ่งหมายถึงมีคำสั่งจำนวนมากถูกดึงเข้ามาก่อนที่จะทราบผลของการกระโดด) แต่กลับทำให้ชีวิตของโปรแกรมเมอร์ MIPS-32 นักพัฒนาคอมไพเลอร์ และผู้ออกแบบโปรเซสเซอร์ยากขึ้นไปตลอด เนื่องจาก ISA แบบเพิ่มขยาย ต้องคงความเข้ากันได้ย้อนหลัง (ดูหัวข้อ 1.2) นอกจากนี้ ยังทำให้โค้ดของ MIPS-32 เข้าใจได้ยากขึ้น (ดูรูปที่ 2.10 หน้าที่ 36 บทที่ 2) ในขณะที่สถาปนิกไม่ควรใส่คุณสมบัติที่ช่วย เฉพาะการนำไปใช้งาน (implementation) แบบใดแบบหนึ่งในช่วงเวลาหนึ่ง พวกเขาก็ไม่ควรเพิ่มคุณสมบัติที่ *ขัดขวาง* วิธีการนำสิ่งที่ออกแบบไว้ไปสร้างให้ทำงานจริงแบบอื่นเช่นกัน ตัวอย่างเช่น ARM-32 และ ISA อื่นๆ มีคำสั่ง **Load Multiple** ตามที่กล่าวในหน้าก่อน คำสั่งเหล่านี้สามารถเพิ่มประสิทธิภาพในโปรเซสเซอร์แบบ pipeline ที่ออก



แยกสถาปัตยกรรม กับนำไปใช้

โปรเซสเซอร์แบบไปป์ไลน์ในปัจจุบัน สามารถคาดการณ์ผลลัพธ์ของคำสั่งกระโดดล่วงหน้าได้โดยใช้ตัวทำนายแบบฮาร์ดแวร์ (hardware predictors) ซึ่งมีความแม่นยำมากกว่า 90% และสามารถทำงานได้กับไปป์ไลน์ทุกขนาด โดยมิกลไกสำหรับล้าง (flush) และเริ่มต้นไปป์ไลน์ใหม่เมื่อมีการคาดเดาผิด

คำสั่งได้ครั้งละคำสั่งเดียว (single-instruction issue) แต่กลับส่งผลเสียต่อโปรเซสเซอร์ที่สามารถออกคำสั่งได้หลายคำสั่งพร้อมกัน (multiple-instruction issue) เหตุผลคือการนำไปใช้งานจริงแบบตรงไปตรงมา ทำให้ไม่สามารถจัดตารางคำสั่ง (schedule) เพื่อการไหลของข้อมูลแบบหลายรายการ ในคำสั่ง Load Multiple ให้ทำงานขนานกับคำสั่งอื่นได้ ส่งผลให้จำนวนคำสั่งที่ประมวลผลได้ต่อหน่วยเวลา (throughput) ของคำสั่งลดลง ในโปรเซสเซอร์ประเภทนั้น



ที่วางสำหรับส่วนขยาย

ความสามารถในการขยายในอนาคต เมื่อกฎของมัวร์ (Moore's Law) สิ้นสุด หนทางหลักเพียงอย่างเดียวในการพัฒนาประสิทธิภาพต่อต้นทุนอย่างมีนัยสำคัญ คือการเพิ่มคำสั่งเฉพาะทางสำหรับงานเฉพาะด้าน เช่น การเรียนรู้เชิงลึก (deep learning), เทคโนโลยีเสมือนผสมความจริง (AR) การเพิ่มประสิทธิภาพเชิงคอมบินาทอเรียล (combinatorial optimization) กราฟิก และอื่นๆ ดังนั้น ในปัจจุบันจึงเป็นสิ่งสำคัญที่ ISA ต้องเผื่อพื้นที่รหัสดำเนินการ (opcode space) สำหรับการขยายความสามารถในอนาคต ในช่วงทศวรรษ 1970 และ 1980 เมื่อกฎของมัวร์ยังคงมีผลเต็มที่ มีการคำนึงถึงการสงวนพื้นที่ opcode สำหรับตัวเร่งความเร็ว (accelerators) ในอนาคตน้อยมาก มีการพิจารณาน้อยมากเกี่ยวกับการสงวนพื้นที่ opcode สำหรับตัวเร่งในอนาคต สถาปนิกให้ความสำคัญกับฟิลด์ที่อยู่ (address fields) และฟิลด์ค่าทันที (immediate fields) ที่มีขนาดใหญ่ขึ้นเพื่อลดจำนวนคำสั่งที่ต้องใช้ต่อโปรแกรม ซึ่งเป็นปัจจัยแรกในสมการประสิทธิภาพในหน้าที่แล้ว

ตัวอย่างหนึ่งของผลกระทบจากการมีพื้นที่ opcode ไม่เพียงพอ เกิดขึ้นเมื่อสถาปนิกของ ARM-32 ต้องการลดขนาดโค้ดโดยเพิ่มคำสั่งขนาด 16 บิต ใน ISA ที่เดิมมีขนาดคำสั่งคงที่ 32 บิต แต่ปัญหาคือ ไม่มีพื้นที่ opcode เหลือแล้ว ดังนั้น วิธีแก้ปัญหาคือ การสร้าง ISA ใหม่ที่มีคำสั่งขนาด 16 บิต (Thumb) และต่อมาพัฒนาเป็น ISA ใหม่ที่รองรับทั้งคำสั่ง 16 บิต และ 32 บิต (Thumb-2) โดยใช้บิตโหมด (mode bit) เพื่อสลับระหว่าง ISA ของ ARM การเปลี่ยนโหมดทำได้โดยโปรแกรมเมอร์หรือคอมไพเลอร์กระโดดไปยังที่อยู่ไบต์ที่มีค่า บิตต่ำสุด (least-significant bit) เป็น 1 ในบิตที่มีนัยสำคัญน้อยที่สุด เหตุผลที่ทำได้เพราะในคำสั่ง 16 บิตและ 32 บิต ต้องมีค่าบิตต่ำสุดเป็น 0

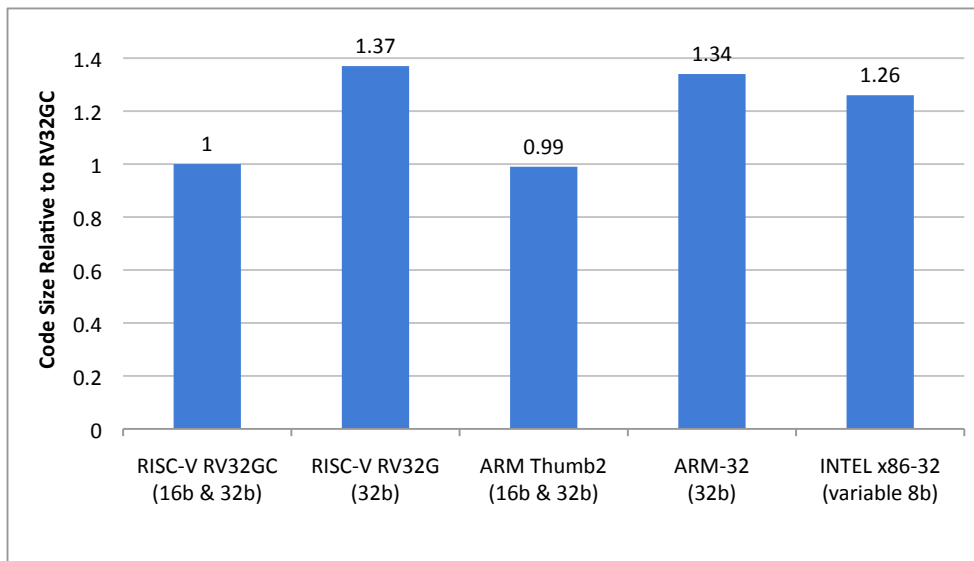
คำสั่ง ldmiaeq ของ ARM-32 ที่กล่าวถึงก่อนหน้านี้ยังมีความซับซ้อนมากขึ้นไปอีก เพราะเมื่อทำการกระโดด (branch) ยังสามารถเปลี่ยนโหมดชุดคำสั่งระหว่าง ARM-32 และ Thumb/Thumb-2 ได้ด้วย



ขนาดโปรแกรม

ขนาดโปรแกรม ยิ่งโปรแกรมมีขนาดเล็กลง พื้นที่บนชิปที่ใช้เก็บโปรแกรม (program memory) ก็ยิ่งน้อยลง ซึ่งเป็นต้นทุนที่สำคัญสำหรับอุปกรณ์แบบฝังตัว ประเด็นนี้เองเป็นแรงบันดาลใจให้สถาปนิกของ ARM เพิ่มคำสั่งที่มีขนาดสั้นลงเข้าไปใน ISA แบบ Thumb และ Thumb-2 ภายหลัง โปรแกรมที่มีขนาดเล็กยังช่วยให้การไหลของคำสั่งน้อยลง ซึ่งช่วยประหยัดพลังงานมากขึ้น เนื่องจากการเข้าถึง DRAM นอกชิปใช้พลังงานมากกว่าการเข้าถึง SRAM บนชิป ดังนั้น การทำให้โค้ดมีขนาดเล็ก จึงเป็นหนึ่งในเป้าหมายของผู้ออกแบบ ISA

ชุดคำสั่งของ x86-32 มีคำสั่งที่สั้นถึง 1 ไบต์ และยาวได้ถึง 15 ไบต์ ซึ่งคำสั่งที่มีความยาวแบบแปรผันเป็นไบต์ของ x86 ควรจะทำให้โปรแกรมมีขนาดเล็กกว่า ISA ที่จำกัดอยู่ที่คำสั่งยาว 32 บิต เช่น ARM-32 และ RISC-V และในทางตรงกันข้าม คำสั่งที่มีความยาวแปรผัน 8 บิตก็น่าจะมีขนาดเล็กกว่า ISA ที่มีเฉพาะคำสั่ง 16 บิตและ 32 บิต เช่น Thumb-2



รูปที่ 1.5: การเปรียบเทียบขนาดโปรแกรมสัมพันธ์ของ RV32G, ARM-32, x86-32, RV32C และ Thumb-2 โดย ISA สองตัวหลังถูกออกแบบมาให้ขนาดโค้ดเล็กลง โปรแกรมที่ใช้คือชุดทดสอบมาตรฐาน SPEC CPU2006 โดยใช้คอมไพเลอร์ GCC ความได้เปรียบด้านขนาดเล็กของ Thumb-2 เมื่อเทียบกับ RV32C เกิดจากการที่ Thumb-2 สามารถลดขนาดโค้ดได้ ผ่านคำสั่ง (Load and Store Multiple) ในช่วงเริ่มต้นของฟังก์ชัน (procedure entry) ในขณะที่ RV32C ไม่รองรับคำสั่งเหล่านี้ เพื่อคง ความสัมพันธ์แบบหนึ่งต่อหนึ่ง (one-to-one mapping) กับคำสั่งของ RV32G และ RV32G เองก็ไม่ได้มีคำสั่ง Load and Store Multiple เพื่อลดความซับซ้อนของการนำไปใช้งาน โดยเฉพาะสำหรับโปรเซสเซอร์ระดับสูง (จะอธิบายเพิ่มเติมในส่วนถัดไป) บทที่ 7 จะอธิบาย RV32C โดย RV32G เป็นชื่อที่ใช้เรียกชุด extension ยอดนิยมของ RISC-V ได้แก่ RV32M, RV32F, RV32D และ RV32A ซึ่งชื่อทางการคือ RV32IMAFD [Waterman 2016]

ตัวอย่างคำสั่ง x86-32 ที่มี
ความยาวถึง 15 ไบต์ คือ
คำสั่ง lock add dword
ptr ds:[esi+ecx*4
+0x12345678],
0xefcdab89 เมื่อแปลง
เป็น machine code (ใน
รูปแบบเลขฐานสิบหก) จะ
ได้ 67 66 f0 3e 81 84 8e
78 56 34 12 89 ab cd
ef โดย 8 ไบต์สุดท้ายคือ
เป็น address จำนวน 2 คำ
(คำ address ขนาด 4 ไบต์
สองตัว) และอีก 7 ไบต์แรก
ใช้ระบุรายละเอียดของคำ
สั่ง ได้แก่ การดำเนินการ
หน่วยความจำแบบอะตอม
มิก, การบวก, การใช้ข้อมูล
ขนาด 32 บิต, ใช้รีจิสเตอร์
data segment, รีจิสเตอร์
สำหรับ address, โหมดการ
กำหนดที่อยู่แบบ scaled
index ตัวอย่างคำสั่งของ
x86-32 ที่มีขนาดเพียง 1
ไบต์ เช่นคำสั่ง inc eax
ซึ่งเมื่อแปลงเป็นโค้ดภาษา
เครื่องจะได้ 40 คำสั่ง x86-
32 รวม operation และ
addressing ที่ซับซ้อนไว้ใน
คำสั่งเดียว ทำให้ยาวได้ถึง
15 ไบต์ คำสั่งประกอบด้วย
ที่อยู่ขนาด 2 เวิร์ด (คำที่อยู่
ขนาด 4 ไบต์ 2 คำ) และที่
อยู่ 7 ไบต์เพื่อระบุการบวก
(add) โดยใช้ข้อมูล 32 บิต
(dword ptr) รีจิสเตอร์เซก
เมนต์ข้อมูล (ds) รีจิสเตอร์ที่
อยู่ (esi, ecx) โหมดการ
กำหนดที่อยู่แบบดัชนีปรับ
ขนาด (esi+ecx*4), และ
การดำเนินการหน่วยความ
จำแบบอะตอมมิก (lock)



การเขียนโปรแกรม

และ RISC-V ที่ใช้ส่วนขยาย RV32C (ดูบทที่ 7) รูปที่ 1.5 แสดงให้เห็นว่า ในขณะที่โค้ดของ ARM-32 และ RISC-V มีขนาดใหญ่กว่าโค้ดของ x86-32 ประมาณ 6% ถึง 9% เมื่อทุกคำสั่งยาว 32 บิต แต่ที่น่าสนใจคือ x86-32 มีขนาดใหญ่กว่าเวอร์ชันที่ถูกบีบอัด (RV32C และ Thumb-2) ที่มีทั้งคำสั่ง 16 บิตและ 32 บิต ถึง 26%

ในขณะที่ ISA ใหม่ที่ใช้คำสั่งแบบความยาวแปรผันขนาด 8 บิต มีแนวโน้มจะทำให้โค้ดมีขนาดเล็กกว่า RV32C และ Thumb-2 แต่สถาปนิกของ x86 รุ่นแรกในทศวรรษ 1970 มีความกังวลที่แตกต่างออกไป ยิ่งไปกว่านั้น จากข้อกำหนดของ ความเข้ากันได้ย้อนหลังในระดับบนารีของ ISA แบบเพิ่มขยาย (หัวข้อ 1.2) ทำให้คำสั่งใหม่ ๆ ของ x86-32 ที่มีอยู่หลายร้อยคำสั่ง มีความยาวมากกว่าที่คาดไว้ เนื่องจากต้องแบกรับภาระของ prefix ขนาด 1 หรือ 2 ไบต์ เพื่อบีบอัด (encode) คำสั่งใหม่ให้เข้าไปอยู่ในพื้นที่ opcode ที่เหลืออยู่อย่างจำกัดของ x86 รุ่นดั้งเดิม

ความง่ายในการเขียนโปรแกรม การคอมไพล์ และการเชื่อมโยง เนื่องจากข้อมูลในรีจิสเตอร์เข้าถึงได้เร็วกว่าข้อมูลในหน่วยความจำมาก จึงเป็นเรื่องสำคัญอย่างยิ่งที่คอมไพเลอร์จะต้องจัดสรรรีจิสเตอร์ (register allocation) ได้อย่างมีประสิทธิภาพ และงานนี้จะง่ายขึ้นมากเมื่อมีรีจิสเตอร์จำนวนมากกว่า ในมุมมองนี้ ARM-32 มี 16 รีจิสเตอร์ ส่วน x86-32 มีเพียง 8 รีจิสเตอร์ ส่วน ISA สมัยใหม่ส่วนใหญ่ รวมถึง RISC-V มีรีจิสเตอร์จำนวนมากถึง 32 ตัว การมีรีจิสเตอร์มาก ช่วยให้การดำเนินงานของทั้งคอมไพเลอร์ และโปรแกรมเมอร์ภาษาแอสเซมบลีง่ายขึ้นอย่างมาก

อีกประเด็นหนึ่งคือ การประเมินความเร็วของชุดคำสั่ง (code sequence) สำหรับคอมไพเลอร์และโปรแกรมเมอร์ภาษาแอสเซมบลี ดังที่เราจะเห็นใน คำสั่ง RISC-V โดยทั่วไปใช้เวลาไม่เกินหนึ่งรอบสัญญาณนาฬิกาต่อคำสั่ง (หากไม่คิด cache miss) แต่ ARM-32 และ x86-32 มีคำสั่งบางคำสั่งที่ใช้หลายรอบสัญญาณนาฬิกา แม้ว่าข้อมูลทั้งหมดจะอยู่ใน cache แล้วก็ตาม แตกต่างจาก ARM-32 และ RISC-V คำสั่งคำนวณของ x86-32 สามารถใช้ตัวถูกดำเนินการที่อยู่ในหน่วยความจำได้โดยตรง โดยไม่จำเป็นต้องอยู่ในรีจิสเตอร์ทั้งหมด ความซับซ้อนของคำสั่ง และการใช้ตัวถูกดำเนินการจากหน่วยความจำ ทำให้ผู้ออกแบบโปรเซสเซอร์ยากที่จะทำให้ประสิทธิภาพคาดเดาได้

การที่ ISA รองรับ โค้ดที่ไม่ขึ้นกับตำแหน่ง Position Independent Code (PIC) เป็นสิ่งที่มีประโยชน์ เพราะช่วยให้สามารถทำการเชื่อมโยงแบบไดนามิกได้ (ดูหัวข้อ 3.5) เนื่องจากโค้ดของไลบรารีที่ใช้ร่วมกัน อาจถูกโหลดไปวางที่ที่อยู่ที่แตกต่างกันในแต่ละโปรแกรม การใช้การเข้าถึงที่อยู่แบบสัมพันธ์กับ PC (PC-relative) สำหรับการกระโดดแบบมีเงื่อนไขและการอ้างอิงข้อมูล เป็นสิ่งที่ช่วยให้ PIC ทำได้ง่ายขึ้นอย่างมาก แม้ว่า ISA เกือบทั้งหมดจะรองรับการกระโดดแบบสัมพันธ์กับ PC แต่ x86-32 และ MIPS-32 กลับไม่รองรับการเข้าถึงที่อยู่แบบสัมพันธ์กับ PC สำหรับการอ้างอิงข้อมูล

■ รายละเอียดเพิ่มเติม: ARM-32, MIPS-32 และ x86-32

หัวข้อที่เรียกว่า Elaboration เป็นส่วนเพิ่มเติมสำหรับผู้อ่านที่สนใจเจาะลึก แต่ไม่จำเป็นต้องอ่าน เพื่อทำความเข้าใจเนื้อหาหลักของหนังสือ ตัวอย่างเช่น ชื่อ ISA ที่เราใช้ในหนังสือไม่ใช่ชื่อทางการ ARM แบบ 32 บิตมีหลายเวอร์ชัน เริ่มตั้งแต่ปี 1986 และเวอร์ชันล่าสุดคือ ARMv7 (ปี 2005) โดยคำว่า ARM-32 มักหมายถึง ARMv7 MIPS ก็มีหลายเวอร์ชัน 32 บิต แต่ในที่นี้หมายถึงเวอร์ชันดั้งเดิมที่เรียกว่า MIPS I (คำว่า “MIPS32” เป็น ISA รุ่นใหม่ที่ต่างออกไป) สถาปัตยกรรมของ Intel เริ่มจาก 8086 (16 บิต) ในปี 1978 และถูกขยายเป็น 32 บิตใน 80386 ปี 1985 โดยคำว่า x86-32 ในที่นี้ โดยทั่วไปหมายถึง IA-32 เนื่องจาก ISA เหล่านี้มีหลายเวอร์ชันย่อยมาก ผู้เขียนจึงเลือกใช้คำเรียกแบบไม่เป็นทางการนี้ เพราะทำให้เข้าใจง่ายและไม่สับสนมากกว่า

1.4 ภาพรวมของหนังสือเล่มนี้

หนังสือเล่มนี้สมมติว่าผู้อ่าน เคยศึกษาชุดคำสั่งอื่นๆ มาก่อน RISC-V หากยังไม่เคย สามารถดูหนังสือสถาปัตยกรรมเบื้องต้นของผู้เขียน ที่ใช้ RISC-V เป็นพื้นฐานได้ที่ [Patterson and Hennessy 2021]

บทที่ 2 เนื้อหาจะเริ่มจาก RV32I ซึ่งเป็นชุดคำสั่งจำนวนเต็มพื้นฐานที่ถูกกำหนดให้คงที่และถือเป็นแกนหลักของ RISC-V บทที่ 3 อธิบายภาษาแอสเซมบลีของ RISC-V ส่วนที่เหลือ นอกเหนือจากที่แนะนำในบทที่ 2 รวมถึงข้อตกลงการเรียกฟังก์ชัน และเทคนิคต่างๆ สำหรับการเชื่อมโยง (linking) ภาษาแอสเซมบลีประกอบด้วย คำสั่งจริงของ RISC-V รวมถึงคำสั่งเพิ่มเติมบางอย่างที่ไม่ได้อยู่ใน ISA โดยตรง คำสั่งเพิ่มเติมเหล่านี้เรียกว่า คำสั่งเทียม (pseudoinstructions) เหล่านี้ ซึ่งเป็นการนำคำสั่งจริงมาประยุกต์อย่างชาญฉลาดเพื่อช่วยให้เขียนโปรแกรมภาษาแอสเซมบลีได้ง่ายขึ้น โดยไม่ต้องเพิ่มความซับซ้อนให้กับ ISA

สามบทถัดไปจะอธิบายส่วนขยายมาตรฐานของ RISC-V ซึ่งเมื่อนำมารวมกับ RV32I แล้ว จะเรียกว่า RV32G (G มาจากคำว่า general) ได้แก่:

- บทที่ 4: การคูณและการหาร (RV32M)
- บทที่ 5: การดำเนินการจุดลอยตัว (RV32F และ RV32D)
- บทที่ 6: การดำเนินการหน่วยความจำอะตอมิก (RV32A)

“การ์ดอ้างอิง” ของ RISC-V ในหน้าที่ 3 และ 4 เป็นสรุปคำสั่ง RISC-V ทั้งหมด ในหนังสือเล่มนี้ครอบคลุม: RV32G, RV64G และ RV32/64V

บทที่ 7 จะอธิบายส่วนขยายเสริม RV32C ซึ่งเป็นตัวอย่างที่ดีของความสง่างามของ RISC-V แนวคิดคือ การจำกัดคำสั่ง 16 บิต ให้เป็นคำสั่งแบบย่อของคำสั่ง 32 บิตใน RV32G ทำให้แทบไม่มีต้นทุนเพิ่ม แอสเซมเบลอร์สามารถเลือกขนาดคำสั่งเองได้ ทำให้โปรแกรมเมอร์และคอมไพเลอร์ แทบไม่ต้องรับรู้ถึงการมีอยู่ของคำสั่ง RV32C ตัวถอดรหัส (hardware decoder) ที่แปลงคำสั่ง 16 บิตของ RV32C เป็นคำสั่ง 32 บิต ใช้เพียง 400 เกต ซึ่งเป็นเพียงไม่กี่เปอร์เซ็นต์ของการนำ RISC-V ไปใช้งานที่ง่ายที่สุด

การ์ดอ้างอิงนี้ยังเรียกว่า **กรีนการ์ด** เนื่องจากในอดีต (ช่วงปี 1960s) มีการสรุป ISA ลงบนกระดาษแข็ง หนึ่งหน้าที่มีพื้นหลังสีเขียว อย่างไรก็ตาม หนังสือเล่มนี้ใช้พื้นหลังสีขาว เพื่อให้อ่านง่ายมากกว่ารักษาความเป็นประวัติศาสตร์

บทที่ 8 แนะนำ RV32V ซึ่งเป็นส่วนขยายเวกเตอร์ คำสั่งแบบเวกเตอร์เป็นอีกตัวอย่างหนึ่งของความสง่างามของ ISA เมื่อเทียบกับคำสั่ง SIMD (Single Instruction Multiple Data) จำนวนมากและซับซ้อนใน ARM-32, MIPS-32 และ x86-32 ในความเป็นจริง คำสั่งที่เพิ่มเข้ามาใน x86-32 จำนวนหลายร้อยคำสั่ง ในรูปที่ 1.2 เป็นคำสั่ง SIMD และยังคงเพิ่มขึ้นอย่างต่อเนื่อง RV32V มีความเรียบง่ายยิ่งกว่า สถาปัตยกรรมเวกเตอร์ส่วนใหญ่ เพราะมันเก็บชนิดข้อมูล (data type) และความยาว (length) ไว้ในรีจิสเตอร์เวกเตอร์ แทนที่จะฝังไว้ในรหัสดำเนินการ RV32V อาจเป็นเหตุผลสำคัญที่สุดในการเปลี่ยนจาก ISA แบบ SIMD ดั้งเดิมมาใช้ RISC-V

บทที่ 9 จะอธิบาย RV32B ส่วนขยายการจัดการบิต (bit manipulation extension) คำสั่งเหล่านี้ช่วยในเรื่อง การคำนวณตำแหน่งที่อยู่ การดำเนินการตรรกะที่ซับซ้อน การหมุนบิต (rotation) การนับบิต (bit count) การตั้งค่าบิต (bit set) การจัดการ error code และการสร้างเลขสุ่มเทียม (pseudo-random numbers)

บทที่ 10 จะแสดงเวอร์ชัน RISC-V แบบตำแหน่งที่อยู่ 64 บิต คือ RV64G ตามที่อธิบายในบทนี้ สถาปนิกของ RISC-V เพียงแค่ขยายขนาดรีจิสเตอร์ให้กว้างขึ้นและเพิ่มคำสั่งบางตัว เช่น word doubleword หรือ long จากคำสั่งของ RV32G ก็สามารถขยายตำแหน่งที่อยู่จาก 32 บิตเป็น 64 บิตได้

บทที่ 11 จะอธิบายคำสั่งระบบ รวมถึงวิธีที่ RISC-V จัดการการแบ่งเพจ (paging) และโหมดสิทธิพิเศษเครื่อง โหมดผู้ใช้ และโหมดผู้ควบคุม

บทสุดท้าย จะให้ภาพรวมสั้นๆ เกี่ยวกับส่วนขยายที่เหลือ ซึ่งปัจจุบันอยู่ระหว่างการพิจารณาโดย RISC-V International

ส่วนที่ใหญ่ที่สุดของหนังสือคือ ภาคผนวก A ซึ่งเป็นสรุปชุดคำสั่ง (instruction set) ตามลำดับตัวอักษร ครอบคลุมสถาปัตยกรรมชุดคำสั่งของ RISC-V ทั้งหมด รวมส่วนขยายทั้งหมดที่กล่าวมาและคำสั่งเทียม คำสั่งทั้งหมดภายในประมาณ 50 หน้า ซึ่งซึ่งสะท้อนถึงความเรียบง่ายของ RISC-V

ภาคผนวก B แสดงการดำเนินการในภาษาแอสเซมบลีที่ใช้บ่อยและคำสั่งที่สอดคล้องกันใน RV32I, ARM-32 และ x86-32 ตามด้วยตัวอย่างโปรแกรมภาษา C ขนาดเล็กและผลลัพธ์จากคอมไพเลอร์สำหรับทั้งสาม ISA ภาคผนวกนี้มีสองวัตถุประสงค์หลัก วัตถุประสงค์แรก สำหรับผู้อ่านที่คุ้นเคยกับ ARM-32 หรือ x86-32 ใช้การจับคู่ (mapping) เพื่อเรียนรู้ RISC-V วัตถุประสงค์ที่สอง สำหรับโปรแกรมเมอร์ที่กำลังแปลงโปรแกรมภาษาแอสเซมบลีเดิม มาเป็น RISC-V

ภาคผนวก C จะเน้นประเด็นบางส่วนของที่ท้าทายต่อความเข้าใจมากที่สุด เมื่อต้องทำการแปลเนื้อหา โดยมีความตั้งใจให้ผู้อ่านสามารถรับรู้ถึง “ความเรียบง่ายและความงดงามพื้นฐาน” ของเนื้อหาตามต้นฉบับของหนังสือเล่มนี้ คำศัพท์บางคำที่พบบ่อย แต่ค่อนข้างเฉพาะทาง เช่น รหัสดำเนินการ (opcode) & รหัสช่วยจำ (mnemonic), ลำดับไบนารี (endian-ness), ช่องหน่วงเวลา (delay slot), การคำนวณแบบส่วนเติมเต็มฐานสอง (2's complement arithmetic), การขยายเครื่องหมายและขยายศูนย์ของบิต (sign- & zero-



ความเรียบง่าย

จำนวนคำสั่ง (ISA)	จำนวนหน้า	จำนวนคำ	จำนวนชั่วโมง	จำนวนสัปดาห์
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	4898	2,186,259	182	4.5

รูปที่ 1.6: จำนวนหน้าและจำนวนคำของคู่มือ ISA [Waterman and Asanović 2017a], [Waterman and Asanović 2017b], [Intel Corporation 2016], [ARM Ltd. 2014] การประมาณเวลาอ่าน (เป็นชั่วโมงหรือสัปดาห์) คำนวณจากความเร็ว 200 คำต่อนาที และอ่านสัปดาห์ละ 40 ชั่วโมง โดยอ้างอิงบางส่วนจากรูปที่ 1 ของ [Baumann 2017]

extension) และการคำนวณแบบล้นหลักและขาดหลัก (overflow & underflow) จะถูกอธิบายไว้อย่างชัดเจน ภาคผนวกประกอบด้วย อภิธานศัพท์ของตัวย่อ (acronyms) และรายการคำศัพท์เทียบเคียง (translator's concordance) เพื่อช่วยเสริมการใช้งานดัชนี (index) ที่อยู่ท้ายเล่มให้มีประโยชน์มากยิ่งขึ้น สุดท้ายนี้ ผู้อ่านจะได้เห็นว่า RISC-V มีบทบาทในการสนับสนุนและส่งเสริม แนวคิดสถาปัตยกรรมแบบเปิดและมาตรฐานแบบเปิด ซึ่งกำลังได้รับความสนใจและความนิยมเพิ่มขึ้นในปัจจุบัน

เราปิดท้ายหนังสือด้วยดัชนี (index)

1.5 ข้อสรุปท้ายบท

เป็นเรื่องง่ายที่จะเห็นด้วยวิธีการเชิงตรรกะอย่างเป็นทางการว่า มีชุดคำสั่ง (instruction set) บางแบบที่ในเชิงนามธรรมแล้วเพียงพอ ในการควบคุมและทำให้เกิดการทำงานของลำดับการดำเนินการใดๆ ก็ได้ ...อย่างไรก็ตาม ปัจจัยที่มีความสำคัญอย่างแท้จริงในมุมมองปัจจุบัน ในการเลือกชุดคำสั่ง กลับเป็นเรื่องเชิงปฏิบัติมากกว่า ได้แก่ ความเรียบง่ายของฮาร์ดแวร์ที่ต้องใช้รองรับชุดคำสั่งนั้น ความชัดเจนในการนำไปใช้กับปัญหาที่สำคัญจริง และความเร็วในการประมวลผลปัญหาเหล่านั้น

—von Neumann et al. 1947 [Burks et al. 1946]

รายงานที่เขียนอย่างดีของ John von Neumann มีอิทธิพลอย่างมาก จนทำให้รูปแบบคอมพิวเตอร์นี้ถูกเรียกกันทั่วไปว่า สถาปัตยกรรมแบบ von Neumann แม้ว่ารายงานนี้จะอิงจากผลงานของผู้อื่นก็ตาม ที่น่าสนใจคือ รายงานนี้ถูกเขียนขึ้นก่อนที่คอมพิวเตอร์แบบเก็บโปรแกรม (stored-program computer) เครื่องแรกจะใช้งานได้จริงถึง 3 ปี!

RISC-V เป็น ISA ที่ค่อนข้างใหม่ ออกแบบจากศูนย์ (clean-slate) มีความเรียบง่าย (minimalist) และเป็นสถาปัตยกรรมแบบเปิด (open) โดยเรียนรู้จากข้อผิดพลาดของ ISA ในอดีต เป้าหมายของผู้ออกแบบ RISC-V คือให้มันสามารถใช้งานได้อย่างมีประสิทธิภาพ กับอุปกรณ์คอมพิวเตอร์ทุกระดับ ตั้งแต่ขนาดเล็กที่สุดไปจนถึงเครื่องที่เร็วที่สุด โดยยึดตามคำแนะนำของ John von Neumann เมื่อกว่า 75 ปีก่อน ISA นี้จึงเน้น “ความเรียบง่าย” เพื่อให้ต้นทุนต่ำ ขณะเดียวกันก็มีจำนวนรีจิสเตอร์ที่เพียงพอ และพฤติกรรมความเร็วของคำสั่งที่ชัดเจน (transparent instruction speed) เพื่อช่วยให้คอมไพเลอร์และโปรแกรมเมอร์สามารถแปลงปัญหาจริงให้เป็นโค้ดที่เหมาะสมและทำงานได้รวดเร็ว

ข้อบ่งชี้ความซับซ้อนประการหนึ่งคือขนาดของเอกสาร รูปที่ 1.6 แสดงขนาดของคู่มือชุดคำสั่งสำหรับ RISC-V, ARM-32, และ x86-32 วัดจากจำนวนหน้าและจำนวนคำ หากคุณอ่านคู่มือแบบเต็มเวลา—วันละ 8 ชั่วโมง สัปดาห์ละ 5 วัน—จะใช้เวลาประมาณครึ่งเดือนในการอ่านคู่มือ ARM-32 หนึ่งรอบ และประมาณหนึ่งเดือนเต็มสำหรับ x86-32 ด้วยความซับซ้อนระดับนี้ อาจไม่มีใครเข้าใจ ARM-32 หรือ 86-32 ได้ทั้งหมด หากใช้เกณฑ์ง่ายๆ นี้วัดความซับซ้อน RISC-V มีความซับซ้อนเพียง $\frac{1}{12}$ ของ ARM-32 และประมาณ $\frac{1}{10}$



ความเรียบง่าย

ถึง $\frac{1}{30}$ ของ x86-32. อันที่จริง สรุปรูปของสถาปัตยกรรมชุดคำสั่ง RISC-V รวมถึงส่วนขยายทั้งหมด ใช้เพียงสองหน้า (คู่มืออ้างอิง)

ISA ที่เรียบง่ายและเปิดนี้ เริ่มต้นขึ้นเมื่อวันที่ 18 พฤษภาคม 2010 และปัจจุบันได้รับการสนับสนุนจากองค์กร (foundation) ซึ่งจะพัฒนา ISA นี้ต่อไป โดยการเพิ่มส่วนขยายแบบทางเลือกเฉพาะเมื่อมีเหตุผลทางเทคนิคที่ชัดเจน หลังจากผ่านการอภิปรายอย่างยาวนาน ความเปิดกว้างของ RISC-V ช่วยให้เกิดการใช้งานร่วมกันได้ฟรี ซึ่งช่วยลดต้นทุนและความเสี่ยงจากความลับที่เป็นอันตรายที่อาจซ่อนอยู่ในโปรเซสเซอร์

อย่างไรก็ตาม ฮาร์ดแวร์เพียงอย่างเดียวไม่สามารถสร้างระบบที่สมบูรณ์ได้ ต้นทุนการพัฒนาซอฟต์แวร์ มักสูงกว่าฮาร์ดแวร์เสียอีก ดังนั้น แม้ว่าฮาร์ดแวร์ที่เสถียรจะสำคัญ แต่ซอฟต์แวร์ที่เสถียรยิ่งสำคัญกว่า ระบบต้องมี ระบบปฏิบัติการ ตัวโหลดบูต ซอฟต์แวร์อ้างอิง และเครื่องมือพัฒนายอดนิยมน องค์กร RISC-V ให้ความสำคัญกับ ISA โดยรวมและชุดคำสั่งพื้นฐานที่ถูกกำหนดให้คงที่ (frozen base) ทำให้ RV32I ซึ่งเป็นแกนของ ซอฟต์แวร์จะไม่มีการเปลี่ยนแปลงอีกต่อไป ด้วยการยอมรับอย่างกว้างขวาง และความเป็นระบบเปิด RISC-V จึงมีศักยภาพในการทำหายความครอบงำของ ISA ที่เป็นกรรมสิทธิ์ซึ่งมีอยู่ทั่วไปได้ คำว่า “สง่างาม” แทบไม่เคยถูกใช้กับสถาปัตยกรรมชุดคำสั่ง (ISA) แต่หลังจากอ่านหนังสือเล่มนี้ คุณอาจเห็นด้วยกับเรามากว่าคำนี้เหมาะสมกับ RISC-V ในหนังสือจะมีการใช้ไอคอน “โมนาลิซ่า” เพื่อเน้นคุณสมบัติที่แสดงถึงความสง่างามนี้

เราได้ลองครอบรอบสิบปีของ RISC-V ด้วยการสัมภาษณ์ผ่านวิดีโอของบุคคล 23 คนจากช่วงทศวรรษแรกของ RISC-V เพื่อแบ่งปันประสบการณ์ผลกระทบต่อชีวิตและองค์กรและมุมมองต่ออนาคต ดูเพิ่มเติมได้ที่ riscv.org/risc-v-10th/



สง่างาม

1.6 เรียนรู้เพิ่มเติม

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.

A. Burks, H. Goldstine, and J. Von Neumann. Preliminary discussion of the logical design of an electronic computer instrument. Technical report, 1946.

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, Volumes 1--4*. 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 Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

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

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/>.

RV32I: ชุดคำสั่งพื้นฐานจำนวนเต็มของ RISC-V

ฟรานเชส เอลิซาเบธ
“ฟราน” อัลเลน

(พ.ศ.1995-2062) ได้รับรางวัลทัวริงจากผลงานของเธอในการพัฒนาคอมพิวเตอร์ให้เหมาะสมที่สุด รางวัลทัวริงเป็นรางวัลสูงสุดในสาขาวิทยาการคอมพิวเตอร์ ซึ่งบางครั้งถูกเรียกว่า รางวัลโนเบลของวิทยาการคอมพิวเตอร์



...วิธีเดียวที่จะบรรลุเป้าหมายด้านประสิทธิภาพเป็นจริงและเข้าถึงได้สำหรับผู้ใช้คือ การออกแบบคอมพิวเตอร์และคอมพิวเตอร์ไปพร้อมกัน ด้วยวิธีนี้ คุณสมบัติที่ซอฟต์แวร์ไม่สามารถใช้ได้จะไม่ถูกใส่ลงในฮาร์ดแวร์...

—ฟรานเชส เอลิซาเบธ “ฟราน” อัลเลน, 1981

2.1 บทนำ

รูปที่ 2.1 เป็นการนำเสนอแบบกราฟิกหนึ่งหน้าของชุดคำสั่งพื้นฐาน RV32I คุณสามารถดูชุดคำสั่ง RV32I ทั้งหมดได้โดยการต่อตัวอักษรที่ขีดเส้นใต้จากซ้ายไปขวาสำหรับแต่ละแผนภาพ สัญลักษณ์ชุดที่ใช้ { } จะแสดงรายการความหลากหลายที่เป็นไปได้ของคำสั่ง โดยใช้ตัวอักษรที่ขีดเส้นใต้ หรือตัวอักษร _ ซึ่งหมายถึงไม่มีตัวอักษรสำหรับรูปแบบนี้ ยกตัวอย่างเช่น

$$\text{set_less_than} \left\{ \begin{array}{c} _ \\ \text{immediate} \end{array} \right\} \left\{ \begin{array}{c} _ \\ \text{unsigned} \end{array} \right\}$$

แสดงคำสั่ง RV32I สีคำสั่งนี้: `slt, slti, sltu, sltiu`

เป้าหมายของแผนภาพเหล่านี้ ซึ่งจะปรากฏเป็นรูปแรกของแต่ละบทถัดไป คือ เพื่อให้ภาพรวมของคำสั่งในบทนั้นอย่างรวดเร็ว และช่วยให้เข้าใจเชิงลึกได้ทันที

2.2 รูปแบบคำสั่ง RV32I



ความเรียบง่าย

รูป 2.2 แสดงรูปแบบคำสั่งพื้นฐานทั้งหกแบบ ได้แก่ R-type สำหรับการดำเนินการระหว่างรีจิสเตอร์กับรีจิสเตอร์; I-type สำหรับค่าคงที่ทันทีขนาดสั้น (12 บิต) และคำสั่งโหลด; S-type สำหรับคำสั่งเก็บข้อมูล; B-type สำหรับการกระโดดแบบมีเงื่อนไข; U-type สำหรับค่าคงที่ทันทีขนาดยาว (20 บิต) และ J-type สำหรับการกระโดดแบบไม่มีเงื่อนไข รูปที่ 2.3 แสดงรายการรหัสดำเนินการ (opcodes) ของคำสั่ง RV32I ในรูปที่ 2.1 โดยใช้รูปแบบคำสั่งจากรูปที่ 2.2

แม้แต่รูปแบบคำสั่งก็แสดงให้เห็นตัวอย่างหลายประการที่ สถาปัตยกรรมชุดคำสั่ง RISC-V แบบเรียบง่าย สามารถปรับปรุงความคุ้มค่าต่อประสิทธิภาพได้ ประการแรก มี



ค่าใช้จ่าย



RV32I

Integer Computation

add {immediate}subtract{and
or
exclusive or}{immediate}{shift left logical
shift right arithmetic
shift right logical}{immediate}load upper immediateadd upper immediate to pc

Control transfer

branch{equal
not equal}branch{greater than or equal
less than}{unsigned}jump and link{register}

Loads and Stores

{load
store}{byte
halfword
word}load{byte
halfword}unsigned

Miscellaneous instructions

fence loads & storesfence.instruction & dataenvironment{break
call}control status register{read & clear bit
read & set bit
read & write}{immediate}

รูปที่ 2.1: แผนภาพของคำสั่ง RV32I ตัวอักษรที่ขีดเส้นใต้จะถูกนำมาต่อกันจากซ้ายไปขวา เพื่อสร้างคำสั่งของ RV32I เครื่องหมายวงเล็บปีกกา { } หมายถึงแต่ละรายการในแนวตั้ง ภายในชุดเป็นรูปแบบที่แตกต่างกันของคำสั่งนั้นๆ เครื่องหมายขีดล่าง _ ภายในชุดหมายถึงตัวเลือกหนึ่งที่เป็นเพียงชื่อคำสั่ง จนถึงจุดนั้นโดยไม่มีตัวอักษรจากชุดนี้ ตัวอย่างเช่น สัญลักษณ์ไกล์มบนซ้ายแสดงถึงคำสั่งหกคำสั่งต่อไปนี้: 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		R-type	
imm[11:0]						rs1	funct3		rd			opcode		I-type	
imm[11:5]			rs2		rs1	funct3		imm[4:0]			opcode		S-type		
imm[12]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]	opcode		B-type		
imm[31:12]									rd			opcode		U-type	
imm[20]	imm[10:1]			imm[11]	imm[19:12]			rd			opcode		J-type		

รูปที่ 2.2: รูปแบบคำสั่ง RISC-V เราใช้การระบุตำแหน่งของแต่ละส่วนย่อยของค่าคงที่ ที่ถูกระบุมาในตัวคำสั่งโดยตรง (immediate) โดยอ้างอิงตามตำแหน่งบิต (imm[x]) ภายในค่าคงที่ที่ถูกสร้างขึ้น แทนการอ้างอิงตำแหน่งบิตในฟิลด์ค่าคงที่ของคำสั่ง ซึ่งเป็นวิธีที่มักใช้กันโดยทั่วไป บทที่ 11 อธิบายว่าคำสั่งในกลุ่มรีจิสเตอร์ควบคุมสถานะ (CSR) ใช้รูปแบบ I-type ที่แตกต่างออกไปเล็กน้อย (รูปที่ 2.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

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

รูปที่ 2.3: แผนผังรหัสดำเนินการของ RV32I มีโครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทรูปแบบ, และชื่อคำสั่ง (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

รูปแบบคำสั่งเพียงหกแบบ และทุกคำสั่งมีความยาว 32 บิตเท่ากัน ซึ่งทำให้การถอดรหัสคำสั่งง่ายขึ้น ในขณะที่ ARM-32 และโดยเฉพาะ x86-32 มีรูปแบบคำสั่งจำนวนมาก ทำให้การถอดรหัสมีต้นทุนสูงในระบบระดับล่าง และเป็นความท้าทายด้านประสิทธิภาพในการออกแบบโปรเซสเซอร์ระดับกลางและระดับสูง ประการที่สอง คำสั่งของ RISC-V รองรับตัวถูกดำเนินการสามตัว แทนที่จะใช้ฟิลด์เดียวร่วมกัน สำหรับรีจิสเตอร์ต้นทางและปลายทาง อย่างไรก็ตาม x86-32 เมื่อการดำเนินการโดยธรรมชาติต้องการตัวถูกดำเนินการสามตัว แต่ ISA รองรับตัวถูกดำเนินการเพียงสองตัว คอมไพเลอร์หรือโปรแกรมเมอร์ภาษาแอสเซมบลีจำเป็นต้องเพิ่มคำสั่งย้ายข้อมูล (move) อีกหนึ่งคำสั่งเพื่อเก็บค่าของตัวถูกดำเนินการปลายทางเดิมไว้ ประการที่สาม ใน RISC-V ตำแหน่งของตัวถูกดำเนินการสำหรับอ่านและเขียน จะอยู่ในตำแหน่งเดิมเสมอในทุกคำสั่ง ซึ่งหมายความว่า สามารถเริ่มการเข้าถึงรีจิสเตอร์ได้ก่อนที่จะถอดรหัสคำสั่งเสร็จสมบูรณ์ ISA อื่นจำนวนมาก (เช่น ARM-32 และ MIPS-32) ใช้ฟิลด์เดียวกันเป็นรีจิสเตอร์ต้นทาง ในบางคำสั่งและเป็นรีจิสเตอร์ปลายทางในคำสั่งอื่น ทำให้ต้องเพิ่มฮาร์ดแวร์เพิ่มเติมในเส้นทางที่อาจเป็นจุดวิกฤตด้านเวลา (critical path) เพื่อเลือกฟิลด์ที่ต้องการ ประการที่สี่ ฟิลด์ค่าคงที่ทันที (immediate fields) ในรูปแบบเหล่านี้จะถูกขยายเครื่องหมาย (sign-extended) เสมอ และบิตเครื่องหมาย (sign bit) จะอยู่ในบิตที่มีนัยสำคัญสูงสุดของคำสั่งเสมอ การออกแบบนี้ทำให้ การขยายเครื่องหมายค่าคงที่ทันที ซึ่งอาจอยู่ในเส้นทางวิกฤตเวลา (Critical Timing Path) สามารถดำเนินการได้ก่อนการถอดรหัสคำสั่งเสร็จสิ้น

การขยายเครื่องหมายของค่าคงที่ทันที ยังช่วยให้คำสั่งเชิงตรรกะ (logical instructions) ทำงานได้มีประสิทธิภาพขึ้นอีกด้วย ตัวอย่างเช่น การคำนวณ $x \& 0xffffffff$ ใน RISC-V สามารถใช้เพียงคำสั่งเดียวคือ `andi` แต่ใน MIPS-32 ต้องใช้ถึงสองคำสั่ง (`addiu` เพื่อโหลดค่าคงที่ แล้วตามด้วยคำสั่ง `and`) เนื่องจาก MIPS ใช้วิธีขยายค่าคงที่ทันทีแบบเติมศูนย์ (zero-extends) สำหรับคำสั่งเชิงตรรกะ ส่วนใน ARM-32 จำเป็นต้องเพิ่มคำสั่งพิเศษ `bic` ซึ่งทำหน้าที่คำนวณ $rx \& \text{immediate}$ เพื่อชดเชยข้อจำกัดจากการใช้การขยายศูนย์กับค่าคงที่ทันที

■ รายละเอียดเพิ่มเติม: รูปแบบคำสั่ง B-type และ J-type คืออะไร?

ตามที่กล่าวไว้ด้านล่าง ฟิลด์ค่าคงที่ทันที (imm) สำหรับคำสั่งกระโดดแบบมีเงื่อนไข ถูกหมุนบิต (rotate) ไป 1 บิต ซึ่งเป็นรูปแบบที่เปลี่ยนแปลงจากรูปแบบ S-type เรียกรูปแบบใหม่นี้ว่ารูปแบบ B-type ในทำนองเดียวกัน ฟิลด์ค่าคงที่ทันทีสำหรับคำสั่งกระโดด ก็ถูกหมุนเช่นกัน โดยเป็นการเปลี่ยนแปลงจากรูปแบบ U-type ที่เรียกใหม่ว่ารูปแบบ J-type ดังนั้น ในความเป็นจริงแล้วมีรูปแบบพื้นฐานเพียงสี่แบบ แต่หากนับแบบแยกให้ครบทุกประเภท ก็สามารถกล่าวได้ว่า RISC-V มีรูปแบบคำสั่งทั้งหมดหกแบบ

เพื่อช่วยผู้เขียนโปรแกรม รูปแบบบิตที่เป็นศูนย์ทั้งหมด (all zeros) ถูกกำหนดให้เป็นคำสั่งที่ไม่ถูกต้อง (illegal instruction) ใน RV32I ดังนั้น หากเกิดการกระโดดผิดพลาดเข้าไปยังบริเวณหน่วยความจำที่ถูกเคลียร์เป็นศูนย์ไว้ ระบบจะเกิด trap ทันที ซึ่งช่วยในการดีบั๊กได้ ในทำนองเดียวกัน รูปแบบบิตที่เป็นหนึ่งทั้งหมด (all ones) ก็ถูกกำหนดให้เป็นคำสั่งที่ไม่ถูกต้องเช่นกัน ซึ่งจะทำให้เกิด trap สำหรับข้อผิดพลาดที่พบบ่อยอื่นๆ เช่น หน่วยความจำแบบไม่ลบเลือน (non-volatile memory) ที่ยังไม่ได้โปรแกรม บัสหน่วยความจำที่ไม่ได้เชื่อมต่อ หรือชิปหน่วยความจำที่ชำรุด

เพื่อให้มีพื้นที่สำหรับการขยาย ISA ในอนาคตให้เพียงพอ ชุดคำสั่งพื้นฐาน RV32I ใช้พื้นที่การเข้ารหัสน้อยกว่า $1/8$ -th ของพื้นที่ทั้งหมดในคำสั่งขนาด 32 บิต นอกจากนี้ ผู้ออกแบบยังเลือก opcode ของ RV32I อย่างรอบคอบ เพื่อให้คำสั่งที่มีการดำเนินการบนเส้นทางข้อมูล (datapath) คล้ายกัน ใช้ค่าบิตของ opcode ร่วมกันให้มากที่สุด ซึ่งช่วยให้ตรรกะควบคุม (control logic) ง่ายขึ้น สุดท้าย ดังที่เราจะเห็นต่อไป ตำแหน่งที่อยู่ของค่า

ooc
การเขียนโปรแกรม



ที่ว่างสำหรับส่วนขยาย



ค่าใช้จ่าย

สั่งกระโดดแบบมีเงื่อนไข (branch) และคำสั่งกระโดดแบบไม่มีเงื่อนไข (jump) ในรูปแบบ B และ J จะต้องถูกเลื่อนบิตไปทางซ้าย 1 บิต เพื่อให้ค่าตำแหน่งที่อยู่ถูกคูณด้วย 2 ส่งผลให้คำสั่ง กระโดดแบบมีเงื่อนไขและกระโดดแบบไม่มีเงื่อนไข มีช่วงการเข้าถึง (range) ที่กว้างขึ้น RISC-V ยังใช้เทคนิคการ “หมุนบิต” (rotate) ของค่าคงที่ทันที จากตำแหน่งตามธรรมชาติ เพื่อช่วยลดภาระจำนวนวงจรปลายทาง (fanout) ของสัญญาณในคำสั่ง และลดต้นทุนของการใช้มัลติเพล็กซ์เซอร์สำหรับค่าคงที่ทันที ลงได้เกือบครึ่งหนึ่ง ซึ่งช่วยให้ตรรกะของ datapath ในการออกแบบระดับล่าง (low-end implementations) เรียบง่ายขึ้นอีกด้วย

การนำ RISC-V ไปใช้งาน (implementations) จะใช้ opcode เดียวกัน สำหรับส่วนขยายเสริม เช่น RV32M, RV32F เป็นต้น ส่วนขยายที่ไม่เป็นมาตรฐานและมีลักษณะเฉพาะของโปรเซสเซอร์แต่ละตัว จะถูกจำกัดให้อยู่ในพื้นที่ opcode ที่สำรองไว้ใน RISC-V



ประสิทธิภาพ

มีอะไรแตกต่าง? เราจะจบแต่ละหัวข้อของบทนี้และบทต่อไป ด้วยการอธิบายว่า RISC-V แตกต่างจาก ISA อื่นอย่างไร ความแตกต่างมักอยู่ที่สิ่งที่ RISC-V สถาปนิกที่ดีไม่ได้แสดงรสนิยมที่ดีเพียงจากสิ่งที่เลือกใส่เข้าไป แต่ยังรวมถึงสิ่งที่เลือกไม่ใส่ด้วย

ใน ARM-32 ฟิลด์ค่าคงที่ทันที ขนาด 12 บิตของ ไม่ได้เป็นเพียงค่าคงที่ตรงๆ แต่เป็นอินพุตให้กับฟังก์ชันที่สร้างค่าคงที่ขึ้นมา โดย 8 บิตแรกจะถูกขยายด้วยศูนย์ (zero-extended) ให้เต็มความกว้าง จากนั้นหมุนขวาตามค่าของ 4 บิตที่เหลือ คูณด้วย 2 แนวคิดนี้มีเป้าหมายเพื่อให้สามารถเข้ารหัสค่าคงที่ที่มีประโยชน์ได้มากขึ้นภายใน 12 บิต ซึ่งอาจช่วยลดจำนวนคำสั่งที่ต้องใช้ในการประมวลผล ARM-32 ยังใช้บิตขนาด 4 บิตที่มีค่ามากในรูปแบบคำสั่งส่วนใหญ่สำหรับการดำเนินการแบบมีเงื่อนไข แม้ว่าการดำเนินการนี้จะถูกใช้ไม่บ่อยนัก แต่เพิ่มความซับซ้อนให้กับ โปรเซสเซอร์แบบทำงานนอกลำดับคำสั่ง (out-of-order processors)

■ รายละเอียดเพิ่มเติม: โปรเซสเซอร์แบบทำงานนอกลำดับคำสั่ง (out-of-order processors)

เป็นโปรเซสเซอร์ความเร็วสูงที่ใช้ไปป์ไลน์ และสามารถดำเนินการคำสั่งแบบฉวยโอกาส (opportunistic) แทนที่จะดำเนินการตามลำดับของโปรแกรมอย่างเคร่งครัด (in-order) คุณสมบัติสำคัญของโปรเซสเซอร์ลักษณะนี้คือ การเปลี่ยนชื่อรีจิสเตอร์ ซึ่งเป็นการแมปชื่อรีจิสเตอร์ในโปรแกรมไปยัง ชุดรีจิสเตอร์จริงภายใน (physical registers) ที่มีจำนวนมากกว่า ปัญหาของการดำเนินการแบบมีเงื่อนไขคือ รีจิสเตอร์จริงตัวใหม่จะต้องถูกเขียนค่าไม่ว่าเงื่อนไขจะเป็นจริงหรือไม่ก็ตาม ดังนั้นจึงจำเป็นต้องอ่านค่าของรีจิสเตอร์ปลายทางเดิมมาเป็นตัวถูกดำเนินการตัวที่สาม เพื่อคัดลอกค่าเดิมไปยังรีจิสเตอร์ปลายทางใหม่ในกรณีที่เงื่อนไขไม่เป็นจริง ตัวถูกดำเนินการที่เพิ่มขึ้นนี้ทำให้ต้นทุนของชุดรีจิสเตอร์ (register file), หน่วยเปลี่ยนชื่อรีจิสเตอร์ (register renamer) และฮาร์ดแวร์สำหรับการประมวลผลแบบนอกลำดับคำสั่งเพิ่มสูงขึ้น

2.3 รีจิสเตอร์ RV32I

รูปที่ 2.4 แสดงรายการรีจิสเตอร์ของ RV32I พร้อมชื่อของรีจิสเตอร์ที่กำหนดตามอินเตอร์เฟซไบนารีแอปพลิเคชัน (ABI) ของ RISC-V ในตัวอย่างโค้ด เราจะใช้ชื่อแบบ ABI เพื่อให้อ่านและเข้าใจได้ง่ายขึ้น เพื่อช่วยในการทำงานของโปรแกรมเมอร์ภาษาแอสเซมบลีและพัฒนาคอมไพเลอร์ คือ RV32I มีรีจิสเตอร์ทั้งหมด 31 ตัว พร้อมด้วย x0 ซึ่งจะมีค่า 0 เสมอ ในขณะที่ ARM-32 มีรีจิสเตอร์เพียง 16 ตัว และ x86-32 มีรีจิสเตอร์ 8 ตัวเท่านั้น!

อะไรที่แตกต่าง? การกำหนดรีจิสเตอร์ตัวหนึ่งให้มีค่าเป็นศูนย์ (x0 ใน RISC-V) ถือเป็น



การเขียนโปรแกรม



ความเรียบง่าย

31	0	
	x0 / zero	Hardwired to zero
	x1 / ra	Return address
	x2 / sp	Stack pointer
	x3 / gp	Global pointer
	x4 / tp	Thread pointer
	x5 / t0	Temporary
	x6 / t1	Temporary
	x7 / t2	Temporary
	x8 / s0 / fp	Saved register, Frame pointer
	x9 / s1	Saved register
	x10 / a0	Function argument และ return value
	x11 / a1	Function argument และ return value
	x12 / a2	Function argument
	x13 / a3	Function argument
	x14 / a4	Function argument
	x15 / a5	Function argument
	x16 / a6	Function argument
	x17 / a7	Function argument
	x18 / s2	Saved register
	x19 / s3	Saved register
	x20 / s4	Saved register
	x21 / s5	Saved register
	x22 / s6	Saved register
	x23 / s7	Saved register
	x24 / s8	Saved register
	x25 / s9	Saved register
	x26 / s10	Saved register
	x27 / s11	Saved register
	x28 / t3	Temporary
	x29 / t4	Temporary
	x30 / t5	Temporary
	x31 / t6	Temporary
32		
31	0	
	pc	Program counter
32		

รูปที่ 2.4: รีจิสเตอร์ของ RV32I ในบท 3 อธิบายเกี่ยวกับ ข้อตกลงการเรียกใช้ฟังก์ชันของ RISC-V (RISC-V calling convention) รวมถึงเหตุผลของการกำหนดรีจิสเตอร์แบบต่างๆ เช่น ตัวชี้ (pointer) ได้แก่ sp (stack pointer), gp (global pointer), tp (thread pointer), fp (frame pointer), กลุ่ม รีจิสเตอร์เก็บค่า s0–s11 (saved registers) และ รีจิสเตอร์ชั่วคราว t0–t6 (temporaries) (รูปที่ 2.1 และตาราง 20.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

ปัจจัยสำคัญอย่างคาดไม่ถึงที่ช่วยให้สถาปัตยกรรมคำสั่ง (ISA) ของ RISC-V เรียบง่ายขึ้นอย่างมาก รูปที่อ้างอิง 3.3 หน้าที่ 42 ในบทที่ 3 แสดงตัวอย่างจำนวนมากของคำสั่งที่อยู่ในสถาปัตยกรรม ARM-32 และ x86-32 ต้องมีเป็น “คำสั่งเฉพาะ” เนื่องจากไม่มีรีจิสเตอร์ศูนย์ แต่ใน RV32I สามารถสังเคราะห์ขึ้นได้จากคำสั่งพื้นฐาน โดยเพียงแค่อัปเดตรีจิสเตอร์ศูนย์เป็นตัวถูกดำเนินการ

ใน ARM-32 ตัวนับโปรแกรม (PC - Program Counter) ถูกนับเป็นหนึ่งใน 16 รีจิสเตอร์ ซึ่งหมายความว่า คำสั่งใดก็ตามที่เปลี่ยนค่ารีจิสเตอร์ ก็อาจมีผลข้างเคียง กลายเป็นคำสั่งกระโดด การที่ PC เป็นรีจิสเตอร์ทำให้ฮาร์ดแวร์สำหรับการทำนายการกระโดด (branch prediction) มีความซับซ้อนมากขึ้น ซึ่งความแม่นยำของมันมีความสำคัญอย่างยิ่งต่อประสิทธิภาพ ของระบบประมวลผลแบบท้อ (Pipelining) เพราะในกรณีนี้ ทุกคำสั่งอาจเป็นการกระโดดแบบมีเงื่อนไข แทนที่จะเป็นเพียง 10–20% ของคำสั่งทั้งหมดในโปรแกรม ตามปกติของ ISA ทั่วไป นอกจากนี้ การใช้ PC เป็นรีจิสเตอร์ ยังทำให้มีจำนวนรีจิสเตอร์อเนกประสงค์ลดลงหนึ่งตัว

การประมวลผลแบบท้อ (Pipelining) ถูกใช้ใน

โปรเซสเซอร์เกือบทั้งหมดในปัจจุบัน ยกเว้นโปรเซสเซอร์ที่มีราคาต่ำสุด เพื่อให้ได้ประสิทธิภาพที่ดี เช่นเดียวกับสายการผลิตในโรงงานอุตสาหกรรม มันเพิ่มอัตราการทำงาน (throughput) โดยการทำให้หลายคำสั่งถูกประมวลผลซ้อนทับกันในเวลาเดียวกัน เพื่อให้ทำงานแบบนี้ได้ ตัวประมวลผลจะทำการ “ทำนายผลของการกระโดด” (branch prediction) ซึ่งสามารถทำได้ด้วยความแม่นยำมากกว่า 90% เมื่อทำนายผิด โปรเซสเซอร์จะต้องนำคำสั่งกลับประมวลผลใหม่ ไม่โครโปรเซสเซอร์ยุคแรกๆ มีไปป์ไลน์ 5 ชั้น ซึ่งหมายความว่า มี 5 คำสั่งถูกประมวลผลซ้อนกันอยู่ ส่วนโปรเซสเซอร์ในปัจจุบัน มีไปป์ไลน์มากกว่า 10 ชั้น สถาปัตยกรรม ARM v8 ซึ่งเป็นรุ่นต่อจาก ARM-32 ได้ยกเลิกการใช้ program counter (PC) เป็นรีจิสเตอร์อเนกประสงค์ (general-purpose register) ซึ่งโดยนัยแล้วเป็นการยอมรับว่าการออกแบบแบบเดิมนั้นเป็นความผิดพลาด



ความเรียบง่าย

2.4 การคำนวณจำนวนเต็มใน RV32I

ภาคผนวก A ให้รายละเอียดของคำสั่งทั้งหมดของ RISC-V รวมถึงรูปแบบและรหัสดำเนินการ ในหัวข้อนี้ และหัวข้อที่คล้ายกันในบทถัดๆ ไป เราจะให้ภาพรวมของ ISA ที่เพียงพอสำหรับโปรแกรมเมอร์ที่มีความรู้ด้านภาษาแอสเซมบลี รวมทั้งเน้นคุณสมบัติที่แสดงให้เห็นถึงตัวชี้วัด ISA ทั้งเจ็ดประการจากบทก่อนหน้า จากบทที่ 1

คำสั่งคำนวณพื้นฐาน (add, sub), คำสั่งตรรกะ (and, or, xor) และคำสั่งการเลื่อนบิต (sll, srl, sra) ที่แสดงในรูปที่ 2.1 เป็นไปตามที่คาดหวังใน ISA ทั่วไป โดยคำสั่งเหล่านี้อ่านค่าขนาด 32 บิตสองค่าจากรีจิสเตอร์ และเขียนผลลัพธ์ขนาด 32 บิตไปยังรีจิสเตอร์ปลายทาง RV32I ยังมีรูปแบบที่ใช้ค่าคงที่ทันที (immediate) ของคำสั่งเหล่านี้ด้วย โดยต่างจาก ARM-32 ตรงที่ค่าคงที่ที่จะถูกจะขยายเครื่องหมาย (sign-extended) เสมอ เพื่อให้สามารถแทนค่าลบได้เมื่อจำเป็น นี่จึงเป็นเหตุผลว่าทำไมจึงไม่จำเป็นต้องมี รูปแบบค่าคงที่ทันทีของคำสั่ง sub

โปรแกรมสามารถสร้างค่าบูลีนจากผลลัพธ์ของการเปรียบเทียบได้ เพื่อรองรับกรณีนี้ RV32I จึงมีคำสั่ง *set less than* ซึ่งจะตั้งค่ารีจิสเตอร์ปลายทางเป็น 1 หากตัวถูกดำเนินการตัวแรกน้อยกว่าตัวที่สอง หรือเป็น 0 ในกรณีอื่น นอกจากนี้ ยังมีคำสั่งที่ใช้เครื่องหมาย slt และคำสั่งที่ไม่ใช่เครื่องหมาย (sltu) สำหรับจำนวนเต็มแบบมีเครื่องหมาย และไม่มีเครื่องหมาย รวมถึงคำสั่งแบบค่าคงที่ทันที สำหรับทั้งคำสั่ง (slti และ sltiu ดังที่เราจะเห็นต่อไป แม้ว่าคำสั่งการกระโดดแบบมีเงื่อนไขของ RV32I จะสามารถตรวจสอบความสัมพันธ์ทุกแบบระหว่างรีจิสเตอร์สองตัวได้ แต่เงื่อนไขบางอย่างอาจเกี่ยวข้องกับความสัมพันธ์ระหว่างหลายคู่ของรีจิสเตอร์ ในกรณีเช่นนี้ คอมไพเลอร์หรือโปรแกรมเมอร์ภาษาแอสเซมบลีสามารถใช้คำสั่ง slt ร่วมกับคำสั่งตรรกะ and, or และ xor เพื่อจัดการเงื่อนไขที่ซับซ้อนมากขึ้นได้

คำสั่งคำนวณจำนวนเต็มอีกสองคำสั่งที่เหลือใน รูปที่ 2.1 ช่วยสนับสนุนการเขียนแอสเซมบลีและการลิงก์ ได้แก่ คำสั่ง (*Load upper immediate*) (`lui`) ซึ่งจะโหลดค่าคงที่ขนาด 20 บิตไปยัง 20 บิต ที่มีนัยสำคัญสูงสุดของรีจิสเตอร์ และสามารถตามด้วยคำสั่งค่าคงที่ปกติ เพื่อสร้างค่าคงที่ขนาด 32 บิต โดยใช้คำสั่ง 2 คำสั่งของ RV32I คำสั่ง (*Add upper immediate to PC*) (`auipc`) ใช้สำหรับสนับสนุนลำดับคำสั่ง 2 คำสั่งในการเข้าถึงออฟเซต (offset) ที่ระบุได้ จากค่า PC สำหรับการเข้าหน่วยความจำ (data accesses) และสำหรับการเปลี่ยนเส้นทางการควบคุมข้อมูล (control-flow transfers) โดยการใช้คำสั่ง `auipc` ร่วมกับค่าคงที่ทันที ขนาด 12 บิตในคำสั่ง `jalr` (ดูด้านล่าง) สามารถกระโดดไปยังตำแหน่งที่อยู่ใดๆ แบบสัมพันธ์กับ PC ขนาด 32 บิตได้ และในทำนองเดียวกัน การใช้คำสั่ง `auipc` ร่วมกับออฟเซต 12 บิต ในคำสั่งโหลดหรือจัดเก็บข้อมูล (load/store) ก็สามารถเข้าถึงข้อมูลแบบสัมพันธ์กับ PC ขนาด 32 บิตได้เช่นกัน

อะไรที่แตกต่าง? ประการแรก ไม่มีคำสั่งคำนวณจำนวนเต็มแบบไบนารี (byte) หรือฮาล์ฟเวิร์ด (half-word) การคำนวณทำกับขนาดเต็มของรีจิสเตอร์เสมอ การเข้าถึงหน่วยความจำใช้พลังงานมากกว่าการคำนวณเชิงเลขคณิตหลายเท่า ดังนั้นการเข้าถึงข้อมูลขนาดเล็ก (narrow data access) จึงช่วยประหยัดพลังงานได้อย่างมาก แต่การคำนวณกับข้อมูลขนาดเล็ก (narrow operations) ไม่ได้ช่วยประหยัดพลังงานในลักษณะเดียวกัน ARM-32 มีคุณสมบัติที่ค่อนข้างแปลก คือสามารถเลื่อนบิต (shift) หนึ่งในตัวถูกดำเนินการได้ในคำสั่งคำนวณเชิงตรรกะและเลขคณิตส่วนใหญ่ ซึ่งทำให้เส้นทางข้อมูล (datapath) ซับซ้อนขึ้น และแทบไม่จำเป็นต้องใช้บอยนิก งานศึกษาระบุว่าพีเจอนี้ไม่ค่อยถูกใช้งาน: [Hohl and Hinds 2016]; และใน RV32I จึงแยกคำสั่งเลื่อนบิต (shift) ออกมาต่างหาก

RV32I ยังไม่มีคำสั่งคูณ (multiply) และหาร (divide) โดยคำสั่งเหล่านี้ถูกจัดอยู่ในส่วนขยายเสริม RV32M (ดูบทที่ 4) แตกต่างจาก ARM-32 และ x86-32 ตรงที่ซอฟต์แวร์สแตคของ RISC-V ทั้งหมด สามารถทำงานได้โดยไม่ต้องใช้คำสั่งคูณและหาร ซึ่งช่วยลดขนาดของชิปในระบบฝังตัว (embedded) สถาปนิกของ MIPS-32 เคยสมมติว่าหน่วยคูณ-หาร (multiply-divide unit) จะไม่มีการทำงานแบบท่อ (non-pipelined) ซึ่งอาจใช้เวลาหลายสิบรอบสัญญาณนาฬิกา สถาปัตยกรรมลักษณะนี้อาจทำให้ผลลัพธ์ของคำสั่งก่อนหน้าสูญหายได้ หากมีคำสั่งคูณใหม่เริ่มทำงานก่อนที่คำสั่งก่อนหน้าจะเสร็จสิ้น ดังนั้นหากไม่ระวัง เมื่อเกิดการขัดจังหวะ (interrupt) โปรแกรมเมอร์อาจพบผลลัพธ์ที่ไม่สามารถอธิบายได้ [Sweetman 2010] แม้ว่าจะไม่ใช่ปัญหาระดับฮาร์ดแวร์โดยตรง แต่แอสเซมเบลอร์ ของ MIPS-32 อาจแทนที่คำสั่งการคูณด้วยลำดับของคำสั่งเลื่อนบิต (shifts) และบวก (adds) เพื่อพยายามเพิ่มประสิทธิภาพ ซึ่งอาจทำให้โปรแกรมเมอร์สับสน เพราะคำสั่งที่ถูกประมวลผลจริงแต่ไม่ตรงกับที่เห็นในโปรแกรมภาษาแอสเซมบลี RV32I ยังไม่มีคำสั่งหมุนบิต (rotate instructions) และไม่มี การตรวจจบการล้นของการคำนวณเลขจำนวนเต็ม (Arithmetic Overflow) อย่างไรก็ตาม ทั้งสองคำสั่งนี้สามารถคำนวณได้ด้วยคำสั่ง RV32I เพียงไม่กี่คำสั่ง (ดู หัวข้อ 2.6)



ความเรียบง่าย

สถาปัตยกรรม ARM v8 ซึ่งเป็นรุ่นต่อจาก ARM-32 ได้ยกเลิกความสามารถในการเลื่อนบิต ภายในคำสั่ง ALU ซึ่งสะท้อนโดยนัยว่าการออกแบบดังกล่าวใน ARM-32 เป็นข้อผิดพลาด



ค่าใช้จ่าย

■ รายละเอียดเพิ่มเติม: ใช้ xor ช่วยให้เกิดกรณีพิเศษ

คุณสามารถสลับค่าของสองตัวแปรได้โดยไม่ต้องใช้รีจิสเตอร์กลาง! ได้ดังนี้สลับค่าของรีจิสเตอร์ $x1$ และ $x2$ เพื่อให้ผู้อ่านพิสูจน์ด้วยตัวเอง คำใบ้ : การดำเนินการ XOR มีคุณสมบัติ สลับที่ได้ ($a \oplus b = b \oplus a$), เชื่อมโยงได้ ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), เป็น ผลผกผันของตัวเอง ($a \oplus a = 0$), และมีเอกลักษณ์ ($a \oplus 0 = a$)

```
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
```

อย่างไรก็ตาม แม่วิธีนี้จะน่าสนใจ แต่ชุดรีจิสเตอร์จำนวนมากของ RISC-V มักทำให้คอมไพเลอร์สามารถหารีจิสเตอร์ชั่วคราว (scratch register) ได้อยู่แล้ว ดังนั้นจึงแทบไม่จำเป็นต้องใช้เทคนิค XOR-swap เลย

2.5 การโหลดและการจัดเก็บข้อมูลใน RV32I

นอกจากจะรองรับการโหลดและเก็บข้อมูลขนาดเวิร์ด (32 บิต) ด้วยคำสั่ง (lw, sw) รูปที่ 2.1 แสดง RV32I แล้วยังมีคำสั่งโหลดข้อมูลขนาดไบต์และฮาล์ฟเวิร์ด (halfwords) ทั้งแบบมีเครื่องหมายและไม่มีเครื่องหมาย (lb, lbu, lh, lhu) และคำสั่งเก็บข้อมูลขนาดไบต์และฮาล์ฟเวิร์ด (sb, sh) สำหรับข้อมูลแบบมีเครื่องหมาย (signed) ทั้งไบต์และฮาล์ฟเวิร์ด จะถูกขยายเครื่องหมาย (sign-extended) เป็น 32 บิต แล้วจึงเขียนลงในรีจิสเตอร์ปลายทาง การขยายข้อมูลขนาดเล็กให้เป็น 32 บิตนี้ ช่วยให้คำสั่งคำนวณจำนวนเต็มที่สามารถทำงานกับข้อมูลทั้ง 32 บิตได้อย่างถูกต้อง แม้ว่าชนิดข้อมูลจริงจะมีขนาดเล็กกว่านั้น ส่วนข้อมูลแบบไม่มีเครื่องหมาย (unsigned) ซึ่งมีประโยชน์สำหรับข้อมูลข้อความหรือจำนวนเต็มแบบไม่มีเครื่องหมาย จะถูกขยายด้วยศูนย์ (zero-extended) เป็น 32 บิตก่อนจะเขียนลงในรีจิสเตอร์ปลายทาง



ความเรียบง่าย

โหมดการกำหนดที่อยู่ (addressing mode) สำหรับคำสั่งโหลดและเก็บข้อมูลมีเพียงโหมดเดียว คือนำค่าคงที่ทันทีขนาด 12 บิต (ที่ถูกขยายเครื่องหมาย) ไปบวกกับค่ารีจิสเตอร์ ซึ่งใน x86-32 เรียกว่า โหมดการกำหนดที่อยู่แบบ displacement [Irvine 2014]

อะไรที่แตกต่าง? RV32I ไม่มีโหมดการกำหนดที่อยู่ที่ซับซ้อนของ ARM-32 และ x86-32 นอกจากนี้ โหมดการกำหนดที่อยู่ทั้งหมดของ ARM-32 ไม่สามารถใช้ได้กับทุกประเภทข้อมูล แต่การกำหนดที่อยู่ใน RV32I ไม่เลือกปฏิบัติต่อประเภทข้อมูลใดๆ RISC-V สามารถเลียนแบบโหมดการกำหนดที่อยู่บางแบบของ x86 ได้ ตัวอย่างเช่น การตั้งค่าฟิลด์ค่าคงที่ทันทีเป็น 0 จะให้ผลเช่นเดียวกับโหมดการกำหนดที่อยู่แบบอ้อมผ่านรีจิสเตอร์ ต่างจาก x86-32 ที่ RISC-V ไม่มีคำสั่งสแต็กพิเศษ โดยการใช้หนึ่งใน 31 รีจิสเตอร์เป็นตัวชี้สแต็ก (stack pointer) (ดูรูปที่ 2.4) โหมดการกำหนดที่อยู่มาตรฐาน จะให้ประโยชน์ส่วนใหญ่ของคำสั่ง push และ pop ได้ โดยไม่ต้องเพิ่มความซับซ้อนให้กับ ISA ต่างจาก MIPS-32 โดย RISC-V ปฏิเสธการใช้ delayed load ซึ่งคล้ายคลึงกับแนวคิดของการกระโดดแบบล่าช้า (delayed branches) โดย MIPS-32 นิยามใหม่ให้คำสั่งโหลดทำให้ข้อมูลไม่พร้อมใช้งาน จนกว่าจะผ่านไปสองคำสั่ง ซึ่งข้อมูลจะปรากฏในไปป์ไลน์ห้าขั้นตอน (five-stage pipeline) การจะใส่คำสั่ง NOP หลังคำสั่งโหลดหรือไม่ ขึ้นอยู่กับคอมไพเลอร์หรือโปรแกรมเมอร์ภาษาแอสเซมบลี หากไม่สามารถย้ายคำสั่งที่มีประโยชน์มาใส่ในช่อง delay

slot ได้ อย่างไรก็ตาม ประโยชน์ของแนวคิดนี้ลดลงและแทบหายไป เมื่อ pipeline มีความยาวมากขึ้นในสถาปัตยกรรมรุ่นหลัง ๆ

ในขณะที่ ARM-32 และ MIPS-32 กำหนดให้ข้อมูลต้องจัดเรียง(align) ตามขนาดของข้อมูลในหน่วยความจำ แต่ RISC-V ไม่ได้บังคับเช่นนั้น การเข้าถึงข้อมูลแบบไม่จัดแนว (misaligned access) บางครั้งจำเป็นเมื่อมีการย้าย (port) โค้ดเก่า แนวทางหนึ่งคือการไม่อนุญาตให้มีการเข้าถึงที่ไม่จัดแนวใน ISA หลัก และเพิ่มคำสั่งพิเศษเพื่อรองรับ เช่น คำสั่ง Load Word Left และ Load Word Right ของ MIPS-32 อย่างไรก็ตาม วิธีนี้จะทำให้การเข้าถึงรีจิสเตอร์ซับซ้อนขึ้น เพราะคำสั่ง lw1 และ lw2 ต้องเขียนข้อมูลเพียงบางส่วนของรีจิสเตอร์ แทนที่จะเขียนทั้งรีจิสเตอร์ในครั้งเดียว ในทางกลับกัน การกำหนดให้คำสั่ง load/store ปกติสามารถรองรับการเข้าถึงที่ไม่จัดแนว (misaligned accesses) ได้เลย ช่วยให้การออกแบบโดยรวมเรียบง่ายขึ้น

■ รายละเอียดเพิ่มเติม: ลำดับไบนารี (Endianness)

RISC-V เลือกใช้การจัดเรียงไบนารีแบบ *little-endian* เนื่องจากเป็นมาตรฐานที่ใช้กันอย่างแพร่หลายในเชิงพาณิชย์ โดยระบบ x86-32 ทั้งหมด รวมถึง Apple iOS, Google Android และ Microsoft Windows สำหรับ ARM ต่างก็ใช้ little-endian ทั้งนี้ ลำดับ endian จะมีความสำคัญเฉพาะในกรณีที่มีการเข้าถึงข้อมูลเดียวกันทั้งในรูปแบบเวิร์ด (word) และไบนารี (bytes) เท่านั้น ดังนั้นในทางปฏิบัติ เรื่อง endianness จึงมีผลกระทบต่อโปรแกรมเมอร์เพียงเล็กน้อย

■ รายละเอียดเพิ่มเติม: ความสอดคล้องของหน่วยความจำ

ความสอดคล้องของหน่วยความจำในสถาปัตยกรรมกำหนดว่า เมื่อมีหลายเธรดทำงานร่วมกัน ค่าใดที่ถูกเขียนลงไปจะถูกอ่านกลับมาเมื่อใด แม้ว่าประเด็นนี้ยังอยู่ระหว่างการพิจารณาขององค์กรของ RISC-V แต่ซอฟต์แวร์ที่เขียนตามโมเดลหน่วยความจำของ ARM-64 สามารถทำงานบนโปรเซสเซอร์ RISC-V ได้



ค่าใช้จ่าย

2.6 การกระโดดแบบมีเงื่อนไขใน RV32I

RV32I สามารถเปรียบเทียบค่าระหว่างรีจิสเตอร์สองตัวและกระโดดตามผลลัพธ์ได้ เช่น เท่ากัน (beq), ไม่เท่ากัน (bne), มากกว่าหรือเท่ากับ (bge) หรือน้อยกว่า (blt) สองกรณีหลังเป็นการเปรียบเทียบแบบมีเครื่องหมาย และ RV32I ยังมีคำสั่งแบบไม่มีเครื่องหมาย คือ bgeu และ bltu สำหรับความสัมพันธ์อีกสองแบบ (มากกว่า และ น้อยกว่าหรือเท่ากับ) สามารถตรวจสอบได้ง่ายโดยการกลับตัวถูกดำเนินการ เช่น $x < y$ เทียบเท่ากับ $y > x$ และ $x \geq y$ เทียบเท่ากับ $y \leq x$

เนื่องจากคำสั่งใน RISC-V ต้องมีความยาวเป็นจำนวนเท่าของ 2 ไบนารี—ดูบทที่ 7 เพื่อเรียนรู้เกี่ยวกับคำสั่งขนาด 2 ไบนารีที่เป็นตัวเลือก—โหมดการกำหนดที่อยู่สำหรับการกระโดด แบบมีเงื่อนไข (branch addressing mode) จึงนำค่าคงที่ทันที ขนาด 12 บิตไปคูณด้วย 2 (หรือเลื่อนซ้าย 1 บิต) จากนั้นทำการขยายเครื่องหมาย (sign-extend) แล้วบวกเข้ากับค่าตัวนับโปรแกรม (PC) การใช้การกำหนดที่อยู่แบบสัมพันธ์กับ PC (PC-Relative Addressing) ช่วยให้สามารถเขียนโปรแกรมแบบไม่ขึ้นกับตำแหน่ง (position-independent code) ได้ ซึ่งช่วยลดภาระของลิงเกอร์และโหลดเดอร์ (ดูบทที่ 3)

คำสั่ง bltu ยังช่วยให้สามารถตรวจสอบขอบเขตของอาร์เรย์แบบมีเครื่องหมายได้ด้วยคำสั่งเดียว เนื่องจากดัชนีที่เป็นลบใดจะถูกเปรียบเทียบกับค่ามากกว่าขอบเขตที่ไม่เป็นลบเสมอ



อะไรที่แตกต่าง? ดังที่กล่าวไว้ก่อนหน้านี้ RISC-V ได้ตัดคุณสมบัติ delayed branch ที่มีชื่อเสียง (ในทางลบ) ออกไป ซึ่งเคยมีใน MIPS-32, Oracle SPARC และสถาปัตยกรรมอื่นๆ นอกจากนี้ RISC-V ยังหลีกเลี่ยงการใช้รหัสเงื่อนไข (condition codes) แบบที่มีใน ARM-32 และ x86-32 สำหรับคำสั่ง กระโดดแบบมีเงื่อนไข รหัสเงื่อนไขเหล่านี้เพิ่มสถานะพิเศษที่ถูกตั้งค่าโดยปริยายจากคำสั่งส่วนใหญ่ ซึ่งทำให้การคำนวณ dependency ในการประมวลผลแบบนอกลำดับ (out-of-order) ซับซ้อนขึ้นโดยไม่จำเป็น

สุดท้าย RV32I ยังไม่มีคำสั่งลูปของ x86-32: loop, loope, loopz, loopne, loopnz

■ **รายละเอียดเพิ่มเติม: การบวกตัวเลขแบบมัลติเวิร์ดโดยไม่ใช้รหัสเงื่อนไข**

สามารถทำได้ใน RV32I โดยใช้คำสั่ง sltu เพื่อคำนวณหาตัวทดบิต (carry-out) ดังนี้:

```
add  a0, a2, a4 # บวก 32 บิตล่าง: a0 = a2 + a4
sltu a2, a0, a2 # a2' = 1 ถ้า (a2+a4) < a2, a2' = 0 มิฉะนั้น
add  a5, a3, a5 # บวก 32 บิตบน: a5 = a3 + a5
add  a1, a2, a5 # บวกค่าตัวทดบิตจาก 32 บิตล่าง
```

■ **รายละเอียดเพิ่มเติม: การอ่าน PC**

ค่า PC ปัจจุบันสามารถอ่านได้โดยการตั้งค่าฟิลด์ค่าคงที่ทันทีของคำสั่ง auipc เป็น 0 สำหรับ x86-32 การอ่าน PC หากต้องเรียกฟังก์ชัน (ซึ่งจะ push ค่า PC ลงใน stack) จากนั้นฟังก์ชันที่ถูกเรียก(callee) จะอ่านค่า PC ที่ถูก push ไว้จากสแต็ก และสุดท้ายจึงคืนค่า PC โดยการ pop stack ดังนั้น การอ่านค่า PC ใน x86-32 ต้องใช้ถึง 1 คำสั่ง store, 2 คำสั่ง load และ 2 การกระโดดแบบ taken jumps! (การกระโดดแบบมีการเปลี่ยน PC ไปที่ปลายทางจริง)!

■ **รายละเอียดเพิ่มเติม: การตรวจสอบการล้นด้วยซอฟต์แวร์**

โปรแกรมส่วนใหญ่ (แต่ไม่ใช่ทั้งหมด) มักจะมองข้ามการล้นหลักของการคำนวณจำนวนเต็ม ดังนั้น RISC-V จึงพึ่งพาการตรวจสอบการล้นหลักในซอฟต์แวร์แทน สำหรับการบวกแบบไม่มีเครื่องหมาย (unsigned addition) การตรวจสอบ overflow ต้องใช้คำสั่งกระโดดแบบมีเงื่อนไขเพิ่มเพียงหนึ่งคำสั่งหลังการบวก:

```
addu t0, t1, t2; bltu t0, t1, overflow
```

สำหรับการบวกแบบมีเครื่องหมาย (signed addition) หากทราบเครื่องหมายของตัวถูกดำเนินการตัวหนึ่ง การตรวจสอบการล้นหลักต้องใช้เพียงคำสั่งกระโดดหนึ่งคำสั่งหลังการบวก:

```
addi t0, t1, +imm; blt t0, t1, overflow
```

กรณีนี้ครอบคลุมสถานการณ์ที่พบบ่อย คือ การบวกตัวถูกดำเนินการกับค่าคงที่แบบทันที (immediate operand)

สำหรับการบวกแบบมีเครื่องหมายทั่วไป ต้องใช้คำสั่งเพิ่มอีกสามคำสั่งหลังการบวก โดยอาศัยการสังเกตว่า ผลรวมจะมีค่าน้อยกว่าหนึ่งในตัวถูกดำเนินการ ก็ต่อเมื่อตัวถูกดำเนินการอีกตัวหนึ่งเป็นค่าลบเท่านั้น

```
add t0, t1, t2 # t0 = t1 + t2
slti t3, t2, 0 # t3 = (t2<0) : t3 = 1 ถ้า t2<0, มิฉะนั้น t3=0
slt t4, t0, t1 # t4 = (t1+t2<t1) : t4=1 ถ้า t0<t1, มิฉะนั้น t4=0
bne t3, t4, overflow # กระโดดไปที่ overflow ถ้า (t2<0) && (t1+t2>=t1)
# | (t2>=0) && (t1+t2<t1)
```

2.7 การกระโดดแบบไม่มีเงื่อนไขใน RV32I

คำสั่ง *jump and link* เพียงคำสั่งเดียว (jal) ในรูปที่ 2.1 ทำหน้าที่สองอย่าง เพื่อรองรับ



```
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;
    }
}
```

รูปที่ 2.5: การเรียงลำดับแบบแทรก (Insertion Sort) ในภาษา C แม้อัลกอริทึมนี้จะเรียบง่าย แต่มีข้อดีหลายอย่างเหนืออัลกอริทึมที่ซับซ้อนกว่า เช่น ใช้หน่วยความจำอย่างมีประสิทธิภาพ ทำงานได้รวดเร็วกับชุดข้อมูลขนาดเล็ก และมีคุณสมบัติข้อมูลเกือบเรียงอยู่แล้ว (adaptive), รักษาลำดับเดิมของข้อมูลที่ค่าเท่ากัน (stable) และสามารถแทรกได้แม้ข้อมูลยังไม่ครบ (online) โค้ดในตัวอย่างทั้งสี่รูปถูกสร้างโดยคอมไพเลอร์ GCC โดยตั้งค่าการปรับแต่ง (optimization) เพื่อลดขนาดโค้ด เนื่องจากทำให้โค้ดอ่านเข้าใจได้ง่ายที่สุด

การเรียกโปรซีเจอร์ (procedure calls) คำสั่งจะบันทึกที่อยู่ของคำสั่งถัดไป (PC+4) ลงในรีจิสเตอร์ปลายทาง ซึ่งโดยปกติคือรีจิสเตอร์เก็บที่อยู่ย้อนกลับ ra (ดูรูปที่ 2.4) เพื่อรองรับการกระโดดแบบไม่มีเงื่อนไข เราสามารถใช้ รีจิสเตอร์ศูนย์ (x0) แทน ra เป็นรีจิสเตอร์ปลายทาง เนื่องจาก x0 ไม่สามารถเปลี่ยนแปลงค่าได้ เช่นเดียวกับการกระโดดแบบมีเงื่อนไข (branches) คำสั่ง jal จะนำค่าคงที่ขนาด ขนาด 20 บิตไปคูณด้วย 2 (หรือเลื่อนซ้าย 1 บิต) จากนั้นขยายเครื่องหมาย (sign-extend) แล้วนำไปบวกกับค่า PC เพื่อสร้างที่อยู่สำหรับการกระโดด

คำสั่งกระโดดและเชื่อมโยงแบบใช้รีจิสเตอร์ (jalr) ก็มีความยืดหยุ่นในลักษณะเดียวกัน สามารถใช้เรียกฟังก์ชันไปยังที่อยู่ที่กำหนดแบบไดนามิก หรือใช้สำหรับคืนค่าจากฟังก์ชันโดยเลือก ra เป็นรีจิสเตอร์ต้นทาง และใช้รีจิสเตอร์ศูนย์ (x0) เป็นรีจิสเตอร์ปลายทาง นอกจากนี้ คำสั่ง Switch หรือ case ซึ่งต้องคำนวณตำแหน่งที่อยู่สำหรับการกระโดดก็สามารถใช้คำสั่ง jalr โดยใช้รีจิสเตอร์ศูนย์ (x0) เป็นรีจิสเตอร์ปลายทางได้เช่นกัน

อะไรที่แตกต่าง? RV32I หลีกเลียงคำสั่งเรียกฟังก์ชันที่ซับซ้อน เช่น คำสั่ง enter และ leave ของ x86-32 หรือ หน้าต่างรีจิสเตอร์ (register windows) ที่พบใน Intel Itanium, Oracle SPARC และ Cadence Tensilica

2.8 คำสั่งเบ็ดเตล็ดใน RV32I

คำสั่งที่เกี่ยวข้องกับรีจิสเตอร์ควบคุมสถานะ ได้แก่ (csrrc, csrrs, csrrw, csrrci, csrrsi, csrrwi) ในรูปที่ 2.1 ใช้สำหรับเข้าถึงรีจิสเตอร์ที่ช่วยในการวัดประสิทธิภาพของโปรแกรมได้อย่างสะดวก ตัวนับขนาด 64 บิตเหล่านี้ ซึ่งสามารถอ่านได้ครั้งละ 32 บิต ใช้สำหรับวัดเวลาจริงตามนาฬิกาโลกจริง, จำนวนรอบสัญญาณนาฬิกา (clock cycles) ที่

หน้าต่างรีจิสเตอร์ มีการเรียงประสิทธิภาพของการเรียกฟังก์ชัน (function call) โดยการมีจำนวนรีจิสเตอร์มากกว่า 32 ตัวอย่างมาก เมื่อมีการเรียกฟังก์ชันใหม่ ฟังก์ชันจะได้รับชุดรีจิสเตอร์ใหม่ หรือที่เรียกว่าหน้าต่าง จำนวน 32 รีจิสเตอร์ เพื่อส่งผ่านอาร์กิวเมนต์ระหว่างฟังก์ชัน หน้าต่างรีจิสเตอร์จะมีการซ้อนทับกัน (overlap) ซึ่งหมายความว่าบางรีจิสเตอร์จะอยู่ในหน้าต่างของฟังก์ชันที่อยู่ติดกันทั้งสองฟังก์ชัน

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

รูปที่ 2.6: จำนวนคำสั่งและขนาดโค้ดสำหรับการเรียงลำดับแบบแทรก สำหรับแต่ละ ISA สำหรับชุดคำสั่งเหล่านี้ บทที่ 7 อธิบาย ARM Thumb-2, microMIPS และ RV32C

CPU ใช้ไปในการรันโปรแกรม และจำนวนคำสั่งที่ประมวลผลเสร็จสมบูรณ์แล้ว (instructions retired)

คำสั่ง `ecall` ใช้สำหรับร้องขอไปยังสภาพแวดล้อมการทำงาน เช่น การเรียกระบบ (system calls) โปรแกรมดีบักเกอร์ใช้คำสั่ง `ebreak` เพื่อโอนการควบคุมไปยังสภาพแวดล้อมการดีบัก

คำสั่ง `fence` ใช้ในการจัดลำดับ (sequence) การเข้าถึง I/O ของอุปกรณ์และหน่วยความจำ เพื่อให้เรดอื่น ๆ หรืออุปกรณ์ภายนอก หรือโคโพรเซสเซอร์เห็นลำดับการทำงานได้ถูกต้อง คำสั่ง `fence.i` ใช้สำหรับ ซิงโครไนซ์ระหว่างลำดับคำสั่ง (instruction stream) และกระแสข้อมูล (data stream) ที่ไหลเข้าสู่ CPU RISC-V ไม่รับประกันว่า การเก็บข้อมูลไปยังหน่วยความจำ (instruction memory) จะสามารถถูกมองเห็นได้ทันทีโดยการอ่านคำสั่ง (instruction fetch) ภายในโปรเซสเซอร์เดียวกัน จนกว่าจะมีการเรียกใช้คำสั่ง `fence.i`

บทที่ 11 ครอบคลุมคำสั่งระบบของ RISC-V

อะไรที่แตกต่าง? RISC-V ใช้ memory-mapped I/O แทนคำสั่ง `in`, `ins`, `insb`, `insw` และ `out`, `outs`, `outsb`, และ `outsw` ของ x86-32 การจัดการสตริงใน RISC-V ใช้การโหลดและเก็บข้อมูลแบบไบนารี แทนคำสั่งสตริงพิเศษ 16 คำสั่งของ x86-32 เช่น `rep`, `movs`, `coms`, `scas`, `lods`, ...



ความเรียบง่าย

■ รายละเอียดเพิ่มเติม: ส่วนขยายมาตรฐาน RV32E สำหรับระบบฝังตัว

เพื่อลดต้นทุนของคอร์ระดับล่าง RV32E เป็นชุดคำสั่งพื้นฐาน ที่มีรีจิสเตอร์น้อยลง โดยลดลง 16 ตัว RV32E เป็นเหตุผลที่ รีจิสเตอร์ประเภทบันทึก (saved registers) และรีจิสเตอร์ชั่วคราว (temporary registers) ถูกแบ่งออกเป็นสองช่วง คือรีจิสเตอร์หมายเลข 0-15 และ 16-31 (รูปที่ 3.2)

2.9 เปรียบเทียบ RV32I, ARM-32, MIPS-32, และ x86-32

เราได้แนะนำชุดคำสั่งพื้นฐานของ RISC-V และได้แสดงความคิดเห็นเกี่ยวกับการเลือกออกแบบเมื่อเปรียบเทียบกับ ARM-32, MIPS-32 และ x86-32 ต่อไปนี้เราจะทำการเปรียบเทียบแบบตัวต่อตัว รูปที่ 2.5 แสดงการเรียงลำดับแบบแทรกในภาษา C ซึ่งจะใช้เป็นเกณฑ์มาตรฐาน (benchmark) ของเรา รูปที่ 2.6 เป็นตารางที่สรุปจำนวนคำสั่งและจำนวนไบต์ ในโค้ดการเรียงลำดับแบบแทรก สำหรับแต่ละ ISA รูปที่แสดงตั้งแต่ 2.8 ถึง 2.11 แสดงโค้ดที่ถูกคอมไพล์สำหรับ RV32I, ARM-32, MIPS-32 และ x86-32 แม้ว่า RISC-V จะเน้นความเรียบง่ายในการออกแบบ แต่เวอร์ชันของ RISC-V กลับใช้จำนวนคำสั่งเท่า

กันหรือน้อยกว่า และขนาดโค้ดของแต่ละสถาปัตยกรรมก็ใกล้เคียงกันมาก ในตัวอย่างนี้ คำสั่ง branch ที่ทำการเปรียบเทียบและตัดสินใจใน RISC-V (compare-and-execute branches) สามารถลดจำนวนคำสั่งได้พอ ๆ กับการใช้โหมดการกำหนดที่อยู่ที่ซับซ้อน รวมถึงคำสั่ง push และ pop ของ ARM-32 และ x86-32 ที่แสดงในรูปที่ 2.9 และ 2.11

เราได้ย้ายตัวอย่างโค้ดไปไว้หลังจบเนื้อหาของบท เพื่อรักษาความต่อเนื่องของการเขียนในบทนี้และบทถัดไป

2.10 ข้อสังเกตสรุป

ผู้ที่ไม่สามารถจดจำอดีตได้ย่อมถูกลิขิตให้ต้องทำซ้ำ

—จอร์จ ซานตายานา ค.ศ. 1905

รูปที่ 2.7 ใช้ตัวชี้วัดเจ็ดประการของการออกแบบ ISA จากบทที่ 1 เพื่อจัดระเบียบบทเรียนที่ได้จากจาก ISA ในอดีตที่กล่าวถึงในส่วนก่อนหน้านี้ และแสดงให้เห็นผลลัพธ์เชิงบวกที่เกิดขึ้นใน RV32I เราไม่ได้ต้องการจะบอกว่า แรกที่มีคุณสมบัติเหล่านี้ แท้จริงแล้ว RV32I ได้สืบทอดแนวคิดหลายอย่างมาจาก RISC-I ซึ่งเป็น “บรรพบุรุษรุ่นทวด” ของมัน [Patterson 2017]:

- พื้นที่ที่อยู่ขนาด 32 บิต มีการเข้าถึงที่อยู่แบบไบต์ (byte-addressable address space) (อ้างอิงหน่วยความจำเป็นไบต์)
- คำสั่งทั้งหมดมีความยาว 32 บิต
- มีรีจิสเตอร์ 31 ตัว ขนาด 32 บิตทั้งหมด โดยรีจิสเตอร์หมายเลข 0 ถูกกำหนดค่าเป็นศูนย์ตลอดเวลา (hardwired to zero)
- การคำนวณทั้งหมดทำระหว่างรีจิสเตอร์ (ไม่มีการคำนวณแบบ รีจิสเตอร์ไปหน่วยความจำ)
- มีคำสั่งโหลด/จัดเก็บ สำหรับเวิร์ด (word) รวมถึงการโหลด/จัดเก็บ สำหรับไบต์และฮาล์ฟเวิร์ด (halfword) ทั้งแบบมีเครื่องหมายและไม่มีเครื่องหมาย
- มีตัวเลือกค่าคงที่ทันที (immediate) สำหรับคำสั่งเลขคณิต คำสั่งตรรกะ และคำสั่งเลื่อนบิต ทุกคำสั่ง
- ค่าคงที่ทันที (immediate) จะถูกขยายเครื่องหมาย (sign-extend) เสมอ
- มีโหมดการเข้าถึงที่อยู่สำหรับข้อมูลเพียงโหมดเดียว (รีจิสเตอร์ + ค่าคงที่ทันที) และการกระโดดแบบสัมพันธ์กับ PC (PC-relative)
- ไม่มีคำสั่งคูณหรือหาร
- มีคำสั่งสำหรับโหลดค่าคงที่ทันทีขนาด 20 บิตเข้าส่วนบนของรีจิสเตอร์ เพื่อให้การสร้างค่าคงที่ 32 บิต ได้ด้วยเพียง 2 คำสั่ง

ลำดับสายวิวัฒนาการ (genealogy) ของคำสั่งทั้งหมดใน RISC-V ได้ถูกบันทึกไว้อย่างละเอียดในงานของ [Chen and Patterson 2016]

ผลกระทบของลินดี [Lin 2017] แนวคิดของ Lindy ระบุว่า อายุขัยในอนาคตของเทคโนโลยีหรือแนวคิดหนึ่งมีสัดส่วนกับอายุที่มันมีอยู่แล้ว กล่าวคือ หากสิ่งใดสามารถผ่านการทดสอบของกาลเวลาได้ ยิ่งมันอยู่รอดมาได้นานในอดีต ก็ยิ่งมีแนวโน้มว่าจะอยู่รอดต่อไปได้นานในอนาคต หากสมมติฐานนี้เป็นจริง สถาปัตยกรรม RISC อาจเป็นแนวคิดที่ดีไปได้อีกยาวนาน

RISC-V ได้เปรียบจากการเริ่มต้นช้ากว่าสถาปัตยกรรมก่อนหน้านี้ประมาณ 25–30 ปี ทำให้ผู้ออกแบบสามารถทำตามคำแนะนำของซานตายานา (Santayana) คือ เลือกใช้แนวคิดที่ดี และหลีกเลี่ยงการทำผิดซ้ำ—รวมถึงข้อผิดพลาดของ RISC-I ด้วย—ยิ่งไปกว่า



	ความผิดพลาดในอดีต			บทเรียนที่ได้รับ RV32I (2011)
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	
ต้นทุน	รองรับการคูณจำนวนเต็ม	รองรับการหารและการคูณจำนวนเต็ม	มีการดำเนินการ 8 บิต และ 16 บิต. มีการหารและการคูณจำนวนเต็ม	ไม่มีการดำเนินการ 8 บิต และ 16 บิต. มีการคูณและการหารจำนวนเต็มในส่วนเพิ่มเติม(RV32M)
ความเรียบง่าย	ไม่มีรีจิสเตอร์ศูนย์. การดำเนินการแบบมีเงื่อนไข. โหมดการเข้าถึงข้อมูลซับซ้อน. คำสั่งแบบสแต็ก (push/pop). มีการเลื่อนบิตสำหรับคำสั่งทางคณิตศาสตร์และตรรกะ	ขยายศูนย์และเครื่องหมายให้แบบให้ค่าทันที. บางคำสั่งทางคณิตศาสตร์ทำให้เกิด overflow	ไม่มีรีจิสเตอร์ศูนย์. การเรียกใช้ฟังก์ชัน/คืนค่าซับซ้อน(เข้า-ออก). คำสั่งแบบสแต็ก (push/pop). โหมดการเข้าถึงข้อมูลซับซ้อน. คำสั่งวงเล็บ	รีจิสเตอร์ศูนย์ x0 มีค่า 0. ค่าคงที่มีค่าเป็นบวก. มีโหมดการเข้าถึงข้อมูล. ไม่มีการดำเนินการแบบมีเงื่อนไข. การเรียกใช้ฟังก์ชัน/คืนค่า และคำสั่งแบบสแต็กไม่ซับซ้อน. ไม่มีการตรวจจับคำสั่งการเกิด overflow. แยกคำสั่งการเลื่อนบิต
ประสิทธิภาพ	รหัสในการกระโดดแบบมีเงื่อนไข. รีจิสเตอร์ต้นทาง-ปลายทางแตกต่างกันในทุกรูปแบบคำสั่ง. โหลดหลายตัว. คำนวณค่าคงที่. PC เป็นรีจิสเตอร์ทั่วไป	รีจิสเตอร์ต้นทาง-ปลายทางแตกต่างกันในทุกรูปแบบคำสั่ง	รหัสในการกระโดดแบบมีเงื่อนไข. มีรีจิสเตอร์อย่างมาก 2 ต่อคำสั่ง	คำสั่งเปรียบเทียบและคำสั่งกระโดดแบบมีเงื่อนไข(ไม่รหัสในการกระโดด). มีรีจิสเตอร์อย่างมาก 3 ต่อคำสั่ง. ไม่มีการโหลดหลายตัว. รีจิสเตอร์ต้นทาง-ปลายทางคงที่ในทุกรูปแบบคำสั่ง. ค่าคงที่ทันที. PC ไม่เป็นรีจิสเตอร์ทั่วไป
การแยกสถาปัตยกรรมกับการนำไปใช้	แสดงความยาวของไบต์เมื่อบันทึกลงใน PC เป็นรีจิสเตอร์ทั่วไป	มีการหน่วงการกระโดดและการหน่วงโหลด. รีจิสเตอร์ HI และ LO เพื่อการคูณและการหารเท่านั้น	รีจิสเตอร์ไม่ใช้รีจิสเตอร์วัตถุประสงค์ทั่วไป (AX, CX, DX, DI, SI มีการใช้งานเฉพาะ)	ไม่มีการหน่วงการกระโดดและการหน่วงโหลด. จิสเตอร์วัตถุประสงค์ทั่วไป
พื้นที่ในการขยายในอนาคต	พื้นที่ opcode ว่างที่จำกัด	พื้นที่ opcode ว่างที่จำกัด		พื้นที่ opcode ว่างที่กว้าง
ขนาดโปรแกรม	มีคำสั่งขนาด 32 บิตเท่านั้น (+Thumb-2 แยกจาก ISA)	มีคำสั่งขนาด 32 บิตเท่านั้น (+microMIPS แยกจาก ISA)	มีคำสั่งขนาดบิตไม่คงที่, เป็นตัวเลือกที่ไม่ดี	มีคำสั่งขนาด 32 บิต + 16 บิต ส่วนขยาย RV32C
ง่ายในการเขียนโปรแกรม / การแปลคำสั่ง / การลิงค์	มี 15 รีจิสเตอร์เท่านั้น. ข้อมูลตรงกันในหน่วยความจำ. โหมดการเข้าถึงข้อมูลไม่สม่ำเสมอ. ตัวนับประสิทธิภาพไม่สอดคล้องกัน	ข้อมูลตรงกันในหน่วยความจำ. โหมดการเข้าถึงข้อมูลไม่สม่ำเสมอ	มี 8 รีจิสเตอร์เท่านั้น. ไม่มีโหมดการเข้าถึงข้อมูลแบบสัมพัทธ์. Inconsistent performance counters	มี 31 รีจิสเตอร์. ข้อมูลกำหนดเพิ่ม. มีโหมดการเข้าถึงข้อมูลแบบสัมพัทธ์. โหมดการเข้าถึงข้อมูลสม่ำเสมอ. ตัวนับประสิทธิภาพที่กำหนดไว้ในกรอบการออกแบบ

รูปที่ 2.7: บทเรียนที่สถาปนิกของ RISC-V ได้เรียนรู้จากความผิดพลาดของชุดคำสั่งในอดีต มักจะเป็นเพียงการหลีกเลี่ยง “การปรับแต่ง” ISA ที่เคยเกิดขึ้นในอดีต บทเรียนและข้อผิดพลาดเหล่านี้จัดหมวดหมู่ตามตัวชี้วัดทั้งเจ็ดจากบทที่ 1 ของการออกแบบ ISA จากบทก่อนหน้า โดยคุณสมบัติหลายอย่างที่อยู่ในหมวดต้นทุน (cost), ความเรียบง่าย (simplicity) และประสิทธิภาพ (performance) อาจสลับหมวดกันได้ เพราะขึ้นอยู่กับมุมมอง แต่ไม่ว่าจะอยู่ในหมวดใดก็ล้วนมีความสำคัญ

นั่น RISC-V International มีแนวทางที่จะพัฒนา ISA อย่างค่อยเป็นค่อยไป ผ่านส่วนขยายเสริม (optional extensions) เพื่อหลีกเลี่ยงปัญหาการเพิ่มพีเจอร์เล็ก ๆ อย่างต่อเนื่อง (incrementalism) ที่เคยสร้างปัญหาให้กับ ISA ที่ประสบความสำเร็จในอดีต

■ **รายละเอียดเพิ่มเติม: RV32I มีเอกลักษณ์เฉพาะตัวหรือไม่?**

ในอดีต ไมโครโปรเซสเซอร์ยุคแรกมีชิปแยกสำหรับการคำนวณ เลขทศนิยม ดังนั้นคำสั่งเหล่านี้จึงเป็นแบบ optional ต่อมา กฎของมัวร์ (Moore's Law) ทำให้ทุกอย่างสามารถรวมอยู่ในชิปเดียวได้ และแนวคิดความเป็นโมดูล (modularity) ใน ISA ก็ลดความสำคัญลง การลดทอน ISA (subsetting) ในโปรเซสเซอร์ขนาดเล็ก และให้ซอฟต์แวร์ช่วยจำลองคำสั่งที่ขาดหายไป (emulation) เป็นแนวคิดที่มีมานานแล้ว เช่นใน IBM 360 model 44 และ Digital Equipment microVAX แต่ RV32I แตกต่างตรงที่: ซอฟต์แวร์สแตกทั้งหมดสามารถทำงานได้ด้วยคำสั่งพื้นฐานเท่านั้น ดังนั้น โปรเซสเซอร์ RV32I ไม่จำเป็นต้อง trap ซ้ำๆ เพื่อจำลองคำสั่งที่ไม่มีใน RV32G สถาปัตยกรรมที่ใกล้เคียงกับ RISC-V ในแง่ก็คือ Tensilica Xtensa ซึ่งออกแบบมาสำหรับงาน embedded Xtensa มี ISA พื้นฐานประมาณ 80 คำสั่ง และเปิดให้ผู้ใช้สามารถเพิ่มคำสั่งเฉพาะ (custom instructions) เพื่อเร่งประสิทธิภาพงานของตนเอง RV32I มีชุดคำสั่งพื้นฐานที่เรียบง่ายกว่า มีรุ่นที่รองรับที่อยู่ 64 บิต และมีส่วนขยายที่รองรับทั้งซูเปอร์คอมพิวเตอร์ และไมโครคอนโทรลเลอร์

2.11 เรียนรู้เพิ่มเติม

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.

D. Sweetman. *See MIPS run*. Morgan Kaufmann, 2010.

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

```

# RV32I (19 คำสั่ง, 76 bytes, หรือ 52 bytes ด้วย RVC)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
0: 00450693 addi a3,a0,4 # a3 ชี้ไปที่ a[i]
4: 00100713 addi a4,x0,1 # i = 1
Outer Loop:
8: 00b76463 bltu a4,a1,10 # ถ้า i < n กระโดดไปที่ Continue Outer Loop
Exit Outer Loop:
c: 00008067 jalr x0,x1,0 # ออกจากฟังก์ชัน
Continue Outer Loop:
10: 0006a803 lw a6,0(a3) # x = a[i]
14: 00068613 addi a2,a3,0 # a2 เป็นพอยเตอร์ของ 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 # ถ้า a[j-1] <= a[i] กระโดดไปที่ 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 # ลด a2 ชี้ไปที่ a[j]
30: fe0796e3 bne a5,x0,1c # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
34: 00279793 slli a5,a5,0x2 # คูณ a5 ด้วย 4
38: 00f507b3 add a5,a0,a5 # a5 ตำแหน่งที่อยู่ของ a[j]
3c: 0107a023 sw a6,0(a5) # a[j] = x
40: 00170713 addi a4,a4,1 # i++
44: 00468693 addi a3,a3,4 # เพิ่ม a3 ชี้ไปที่ a[i]
48: fc1ff06f jal x0,8 # กระโดดไปที่ Outer Loop

```

รูปที่ 2.8: โค้ด RV32I สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5. ทางด้านซ้ายเป็นแอดเดรสในรูปแบบเลขฐานสิบหก (hexadecimal) ถัดมาเป็นโค้ดภาษาเครื่องในรูปแบบเลขฐานสิบหก และตามด้วยคำสั่งภาษาแอสเซมบลีพร้อมคอมเมนต์ RV32I จัดสรรรีจิสเตอร์ 2 ตัว ให้ชี้ไปยัง $a[j]$ และ $a[j-1]$ RV32I มีรีจิสเตอร์จำนวนมากเพียงพอ โดยบางส่วนถูกกำหนดไว้ตาม ABI สำหรับใช้ในการเรียกฟังก์ชัน แตกต่างจาก ISA อื่น RV32I ไม่จำเป็นต้องบันทึกและกู้คืนรีจิสเตอร์เหล่านี้ ลงในหน่วยความจำ แม้ว่าขนาดโค้ดจะใหญ่กว่า x86-32 แต่การใช้คำสั่งเสริม RV32C (ดูบทที่ 7) จะช่วยลดช่องว่างของขนาดโค้ดได้

สังเกตว่า คำสั่งเปรียบเทียบและการกระโดดของ RISC-V ช่วยหลีกเลี่ยงการใช้คำสั่งเปรียบเทียบ 3 คำสั่งที่ ARM-32 และ x86-32 จำเป็นต้องใช้

```
# ARM-32 (19 คำสั่ง, 76 bytes; หรือ 18 คำสั่ง/46 bytes ด้วย Thumb-2)
# r0ชี้ไปที่ a[0], r1 คือ n, r2 คือ j, r3 คือ i, r4 คือ x
0: e3a03001 mov r3, #1          # i = 1
4: e1530001 cmp r3, r1          # i และ n (ไม่จำเป็น?)
8: e1a0c000 mov ip, r0          # ip = a[0]
c: 212ffff1e bxcs lr           # ไม่คืนตำแหน่งที่อยู่
10: e92d4030 push {r4, r5, lr}   # บันทึก r4, r5, คืนตำแหน่งที่อยู่
Outer Loop:
14: e5bc4004 ldr r4, [ip, #4]!    # x = a[i] ; เพิ่ม ip
18: e1a02003 mov r2, r3          # j = i
1c: e1a0e00c mov lr, ip          # lr = a[0] (ใช้ lr เป็น scratch reg)
Inner Loop:
20: e51e5004 ldr r5, [lr, #-4]    # r5 = a[j-1]
24: e1550004 cmp r5, r4          # เปรียบเทียบ a[j-1] vs. x
28: da000002 ble 38              # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
2c: e2522001 subs r2, r2, #1     # j--
30: e40e5004 str r5, [lr], #-4   # a[j] = a[j-1]
34: 1affffff9 bne 20             # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
38: e2833001 add r3, r3, #1      # i++
3c: e1530001 cmp r3, r1          # i และ n
40: e7804102 str r4, [r0, r2, lsl #2] # a[j] = x
44: 3affffff2 bcc 14             # ถ้า i < n กระโดดไปที่ Outer Loop
48: e8bd8030 pop {r4, r5, pc}    # คืน r4, r5, และ คืนตำแหน่งที่อยู่
```

รูปที่ 2.9: โค้ด ARM-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 ทางด้านซ้ายเป็นแอดเดรสในรูปแบบเลขฐานสิบหก (hexadecimal) ถัดมาเป็นโค้ดภาษาเครื่องในรูปแบบเลขฐานสิบหก และตามด้วยคำสั่งภาษาแอสเซมบลีพร้อมคอมเมนต์ เนื่องจากมีรีจิสเตอร์ไม่เพียงพอ ARM-32 จึงต้องบันทึกรีจิสเตอร์สองตัวลงใน stack เพื่อใช้งานภายหลัง พร้อมกับบันทึกที่อยู่ย้อนกลับ (return address) ไปด้วย ARM-32 ใช้โหมดการเข้าถึงที่อยู่ทำการสเกลค่า (scale) ค่า i และ j ให้เป็นที่อยู่แบบไบต์ เนื่องจากการกระโดดแบบมีเงื่อนไขอาจทำให้มีการสลับ ISA ระหว่าง ARM-32 และ Thumb-2 คำสั่ง bxcs จึงทำการตั้งค่าบิตลำดับต่ำสุด (least significant bit) ของที่อยู่ย้อนกลับให้เป็น 0 ก่อนที่จะบันทึกรหัสเงื่อนไขช่วยลดความจำเป็นในการใช้คำสั่งเปรียบเทียบหนึ่งคำสั่ง สำหรับตรวจสอบค่า j หลังจากลดค่าลงแล้ว แต่ยังคงมีการเปรียบเทียบอีกสามจุดในโค้ด

```

# MIPS-32 (24 คำสั่ง, 96 bytes, หรือ 56 bytes ด้วย microMIPS)
# a1 คือ n, a3 เป็นตัวชี้ไปที่ a[0], v0 คือ j, v1 คือ i, t0 คือ x
0: 24860004 addiu a2,a0,4 # a2 เป็นตัวชี้ไปที่ a[i]
4: 24030001 li v1,1 # i = 1
Outer Loop:
8: 0065102b sltu v0,v1,a1 # กำหนดให้ i < n
c: 14400003 bnez v0,1c # ถ้า i < n กระโดดไปที่ Continue Outer Loop
10: 00c03825 move a3,a2 # a3 เป็นตัวชี้ไปที่ a[j] (ช่องเสียเบ็ดเตล็ด)
14: 03e00008 jr ra # ออกจากฟังก์ชัน
18: 00000000 nop # ไม่ดำเนินการ
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 # ไม่ดำเนินการ
2c: 0109502a slt t2,t0,t1 # set a[i] < a[j-1]
30: 11400005 beqz t2,48 # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
34: 00000000 nop # ไม่ดำเนินการ
38: 2442ffff addiu v0,v0,-1 # j--
3c: ace90000 sw t1,0(a3) # a[j] = a[j-1]
40: 1440ffff bnez v0,24 # ถ้า j != 0 กระโดดไปที่ Inner Loop
44: 24e7fffc addiu a3,a3,-4 # ลด a3 ชี้ไปที่ a[j] (ช่องเสียเบ็ดเตล็ด)
Exit Inner Loop:
48: 00021080 sll v0,v0,0x2 #
4c: 00821021 addu v0,a0,v0 # v0 เป็นที่อยู่ของ a[j]
50: ac480000 sw t0,0(v0) # a[j] = x
54: 24630001 addiu v1,v1,1 # i++
58: 1000ffeb b 8 # กระโดดไปที่ Outer Loop
5c: 24c60004 addiu a2,a2,4 # เพิ่ม a2 ชี้ไปที่ a[i] (ช่องเสียเบ็ดเตล็ด)

```

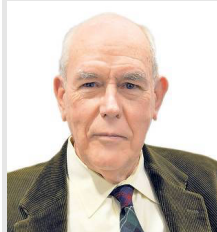
รูปที่ 2.10: โค้ด MIPS-32 สำหรับการเรียงลำดับแบบแทรก ในรูปที่ 2.5 ทางด้านซ้ายเป็นตำแหน่งที่อยู่ในรูปแบบเลขฐานสิบหก (hexadecimal) ถัดมาเป็นโค้ดภาษาเครื่องในรูปแบบเลขฐานสิบหก และตามด้วยคำสั่งภาษาแอสเซมบลี พร้อมคอมเมนต์อธิบาย โค้ด MIPS-32 มีคำสั่ง nop 3 คำสั่ง ทำให้ดัดยาวขึ้น โดย 2 คำสั่งเกิดจาก delayed branches และอีก 1 คำสั่งเกิดจาก delayed load คอมไพเลอร์ไม่สามารถหาคำสั่งที่มีประโยชน์มาใส่ในช่องหน่วงเวลาเหล่านั้นได้ นอกจากนี้ Delayed branches ยังทำให้โค้ดเข้าใจยากขึ้น เนื่องจากคำสั่งที่อยู่ถัดจาก branch จะถูกประมวลผลด้วย แม้ว่าจะมีการกระโดด (branch หรือ jump) เกิดขึ้นก็ตาม ตัวอย่างเช่น คำสั่งสุดท้าย (addiu) ที่ตำแหน่งที่อยู่ 5c เป็นส่วนหนึ่งของการลูป แม้ว่าจะอยู่ถัดจากคำสั่งกระโดดแบบมีเงื่อนไขก็ตาม

```
# x86-32 (20 คำสั่ง, 45 bytes)
# eax คือ j, ecx คือ x, edx คือ i
# ตัวชี้ไปที่ a[0] ในหน่วยความจำที่ตำแหน่งที่อยู่ esp+0xc, n ในหน่วยความจำที่ตำแหน่งที่อยู่ esp+0x10
0: 56          push esi          # บันทึก esi บน stack (esi needed below)
1: 53          push ebx          # บันทึก ebx บน stack (ebx needed below)
2: ba 01 00 00 mov  edx,0x1      # i = 1
7: 8b 4c 24 0c  mov  ecx,[esp+0xc] # ecx เป็นตัวชี้ไปที่ a[0]
Outer Loop:
b: 3b 54 24 10  cmp  edx,[esp+0x10] # เปรียบเทียบ i กับ n
f: 73 19        jae  2a <Exit Loop>    # ถ้า i >= n กระโดดไปที่ 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        # เปรียบเทียบ a[j-1] vs. x
1c: 7e 06        jle  24 <Exit Loop>    # ถ้า a[j-1] <= a[i] กระโดดไปที่ 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>    # ถ้า j != 0 กระโดดไปที่ 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>    # กระโดดไปที่ Outer Loop
Exit Outer Loop:
2a: 5b          pop  ebx            # กู้คืน ค่าเดิมของ ebx จาก stack
2b: 5e          pop  esi            # กู้คืน ค่าเดิมของ esi จาก stack
2c: c3          ret              # ออกจากฟังก์ชัน
```

รูปที่ 2.11: โค้ด x86-32 สำหรับอัลกอริทึมการเรียงลำดับแบบแทรก ใน 2.5 แสดงดังนี้ ทางด้านซ้ายเป็นตำแหน่งที่อยู่ในรูปแบบเลขฐานสิบหก (hexadecimal) ถัดมาเป็นโค้ดภาษาเครื่องในรูปแบบเลขฐานสิบหก และตามด้วยคำสั่งภาษาแอสเซมบลี พร้อมคอมเมนต์อธิบาย มีรูปแบบภาษาแอสเซมบลีของ x86 อยู่สองรูปแบบที่นิยม ได้แก่ รูปแบบ Intel/Microsoft และ AT&T/Linux เราใช้รูปแบบ Intel ในที่นี้เราใช้รูปแบบ Intel ส่วนหนึ่ง เพราะมันมีลำดับตัวถูกดำเนินการเหมือนกับ RISC-V, ARM-32 และ MIPS-32 คือให้ตัวถูกดำเนินการปลายทางอยู่ทางซ้าย และตัวถูกดำเนินการด้านหน้าอยู่ทางขวา ในขณะที่รูปแบบ AT&T จะสลับกัน (และใช้เครื่องหมาย % นำหน้าชื่อรีจิสเตอร์) แม้จะดูเป็นเรื่องเล็กน้อย แต่สำหรับโปรแกรมเมอร์บางคน เรื่องนี้ถือเป็นประเด็นที่ถกเถียงกันอย่างจริงจัง การเลือกใช้รูปแบบในที่นี้ ขึ้นอยู่กับเหตุผลด้านการสอน (pedagogy) ไม่ใช่ความเชื่อหรือแนวทางแบบเคร่งครัด (orthodoxy)

ภาษาแอสเซมบลี RISC-V

อีวาล ชัทเธอร์แลนด์ (1938–) ได้รับการขนานนามว่าเป็น บิดาแห่งคอมพิวเตอร์กราฟิก จากการประดิษฐ์ Sketchpad —ซึ่งเป็นระบบต้นแบบในปี 1962 ของส่วนต่อประสานกราฟิกกับผู้ใช้ (GUI) ในคอมพิวเตอร์สมัยใหม่— และผลงานนี้ทำให้เขาได้รับรางวัลทัวริง (Turing Award)



มันเป็นสิ่งที่น่าสนใจอย่างมาก เมื่อเราสามารถนำปัญหาที่คิดว่ายาก มาพบวิธีแก้ที่เรียบง่ายได้ วิธีแก้ปัญหานั้นที่ดีที่สุดมักจะเป็นวิธีที่เรียบง่ายเสมอ

—อีวาล ชัทเธอร์แลนด์

3.1 บทนำ

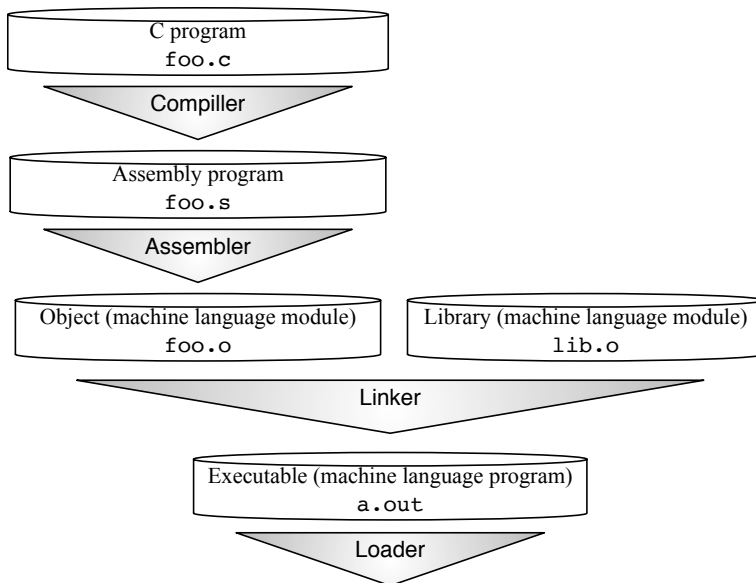
รูปที่ 3.1 แสดงขั้นตอนพื้นฐานสี่ขั้นตอนของการแปลโปรแกรม โดยเริ่มจากโปรแกรมภาษา C และสิ้นสุดด้วยที่โปรแกรมภาษาเครื่อง ที่พร้อมรันบนคอมพิวเตอร์ ในบทนี้ครอบคลุมขั้นตอนสามขั้นตอนหลัง แต่เราเริ่มต้นด้วยบทบาทของแอสเซมเบลอร์ในข้อตกลงในการเรียกฟังก์ชันของ RISC-V (RISC-V calling convention)

3.2 ข้อตกลงในการเรียกฟังก์ชัน

มีขั้นตอนทั่วไป 6 ขั้นตอนในการเรียกฟังก์ชัน: Patterson and Hennessy 2021

1. วางอาร์กิวเมนต์ไว้ในตำแหน่งหน่วยความจำ หรือรีจิสเตอร์ที่ฟังก์ชันสามารถเข้าถึงได้ [เป็นการตัดสินใจในระดับ API หรือสถาปัตยกรรมโปรแกรม]
2. กระโดดไปยังฟังก์ชัน (โดยใช้คำสั่ง `jal` ของ RV32I)
3. จัดสรรพื้นที่เก็บข้อมูลภายใน (local storage) ที่ฟังก์ชันต้องใช้ และบันทึกค่ารีจิสเตอร์ที่จำเป็นเอาไว้
4. ดำเนินงานตามที่ฟังก์ชันต้องการ
5. นำผลลัพธ์ของฟังก์ชันไปเก็บไว้ในตำแหน่งที่โปรแกรมผู้เรียกสามารถเข้าถึงได้ พร้อมทั้งกู้คืนรีจิสเตอร์และคืนพื้นที่เก็บข้อมูลภายใน
6. เนื่องจาก ฟังก์ชัน สามารถ ถูก เรียก จาก หลาย ตำแหน่ง ใน โปรแกรม จึง ต้อง ส่ง การ ควบคุมกลับไปยังจุดที่เรียกฟังก์ชัน (โดยใช้คำสั่ง `ret`)

เพื่อให้ได้ประสิทธิภาพที่ดี เพื่อให้ได้ประสิทธิภาพที่ดี ควรพยายามเก็บตัวแปรไว้ในรีจิสเตอร์แทนหน่วยความจำ แต่ในขณะเดียวกัน ก็ควรหลีกเลี่ยงการเข้าถึงหน่วยความจำบ่อยๆ เพื่อบันทึกและเรียกคืนค่ารีจิสเตอร์เหล่านั้น



รูปที่ 3.1: ขั้นตอนการแปลจากโค้ดภาษา C ไปเป็นโปรแกรมที่สามารถทำงานได้ ขั้นตอนเหล่านี้คือขั้นตอนเชิงตรรกะ แม้ว่าบางขั้นตอนจะถูกรวมเข้าด้วยกันเพื่อเร่งกระบวนการแปล เราใช้รูปแบบนามสกุลไฟล์แบบ Unix สำหรับไฟล์แต่ละประเภท โดยนามสกุลที่เทียบเท่าในระบบ MS-DOS คือ .C, .ASM, .OBJ, .LIB และ .EXE

โซคตที่ RISC-V มีจำนวนรีจิสเตอร์เพียงพอ ทำให้สามารถได้ข้อดีทั้งสองด้าน โดยสามารถเก็บตัวถูกดำเนินการไว้ในรีจิสเตอร์ และลดความจำเป็นในการบันทึกและกู้คืนค่ารีจิสเตอร์ลง แนวคิดสำคัญคือ การแบ่งรีจิสเตอร์ออกเป็นสองประเภท ได้แก่ รีจิสเตอร์ที่ไม่รับประกันว่าจะคงค่าเดิมหลังจากเรียกฟังก์ชัน เรียกว่า รีจิสเตอร์ชั่วคราว (temporary registers) และรีจิสเตอร์ที่รับประกันว่าจะคงค่าเดิม เรียกว่า รีจิสเตอร์ที่ต้องรักษาค่า (saved register) ฟังก์ชันที่ไม่มีการเรียกฟังก์ชันอื่นต่อ เรียกว่า ฟังก์ชันใบ (leaf functions) เมื่อฟังก์ชันแบบ leaf มีจำนวนอาร์กิวเมนต์และตัวแปรภายในไม่มาก เราสามารถเก็บข้อมูลทั้งหมดไว้ในรีจิสเตอร์ได้ โดยไม่ต้อง “ย้ายออก” (spilling) ไปเก็บในหน่วยความจำ หากเงื่อนไขเหล่านี้เป็นจริง โปรแกรมจะไม่จำเป็นต้องบันทึกค่าของรีจิสเตอร์ลงในหน่วยความจำ และที่น่าสนใจคือ ฟังก์ชันจำนวนมากที่อยู่ในกรณีที่เหมาะสมแบบนี้

รีจิสเตอร์อื่นๆ ภายในระหว่างการเรียกฟังก์ชัน จะต้องถูกจัดให้อยู่ในกลุ่มเดียวกับ รีจิสเตอร์แบบรักษาค่า (saved registers) ซึ่งค่าจะต้องถูกคงไว้ข้ามการเรียกฟังก์ชัน หรือรีจิสเตอร์ชั่วคราว ซึ่งไม่จำเป็นต้องเก็บรักษาค่า ฟังก์ชันจะเปลี่ยนค่าในรีจิสเตอร์ที่ใช้เก็บผลลัพธ์ (return value) ดังนั้นรีจิสเตอร์เหล่านี้จึงมีลักษณะเหมือนรีจิสเตอร์ชั่วคราว ไม่จำเป็นต้องรักษาค่า return address หรืออาร์กิวเมนต์ของฟังก์ชัน ดังนั้นรีจิสเตอร์เหล่านี้ก็ถือว่าเป็นแบบชั่วคราวเช่นกัน ผู้เรียกฟังก์ชัน (caller) สามารถคาดหวังได้ว่ารีจิสเตอร์ที่เหลืออยู่ คือ stack pointer จะไม่เปลี่ยนแปลงหลังจากการเรียกฟังก์ชัน รูปที่ 3.2 แสดง



Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

รูปที่ 3.2: ชื่อย่อแบบแอสเซมบลีสำหรับรีจิสเตอร์จำนวนเต็มและรีจิสเตอร์จุดลอยตัวของ RISC-V RISC-V มีจำนวนรีจิสเตอร์เพียงพอ ทำให้ ABI สามารถจัดสรรรีจิสเตอร์บางส่วนให้ฟังก์ชันหรือเมธอดใช้งานได้อย่างอิสระ โดยไม่จำเป็นต้องบันทึกหรือกู้คืนค่า หากฟังก์ชันนั้นไม่ได้เรียกฟังก์ชันอื่นต่อ รีจิสเตอร์ที่ต้องรักษาค่าข้ามการเรียกฟังก์ชัน เรียกว่า *callee save* ส่วนรีจิสเตอร์ที่ไม่ต้องการรักษาค่า เรียกว่า *caller Saved* บทที่ 5 จะอธิบายเกี่ยวกับรีจิสเตอร์จุดลอยตัว $\mathbb{F}$ (ตารางที่ 20.1 จากเอกสาร [Waterman and Asanović 2017] เป็นที่มาของภาพประกอบนี้)

ชื่อรีจิสเตอร์ตามมาตรฐาน ABI ของ RISC-V และระบุว่ารีจิสเตอร์ใดต้องรักษาค่าหรือไม่เมื่อมีการเรียกฟังก์ชัน

เมื่อพิจารณาโค้ด C เราจะเห็นโค้ดมาตรฐานของ RV32I สำหรับการเข้าสู่ฟังก์ชัน (function entry) และออกจากฟังก์ชัน โค้ดส่วนต้นของฟังก์ชัน (prologue) มีดังนี้:

```
entry_label:
    addi sp, sp, -framesize    # จัดสรรพื้นที่สำหรับ stack frame ของฟังก์ชัน
                                # โดยปรับค่า stack pointer (รีจิสเตอร์ sp) ลง
    sw   ra, framesize-4(sp)   # บันทึกค่า return address (รีจิสเตอร์ ra) ลงใน stack
                                # บันทึกรีจิสเตอร์อื่นลงสแต็กหากจำเป็น
    ... # ส่วนคำสั่งหลักของฟังก์ชัน (การทำงานจริงของฟังก์ชัน)
```

หากจำนวนอาร์กิวเมนต์และตัวแปรของฟังก์ชันมีมากกว่าจะเก็บในรีจิสเตอร์ได้ *ส่วนต้นของฟังก์ชัน* จะจัดสรรพื้นที่บนสแต็กสำหรับสิ่งที่เรียกว่า *stack frame* หลังจากฟังก์ชันทำงานเสร็จแล้ว *ส่วนท้ายของฟังก์ชัน (epilogue)* หลังจากฟังก์ชันทำงานเสร็จ จะต้องคืนสถานะเดิมและกลับไปยังจุดที่เรียก:

```
# คืนค่ารีจิสเตอร์จาก stack หากมีการบันทึกไว้ก่อนหน้านี้
lw   ra, framesize-4(sp)    # คืนค่า return address (รีจิสเตอร์ ra) กลับมาจาก stack
addi sp, sp, framesize      # คืนพื้นที่ของ stack frame
ret                                # กลับไปที่จุดที่เรียกฟังก์ชัน
```

เราจะเห็นตัวอย่างที่ปฏิบัติตาม ABI จากนั้น แต่ก่อนอื่น เราต้องอธิบายงานแอสเซมบลีที่เหลือ นอกเหนือจากการแปลงชื่อรีจิสเตอร์ ของ ABI เป็นหมายเลขรีจิสเตอร์

■ **รายละเอียดเพิ่มเติม:** รีจิสเตอร์แบบรักษาค่า (saved) และรีจิสเตอร์ชั่วคราว (temporary) ไม่ได้ถูกจัดเรียงติดกัน

เพื่อรองรับ RV32E ซึ่งเป็นเวอร์ชันสำหรับระบบฝังตัวของ RISC-V ที่มีรีจิสเตอร์เพียง 16 ตัว (ดูบทที่ 12) RV32E ใช้หมายเลขรีจิสเตอร์หมายเลข x0 ถึง x15 ดังนั้นรีจิสเตอร์แบบรักษาค่าและรีจิสเตอร์ชั่วคราวบางส่วนจึงอยู่ในช่วงนี้ และส่วนที่เหลืออยู่ใน 16 ตัวท้าย RV32E มีขนาดเล็กกว่า แต่ในปัจจุบันยังไม่มีกรณีสนับสนุนจากคอมไพเลอร์ เนื่องจากไม่สอดคล้องกับ RV32I

3.3 แอสเซมบลี

ข้อมูลนำเข้าในขั้นตอนนีสำหรับระบบ Unix คือไฟล์ที่มีนามสกุล .s เช่น foo.s ส่วนในระบบ MS-DOS จะใช้นามสกุล .ASM

งานของขั้นตอนแอสเซมเบลอร์ ในรูปที่ 3.1 ไม่ได้มีแค่การแปลงคำสั่งให้เป็น object code ที่โปรเซสเซอร์เข้าใจเท่านั้น แต่ยังขยายชุดคำสั่งให้รองรับการใช้งานที่สะดวกขึ้นสำหรับโปรแกรมเมอร์ภาษาแอสเซมบลีหรือผู้พัฒนาคอมไพเลอร์อีกด้วย คำสั่งในกลุ่มนี้ถูกสร้างขึ้นจากการนำคำสั่งพื้นฐานมาจัดเรียงอย่างชาญฉลาด เรียกว่า *คำสั่งเทียม* (pseudoinstructions) รูปที่ 3.3 และ 3.4 ได้แสดงรายการคำสั่งเทียมของ RISC-V โดยกลุ่มแรกทั้งหมดใช้รีจิสเตอร์ x0 ที่มีค่าเป็นศูนย์เสมอ ส่วนกลุ่มที่สองไม่จำเป็นต้องใช้รีจิสเตอร์ x0 ตัวอย่างเช่น คำสั่ง ret เป็นคำสั่งเทียมที่แอสเซมเบลอร์จะแทนที่ด้วยคำสั่ง jalr x0, x1, 0 (ดูรูปที่ 3.3) คำสั่งเทียมส่วนใหญ่ของ RISC-V ใช้รีจิสเตอร์ x0 การกัน



คำสั่งหลอก	คำสั่งพื้นฐาน	คำสั่งพื้นฐาน
nop	addi x0, x0, 0	No operation
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
snez rd, rs	sltu rd, x0, rs	Set if $\neq$ zero
sltz rd, rs	slt rd, rs, x0	Set if $<$ zero
sgtz rd, rs	slt rd, x0, rs	Set if $>$ zero
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if $\neq$ zero
blez rs, offset	bge x0, rs, offset	Branch if $\leq$ zero
bgez rs, offset	bge rs, x0, offset	Branch if $\geq$ zero
bltz rs, offset	blt rs, x0, offset	Branch if $<$ zero
bgtz rs, offset	blt x0, rs, offset	Branch if $>$ zero
j offset	jal x0, offset	Jump
jr rs	jalr x0, rs, 0	Jump register
ret	jalr x0, x1, 0	Return from subroutine
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine
rdinstret[h] rd	csrrs rd, instret[h], x0	Read instructions-retired counter
rdcycle[h] rd	csrrs rd, cycle[h], x0	Read cycle counter
rdtime[h] rd	csrrs rd, time[h], x0	Read real-time clock
csrr rd, csr	csrrs rd, csr, x0	Read CSR
csrw csr, rs	csrrw x0, csr, rs	Write CSR
csrs csr, rs	csrrs x0, csr, rs	Set bits in CSR
csrc csr, rs	csrrc x0, csr, rs	Clear bits in CSR
csrwi csr, imm	csrrwi x0, csr, imm	Write CSR, immediate
csrsi csr, imm	csrrsi x0, csr, imm	Set bits in CSR, immediate
csrci csr, imm	csrrci x0, csr, imm	Clear bits in CSR, immediate
frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register
frrm rd	csrrs rd, frm, x0	Read FP rounding mode
frrm rs	csrrw x0, frm, rs	Write FP rounding mode
frflags rd	csrrs rd, fflags, x0	Read FP exception flags
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags

รูปที่ 3.3: คำสั่งเทียม ของ RISC-V จำนวน 32 คำสั่ง ที่ใช้รีจิสเตอร์ x0, รีจิสเตอร์ศูนย์ ภาค
 แผนก A ได้รวบรวมถึงคำสั่งเทียมและคำสั่งจริงของ RISC-V คำสั่งที่ใช้ในการอ่านตัวนับ
 ขนาด 64 บิต สามารถอ่าน 32 บิตส่วนบนใน RV32I โดยใช้เวอร์ชันมีตัวอักษร “h” และ
 อ่านค่า 32 บิตส่วนล่าง โดยใช้คำสั่งปกติ (ตารางที่ 20.2 และ 20.3 ของ [Waterman and
 Asanović 2017] เป็นพื้นฐานของรูปนี้)

คำสั่งหลัก	คำสั่งพื้นฐาน	คำสั่งพื้นฐาน
lla rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	โหลดที่อยู่ภายใน (Load local address)
la rd, symbol	<i>PIC:</i> auipc rd, GOT[symbol][31:12] l{w d} rd, rd, GOT[symbol][11:0] <i>Non-PIC:</i> Same as lla rd, symbol	Load address
l{b h w d} rd, symbol	auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0] (rd)	Load global
s{b h w d} 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)	Floating-point load global
fs{w d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0] (rt)	Floating-point store global
li rd, immediate	<i>Myriad sequences</i>	โหลดค่าคงที่ (Load immediate)
mv rd, rs	addi rd, rs, 0	คัดลอกรีจิสเตอร์ (Copy register)
not rd, rs	xori rd, rs, -1	ส่วนกลับของบิต (One's complement)
sext.w rd, rs	addiw rd, rs, 0	ส่วนกลับของบิต (One's complement)
seqz rd, rs	sltiu rd, rs, 1	ขยายเครื่องหมายของ word (Sign extend word)
fmv.s rd, rs	fsgnj.s rd, rs, rs	ตั้งค่าเป็น 1 ถ้าเท่ากับศูนย์ (Set if = zero)
fabs.s rd, rs	fsgnjx.s rd, rs, rs	ค่าสัมบูรณ์ทศนิยมความแม่นยำเดี่ยว (Single-precision absolute value)
fneg.s rd, rs	fsgnjn.s rd, rs, rs	เปลี่ยนเครื่องหมายทศนิยมความแม่นยำเดี่ยว (Single-precision negate)
fmv.d rd, rs	fsgnj.d rd, rs, rs	คัดลอกรีจิสเตอร์ทศนิยมความแม่นยำคู่ (Copy double-precision register)
fabs.d rd, rs	fsgnjx.d rd, rs, rs	ค่าสัมบูรณ์ทศนิยมความแม่นยำคู่ (Double-precision absolute value)
fneg.d rd, rs	fsgnjn.d rd, rs, rs	เปลี่ยนเครื่องหมายทศนิยมความแม่นยำคู่ (Double-precision negate)
bgt rs, rt, offset	blt rt, rs, offset	แบ่งแขนงถ้ามากกว่า (Branch if >)
ble rs, rt, offset	bge rt, rs, offset	แบ่งแขนงถ้าไม่น้อยกว่าหรือเท่ากับ (Branch if ≤)
bgtu rs, rt, offset	bltu rt, rs, offset	แบ่งแขนงถ้ามากกว่า (ไม่มีเครื่องหมาย) (Branch if >, unsigned)
bleu rs, rt, offset	bgeu rt, rs, offset	แบ่งแขนงถ้าไม่น้อยกว่าหรือเท่ากับ (ไม่มีเครื่องหมาย) (Branch if ≤, unsigned)
jal offset	jal x1, offset	กระโดดและเชื่อมโยง (Jump and link)
jalr rs	jalr x1, rs, 0	กระโดดและเชื่อมโยงด้วยรีจิสเตอร์ (Jump and link register)
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	เรียกซับรูทีนที่อยู่ไกล (Call far-away subroutine)
fence	fence iorw, iorw	รั้วสำหรับหน่วยความจำและ I/O ทั้งหมด (Fence on all memory and I/O)
fscsr rd, rs	csrrw rd, fcsr, rs	สลับโหมดการปัดเศษทศนิยม (Swap FP rounding mode)
fsm rd, rs	csrrw rd, frm, rs	สลับโหมดการปัดเศษทศนิยม (Swap FP rounding mode)
fsflags rd, rs	csrrw rd, fflags, rs	สลับแฟล็กข้อยกเว้นทศนิยม (Swap FP exception flags)

รูปที่ 3.4: คำสั่งเทียม 28 คำสั่ง ของ RISC-V ที่ไม่ขึ้นกับรีจิสเตอร์ x0 ,รีจิสเตอร์ศูนย์ สำหรับ la, GOT ย่อมาจาก Global Offset Table ซึ่งเก็บที่อยู่รันไทม์ของสัญลักษณ์ในไลบรารี ที่เชื่อมโยงแบบไดนามิก ภาคผนวก A รวมถึงคำสั่งเทียม RISC-V เช่นเดียวกับคำสั่งปกติ (ตารางที่ 20.2 และ 20.3 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

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

รูปที่ 3.5: โปรแกรม Hello World ภาษาซี (hello.c)

<pre>.text .align 2 .globl main main: addi sp, sp, -16 sw ra, 12(sp) lui a0, %hi(string1) addi a0, a0, %lo(string1) lui a1, %hi(string2) addi a1, a1, %lo(string2) call printf lw ra, 12(sp) addi sp, sp, 16 li a0, 0 ret .section .rodata .balign 4 string1: .string "Hello, %s\n" string2: .string "world"</pre>	<pre># ตัวบ่งชี้: บ้อนรหัสเซกเมนต์ # ตัวบ่งชี้: กัด 2^2 รหัสการจัดการตำแหน่งไบต์ # ตัวบ่งชี้: ประกาศสัญลักษณ์หลักทั่วไป # ตำแหน่งเริ่มฟังก์ชันหลัก # จัดสรรสแต็กเฟรม # บันทึกค่าที่อยู่ # จำนวนที่อยู่ของ # string1 # จำนวนที่อยู่ของ # string2 # เรียกฟังก์ชันพิมพ์ call function printf # คืนค่าตำแหน่งที่อยู่ restore return address # จัดสรรสแต็กเฟรม deallocate stack frame # คืนค่า 0 load return value 0 # คืนค่า return # ตัวบ่งชี้: การป้อนเช็คเมนต์ข้อมูลแบบอ่านอย่างเดียว # ตัวบ่งชี้: กัด 4 ข้อมูลการจัดการตำแหน่งไบต์ # ตำแหน่งสำหรับตัวอักขระแรก label for first string # ตัวบ่งชี้: สตริงที่สิ้นสุดด้วยค่า null # ตำแหน่งสำหรับตัวอักขระตัวที่สอง label for second string # ตัวบ่งชี้: สตริงที่สิ้นสุดด้วยค่า null</pre>
--	---

รูปที่ 3.6: โปรแกรม Hello World ภาษาแอสเซมบลีใน RISC-V (hello.s)

รีจิสเตอร์ 1 ตัว ให้มีค่าเป็นศูนย์ตลอดเวลาทำให้ชุดคำสั่ง RISC-V ง่ายขึ้นมาก ทำให้สามารถสร้างคำสั่งยอदनิยมได้ง่าย :—เช่น คำสั่งกระโดด (jump), การคืนค่า (return) และคำสั่งกระโดดเมื่อเท่ากับศูนย์ (branch on equal to zero)—เป็นคำสั่งเทียม

รูปที่ 3.5 แสดงโปรแกรมคลาสสิก “Hello world” ในภาษาซี คอมไพเลอร์สร้างเอาต์พุตเป็นภาษาแอสเซมบลี ในรูปที่ 3.6 โดยใช้ข้อตกลงการเรียกฟังก์ชัน ในรูปที่ 3.2 และคำสั่งเทียม จากรูปที่ 3.3 และ 3.4

คำสั่งที่ขึ้นต้นด้วยเครื่องหมายจุด (.) เรียกว่า *คำสั่งกำกับของแอสเซมเบลอร์* (assembler directives) คำสั่งเหล่านี้ไม่ใช่โค้ดที่ต้องแปลเป็นภาษาเครื่อง แต่เป็นคำสั่งที่ใช้สั่งงานตัวแอสเซมเบลอร์โดยตรง คำสั่งเหล่านี้ใช้บอกตำแหน่งในการวางโค้ดและข้อมูลในหน่วยความจำ รวมถึงกำหนดค่าคงที่ (constants) สำหรับใช้ในโปรแกรม และจัดโครงสร้างของโปรแกรม

- `.text`—เข้าสู่ส่วนของโค้ด (text section)
- `.align 2`—จัดแนวโค้ดถัดไปให้ตรงกับขนาด 2^2 ไบต์ (4 ไบต์)

```

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

```

รูปที่ 3.7: โปรแกรม Hello World ในระดับภาษาเครื่องใน RISC-V (hello.o) คำสั่งจำนวน 6 คำสั่ง (อยู่ในช่วงตำแหน่งหน่วยความจำ 8 ถึง 1c) จะถูกรันทดสอบโดยลิงเกอร์ โดยในขั้นตอนนี้ ฟิลด์ที่เป็นตำแหน่งที่อยู่ของคำสั่งเหล่านั้นยังถูกใส่เป็นค่าศูนย์ (0) ไว้ก่อน แล้วใช้ตารางสัญลักษณ์ (symbol table) ภายในไฟล์ object บันทึกชื่อเลเบล (เช่น loop, main) และตำแหน่งที่อยู่ของคำสั่งทั้งหมดที่ต้องการแก้ไขโดยลิงเกอร์

```

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

```

รูปที่ 3.8: โปรแกรม Hello World ภาษาเครื่องใน RISC-V หลังการเชื่อมโยง ในระบบ Unix ไฟล์จะมีชื่อว่า a.out

คำสั่ง (Directive)	คำอธิบาย
<code>.text</code>	รายการต่อมาจะถูกเก็บไว้ในส่วนข้อความ (รหัสเครื่อง)
<code>.data</code>	รายการต่อมาจะถูกเก็บไว้ในส่วนข้อมูล (ตัวแปรทั่วไป)
<code>.bss</code>	รายการต่อมาจะถูกเก็บไว้ในส่วน bss (ตัวแปรทั่วไปที่ถูกกำหนดค่าเป็น 0)
<code>.section .foo</code>	รายการต่อมาจะถูกเก็บไว้ในส่วนที่ชื่อ <code>.foo</code>
<code>.align n</code>	จัดเรียงข้อมูลถัดไปให้ตรงกับขอบเขต 2^n ไบต์ เช่น <code>.align 2</code> จัดเรียงให้ตรงกับขอบเขต word (4 ไบต์)
<code>.balign n</code>	จัดเรียงข้อมูลถัดไปให้ตรงกับขอบเขต n ไบต์ เช่น <code>.balign 4</code> จัดเรียงให้ตรงกับขอบเขต word (4 ไบต์)
<code>.globl sym</code>	ประกาศว่าป้ายชื่อ <code>sym</code> เป็นสัญลักษณ์สากลและสามารถอ้างอิงจากไฟล์อื่นได้
<code>.string "str"</code>	เก็บสตริง <code>str</code> ในหน่วยความจำและสิ้นสุดด้วย <code>null</code>
<code>.byte b1,..., bn</code>	เก็บปริมาณ n ค่าขนาด 8 บิตในไบต์ต่อเนื่องของหน่วยความจำ
<code>.half w1,..., wn</code>	เก็บปริมาณ n ค่าขนาด 16 บิตใน halfword ต่อเนื่องของหน่วยความจำ
<code>.word w1,..., wn</code>	เก็บปริมาณ n ค่าขนาด 32 บิตใน word ต่อเนื่องของหน่วยความจำ
<code>.dword w1,..., wn</code>	เก็บปริมาณ n ค่าขนาด 64 บิตใน doubleword ต่อเนื่องของหน่วยความจำ
<code>.float f1,..., fn</code>	เก็บตัวเลขทศนิยมความแม่นยำเดี่ยว n ค่าใน word ต่อเนื่องของหน่วยความจำ
<code>.double d1,..., dn</code>	เก็บตัวเลขทศนิยมความแม่นยำคู่ n ค่าใน doubleword ต่อเนื่องของหน่วยความจำ
<code>.option rvc</code>	บีบอัดคำสั่งถัดไป (คูบทที่เกี่ยวกับ 7)
<code>.option norvc</code>	ไม่บีบอัดคำสั่งถัดไป
<code>.option relax</code>	อนุญาตให้ลิงเกอร์ทำการผ่อนคลาย (relaxations) สำหรับคำสั่งถัดไป
<code>.option norelax</code>	ไม่อนุญาตให้ลิงเกอร์ทำการผ่อนคลายสำหรับคำสั่งถัดไป
<code>.option pic</code>	คำสั่งถัดไปเป็นรหัสที่ไม่ขึ้นกับตำแหน่ง (position-independent code)
<code>.option nopic</code>	คำสั่งถัดไปเป็นรหัสที่ขึ้นกับตำแหน่ง (position-dependent code)
<code>.option push</code>	บันทึกการตั้งค่าปัจจุบันของ <code>.option</code> ทั้งหมดลงในสแต็ก เพื่อให้ <code>.option pop</code> ถัดไปสามารถกู้คืนค่าได้
<code>.option pop</code>	กู้คืนการตั้งค่าของ <code>.option</code> ทั้งหมดจากสแต็กตามเวลาที่ทำ <code>.option push</code> ล่าสุด

รูปที่ 3.9: คำสั่งแอสเซมเบลอร์ทั่วไปของ RISC-V

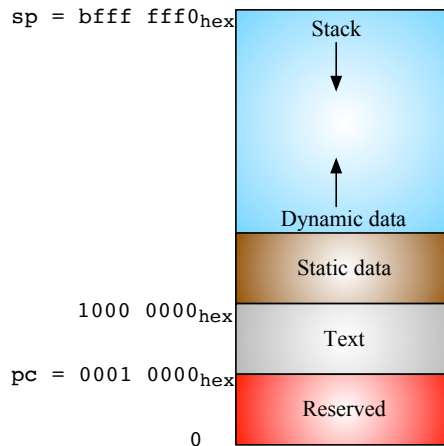
- `.globl main`—ประกาศให้ “main” เป็นสัญลักษณ์แบบโกลบอล (เรียกใช้ข้ามไฟล์ได้)
- `.section .rodata`—เข้าส่วนข้อมูลแบบอ่านอย่างเดียว (read-only data)
- `.balign 4`—จัดแนวข้อมูลให้ตรงกับ 4 ไบต์
- `.string "Hello, %s!\n"`—สร้างสตริง “Hello” ที่ลงท้ายด้วย `null (\0)`
- `.string "world"`—สร้างสตริง “world” ที่ลงท้ายด้วย `null (\0)`

แอสเซมเบลอร์สร้างไฟล์อ็อบเจกต์ แสดงในรูปที่ 3.7 โดยใช้รูปแบบมาตรฐาน ELF (Executable and Linkable Format) [TIS Committee 1995]

3.4 ตัวเชื่อมโยง (Linker)

แทนที่จะต้องคอมไพล์ซอร์สโค้ดทั้งหมดใหม่ทุกครั้งที่มีไฟล์ใดไฟล์หนึ่งถูกแก้ไข ตัวเชื่อมโยง (linker) ช่วยให้เราสามารถคอมไพล์และแอสเซมบลีแต่ละไฟล์แยกกันได้ จากนั้นมันจะ “เย็บ

โปรแกรม “Hello world” มักจะเป็นโปรแกรมแรกที่ถูกนำมาทดสอบบนหน่วยประมวลผลที่ออกแบบใหม่ โดยทั่วไปผู้ออกแบบสถาปัตยกรรมจะถือว่าการที่ระบบปฏิบัติการสามารถทำงานได้ดีพอที่จะแสดงข้อความ “Hello world” เป็นสัญญาณสำคัญว่า ซิปตัวใหม่ทำงานได้โดยรวมแล้วพวกเขาจะส่งผลลัพธ์นี้ทางอีเมลให้ผู้บริหาร และเพื่อนร่วมงาน จากนั้นก็ร่วมเฉลิมฉลอง



รูปที่ 3.10: การจัดสรรหน่วยความจำของโปรแกรมและข้อมูล RV32I ตำแหน่งที่อยู่หน่วยความจำที่มีค่าสูงจะอยู่ด้านบนของรูป และตำแหน่งที่มีค่าต่ำจะอยู่ด้านล่าง ตามข้อตกลงซอฟต์แวร์ของ RISC-V นี้ ตัวชี้สแต็ก (stack pointer) (sp) จะเริ่มต้นที่ `bfff fff0hex` และจะขยายลงด้านล่างไปยังส่วนของข้อมูลแบบคงที่ (Static data) ส่วน text (โค้ดโปรแกรม) จะเริ่มต้นที่ `0001 0000hex` และรวมถึงไลบรารีที่ถูกเชื่อมโยงแบบคงที่ (statically-linked libraries) ส่วนข้อมูลแบบคงที่ที่จะเริ่มต้นถัดขึ้นไปเหนือส่วน text โดยในตัวอย่างนี้กำหนดให้เริ่มที่ตำแหน่ง `1000 0000hex` ข้อมูลแบบไดนามิก ซึ่งจัดสรรในภาษา C โดยใช้ `malloc()` จะอยู่ถัดขึ้นไปเหนือข้อมูลแบบคงที่ ส่วนนี้เรียกว่า heap และจะเติบโตขึ้นด้านบนไปทางสแต็ก และส่วนนี้ยังรวมถึงไลบรารีที่เชื่อมโยงแบบไดนามิก (dynamically-linked libraries)

รวม” โค้ดอ็อบเจกต์ที่ได้ใหม่เข้ากับโมดูลภาษาเครื่องที่มีอยู่แล้ว เช่น ไลบรารี ชื่อ linker มาจากหน้าที่อย่างหนึ่งของมัน คือการแก้ไข (edit) การเชื่อมโยง (links) เช่น คำสั่ง คำสั่งกระโดด (jump) และคำสั่งกระโดดและเชื่อมโยง (jump and link) ภายในไฟล์ object จริงๆ แล้วคำว่า “linker” ย่อมาจาก “link editor” ซึ่งเป็นชื่อดั้งเดิมของขั้นตอนนี้ ในรูปที่ 3.1 ในระบบ Unix ข้อมูลนำเข้าของตัวเชื่อมโยงคือไฟล์ที่มีนามสกุล `.o` (เช่น `foo.o`, `libc.o`) และผลลัพธ์คือไฟล์ `a.out` สำหรับระบบ MS-DOS อินพุตของ linker คือไฟล์ที่มีนามสกุล `.OBJ` หรือ `.LIB` และเอาต์พุตคือไฟล์ `.EXE`

ภาพ 3.10 แสดงตำแหน่งที่อยู่ของส่วนต่างๆ ในหน่วยความจำที่ถูกจัดสรรสำหรับโค้ดและข้อมูลในโปรแกรม RISC-V ทั่วไป ตัวเชื่อมโยงจะต้องปรับที่อยู่ของคำสั่งและข้อมูลในไฟล์อ็อบเจกต์ทั้งหมด เพื่อให้ตรงกับตำแหน่งที่อยู่หน่วยความจำตามที่กำหนดไว้ในรูป ถ้าไฟล์นำเข้าเป็น โค้ดที่ไม่ขึ้นกับตำแหน่ง (PIC: position independent code) ตัวเชื่อมโยงจะทำงานง่ายขึ้นมาก (PIC) หมายความว่า การกระโดดไปยังคำสั่งต่าง ๆ (branches) และการอ้างอิงข้อมูล (data references) ภายในไฟล์นั้น ยังคงถูกต้องไม่ว่าจะนำโค้ดไปวางไว้ที่ตำแหน่งใดในหน่วยความจำก็ตาม ตามที่กล่าวไว้ในบทที่ 2 การกระโดดแบบอ้างอิง



ตำแหน่งสัมพันธ์กับตัวนับคำสั่ง (PC-relative branch) ในบท RV32I ช่วยให้การทำ PIC เป็นเรื่องง่ายขึ้นมาก

นอกจากชุดคำสั่งแล้ว ไฟล์ object แต่ละไฟล์ยังมีตารางสัญลักษณ์ (symbol table) ที่รวบรวมเลเบล (labels) ทั้งหมดในโปรแกรม ที่จำเป็นต้องถูกกำหนดตำแหน่งที่อยู่จริงในขั้นตอนของการเชื่อมโยง รายการนี้รวมถึงเลเบลที่อ้างอิงถึงข้อมูล และที่อ้างอิงถึงโค้ด รูปที่ 3.6 มีเลเบลของข้อมูล 2 ตัว ที่ต้องถูกกำหนดค่า คือ (string1 และ string2) และเลเบลของโค้ด 2 ตัว ที่ต้องถูกกำหนดค่า คือ (main และ printf) เนื่องจากที่อยู่แบบ 32 บิต ไม่สามารถใส่ลงในคำสั่ง 32 บิต ได้ภายในคำสั่งเดียว ดังนั้นตัวเชื่อมโยงจึงต้องปรับคำสั่ง 2 คำสั่ง ต่อ 1 เลเบล ในโค้ด RV32I ดังที่แสดงในรูปที่ 3.6: คำสั่ง lui และ addi เพื่อประกอบเป็นตำแหน่งที่อยู่ 32 บิต และใช้คำสั่ง auipc กับ jalr สำหรับคำสั่งกระโดดไปยังโค้ด รูปที่ 3.8 แสดงไฟล์ a.out ที่ได้หลังจากการลิงก์เสร็จสมบูรณ์ ในรูปที่ 3.7.

คอมไพเลอร์ของ RISC-V รองรับ ABI หลายรูปแบบ ขึ้นอยู่กับว่ามีส่วนขยาย F และ D หรือไม่ สำหรับสถาปัตยกรรม RV32 จะมี ABI อยู่ 3 แบบ ได้แก่ ilp32, ilp32f และ ilp32d สำหรับ ilp32 หมายถึงชนิดข้อมูลในภาษา C ได้แก่ int, long และ pointer มีขนาด 32 บิตทั้งหมด ส่วนตัวอักษรต่อท้าย (f หรือ d) ใช้ระบุวิธีการส่งค่าทศนิยม ใน ilp32 ค่าทศนิยมจะถูกส่งผ่านรีจิสเตอร์จำนวนเต็ม ใน ilp32f ค่าทศนิยมแบบความแม่นยำเดียว (single-precision) จะถูกส่งผ่านรีจิสเตอร์ทศนิยม ใน ilp32d ค่าทศนิยมแบบความแม่นยำคู่ (double-precision) ถูกส่งผ่านรีจิสเตอร์ทศนิยมเช่นกัน

โดยธรรมชาติแล้ว หากต้องการส่งค่าทศนิยมผ่านรีจิสเตอร์ทศนิยม จะต้องมีส่วนขยาย ISA แบบ F หรือ D ที่รองรับ (ดูบทที่ 5) ดังนั้น หากต้องการคอมไพล์โค้ดสำหรับ RV32I (GCC flag '-march=rv32i') จะต้องใช้ ABI แบบ ilp32 (GCC flag '-mabi=ilp32') ในทางกลับกัน ถึงแม้จะมีคำสั่งสำหรับคำนวณเลขทศนิยม ก็ไม่ได้หมายความว่าจำเป็นต้องใช้รีจิสเตอร์ทศนิยมเสมอไป ตัวอย่างเช่น RV32IFD สามารถใช้ร่วมกับ ABI ได้ทั้งสามแบบ ได้แก่ ilp32, ilp32f และ ilp32d

ตัวเชื่อมโยง (linker) จะตรวจสอบว่า ABI ของโปรแกรม ตรงกับ ABI ของ ไลบรารี ทั้งหมดหรือไม่ แม้ว่าคอมไพเลอร์จะรองรับหลายรูปแบบของ ISA และ ABI แต่ในเครื่องอาจมีไลบรารีติดตั้งอยู่เพียงบางชุดเท่านั้น ดังนั้น ปัญหาที่พบบ่อยคือพยายามลิงก์โปรแกรม โดยที่ไม่มีไลบรารีที่เข้ากันได้ ในกรณีนี้ ตัวเชื่อมโยงมักจะไม่แสดงข้อความแจ้งข้อผิดพลาดที่ชัดเจน แต่จะพยายามลิงก์กับไลบรารีที่ไม่เข้ากัน แล้วจึงแจ้งว่าไม่สามารถใช้งานร่วมกันได้ ปัญหานี้มักเกิดขึ้นเมื่อคอมไพล์บนเครื่องหนึ่งเพื่อไปใช้งานบนอีกเครื่องหนึ่ง (cross compiling)

■ รายละเอียดเพิ่มเติม: Linker relaxation

The jump and link instruction has a 20-bit PC-relative address field, so a single instruction can jump far. While the compiler produces two instructions for each external function, quite often only one instruction is necessary. Since this optimization saves both time and space, the linker will make passes over the code to replace two instructions with one whenever it can. Because a pass might shrink distance between a call and the function so that it now fits in a single instruction, the linker keeps optimizing the code until there are no further changes. This process is called *Linker relaxation*, with the name referring to relaxation techniques for solving systems of equations. คำสั่งกระโดดและเชื่อมโยง (jump and link) มีฟิลด์ที่อยู่สัมพันธ์กับ PC ขนาด 20 บิต ทำให้สามารถกระโดดได้ไกลด้วยคำสั่งเดียว แม้ว่าคอมไพเลอร์จะสร้างคำสั่ง 2 คำสั่งสำหรับเรียกฟังก์ชันภายนอก แต่ในหลายกรณีจริงๆ แล้วใช้คำสั่งเพียงคำสั่งเดียวเท่านั้น เนื่องจากการปรับให้เหมาะสมนี้ช่วยประหยัดทั้งเวลาและพื้นที่ ตัวเชื่อมโยงจะทำการตรวจสอบโค้ดหลายรอบเพื่อแทนที่คำสั่ง 2 คำสั่งด้วย 1 คำสั่ง เมื่อเป็นไปได้ เนื่องจากการปรับแต่ละครั้งอาจทำให้ระยะห่างระหว่างคำสั่งเรียกและฟังก์ชันสั้นลงจนสามารถใช้คำสั่งเดียวได้ ตัวเชื่อมโยงจะทำการปรับโค้ดที่ละรอบจนได้รูปแบบที่ดีที่สุด กระบวนการนี้เรียกว่า (*Linker relaxation*) โดยชื่อนี้มาจากเทคนิคการค่อย ๆ ปรับให้ดีขึ้นทีละรอบ (relaxation) ที่ใช้ในการแก้ระบบสมการ นอกจากการเรียกฟังก์ชันแล้ว ตัวเชื่อมโยงของ RISC-V ยังปรับการอ้างอิงข้อมูล โดยใช้ global pointer (gp) หากข้อมูลอยู่ในช่วง ± 2 KiB จาก gp ซึ่งจะช่วยลดการใช้คำสั่ง lui หรือ auipc ในทำนองเดียวกัน ยังสามารถปรับการอ้างอิง thread-local storage โดยใช้ tp หากข้อมูลอยู่ในช่วง ± 2 KiB จาก tp

3.5 การเชื่อมโยงแบบคงที่เทียบกับแบบไดนามิก

ส่วนก่อนหน้านี้ได้อธิบาย การเชื่อมโยงแบบคงที่ ซึ่งเป็นการนำโค้ดของไลบรารีทั้งหมดที่อาจถูกใช้งานมาลิงก์รวมกับโปรแกรม และโหลดเข้ามามีพร้อมกันก่อนเริ่มรัน ไลบรารีเหล่านี้อาจมีขนาดใหญ่ ดังนั้นการนำไลบรารีที่นิยมใช้ไปเชื่อมโยงกับหลายโปรแกรมจะทำให้สิ้นเปลืองหน่วยความจำ นอกจากนี้ ไลบรารีจะถูกผูกติด ตั้งแต่ตอนเชื่อมโยง—แม้ว่าจะมีการอัปเดตแก้ไขบั๊กในภายหลังก็ตาม —โปรแกรมที่ลิงก์แบบคงที่ก็จะยังคงใช้เวอร์ชันเก่าที่มีข้อผิดพลาดอยู่

เพื่อแก้ปัญหาทั้งสอง ระบบส่วนใหญ่จึงใช้การเชื่อมโยงแบบไดนามิก (dynamic linking) ฟังก์ชันจากไลบรารีภายนอกจะถูกโหลดและลิงก์ ก็ต่อเมื่อถูกเรียกใช้งานครั้งแรกเท่านั้น หากฟังก์ชันนั้นไม่เคยถูกเรียก ก็จะไม่ถูกโหลดและไม่ถูกลิงก์เลย การเรียกครั้งแรกจะใช้การเชื่อมโยงที่เร็วมาก ทำให้การดำเนินการเพิ่มเติมจะเกิดขึ้นแค่ครั้งเดียว โปรแกรมจะใช้เวอร์ชันล่าสุดของไลบรารีตอนเริ่มต้น ถ้ามีหลายโปรแกรมใช้ไลบรารีเดียวกัน โค้ดของไลบรารีจะอยู่ในหน่วยความจำเพียงชุดเดียว

โค้ดที่คอมไพเลอร์สร้างขึ้นจะคล้ายกับการลิงก์แบบคงที่ (static linking) แทนที่จะกระโดดไปยังฟังก์ชันจริงโดยตรง มันจะกระโดดไปยังฟังก์ชันสั้นๆ (stub function) ที่มีเพียงประมาณ 3 คำสั่ง โดยฟังก์ชัน stub จะโหลดที่อยู่ของฟังก์ชันจริงจากตารางในหน่วยความจำ แล้วกระโดดไปทำงานที่ฟังก์ชันนั้น อย่างไรก็ตาม ในการเรียกครั้งแรก ตารางนี้จะยังไม่มีที่อยู่ของฟังก์ชันจริง แต่จะเก็บที่อยู่ของตัวจัดการเชื่อมโยงแบบไดนามิกแทน เมื่อถูกเรียกใช้งาน การเชื่อมโยงแบบไดนามิก (dynamic linker) จะใช้ symbol table เพื่อค้นหาฟังก์ชันจริง คัดลอกมันเข้าไปในหน่วยความจำ และอัปเดตตารางให้ชี้ไปยังฟังก์ชันจริง หลังจากนั้น ทุกการเรียกฟังก์ชันจะมีค่าดำเนินการเพียงแค่การทำงาน (ประมาณ 3 คำ

สถาปนิกมักจะวัดประสิทธิภาพของโปรเซสเซอร์ โดยใช้การวัดประสิทธิภาพที่เชื่อมโยงแบบคงที่ แม้ว่าโปรแกรมจริงส่วนใหญ่จะใช้การเชื่อมโยงแบบไดนามิก ข้ออ้างคือผู้ใช้ที่สนใจด้านประสิทธิภาพควรใช้การลิงก์แบบคงที่ แต่นั่นเป็นเหตุผลที่ไม่ดีนัก มันสมเหตุสมผลกว่าที่จะเร่งประสิทธิภาพของโปรแกรมจริง ไม่ใช่การทำให้ผลทดสอบประสิทธิภาพ (benchmarks) ดีขึ้น

สั่ง) ของ stub เท่านั้น

3.6 โหลดเดอร์ (Loader)

โปรแกรมอย่าง que แสดงในรูป 3.8 เป็นไฟล์ที่สามารถรันได้ (executable) ที่ถูกเก็บอยู่ในหน่วยเก็บข้อมูลของคอมพิวเตอร์ เมื่อมีการสั่งให้รันโปรแกรมนั้น หน้าแรกของโหลดเดอร์ (loader) คือ โหลดโปรแกรมเข้าสู่หน่วยความจำ แล้วกระโดดไปยังที่อยู่หน่วยความจำเริ่มต้นของโปรแกรม ในระบบปัจจุบัน “โหลดเดอร์” ก็คือส่วนหนึ่งของระบบปฏิบัติการ กล่าวคือ การโหลดไฟล์ a.out เป็นหน้าที่หนึ่งของระบบปฏิบัติการ

การโหลดโปรแกรมที่ใช้การเชื่อมโยงแบบไดนามิกจะซับซ้อนมากขึ้นเล็กน้อย แทนที่จะเริ่มโปรแกรมโดยตรง ระบบปฏิบัติการจะเริ่มต้นเชื่อมโยงแบบไดนามิกก่อน จากนั้น จะเริ่มโปรแกรมที่ต้องการ และจัดการการเรียกฟังก์ชันภายนอกครั้งแรก โดยคัดลอกฟังก์ชันเข้าสู่หน่วยความจำ และปรับแก้โปรแกรมหลังจากการเรียกแต่ละครั้งให้ชี้ไปยังฟังก์ชันที่ถูกต้อง

3.7 ข้อสังเกตสรุป

ทำให้มันเรียบง่ายเข้าไว้ อย่าทำให้ซับซ้อนเกินจำเป็น

—เคลลี จอห์นสัน วิศวกรการบิน เป็นผู้ตั้งแนวคิดที่เรียกว่า “KISS Principle” (Keep It Simple, Stupid) 1960



แอสเซมเบลอร์ช่วยเสริมความสามารถของ ISA ที่เรียบง่ายของ RISC-V ด้วยคำสั่งเทียม (pseudoinstructions) จำนวน 60 คำสั่ง ซึ่งทำให้โค้ด RISC-V อ่านและเขียนง่ายขึ้น โดยไม่เพิ่มต้นทุนฮาร์ดแวร์ เพียงแค่กำหนดให้รีจิสเตอร์ตัวหนึ่งของ RISC-V ให้มีค่าเป็นศูนย์ (x0) ก็สามารถสร้างคำสั่งที่มีประโยชน์เหล่านี้ได้จำนวนมาก คำสั่ง (lui) และคำสั่ง (auipc) ช่วยให้คอมไพเลอร์และตัวเชื่อมโยงปรับตำแหน่งที่อยู่ของข้อมูล และฟังก์ชันภายนอกได้ง่ายขึ้น การกระโดดสัมพันธ์กับ PC (PC-relative branching) ช่วยให้การเชื่อมโยงทำงานกับโค้ดแบบไม่ขึ้นกับตำแหน่ง (PIC) ได้ง่ายขึ้น การมีรีจิสเตอร์จำนวนมาก ช่วยให้สามารถออกแบบข้อตกลงการเรียกฟังก์ชัน (calling convention) ช่วยให้การเรียกและคืนค่าจากฟังก์ชันทำได้เร็วขึ้น โดยลดจำนวนครั้งของการย้ายข้อมูล (เก็บค่าลงหน่วยความจำและโหลดค่ากลับขึ้นมา) ระหว่างรีจิสเตอร์กับหน่วยความจำ (spills และ restores)

RISC-V นำเสนอชุดกลไกที่เรียบง่าย แต่ทรงประสิทธิภาพอย่างมีรสนิม ที่ช่วยลดต้นทุน เพิ่มประสิทธิภาพ และทำให้การเขียนโปรแกรมง่ายขึ้น

3.8 เรียนรู้เพิ่มเติม

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

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

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

4

RV32M: การคูณและการหาร

วิลเลียมแห่งอ็อกแคม
(1287–1347) นักเทววิทยา
ชาวอังกฤษ ผู้สนับสนุน
แนวคิดที่ปัจจุบันเรียก
ว่า “มีดโกนของอ็อกคัม
(Occam's razor)” ซึ่งเป็น
หลักการเลือกความเรียบง่าย
เป็นหลักในการอธิบายหรือ
ออกแบบทางวิทยาศาสตร์



ไม่ควรเพิ่มสมมติฐานหรือความซับซ้อนโดยไม่จำเป็น

— วิลเลียมแห่งอ็อกแคม, 1320

4.1 บทนำ

RV32M เป็นส่วนขยายของ RISC-V ที่เพิ่มคำสั่งสำหรับการคูณ (multiply) และหาร (divide) ให้กับ RV32I รูปที่ 4.1 เป็นการแสดงภาพรวมของชุดคำสั่ง RV32M และรูปที่ 4.2 แสดงรายการรหัสดำเนินการ (opcodes) ของคำสั่งเหล่านี้ การหารเป็นเรื่องที่เข้าใจได้ไม่ยาก ให้จำว่า:

$$\text{ผลลัพธ์} = (\text{ตัวตั้ง} - \text{เศษ}) \div \text{ตัวหาร}$$

หรืออีกแบบหนึ่ง

$$\text{ตัวตั้ง} = \text{ผลลัพธ์} \times \text{ตัวหาร} + \text{เศษ}$$

$$\text{เศษ} = \text{ตัวตั้ง} - (\text{ผลลัพธ์} \times \text{ตัวหาร})$$

ใน RV32M มีคำสั่งสำหรับทั้งจำนวนเต็ม แบบมีเครื่องหมาย (signed) และไม่มีเครื่องหมาย (unsigned) ได้แก่คำสั่ง การหารแบบมีเครื่องหมาย (div) และ คำสั่งการหาร

RV32M

multiply

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

$\left\{ \begin{array}{l} \text{divide} \\ \text{remainder} \end{array} \right\} \left\{ \begin{array}{l} \text{—} \\ \text{unsigned} \end{array} \right\}$

รูปที่ 4.1: แผนภาพของคำสั่ง 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

รูปที่ 4.2: แผนผังรหัสดำเนินการของ RV32M แสดงรายละเอียดของโครงสร้างคำสั่ง รหัสดำเนินการ ประเภทของรูปแบบคำสั่ง และชื่อคำสั่ง (ตารางที่ 19.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

แบบไม่มีเครื่องหมาย (divu) โดยผลลัพธ์ (Quotient) จะถูกเก็บในรีจิสเตอร์ปลายทาง ในบางกรณีที่โปรแกรมเมอร์ต้องการเศษ (remainder) แทน RV32M จึงมีคำสั่ง เศษแบบมีเครื่องหมาย (rem) และ คำสั่งเศษแบบไม่มีเครื่องหมาย (remu) ซึ่งจะเขียนค่าเศษ ลงในรีจิสเตอร์แทนผลลัพธ์

สมการการคูณ (multiply) คือ:

$$\text{ผลคูณ} = \text{ตัวถูกคูณ} \times \text{ตัวคูณ}$$

การคูณซับซ้อนกว่าการหาร เพราะขนาดของผลคูณ เท่ากับผลรวมของขนาดของตัวคูณและตัวถูกคูณ เช่น การคูณจำนวน 32 บิต สองจำนวน จะได้ผลคูณ 64 บิต เพื่อรองรับผลลัพธ์ 64 บิตอย่างถูกต้อง RISC-V มีคำสั่งคูณ 4 แบบ —คำสั่ง mul เพื่อให้ได้ผลคูณของจำนวนเต็ม 32 บิต ซึ่งเป็น 32 บิตล่างของผลคูณเต็ม—คำสั่ง mulh เพื่อให้ได้ 32 บิตบนของผลคูณ 64 บิต เมื่อตัวคูณทั้งสองมีเครื่องหมาย (signed) —คำสั่ง mulhu เมื่อถ้าตัวคูณทั้งสองไม่มีเครื่องหมาย และคำสั่ง mulhsu เมื่อตัวคูณหนึ่งมีเครื่องหมายและอีกตัวไม่มีเครื่องหมาย เนื่องจากการเขียนผลคูณ 64 บิตลงในรีจิสเตอร์ 32 บิตสองตัวในคำสั่งเดียว จะทำให้ฮาร์ดแวร์ซับซ้อนขึ้น ดังนั้น RV32M จึงใช้สองคำสั่งเพื่อให้ได้ผลคูณ 64 บิต

คำสั่ง srl สามารถใช้สำหรับการคูณแบบไม่มีเครื่องหมายด้วย 2^i ตัวอย่างเช่น หาก $a2 = 16 (2^4)$ คำสั่ง `srl t2, a1, 4` จะให้ผลลัพธ์ คำสั่ง `sll` สามารถทำการคูณทั้งแบบมีเครื่องหมายและแบบไม่มีเครื่องหมายด้วยค่า 2^i ได้ ตัวอย่างเช่น ถ้า $a2 = 16 (2^4)$ คำสั่ง `slli t2, a1, 4` จะให้ผลลัพธ์เหมือนกับ `mul t2, a1, a2`

ในไมโครโปรเซสเซอร์จำนวนมาก การหารจำนวนเต็ม (integer division) เป็นการดำเนินการที่ค่อนข้างช้า ดังที่กล่าวไว้ก่อนหน้านี้ การเลื่อนบิตไปทางขวา (right shift) สามารถใช้แทนการหารแบบไม่มีเครื่องหมาย (unsigned division) ด้วยค่าอยู่ในรูปกำลังของ 2 ได้ นอกจากนี้ การหารด้วยค่าคงที่อื่นๆ ก็สามารถปรับให้เร็วขึ้นได้เช่นกัน โดยใช้วิธีการคูณด้วยค่า “ส่วนกลับโดยประมาณ” (approximate reciprocal) จากนั้นทำการปรับแก้ (correction) กับครึ่งบนของผลคูณ ตัวอย่างเช่น รูปที่ 4.3 แสดงโค้ดสำหรับการหารแบบไม่มีเครื่องหมายด้วย 3

อะไรที่ต่างกัน? ARM-32 มีคำสั่งคูณมานานแล้วแต่ไม่มีคำสั่งหาร โดยคำสั่งหารเพิ่งถูกกำหนดให้เป็นข้อบังคับ (mandatory) ในปี 2005 ซึ่งช้ากว่าการเปิดตัวโปรเซสเซอร์ ARM รุ่นแรกเกือบ 20 ปี ใน MIPS-32 ใช้รีจิสเตอร์พิเศษชื่อ (HI และ LO) เป็นรีจิสเตอร์ปลายทาง



สำหรับโปรเซสเซอร์เกือบทั้งหมด การคูณจะช้ากว่าการเลื่อนบิต (shifts) หรือการบวก (adds) และการหารจะช้ากว่าการคูณมาก


```
# การคำนวณการหารแบบไม่มีเครื่องหมายของ a0 ด้วย 3 โดยใช้การคูณ
0: aaaab2b7      lui    t0, 0xaaaab # t0 = 0xaaaaaaab
4: aab28293      addi   t0, t0, -1365 #    = ~ 2^32 / 1.5
8: 025535b3      mulhu  a1, a0, t0    # a1 = ~ (a0 / 1.5)
c: 0015d593      srli   a1, a1, 0x1    # a1 = (a0 / 3)
```

รูปที่ 4.3: โค้ด RV32M สำหรับการหารด้วยค่าคงที่โดยใช้การคูณ ต้องอาศัยการวิเคราะห์เชิงตัวเลขอย่างรอบคอบ เพื่อพิสูจน์ว่าอัลกอริทึมนี้สามารถใช้ได้กับตัวตั้ง (dividend) ทุกค่า และสำหรับตัวหารบางค่า ขั้นตอนการแก้ไข (correction step) อาจมีความซับซ้อนมากขึ้น รายละเอียดของการพิสูจน์ความถูกต้อง รวมถึงอัลกอริทึมสำหรับสร้างค่าส่วนกลับโดยประมาณ (reciprocal) และขั้นตอนการแก้ไข สามารถดูได้ใน [Granlund and Montgomery 1994].

เพียงที่เดียวสำหรับคำสั่งคูณและหาร แม้ว่าการออกแบบนี้ จะลดความซับซ้อนของการนำไปใช้งาน MIPS ในยุคแรก แต่ก็มียกเว้นคือ ต้องใช้คำสั่งย้าย (move) เพิ่มอีกหนึ่งคำสั่งเพื่อนำผลลัพธ์ไปใช้งาน ซึ่งทำให้ประสิทธิภาพลดลงได้ นอกจากนี้ รีจิสเตอร์ HI และ LO ยังเพิ่มสถานะของสถาปัตยกรรม ส่งผลให้การสลับระหว่างงานช้าลงเล็กน้อย

■ **รายละเอียดเพิ่มเติม:** คำสั่ง `mulh` และ `mulhu` สามารถใช้ตรวจสอบการล้น (overflow) ในการคูณได้

จะไม่เกิดการล้น (overflow) ในการคูณแบบไม่มีเครื่องหมาย ด้วยคำสั่ง `mul` ถ้าผลลัพธ์ของคำสั่ง `mulhu` มีค่าเป็นศูนย์ ในทำนองเดียวกัน จะไม่เกิดการล้นในการคูณแบบมีเครื่องหมายเมื่อใช้คำสั่ง `mul` ถ้าบิตทั้งหมดในผลลัพธ์ของคำสั่ง `mulh` ตรงกับบิตเครื่องหมายของผลลัพธ์ของคำสั่ง `mul` กล่าวคือ ถ้าเป็นจำนวนบวกต้องได้ค่า 0 ถ้าเป็นจำนวนลบ ต้องได้ค่า `ffff`_{hex}

■ **รายละเอียดเพิ่มเติม:** การตรวจสอบว่า หารด้วยศูนย์ ทำได้ง่าย

เพียงเพิ่มคำสั่ง `beqz` เพื่อตรวจสอบตัวหารก่อนทำการหาร RV32I ไม่ได้ทำให้เกิดข้อบกพร่อง (trap) เมื่อหารด้วยศูนย์ โปรแกรมส่วนใหญ่ไม่ต้องการพฤติกรรมนี้ และโปรแกรมที่ต้องการสามารถตรวจสอบค่าศูนย์ได้เองในซอฟต์แวร์แน่นอนว่า การหารด้วยค่าคงที่ที่ไม่จำเป็นต้องตรวจสอบ

■ **รายละเอียดเพิ่มเติม:** คำสั่ง `mulhsu` มีประโยชน์สำหรับการคูณแบบมีเครื่องหมายมัลติเวิร์ด

คำสั่ง `Tmulhsu` สร้างครึ่งบนของผลคูณ (upper half) เมื่อตัวคูณมีเครื่องหมาย และตัวถูกคูณไม่มีเครื่องหมาย ขั้นตอนนี้นักใช้เป็นขั้นตอนย่อยของการคูณเลขมัลติเวิร์ด (multi-word) แบบมีเครื่องหมาย โดยเฉพาะเมื่อคูณเวิร์ด (word) ที่มีนัยสำคัญสูงสุดของตัวคูณ (ที่มีบิตเครื่องหมาย) กับเวิร์ด ที่มีนัยสำคัญต่ำกว่าของตัวคูณ (ที่ไม่มีเครื่องหมาย) คำสั่งนี้ช่วยเพิ่มประสิทธิภาพของการคูณได้ประมาณ 15%

4.2 ข้อสังเกตสรุป

ส่วนประกอบที่ถูกที่สุด รวดเร็วที่สุด และน่าเชื่อถือที่สุด คือส่วนประกอบที่ไม่ต้องมีเลย

—ซี. กอร์ดอน แบล สถาปนิกของมินิคอมพิวเตอร์ที่มีชื่อเสียง



ค่าใช้จ่าย

เพื่อให้ได้โปรเซสเซอร์ RISC-V ที่เล็กที่สุดสำหรับแอปพลิเคชันระบบฝังตัว คำสั่งคูณและหารจึงถูกจัดเป็นส่วนขยายแบบเลือกใช้ตัวแรกของ RISC-V อย่างไรก็ตาม โปรเซสเซอร์

RISC-V จำนวนมากก็ยังคงเลือกที่จะรองรับ RV32M

4.3 เรียนรู้เพิ่มเติม

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/>.

Zfh, RV32F และ RV32D: จำนวน ลอยตัว แบบ ครึ่ง ความแม่นยำ, ความ แม่นยำเดี่ยวและความแม่นยำสองเท่า

องตวน เดอ แซงเด็กูเปรี (1900–1944) เป็นนักเขียนและนักบินชาวฝรั่งเศส เป็นที่รู้จักจากหนังสือ เจ้าชายน้อย (The Little Prince)



การบรรลุถึงความสมบูรณ์แบบนั้น ไม่ใช่เมื่อไม่มีอะไรเพิ่มเติมอีก หากต่อเมื่อไม่มีอะไรให้ตัดออกไปได้อีกแล้ว

—องตวน เดอ แซงเด็กูเปรี, แดร์ เด ฮอมเมส, 1939

5.1 บทนำ

แม้ว่า RV32F และ RV32D จะเป็นส่วนขยายชุดคำสั่งที่แยกจากกัน และเป็นแบบเลือกใช้ (optional) แต่ในทางปฏิบัติมักถูกนำมาใช้ร่วมกัน ราบใดที่โปรเซสเซอร์รองรับอย่างน้อย RV32F ก็สามารถรองรับ Zfh ได้เช่นกัน เนื่องจากคำสั่งจำนวนลอยตัวส่วนใหญ่มีเวอร์ชันแบบครึ่งความแม่นยำ (half-precision, 16 บิต), ความแม่นยำเดี่ยว (single-precision, 32 บิต), และความแม่นยำสองเท่า (double-precision, 64 บิต) เพื่อความกระชับ เราจะนำเสนอคำสั่งเหล่านี้รวมไว้ในบทเดียว รูป 5.1 เป็นการแสดงภาพรวมของชุดคำสั่งส่วนขยาย Zfh, RV32F และ RV32D ในตอนท้ายของบท รูปที่ 5.13 แสดงรายการรหัสดำเนินการของ Zfh รูปที่ 5.11 แสดงรายการรหัสดำเนินการของ RV32F, และรูปที่ 5.12 แสดงรายการรหัสดำเนินการของ RV32D เช่นเดียวกับสถาปัตยกรรมชุดคำสั่งสมัยใหม่เกือบทั้งหมด RISC-V ปฏิบัติตามมาตรฐานจำนวนลอยตัว IEEE 754-2019 [IEEE Standards Committee 2019]

5.2 รีจิสเตอร์จำนวนลอยตัว

Zfh, RV32F และ RV32D RISC-V ใช้รีจิสเตอร์ชุด ϵ จำนวน 32 ตัว แยกจากรีจิสเตอร์ x เหตุผลหลักของการมีรีจิสเตอร์สองชุดคือ โปรเซสเซอร์สามารถเพิ่มประสิทธิภาพได้โดยการเพิ่มความจุ (capacity) และแบนด์วิดท์ของรีจิสเตอร์เป็นสองเท่า โดยไม่ต้องเพิ่มขนาดของช่องระบุนรีจิสเตอร์ ในรูปแบบคำสั่ง RISC-V ที่มีพื้นที่จำกัด ผลกระทบต่อชุดคำสั่งคือ การมีรีจิสเตอร์ f ทำให้ต้องมีคำสั่งโหลดและจัดเก็บสำหรับรีจิสเตอร์ ϵ รวมถึงคำสั่งย้ายข้อมูลระหว่างรีจิสเตอร์ x และรีจิสเตอร์ ϵ รูปที่ 5.2 แสดงรายชื่อรีจิสเตอร์ของ Zfh, RV32D และ RV32F พร้อมชื่อเรียกตามมาตรฐาน ABI ของ RISC-V

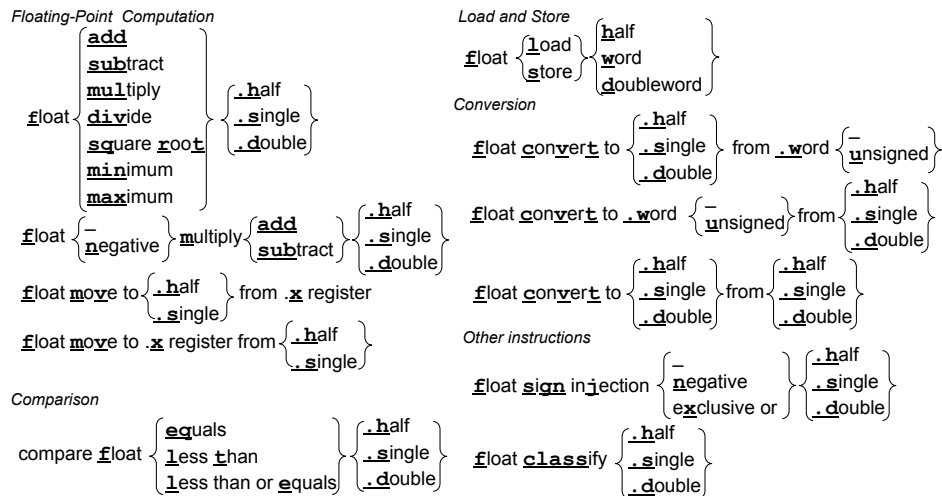
หากโปรเซสเซอร์มีทั้ง RV32F และ RV32D ข้อมูลจำนวนลอยตัวความแม่นยำเดี่ยว (32

แบบครึ่งความแม่นยำใช้ชื่อว่า Zfh เนื่องจากถูกเพิ่มเข้ามาภายหลัง หลังจากที่ RISC-V เริ่มใช้รูปแบบการตั้งชื่อแบบ Z



ประสิทธิภาพ

Zfh, RV32F, and RV32D



รูปที่ 5.1: แผนภาพของคำสั่ง Zfh, RV32F และ RV32D

บิต) ใช้เฉพาะ 32 บิตล่าง ของรีจิสเตอร์ f ในทำนองเดียวกัน จำนวนลอยตัวแบบครึ่งความแม่นยำ (16 บิต) ใช้เฉพาะ 16 บิตล่างของรีจิสเตอร์ f ต่างจาก $x0$ ใน RV32I รีจิสเตอร์ $f0$ ไม่ได้ถูกกำหนดค่าเป็น 0 คงที่ (hardwired) แต่สามารถเปลี่ยนแปลงค่าได้เหมือนรีจิสเตอร์ f อื่นๆ

มาตรฐาน IEEE 754-2019 กำหนดวิธีการปัดเศษจำนวนลอยตัวหลายแบบ ซึ่งมีประโยชน์ในการคำนวณขอบเขตข้อผิดพลาดและพัฒนาไลบรารีเชิงตัวเลข โหมดการปัดที่แม่นยำที่สุดและใช้กันทั่วไปคือการ ปัดไปยังค่าที่ใกล้ที่สุดแบบเลขคู่ (RNE: Round to Nearest Even) โหมดการปัดถูกกำหนดในรีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว `fcsr` รูปที่ 5.3 แสดงโครงสร้างของ `fcsr` รวมถึงตัวเลือกการปัดเศษ และเก็บแฟล็กข้อยกเว้นที่สะสมได้ตามที่มาตรฐานกำหนด

อะไรที่แตกต่าง? ทั้ง ARM-32 และ MIPS-32 มีรีจิสเตอร์จำนวนลอยตัวแบบความแม่นยำเดียว 32 ตัว แต่มีรีจิสเตอร์แบบความแม่นยำสองเท่าเพียง 16 ตัว ทั้งสองสถาปัตยกรรมใช้วิธีนำรีจิสเตอร์แบบ single-precision 2 ตัว มาแมปเป็น ครึ่งซ้าย 32 บิต และครึ่งขวา 32 บิต ของรีจิสเตอร์แบบความแม่นยำสองเท่า การคำนวณจำนวนลอยตัวในยุคแรกๆ ของ x86-32 ไม่มีรีจิสเตอร์ทั่วไป แต่ใช้ โครงสร้างแบบสแต็ก (stack) แทน โดยแต่ละ entry ในสแต็ก มีขนาด 80 บิต เพื่อเพิ่มความแม่นยำ ดังนั้นการโหลดจะแปลงตัวถูกดำเนินการขนาด 32 บิตหรือ 64 บิต เป็น 80 บิต และจะแปลงกลับจาก 80 บิตสำหรับการจัดเก็บ ต่อมา x86-32 ได้เพิ่มรีจิสเตอร์จำนวนลอยตัวแบบดั้งเดิม 64 บิต จำนวน 8 ตัว พร้อมคำสั่งที่เกี่ยวข้อง (หัวข้อ 8.12 ระบุว่า x86 ยังมีการพัฒนาคำสั่งจำนวนลอยตัวเพิ่มเติมอีกหลายครั้ง โดยแต่ละครั้งเพิ่มทั้งขนาดและจำนวนรีจิสเตอร์เป็นสองเท่า) ต่างจาก RV32FD และ MIPS-32 ทั้ง ARM-32 และ x86-32 ไม่มีคำสั่งสำหรับการย้ายข้อมูล

โดยตรง ระหว่างรีจิสเตอร์จำนวนลอยตัวและรีจิสเตอร์จำนวนเต็ม วิธีเดียวที่ทำได้คือ เก็บ
ค่าจากรีจิสเตอร์จำนวนลอยตัวลงหน่วยความจำ แล้วโหลดค่าจากหน่วยความจำไปยังรีจิส
เตอร์จำนวนเต็ม (และทำกลับกันในกรณีตรงข้าม)

■ รายละเอียดเพิ่มเติม: RV32FD อนุญาตให้กำหนดโหมดการปัดเศษสำหรับแต่ละคำสั่งได้

เรียกว่า การปัดเศษแบบคงที่ (static rounding) เพิ่มประสิทธิภาพเมื่อต้องการเปลี่ยนโหมดการปัดเศษสำหรับ
คำสั่งเดียวเท่านั้น โดยค่าเริ่มต้นจะใช้โหมดการปัดแบบไดนามิกในคำสั่ง `fcsr` การปัดแบบคงที่ ที่ถูกระบุ
เป็นอาร์กิวเมนต์สุดท้ายที่เป็นทางเลือก เช่น `fadd.s ft0, ft1, ft2, rtz` จะปัดเข้าหาศูนย์ โดยไม่
ขึ้นกับคำสั่ง `fcsr` คำบรรยายของรูปที่ 5.3 แสดงชื่อของโหมดการปัดเศษ

การถ่ายโอนข้อมูลแบบ
ความแม่นยำเดียวใช้ตัว
อักษร “w” แทน “s”
เนื่องจากข้อมูลชนิดนี้ เป็น
ข้อมูลขนาด 32 บิตเท่านั้น

5.3 การโหลด การจัดเก็บ และการคำนวณจำนวนลอยตัว

RISC-V มีคำสั่งโหลด 3 คำสั่ง ได้แก่ คำสั่ง (`f1h`, `f1w`, `f1d`) และคำสั่งจัดเก็บ 3 คำสั่ง
ได้แก่ คำสั่ง (`fsh`, `fsw`, `fsd`) สำหรับส่วนขยาย `Zfh`, `RV32F` และ `RV32D` คำสั่งเหล่านี้
ใช้โหมดการอ้างอิงที่อยู่ และรูปแบบคำสั่งเหมือนกับคำสั่ง `lw` และ `sw`

นอกเหนือจากการดำเนินการทางคณิตศาสตร์พื้นฐานทั้งสี่ ได้แก่: บวก (`fadd.h`,
`fadd.s`, `fadd.d`), ลบ (`fsub.h`, `fsub.s`, `fsub.d`), คูณ (`fmul.h`, `fmul.s`,
`fmul.d`), หาร (`fdiv.h`, หาร `fdiv.s`, `fdiv.d`) แล้ว ส่วนขยาย `Zfh`, `RV32F` และ
`RV32D` ยังมีคำสั่งการหาค่ารากที่สอง (square root) ได้แก่: (`fsqrt.h`, `fsqrt.s`,
`fsqrt.d`) นอกจากนี้ยังมีคำสั่ง: ค่าต่ำสุด (`fmin.h`, `fmin.s`, `fmin.d` และค่าสูงสุด
`fmax.h`, `fmax.s`, `fmax.d`) ซึ่งจะเขียนค่าที่น้อยกว่าหรือมากกว่า จากตัวถูกดำเนินการ
(operands) สองตัว โดยไม่ต้องใช้คำสั่งกระโดดแบบมีเงื่อนไข (branch)

อัลกอริทึมจำนวนลอยตัวจำนวนมาก เช่น การคูณเมทริกซ์ มักมีรูปแบบ การคูณแล้ว
บวกหรือการลบทันที ดังนั้น RISC-V จึงมีคำสั่งที่ คูณตัวถูกดำเนินการสองค่าแล้วนำผล
ไปบวก กับค่าอีกค่า (`fmadd.h`, `fmadd.s`, `fmadd.d`) แล้วลบ (`fmsub.h`, `fmsub.s`,
`fmsub.d`) ตัวถูกดำเนินการตัวที่สามกับผลคูณนั้น ก่อนจะเขียนผลลัพธ์ นอกจากนี้ยังมี
คำสั่งที่ กลับเครื่องหมายของผลคูณ (negate the product) แล้วจึงบวกหรือลบกับตัวถูก
ดำเนินการตัวที่สาม: `fnmadd.h`, `fnmadd.s`, `fnmadd.d`, `fnmsub.h`, `fnmsub.s`,
`fnmsub.d` เหตุผลที่ใช้คำสั่งแบบ คูณแล้วบวกแบบผสม (fused) เหล่านี้ถูกกำหนดใน
มาตรฐาน IEEE 754-2019 เพราะให้ความแม่นยำที่สูงกว่า คำสั่งคูณแล้วบวกแบบผสม
ปัดเศษครั้งเดียว (หลังการบวก) โดยปกติจะ ปัดเศษสองครั้ง (ปัดเศษหลังการคูณแล้ว
ปัดเศษอีกครั้งหลังการบวก) การข้ามการปัดเศษระหว่างขั้นตอนมีผล มากเมื่อ ค่าที่คูณและ
ค่าที่นำมาบวก มีขนาดใกล้เคียงกันแต่มีเครื่องหมายตรงข้าม ทำให้บิตสำคัญของตัวเลข
(mantissa bits) หักล้างกัน (cancel) จำนวนมาก คำสั่งเหล่านี้จำเป็นต้องใช้รูปแบบคำสั่ง
ใหม่เพื่อระบุรีจิสเตอร์ได้ 4 ตัว ซึ่งเรียกว่ารูปแบบ R4 รูปที่ 5.11 และ 5.12 แสดงรูปแบบ
R4 ซึ่งเป็นรูปแบบที่ดัดแปลงจากรูปแบบ R

แทนที่จะมีคำสั่งกระโดดตามเงื่อนไขสำหรับจำนวนลอยตัวโดยตรง `Zfh`, `RV32F` และ



ประสิทธิภาพ

ต่างจากการเลขจำนวนเต็ม
ผลลัพธ์ของการคูณจำนวน
ลอยตัว มีขนาดเท่ากับตัว
ถูกดำเนินการ อีกทั้ง `Zfh`,
`RV32F` และ `RV32D` ไม่มี
คำสั่งหาเศษ (remainder)
สำหรับจำนวนลอยตัว

63	32	31	16	15	0
				f0 / ft0	FP Temporary
				f1 / ft1	FP Temporary
				f2 / ft2	FP Temporary
				f3 / ft3	FP Temporary
				f4 / ft4	FP Temporary
				f5 / ft5	FP Temporary
				f6 / ft6	FP Temporary
				f7 / ft7	FP Temporary
				f8 / fs0	FP Saved register
				f9 / fs1	FP Saved register
				f10 / fa0	FP Function argument และ return value
				f11 / fa1	FP Function argument และ return value
				f12 / fa2	FP Function argument
				f13 / fa3	FP Function argument
				f14 / fa4	FP Function argument
				f15 / fa5	FP Function argument
				f16 / fa6	FP Function argument
				f17 / fa7	FP Function argument
				f18 / fs2	FP Saved register
				f19 / fs3	FP Saved register
				f20 / fs4	FP Saved register
				f21 / fs5	FP Saved register
				f22 / fs6	FP Saved register
				f23 / fs7	FP Saved register
				f24 / fs8	FP Saved register
				f25 / fs9	FP Saved register
				f26 / fs10	FP Saved register
				f27 / fs11	FP Saved register
				f28 / ft8	FP Temporary
				f29 / ft9	FP Temporary
				f30 / ft10	FP Temporary
				f31 / ft11	FP Temporary
32	16		16		

รูปที่ 5.2: รีจิสเตอร์จำนวนลออยตัวของ Zfh, RV32F, และ RV32D รีจิสเตอร์ความแม่นยำเดียวจะใช้พื้นที่ครึ่งขวา (rightmost half) ของรีจิสเตอร์ความแม่นยำสองเท่าทั้ง 32 ตัว และรีจิสเตอร์แบบครึ่งความแม่นยำ จะใช้พื้นที่ครึ่งขวาของรีจิสเตอร์แบบความแม่นยำเดียว บทที่ 3 อธิบายถึงข้อตกลงการเรียกฟังก์ชัน (calling convention) ของ RISC-V สำหรับรีจิสเตอร์จำนวนลออยตัว รวมถึงเหตุผลในการจัดแบ่งรีจิสเตอร์เป็นรีจิสเตอร์ส่งค่าอาร์กิวเมนต์ (fa0–fa7), รีจิสเตอร์ที่ต้องเก็บค่าไว้ (fs0–fs11) และรีจิสเตอร์ชั่วคราว (ft0–ft11) (รูปนี้อ้างอิงจาก ตารางที่ 20.1 ใน [Waterman and Asanović 2017])

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

รูปที่ 5.3: รีจิสเตอร์ควบคุมและสถานะของจำนวนลอยตัว รีจิสเตอร์นี้เก็บโหมดการปัดเศษ (rounding modes) และแฟล็กของข้อยกเว้น (exception flags) โหมดการปัดประกอบด้วย: ปัดไปยังค่าที่ใกล้ที่สุด โดยกรณีเสมอให้เป็นเลขคู่ (rte, 000 ใน frm); ปัดเข้าหาศูนย์ (rtz, 001); ปัดลงเข้าหา $-\infty$ (rdn, 010); ปัดขึ้นเข้าหา $+\infty$ (rup, 011); และปัดไปยังค่าที่ใกล้ที่สุด โดยกรณีเสมอให้ไปทางค่าที่มีค่าสัมบูรณ์มากกว่า (rmm, 100) แฟล็กข้อยกเว้นสะสม มีทั้งหมด 5 ตัว ใช้ระบุเงื่อนไขข้อยกเว้นที่เกิดขึ้นจากคำสั่งจำนวนลอยตัว นับตั้งแต่ครั้งล่าสุดที่ซอฟต์แวร์รีเซ็ตค่า ประกอบด้วย: NV คือ การดำเนินการไม่ถูกต้อง (Invalid Operation); DZ คือ การหารด้วยศูนย์ (Divide by Zero); OF คือ ค่ามากกว่าค่าสูงสุด (Overflow); UF คือ ค่าต่ำกว่าค่าต่ำสุด (Underflow); และ NX คือ ผลลัพธ์ไม่แม่นยำ (Inexact) (อ้างอิงจากรูปที่ 8.2 ใน [Waterman and Asanović 2017])

RV32D ใช้คำสั่งเปรียบเทียบ ซึ่งจะกำหนดค่าให้รีจิสเตอร์จำนวนเต็มเป็น 1 ถ้าเงื่อนไขเป็นจริง หรือ 0 ถ้าเงื่อนไขเป็นเท็จ ตามผลการเปรียบเทียบของรีจิสเตอร์จำนวนลอยตัวสองตัว มีคำสั่งเท่ากับ (feq.h, feq.s, feq.d), น้อยกว่า (flt.h, flt.s, flt.d), น้อยกว่าหรือเท่ากับ (fle.h, fle.s, fle.d) คำสั่งเหล่านี้ทำให้สามารถใช้คำสั่งกระโดด ของจำนวนเต็มในการตัดสินใจ จากเงื่อนไขจำนวนลอยตัวได้ ตัวอย่างเช่น โค้ดนี้จะกระโดดไปที่ Exit หาก $f1 < f2$

```
flt.d x5, f1, f2    # x5 = 1 ถ้า f1 < f2; ไม่เช่นนั้น x5 = 0
bne    x5, x0, Exit # ถ้า x5 != 0, กระโดดไปที่ Exit
```

5.4 การย้ายข้อมูลและการแปลงชนิดของจำนวนลอยตัว

Zfh, RV32F และ RV32D มีคำสั่งที่รองรับการแปลงข้อมูลในทุกรูปแบบที่สำคัญระหว่างจำนวนเต็ม 32 บิตแบบมีเครื่องหมาย, จำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย, จำนวนลอยตัว 16 บิต, จำนวนลอยตัว 32 บิต, และจำนวนลอยตัว 64 บิต รูป 5.4 แสดงคำสั่งแปลงทั้งหมด 10 คำสั่ง โดย: โดยแบ่งตามประเภทข้อมูลต้นทางและประเภทข้อมูลปลายทาง

นอกจากนี้ RV32F ยังมีคำสั่งสำหรับการย้ายข้อมูลจากรีจิสเตอร์จำนวนลอยตัว f ไปยังรีจิสเตอร์จำนวนเต็ม x (fmv.x.h, fmv.x.w) และย้ายกลับจากรีจิสเตอร์จำนวนเต็ม x ไปยังรีจิสเตอร์จำนวนลอยตัว f (fmv.h.x, fmv.w.x)

5.5 คำสั่งจำนวนลอยตัวสำหรับไลบรารีคณิตศาสตร์

Zfh, RV32F และ RV32D มีคำสั่งพิเศษ ที่ช่วยงานในไลบรารีคณิตศาสตร์และสามารถใช้

To	From				
	32b signed integer (w)	32b unsigned integer (wu)	16b floating point (h)	32b floating point (s)	64b floating point (d)
32b signed int (w)	–	–	fcvt.w.h	fcvt.w.s	fcvt.w.d
32b unsigned int (wu)	–	–	fcvt.wu.h	fcvt.wu.s	fcvt.wu.d
16b fl. pt. (h)	fcvt.h.w	fcvt.h.wu	–	fcvt.h.s	fcvt.h.d
32b fl. pt. (s)	fcvt.s.w	fcvt.s.wu	fcvt.s.h	–	fcvt.s.d
64b fl. pt. (d)	fcvt.d.w	fcvt.d.wu	fcvt.d.h	fcvt.d.s	–

รูปที่ 5.4: คำสั่งแปลงข้อมูลของ Zfh, RV32F และ RV32D ตารางนี้แสดงการแปลงข้อมูลโดยคอลัมน์คือชนิดข้อมูลต้นทาง และแถวคือชนิดข้อมูลปลายทางที่แปลงแล้ว

เป็นคำสั่งเทียม (pseudoinstructions) ที่มีประโยชน์ได้ (มาตรฐาน IEEE 754 กำหนดให้ต้องมีวิธีคัดลอกและจัดการบิตเครื่องหมาย และการจำแนกประเภทข้อมูลจำนวนลอยตัว ซึ่งเป็นที่มาของคำสั่งเหล่านี้)

คำสั่งแรกคือคำสั่ง *ฉีดเครื่องหมาย (sign-injection)* คัดลอกค่าทั้งหมดจากรีจิสเตอร์ต้นทางตัวแรก ยกเว้นบิตเครื่องหมาย โดยค่าของบิตเครื่องหมายจะขึ้นอยู่กับคำสั่งที่ใช้ดังนี้:

1. (ฉีดเครื่องหมายจำนวนลอยตัว fsgnj.h, fsgnj.s, fsgnj.d) บิตเครื่องหมายของผลลัพธ์คือบิตเครื่องหมายของ rs2
2. (ฉีดเครื่องหมายจำนวนลอยตัวแบบลบ fsgnjn.h, fsgnjn.s, fsgnjn.d) บิตเครื่องหมายของผลลัพธ์ตรงข้ามกับบิตเครื่องหมายของ rs2
3. (ฉีดเครื่องหมายจำนวนลอยตัวแบบ XOR fsgnjx.h, fsgnjx.s, fsgnjx.d) บิตเครื่องหมายเป็นผลการดำเนินการ XOR ระหว่างบิตเครื่องหมายของ rs1 และ rs2

นอกจากช่วยในการจัดการเครื่องหมายในไลบรารีคณิตศาสตร์แล้ว คำสั่งเหล่านี้ยังถูกใช้สร้างคำสั่งเทียมที่นิยม สามคำสั่ง (ดูรูปที่ 3.4 ในหน้า 43)

1. คัดลอกรีจิสเตอร์จำนวนลอยตัว:

fmv.h rd, rs คือคำสั่ง fsgnj.h rd, rs, rs,
 fmv.s rd, rs คือคำสั่ง fsgnj.s rd, rs, rs, และ
 fmv.d rd, rs คือคำสั่ง fsgnj.d rd, rs, rs.

2. แปลงเครื่องหมายเป็นลบ

fneg.h rd, rs แปลงเป็น fsgnjn.h rd, rs, rs,
 fneg.s rd, rs แปลงเป็น fsgnjn.s rd, rs, rs, และ
 fneg.d rd, rs แปลงเป็น fsgnjn.d rd, rs, rs,

3. ค่าสัมบูรณ์ (เนื่องจาก $0 \oplus 0 = 0$ และ $1 \oplus 1 = 0$)

fabs.h rd, rs กลายเป็น fsgnjx.h rd, rs, rs
 fabs.s rd, rs กลายเป็น fsgnjx.s rd, rs, rs
 fabs.d rd, rs กลายเป็น fsgnjx.d rd, rs, rs

คำสั่งที่ไม่ธรรมดาอีกคำสั่งหนึ่งคือ คำสั่งจำแนกประเภท (fclass.h, fclass.s, fclass.d) ใช้สำหรับตรวจสอบว่าค่าจำนวนลอยตัวอยู่ในประเภทใด คำสั่งจำแนกประเภทเหล่านี้ เป็นเครื่องมือที่ใช้ในการพัฒนาไลบรารีคณิตศาสตร์ โดยจะตรวจสอบตัวถูกดำเนินการต้นทางว่ามีคุณสมบัติใดจาก 10 ประเภท (ดูตารางด้านล่าง) จากนั้นเขียนผลลัพธ์เป็นมาส์กลงในรีจิสเตอร์จำนวนเต็ม ใช้ 10 บิตล่างของรีจิสเตอร์ปลายทางเพื่อระบุผลลัพธ์ จะมีเพียง 1 บิตที่ถูกตั้งค่าเป็น 1 และบิตที่เหลือตั้งค่าเป็น 0

$x[rd]$ bit	ความหมาย
0	ค่าใน $f[rs1]$ เป็น ลบอนันต์ ($-\infty$)
1	ค่าใน $f[rs1]$ เป็น ค่าจำนวนลอยตัวปกติค่าลบ
2	ค่าใน $f[rs1]$ เป็น จำนวนลอยตัวค่าต่ำกว่าปกติค่าลบ
3	ค่าใน $f[rs1]$ เป็น ศูนย์ลบ (-0)
4	ค่าใน $f[rs1]$ เป็น ศูนย์บวก ($+0$)
5	ค่าใน $f[rs1]$ เป็น จำนวนลอยตัวค่าต่ำกว่าปกติค่าบวก
6	ค่าใน $f[rs1]$ เป็น จำนวนลอยตัวปกติค่าบวก
7	ค่าใน $f[rs1]$ เป็น บวกอนันต์ ($+\infty$)
8	ค่าใน $f[rs1]$ เป็น NaN แบบส่งสัญญาณ
9	ค่าใน $f[rs1]$ เป็น NaN แบบเงียบ

5.6 ส่วนขยายมาตรฐาน “Q” สำหรับ จำนวน ลอยตัว ความ แม่นยำ สี่ เท่า (Quad-Precision Floating-Point)

ส่วนขยาย Q เพิ่มคำสั่งจำนวนลอยตัวแบบไบนารีขนาด 128 บิต (quad-precision) ซึ่งเป็นไปตามมาตรฐาน IEEE 754-2019 รีจิสเตอร์จำนวนลอยตัว ถูกขยายให้สามารถเก็บค่าได้ทั้งแบบความแม่นยำเดียว (single-precision), ความแม่นยำสองเท่า (double-precision) หรือความแม่นยำสี่เท่า (quad-precision) การรองรับส่วนขยายจำนวนลอยตัวความแม่นยำสี่เท่า จำเป็นต้องใช้ RV64IFD เนื่องจากการรองรับจำนวนลอยตัวขนาด 128 บิต ต้องใช้ฮาร์ดแวร์จำนวนมาก ปัจจุบันยังไม่พบการใช้งานเชิงพาณิชย์ของ RV64Q

5.7 การเปรียบเทียบ RV32FD, ARM-32, MIPS-32 และ x86-32 ด้วย DAXPY

ต่อไปเราจะทำการเปรียบเทียบแบบตัวต่อตัว โดยใช้ DAXPY เป็นเกณฑ์วัดประสิทธิภาพสำหรับจำนวนจำนวนลอยตัว (รูปที่ 5.5) โปรแกรมนี้คำนวณสมการ $Y = a \times X + Y$ ใช้การคำนวณแบบความแม่นยำสองเท่าโดย x และ y เป็นเวกเตอร์ และ a เป็นสเกลาร์ รูปที่ 5.6 สรุปจำนวนคำสั่งและและขนาดโค้ด (หน่วยเป็นไบต์) ของโปรแกรม DAXPY สำหรับ ISA ทั้งสี่แบบ โค้ดของแต่ละ ISA แสดงอยู่ในรูป 5.7 ถึง 5.10

เช่นเดียวกับกรณีของการเรียงลำดับการแทรก (Insertion Sort) ในบทที่ 2 แม้ว่า RISC-V จะเน้นความเรียบง่ายเป็นหลัก แต่ในตัวอย่างนี้ เวอร์ชันของ RISC-V ก็ยังมีจำนวนคำสั่ง

ชื่อ DAXPY มาจากสูตรคำนวณโดยตรง: Double-precision A times X Plus Y เวอร์ชันที่ใช้ความแม่นยำเดียวเรียกว่า เรียกว่า SAXPY

```
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];
    }
}
```

รูปที่ 5.5: โปรแกรม DAXPY ในภาษา C ที่เน้นการคำนวณจำนวนจำนวนลอยตัว เวอร์ชันที่ใช้นี้เป็นเวอร์ชันที่ปรับให้ง่ายลงจาก DAXPY มาตรฐาน ซึ่งปกติจะมีพารามิเตอร์ stride และใช้การส่งอาร์กิวเมนต์แบบอ้างอิง (pass by reference) เพื่อให้สามารถทำงานร่วมกับภาษา Fortran ได้

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

รูปที่ 5.6: จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับสำหรับ ISA ทั้งสี่แบบ โดยจะแสดงจำนวนคำสั่งต่อหนึ่งรอบลูปและจำนวนคำสั่งรวมทั้งหมด บทที่ 7 อธิบายเกี่ยวกับ ARM Thumb-2, microMIPS และ RV32C

เท่ากันหรือน้อยกว่า และขนาดโค้ดของแต่ละสถาปัตยกรรมก็ใกล้เคียงกันมาก ในกรณีนี้คำสั่งเปรียบเทียบแล้วดำเนินการกระโดดแบบมีเงื่อนไข ของ RISC-V สามารถลดจำนวนคำสั่งได้พอๆ กับโหมดการเข้าถึงที่อยู่ที่ซับซ้อนกว่า และคำสั่ง push และ pop ของ ARM-32 และ x86-32

5.8 ข้อสังเกตสรุป

น้อยแต่มาก

— คำกล่าวของ โรเบิร์ต บราวน์, ในปี 1855 สำนักสถาปัตยกรรมแบบมินิมัลลิสต์ (ด้านการออกแบบอาคาร) ได้นำบทวินีไปใช้เป็นหลักการพื้นฐาน ในช่วงทศวรรษ 1980



ความเรียบง่าย



ประสิทธิภาพ

มาตรฐานจำนวนลอยตัว IEEE 754-2019 [IEEE Standards Committee 2019] ได้กำหนดประเภทข้อมูลจำนวนลอยตัว ความแม่นยำของการคำนวณ และชุดการดำเนินการที่จำเป็น ความสำเร็จของมาตรฐานนี้ ช่วยลดความยากในการพอร์ตโปรแกรมจำนวนลอยตัวระหว่างระบบต่าง ๆ อย่างมาก และยังทำให้ ISA สำหรับจำนวนลอยตัว (Floating-point ISAs) มีความเป็นมาตรฐานเดียวกันมากกว่าเมื่อเทียบกับชุดคำสั่งในบทอื่นๆ

■ รายละเอียดเพิ่มเติม: การปรับปรุงมาตรฐาน IEEE 754

มาตรฐานฉบับปรับปรุง IEEE 754-2008 [IEEE Standards Committee 2008] ได้เพิ่มรูปแบบใหม่หลายรูปแบบ นอกเหนือจากความแม่นยำเดียว (binary32) และความแม่นยำสองเท่า (binary64) ของมาตรฐานเดิม รูปแบบใหม่ที่เพิ่มเข้ามา ได้แก่ ครึ่งความแม่นยำ (binary16) และความแม่นยำแปดเท่า (binary256) สิ่งเพิ่มเข้ามาอย่างค่อนข้างน่าประหลาดใจ คือ จำนวนลอยตัวแบบฐานสิบ (decimal floating point) ซึ่ง RISC-V ได้กันพื้นที่ไว้เบื้องต้นในสไลด์ RV32L รูปแบบฐานสิบมีสามรูปแบบ ตามชื่อ *decimal32*, *decimal64* และ *decimal128* เรายังไม่พบการใช้งานฮาร์ดแวร์เชิงพาณิชย์ ที่รองรับจำนวนจำนวนลอยตัวฐานสิบ สำหรับโปรเซสเซอร์ RISC-V มาตรฐาน IEEE 754-2019 ซึ่งเผยแพร่ในเดือนกรกฎาคม ปี 2019 เป็นการปรับปรุงเล็กน้อยจาก IEEE 754-2008 โดยเน้นไปที่ การอธิบายให้ชัดเจนขึ้น การแก้ไขข้อบกพร่อง และเพิ่มการดำเนินการที่แนะนำใหม่

5.9 เรียนรู้เพิ่มเติม

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

IEEE Standards Committee. 754-2019 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2019.

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 คำสั่ง in loop; 11 คำสั่ง/44 bytes total; 28 bytes RVC)
# a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], fa0 คือ a
0: 02050463 beqz    a0,28          # ถ้า n == 0 กระโดดไปที่ Exit
4: 00351513 slli    a0,a0,0x3      # a0 = n*8
8: 00a60533 add     a0,a2,a0       # a0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
Loop:
c: 0005b787 fld     fa5,0(a1)      # fa5 = x[]
10: 00063707 fld     fa4,0(a2)     # fa4 = y[]
14: 00860613 addi    a2,a2,8        # a2++ (เพิ่มตัวชี้ไปที่ y)
18: 00858593 addi    a1,a1,8        # a1++ (เพิ่มตัวชี้ไปที่ 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        # ถ้า i != n กระโดดไปที่ Loop
Exit:
28: 00008067                ret     # คืนค่า
```

รูปที่ 5.7: โค้ด RV32D สำหรับ DAXPY แสดงในรูปที่ 5.5 ด้านซ้ายเป็นที่อยู่หน่วยความจำ ในรูปเลขฐานสิบหก (hexadecimal) ถัดมาเป็นโค้ดภาษาเครื่อง ในรูปเลขฐานสิบหก จากนั้นเป็นคำสั่งภาษาแอสเซมบลี พร้อมคำอธิบาย (comment) คำสั่งแบบเปรียบเทียบแล้วกระโดด (compare-and-branch) ของ RISC-V ช่วยหลีกเลี่ยงการใช้คำสั่งเปรียบเทียบ 2 คำสั่ง ที่พบในโค้ดของ ARM-32 และ x86-32

```
# ARM-32 (6 คำสั่ง in loop; 10 คำสั่ง/40 bytes total; 28 bytes Thumb-2)
# r0 คือ n, d0 คือ a, r1 เป็นตัวชี้ไปที่ x[0], r2 เป็นตัวชี้ไปที่ y[0]
0: e3500000 cmp     r0, #0          # เปรียบเทียบ n to 0
4: 0a000006 beq     24 <daxpy+0x24> # ถ้า n == 0 กระโดดไปที่ Exit
8: e0820180 add     r0, r2, r0, lsl #3 # r0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
Loop:
c: ecb16b02 vldmia   r1!, {d6}      # d6 = x[i], เพิ่มตัวชี้ไปที่ 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], เพิ่มตัวชี้ไปที่ y
1c: e1520000 cmp     r2, r0          # i vs. n
20: 1afffff9 bne     c <daxpy+0xc>    # ถ้า i != n กระโดดไปที่ Loop
Exit:
24: e12ffff1e bx      lr            # คืนค่า
```

รูปที่ 5.8: โค้ด ARM-32 สำหรับ DAXPY แสดงในรูปที่ 5.5 โหมดการเข้าถึงที่อยู่แบบเพิ่มค่าอัตโนมัติ (autoincrement addressing mode) ของ ARM-32 ช่วยลดจำนวนคำสั่งได้ 2 คำสั่ง เมื่อเทียบกับ RISC-V ต่างจากกรณีของการเรียงลำดับแบบแทรก (Insertion Sort) การทำ DAXPY ใน ARM-32 ไม่จำเป็นต้องใช้คำสั่ง push และ pop ของรีจิสเตอร์

```

# MIPS-32 (7 คำสั่ง in loop; 12 คำสั่ง/48 bytes total; 32 bytes microMIPS)
# a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], f12 คือ a
0: 10800009 beqz    a0,28 <daxpy+0x28>    # ถ้า n = 0 กระโดดไปที่ Exit
4: 000420c0 sll     a0,a0,0x3              # a0 = n*8  หน่วยเวลาการกระโดดตามเงื่อนไข
8: 00c42021 addu    a0,a2,a0              # a0 = ที่อยู่ของ y[n] (ส่วนสุดท้าย)
Loop:
c: 24c60008 addiu   a2,a2,8                # a2++ (เพิ่มตัวชี้ไปที่ y)
10: d4a00000 ldc1   $f0,0(a1)             # f0 = x[i]
14: 24a50008 addiu   a1,a1,8                # a1++ (เพิ่มตัวชี้ไปที่ 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>    # ถ้า i != n กระโดดไปที่ Loop
24: f4c0ffff sdc1   $f0,-8(a2)            # y[i] = a*x[i] + y[i] (ช่องเสียเพิ่มเติม)
Exit:
28: 03e00008 jr     ra                    # คืนค่า
2c: 00000000 nop                          # ไม่ทำการหน่วยเวลา

```

รูปที่ 5.9: โค้ด MIPS-32 สำหรับ DAXPY แสดงในรูป 5.5 จากช่องหน่วยการกระโดดทั้ง 3 ตำแหน่ง (branch delay slots) มี 2 ตำแหน่งที่ถูกใช้ด้วยคำสั่งที่มีประโยชน์ ความสามารถในการตรวจสอบความเท่ากันของรีจิสเตอร์สองตัวโดยตรงช่วยหลีกเลี่ยงการใช้คำสั่งเปรียบเทียบ 2 คำสั่ง ที่พบใน ARM-32 และ x86-32 นอกจากนี้ การโหลดข้อมูลแบบจำนวนลอยตัวไม่มีช่องหน่วยเวลา (delay slot) ต่างจากการโหลดข้อมูลจำนวนเต็ม

```

# x86-32 (6 คำสั่ง in loop; 16 คำสั่ง/50 bytes total)
# eax คือ i, n อยู่ในหน่วยความจำที่ esp+0x8, a อยู่ในหน่วยความจำที่ esp+0xc
# ตัวชี้ไปที่ x[0] อยู่ในหน่วยความจำที่ esp+0x14
# ตัวชี้ไปที่ y[0] อยู่ในหน่วยความจำที่ esp+0x18
0: 53                push     ebx                # บันทึก ebx
1: 8b 4c 24 08        mov     ecx,[esp+0x8]          # ecx มีสำเนาของ n
5: c5 fb 10 4c 24 0c vmovsd   xmm1,[esp+0xc]      # xmm1 มีสำเนาของ a
b: 8b 5c 24 14        mov     ebx,[esp+0x14]        # ebx ชี้ไปที่ x[0]
f: 8b 54 24 18        mov     edx,[esp+0x18]        # edx ชี้ไปที่ y[0]
13: 85 c9             test    ecx,ecx              # เปรียบเทียบ n to 0
15: 74 19             je      30 <daxpy+0x30>      # ถ้า n = 0 กระโดดไปที่ Exit
17: 31 c0             xor     eax,eax              # i=0 (since $x^x==$0)
Loop:
19: c5 fb 10 04 c3     vmovsd   xmm0,[ebx+eax*8]      # xmm0 = x[i]
1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
24: c5 fb 11 04 c2     vmovsd   xmm0,xmm1,[edx+eax*8] # y[i] = a*x[i] + y[i]
29: 83 c0 01          add     eax,0x1              # i++
2c: 39 c1             cmp     ecx,eax              # เปรียบเทียบ i vs n
2e: 75 e9             jne     19 <daxpy+0x19>      # ถ้า i!=n กระโดดไปที่ Loop
Exit:
30: 5b                pop     ebx                # กู้คืน ebx
31: c3                ret                       # คืนค่า

```

รูปที่ 5.10: โค้ด x86-32 สำหรับ DAXPY แสดงในรูป 5.5 ข้อจำกัดของรีจิสเตอร์ใน x86-32 เห็นได้ชัดในตัวอย่างนี้ โดยมีตัวแปรถึง 4 ตัวที่ต้องเก็บไว้ในหน่วยความจำ ทั้งที่ใน ISA อื่น ๆ ตัวแปรเหล่านี้ถูกเก็บไว้ในรีจิสเตอร์ ตัวอย่างนี้ยังแสดงรูปแบบการเขียนคำสั่ง (idioms) ที่พบบ่อยใน x86-32 เช่น การเปรียบเทียบรีจิสเตอร์กับศูนย์ (test ecx, ecx) หรือการกำหนดค่ารีจิสเตอร์ให้เป็นศูนย์ (xor eax, eax)

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

รูปที่ 5.11: แผนผังรหัสดำเนินการของ RV32F แสดง โครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทของรูปแบบ และชื่อคำสั่ง ความแตกต่างหลักในการเข้ารหัส (encoding) ระหว่างรูปแบบกับรูปถัดไปคือ บิตที่ 12 เป็นค่า 0 สำหรับสองคำสั่งแรก และบิตที่ 25 เป็นค่า 0 สำหรับคำสั่งที่เหลือ ในขณะที่ใน RV32D ทั้งสองบิตนี้จะเป็นค่า 1 (รูปนี้อ้างอิงจาก ตารางที่ 19.2 ใน [Waterman and Asanović 2017])

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

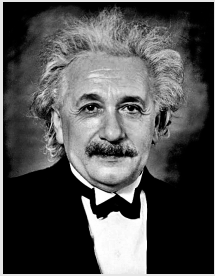
รูปที่ 5.12: แผนผังรหัสดำเนินการ ของ RV32D แสดง โครงสร้างคำสั่ง, รหัสดำเนินการ, ประเภทของรูปแบบ และชื่อคำสั่ง มีคำสั่งบางคำสั่งในสองรูปนี้ ที่ไม่ได้แตกต่างกันเพียงแค่ว่าขนาดข้อมูล (data width) โดยในรูปนี้มีคำสั่งเฉพาะคือ คำสั่ง `fcvt.s.d` และ `fcvt.d.s` ในขณะที่อีกรูปหนึ่งมีคำสั่ง `fmv.xw` และ `fmv.wx` (รูปนี้อ้างอิงจาก ตารางที่ 19.2 ใน [Waterman and Asanović 2017])

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]						rs1	011		rd	0000111	I flh			
imm[11:5]				rs2		rs1	011		imm[4:0]	0100111	S fsh			
rs3	10			rs2		rs1	rm		rd	1000011	R4 fmadd.h			
rs3	10			rs2		rs1	rm		rd	1000111	R4 fmsub.h			
rs3	10			rs2		rs1	rm		rd	1001011	R4 fnmsub.h			
rs3	10			rs2		rs1	rm		rd	1001111	R4 fnmadd.h			
0000010					rs2		rs1	rm		rd	1010011	R fadd.h		
0000110					rs2		rs1	rm		rd	1010011	R fsub.h		
0001010					rs2		rs1	rm		rd	1010011	R fmul.h		
0001110					rs2		rs1	rm		rd	1010011	R fdiv.h		
0101110				00000		rs1	rm		rd	1010011	R fsqrt.h			
0010010					rs2		rs1	000		rd	1010011	R fsgnj.h		
0010010					rs2		rs1	001		rd	1010011	R fsgnjn.h		
0010010					rs2		rs1	010		rd	1010011	R fsgnjx.h		
0010110					rs2		rs1	000		rd	1010011	R fmin.h		
0010110					rs2		rs1	001		rd	1010011	R fmax.h		
0100000				00010		rs1	rm		rd	1010011	R fcvt.s.h			
0100010				00000		rs1	rm		rd	1010011	R fcvt.h.s			
0100001				00010		rs1	rm		rd	1010011	R fcvt.d.h			
0100010				00001		rs1	rm		rd	1010011	R fcvt.h.d			
1010010					rs2		rs1	010		rd	1010011	R feq.h		
1010010					rs2		rs1	001		rd	1010011	R flt.h		
1010010					rs2		rs1	000		rd	1010011	R fle.h		
1110010				00000		rs1	001		rd	1010011	R fclass.h			
1100010				00000		rs1	rm		rd	1010011	R fcvt.w.h			
1100010				00001		rs1	rm		rd	1010011	R fcvt.wu.h			
1101010				00000		rs1	rm		rd	1010011	R fcvt.h.w			
1101010				00001		rs1	rm		rd	1010011	R fcvt.h.wu			
1110010				00000		rs1	000		rd	1010011	R fmv.x.h			
1111010				00000		rs1	000		rd	1010011	R fmv.h.x			

รูปที่ 5.13: แผนผังรหัสดำเนินการ Zfh แสดงรูปแบบคำสั่ง, รหัสดำเนินการ, ประเภทของรูปแบบ และชื่อคำสั่ง รูปนี้รวมคำสั่งทั้งสองประเภท ที่ปรากฏเฉพาะในรูปก่อนหน้าทั้งหมดเข้าด้วยกัน (รูปนี้อ้างอิงจาก ตารางที่ 19.2 ใน [Waterman and Asanović 2017])

RV32A: คำสั่งแบบอะตอมมิก

อัลเบิร์ต ไอน์สไตน์ (ค.ศ. 1879–1955) เป็นนักวิทยาศาสตร์ที่มีชื่อเสียงที่สุดของศตวรรษที่ 20 เขาเป็นผู้คิดค้นทฤษฎีสัมพัทธภาพ และสนับสนุนการสร้างระเบิดปรมาณู ในช่วงสงครามโลกครั้งที่สอง



ทุกสิ่งควรถูกทำให้เรียบง่ายที่สุดเท่าที่เป็นไปได้ แต่ต้องไม่ง่ายไปกว่านั้น

—Albert Einstein, 1933

6.1 บทนำ

เราสมมติว่าผู้อ่านเข้าใจการสนับสนุนของสถาปัตยกรรมชุดคำสั่ง (ISA) สำหรับการประมวลผลหลายตัวประมวลผล (multiprocessing) อยู่แล้ว ดังนั้นหน้าที่ของเราจึงมีเพียงอธิบายคำสั่ง RV32A และหน้าที่การทำงานของมัน หากผู้อ่านยังไม่มีพื้นฐานเพียงพอ หรือต้องการทบทวน แนะนำให้ศึกษาหัวข้อ “การซิงโครไนซ์ (synchronization) วิทยาการคอมพิวเตอร์” ในวิกิพีเดียเพิ่มเติม

([https://en.wikipedia.org/wiki/Synchronization_\(computer_science\)](https://en.wikipedia.org/wiki/Synchronization_(computer_science)))

หรืออ่านหัวข้อ 2.1 ของหนังสือสถาปัตยกรรม RISC-V ที่เกี่ยวข้องของเรา [Patterson and Hennessy 2021]

RV32A มีการดำเนินการแบบอะตอมมิกสำหรับการซิงโครไนซ์ 2 ประเภท คือ

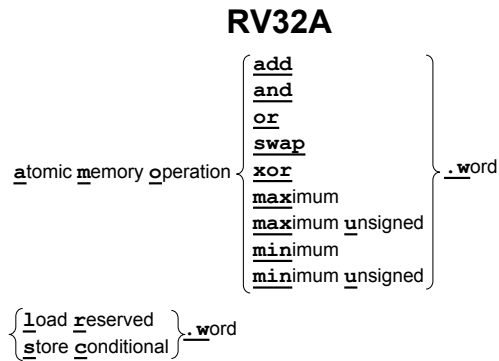
- การดำเนินการกับหน่วยความจำแบบอะตอมมิก (Atomic Memory Operations - AMO) และ
- การโหลดแบบสงวนสิทธิ์ (Load Reserved) และการเก็บข้อมูลแบบมีเงื่อนไข (Store Conditional)

รูปที่ 6.1 เป็นภาพแสดงของชุดคำสั่งส่วนขยาย RV32A ในเชิงกราฟิก และรูป 6.2 แสดงรหัสดำเนินการ (opcodes) และรูปแบบคำสั่ง (instruction formats ของคำสั่งเหล่านั้น

คำสั่ง AMO จะทำการดำเนินการกับข้อมูลในหน่วยความจำแบบอะตอมมิก และตั้งคาร์ริจิสเตอร์ปลายทางให้เก็บค่าดั้งเดิมของหน่วยความจำนั้น คำว่า อะตอมมิก (atomic) หมายความว่า จะต้องไม่มีการขัดจังหวะเกิดขึ้นระหว่างการอ่านและการเขียนหน่วยความจำ และไม่มีการโปรเซสเซอร์ตัวอื่นสามารถแก้ไขค่าหน่วยความจำนั้นได้ ระหว่างช่วงการอ่านและเขียนของคำสั่ง AMO

คำสั่งโหลดแบบสงวนสิทธิ์ (Load reserved) และการเก็บข้อมูลแบบมีเงื่อนไข (Store conditional) จะสร้างการดำเนินการแบบอะตอมมิกผ่านคำสั่งสองคำสั่งร่วมกัน คำสั่ง

ทั้ง AMO และ LR/SC ต้องใช้ที่อยู่หน่วยความจำที่จัดแนวตามธรรมชาติ เพราะเป็นภาระที่ยุงยากสำหรับฮาร์ดแวร์ในการรับประกันความเป็นอะตอมมิก (atomicity) เมื่อข้อมูลข้ามขอบเขตของแคชไลน์



รูปที่ 6.1: ไตอะแกรมของคำสั่ง RV32A

31	27	26	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

รูปที่ 6.2: แผนผังรหัสดำเนินการ RV32A แสดงโครงสร้างคำสั่ง, รหัสดำเนินการ, รูปแบบของคำสั่ง และชื่อคำสั่ง (อ้างอิงจากตารางที่ 19.2 ของ [Waterman and Asanović 2017])

โหลดแบบสแกนสิทธิ์ (LR) จะอ่านค่าจากหน่วยความจำ เขียนค่าลงในรีจิสเตอร์ปลายทาง และบันทึกการจอง (reservation) ที่ตำแหน่งหน่วยความจำนั้น คำสั่งเก็บข้อมูลแบบมีเงื่อนไข (SC) จะเขียนข้อมูลลงหน่วยความจำที่ตำแหน่งหน่วยความจำในรีจิสเตอร์ต้นทางก็ต่อเมื่อ ยังมีการจองอยู่ที่ตำแหน่งหน่วยความจำนั้นอยู่ ถ้าเขียนสำเร็จจะเก็บศูนย์ลงในรีจิสเตอร์ปลายทาง แต่ถ้าเขียนไม่สำเร็จ จะเขียนรหัสข้อผิดพลาดที่ไม่เป็นศูนย์แทน

คำถามสำคัญคือ ทำไม RV32A ต้องมีวิธีการดำเนินการแบบอะตอมมิก ถึงสองแบบ คำตอบคือ เพราะมีการใช้งานที่แตกต่างกันอยู่สองแบบอย่างชัดเจน

นักพัฒนาภาษาการโปรแกรมมักคาดหวังให้สถาปัตยกรรมพื้นฐานรองรับคำสั่ง อะตอมมิกแบบเปรียบเทียบและสลับ (compare-and-swap) ได้ กล่าวคือ: คำสั่งจะทำการเปรียบเทียบค่าจากรีจิสเตอร์กับค่าในหน่วยความจำที่ระบุโดยรีจิสเตอร์อีกตัวหนึ่ง และถ้าค่าทั้งสองเท่ากัน ก็สลับค่าจากรีจิสเตอร์ตัวที่สาม เข้าไปแทนค่าที่อยู่ในหน่วยความจำนั้น เหตุผลที่โปรแกรมเมอร์คาดหวังเช่นนั้น เพราะคำสั่ง compare-and-swap (CAS) เป็นตัวเลือกพื้นฐานสำหรับการซิงโครไนซ์ ที่ใช้งานได้ทั่วไป (universal synchronization primitive) กล่าวคือ ปฏิบัติการซิงโครไนซ์แบบคำเดียว (single-word) ทุกแบบ สามารถสร้างขึ้นได้จาก CAS[Herlihy 1991]

แม้ว่านี่จะเป็นเหตุผลที่หนักแน่นสำหรับการเพิ่มคำสั่งดังกล่าวเข้าไปใน ISA แต่มันต้องใช้รีจิสเตอร์ต้นทางถึงสามตัวในคำสั่งเดียว การเพิ่มจำนวน source operand จากสองเป็นสาม ทำให้การออกแบบส่วนต่อประสานหน่วยความจำซับซ้อนขึ้น และทำให้ทางเดินข้อมูลจำนวนเต็มรวมถึงรูปแบบคำสั่งซับซ้อนขึ้นด้วย (คำสั่ง multiply-add ใน RV32FD ที่มีตัวถูกดำเนินการสามตัวแต่ส่งผลต่อทางเดินข้อมูลทศนิยม ไม่ใช่ทางเดินข้อมูลจำนวนเต็ม) RISC-V ใช้คำสั่ง LR/SC แทน CAS ที่ใช้ รีจิสเตอร์ต้นทางเพียงสองตัวและสามารถนำมาใช้สร้างการดำเนินการอะตอมมิก compare-and-swap ได้ (ดูครึ่งบนของรูปที่ 6.3)

เหตุผลที่ RV32A ยังมีคำสั่ง AMO เพิ่มเข้ามา คือ AMO สามารถขยายขนาดไปยังระบบมัลติโพรเซสเซอร์ขนาดใหญ่ได้ดีกว่า LR/SC นอกจากนี้ AMO ยังสามารถใช้ดำเนินการรวมค่าแบบรีดิกชันได้อย่างมีประสิทธิภาพ และยังมีประโยชน์สำหรับการสื่อสารกับอุปกรณ์ I/O เนื่องจาก AMO ทำการอ่านและเขียนระหว่าง CPU กับหน่วยความจำหรืออุปกรณ์ใน transaction แบบอะตอมมิกเพียงครั้งเดียว ความเป็นอะตอมมิกนี้ช่วยลดความซับซ้อนของไดเรกทอรีอุปกรณ์ และเพิ่มประสิทธิภาพ I/O ส่วนครึ่งล่างของรูป 6.3 แสดงตัวอย่างการเขียนส่วนวิกฤต (critical section) โดยใช้การสลับแบบอะตอมมิก

■ รายละเอียดเพิ่มเติม: โมเดลความสอดคล้องของหน่วยความจำ

RISC-V ใช้โมเดล ความสอดคล้องของหน่วยความจำแบบผ่อนคลาย (relaxed memory consistency model) ดังนั้น เรดอื่นอาจเห็นการเข้าถึงหน่วยความจำในลำดับที่ไม่ตรงกับลำดับจริงได้ รูป 6.2 แสดงให้เห็นว่าคำสั่งทั้งหมดของ RV32A มี บิตการเข้าถึง - acquire bit (aq) และ บิตการปล่อย - release bit (rl) ถ้าการดำเนินการอะตอมมิก ถ้าตั้งบิต aq เรดอื่นจะมองเห็นคำสั่ง AMO นี้เรียงลำดับถูกต้องกับการเข้าถึงหน่วยความจำที่เกิดขึ้น *หลังจากนั้น* (subsequent accesses) หรือถ้าตั้งบิต rl เรดอื่นจะมองเห็นปฏิบัติการอะตอมมิกนี้เรียงลำดับถูกต้องกับการเข้าถึงหน่วยความจำที่เกิดขึ้น *ก่อนหน้านี้* (previous accesses) หากต้องการศึกษาเพิ่มเติม [Adve and Gharachorloo 1996] เป็นเอกสารที่อธิบายที่ดีเกี่ยวกับหัวข้อนี้



ความเรียบง่าย



ประสิทธิภาพ

```
# Compare-and-swap (CAS) memory word M[a0] using lr/sc instructions
# Expected old value in a1 desired new value in a2
0: 100526af    lr.w    a3, (a0)    # โหลดค่าเก่า
4: 06b69e63    bne     a3, a1, 80          # เปรียบเทียบค่าเก่ากับ a1
8: 18c526af    sc.w    a3, a2, (a0)        # ถ้าเท่ากันให้เก็บค่าใหม่
c: fe069ae3    bnez     a3, 0              # ถ้าไม่สำเร็จให้ลองใหม่
... เปรียบเทียบ - รหัสหลังจากสลับสำเร็จ ...
80:                                     # เปรียบเทียบ-สลับไม่สำเร็จ

# Critical section guarded by test-and-set spinlock using an คำสั่ง AMO
0: 00100293    li      t0, 1              # เริ่มต้นการล๊อค
4: 0c55232f    amoswap.w.aq t1, t0, (a0)   # ลองรับล๊อคเกอร์
8: fe031ee3    bnez     t1, 4              # ถ้าพลาดก็พยายามต่อไป
... รหัสส่วนสำคัญ ...
20: 0a05202f    amoswap.w.rl x0, x0, (a0)   # ปลดล๊อค
```

รูปที่ 6.3: ตัวอย่างการชิงโครโนซ์ มี 2 แบบ ตัวอย่างแรกใช้ การโหลดแบบสงวนสิทธิ์และการเก็บแบบมีเงื่อนไข (load reserved/store conditional) `lr.w`, `sc.w` เพื่อสร้างการเปรียบเทียบและสลับ (compare-and-swap) และตัวอย่างที่สอง ใช้การสลับแบบอะตอมมิก `amoswap.w` เพื่อสร้าง mutex (mutual exclusion)

อะไรที่แตกต่าง? MIPS-32 รุ่นแรกไม่มีระบบสำหรับการชิงโครโนซ์ แต่ภายหลังสถาปนิกได้เพิ่มคำสั่ง โหลดแบบสงวนสิทธิ์และการเก็บข้อมูลแบบมีเงื่อนไขเข้ามา

6.2 ข้อสังเกต สรุป

RV32A เป็นส่วนขยายแบบเลือกใช้ (optional) และโปรเซสเซอร์ RISC-V จะเรียบง่ายกว่าถ้าไม่มี RV32A อย่างไรก็ตาม ดังที่ไอน์สไตน์ได้กล่าวไว้: ทุกสิ่งควรถูกทำให้เรียบง่ายที่สุดเท่าที่เป็นไปได้ แต่ไม่ง่ายไปกว่านั้น หลายสถานการณ์จำเป็นต้องใช้ RV32A

6.3 เรียนรู้เพิ่มเติม

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 Second Edition: The Hardware Software Interface*. Morgan Kaufmann, 2021.

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/>.

E.F. ชุมัคเกอร์

(พ.ศ. 1911–1977) เป็นผู้เขียนหนังสือเศรษฐศาสตร์ที่สนับสนุนแนวคิด เทคโนโลยีที่เหมาะสมกับมนุษย์ การกระจายศูนย์ และเทคโนโลยีที่เหมาะสม หนังสือของเขาถูกแปลเป็นหลายภาษา และได้รับการยกย่องว่าเป็นหนึ่งใน 100 หนังสือที่ทรงอิทธิพลที่สุด นับตั้งแต่สงครามโลกครั้งที่สอง

**small
is
beautiful**
a study of economics
as if people mattered
EF Schumacher



ขนาดโปรแกรม



ความเรียบง่าย

เล็กๆ ก็สวยงาม

—E.F. ชุมัคเกอร์, 1973

7.1 บทนำ

สถาปัตยกรรมชุดคำสั่ง (ISA) ก่อนหน้ามักเพิ่มจำนวนคำสั่ง และรูปแบบคำสั่งจำนวนมาก เพื่อให้ได้มิติขนาดลดลง เช่น ใช้คำสั่งที่มีตัวถูกดำเนินการ สองตัวแทนสามตัว หรือใช้คำสั่งที่ทันที (immediate fields) ขนาดเล็ก ARM และ MIPS คิดค้น ISA ใหม่ถึงสองครั้งเพื่อจุดประสงค์นี้ ได้แก่ ARM Thumb และ Thumb-2 รวมถึง MIPS16 และ microMIPS แนวทางเหล่านี้ ทำให้โปรเซสเซอร์ซับซ้อนขึ้น ทำให้คอมพิวเตอร์ซับซ้อนขึ้น และเพิ่มภาระทางความคิดให้โปรแกรมเมอร์

RV32C ใช้วิธีที่แตกต่าง คำสั่ง 16 บิต *ทุกคำสั่ง* ต้องแมปไปยังคำสั่ง RISC-V ปกติขนาด 32 บิต *เพียงคำสั่งเดียว* นอกจากนี้ มีเพียงแอสเซมเบลอร์และลิงเกอร์เท่านั้นที่รู้เรื่องคำสั่ง 16 บิตเหล่านี้ และขึ้นอยู่กับแอสเซมเบลอร์และลิงเกอร์ที่จะเลือกแทนที่คำสั่ง 32 บิตด้วยคำสั่งขนาด 16 บิตที่เทียบเท่า นักพัฒนาคอมพิวเตอร์และโปรแกรมเมอร์ภาษาแอสเซมบลีไม่จำเป็นต้องรู้รายละเอียด และรูปแบบของ RV32C แต่จะได้โปรแกรมที่มีขนาดเล็กลง รูปที่ 7.1 แสดงผลแบบกราฟิกของชุดคำสั่งส่วนขยาย RV32C

สถาปนิก RISC-V เลือก คำสั่ง ใน ส่วน ขยาย RVC เพื่อให้ได้การ บีบอัด โค้ด ที่มีประสิทธิภาพสำหรับโปรแกรมหลากหลาย โดยใช้ข้อสังเกตสามประการ เพื่อให้คำสั่งในขนาด 16 บิต: ประการแรก มีรีจิสเตอร์ที่นิยมใช้สิบตัว เช่น (a0–a5, s0–s1, sp และ ra) ถูกใช้งานบ่อยกว่าตัวอื่นๆ อย่างมาก ประการที่สอง คำสั่งจำนวนมากมักจะเขียนทับตัวถูกดำเนินการต้นทาง (source operand) ตัวใดตัวหนึ่งของมันเอง ประการที่สาม ตัวถูกดำเนินการแบบค่าทันที (immediate operands) ส่วนใหญ่มีขนาดเล็ก และบางคำสั่งมักใช้ค่าที่มีรูปแบบเฉพาะ ดังนั้น คำสั่ง RV32C ส่วนใหญ่จึงสามารถเข้าถึงได้เฉพาะรีจิสเตอร์ที่นิยมใช้ บางคำสั่งเขียนทับตัวถูกดำเนินการต้นทางโดยอัตโนมัติ และค่าคงที่ทันทีจะถูกลดขนาดลง ในการโหลด (loads) และจัดเก็บ (stores) จะใช้เฉพาะ offset แบบ unsigned ที่เป็นจำนวนเท่าของขนาดข้อมูล

รูปที่ 7.3 และ 7.4 ตัวอย่างโค้ดที่คอมไพล์แล้วของ Insertion Sort และ DAXPY โดย

RV32C

Integer Computation

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

Loads and Stores

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

Control transfer

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

Other instructions

$\underline{\text{c.environment break}}$

รูปที่ 7.1: แผนภาพของคำสั่ง RV32C แสดงชุดคำสั่งแบบบีบอัด โดย ฟิลด์ค่าคงที่ทันทีของคำสั่งเลื่อนบิต (shift) และ c.addi4spn จะถูกขยายศูนย์ ส่วนคำสั่งอื่นๆ จะถูกขยายเครื่องหมาย

ใช้ RV32C รวมถึงแสดงรายการคำสั่ง RV32C สำหรับ Insertion Sort และ DAXPY เพื่อแสดงให้เห็นผลกระทบของการบีบอัดโค้ดอย่างชัดเจน แต่ปกติแล้ว คำสั่งเหล่านี้จะมองไม่เห็นในโปรแกรมภาษาแอสเซมบลี ข้อความอธิบายในโค้ดแสดงคำสั่งแบบ 32 บิตที่เทียบเท่ากับคำสั่ง RV32C โดยแสดงไว้ในวงเล็บ ภาคผนวก A จะแสดงคำสั่ง RISC-V แบบ 32 บิตที่สอดคล้องกับคำสั่ง RV32C แบบ 16 บิตแต่ละคำสั่ง

ตัวอย่าง ที่ตำแหน่งที่อยู่ 4 ใน Insertion Sort ในรูป 7.3 แอสเซมเบลอร์แทนที่คำสั่ง RV32I 32 บิต

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

ด้วยคำสั่ง RV32 แบบ 16 บิตดังนี้:

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

คำสั่งโหลดค่าคงที่ทันที c.li ของ RV32C มีขนาดเล็กกว่า เพราะต้องการระบุเพียงรีจิสเตอร์ 1 ตัวและค่าคงที่ทันทีขนาดเล็ก โค้ดภาษาเครื่องของคำสั่ง c.li มีเพียง 4 หลักเลขฐานสิบหก ในรูป 7.3 ซึ่งแสดงว่าคำสั่งว่าคำสั่ง c.li มีขนาดเพียง 2 ไบต์ (16 บิต)

อีกตัวอย่างคือ ที่ตำแหน่งที่อยู่ 10 ในรูป 7.3 แอสเซมเบลอร์แทนที่คำสั่ง:

```
add a2,x0,a3 # a2 เป็นตัวชี้ไปที่ a[j]
```

ด้วยคำสั่ง RV32C 16 บิต ดังนี้:

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

รูปที่ 7.2: คำสั่งและขนาดของโค้ดสำหรับ Insertion Sort และ DAXPY ใน ISA แบบบีบอัด

```
c.mv a2, a3    # (ขยายเป็น add a2, x0, a3) a2 เป็นตัวชี้ไปที่ a[j]
```

คำสั่ง `c.li` ของ RV32C มีขนาดเพียง 16 บิต เพราะระบบแคร์รีจิสเตอร์ 2 ตัว

ในการออกแบบฮาร์ดแวร์ แม้ว่าผู้ออกแบบโปรเซสเซอร์จะไม่สามารถละเลย RV32C ได้ แต่มีเทคนิคหนึ่งที่จะช่วยให้การรองรับคำสั่งเหล่านี้มีต้นทุนต่ำ โดยตัวถอดรหัส (decoder) จะทำการแปลงคำสั่ง 16 บิตทั้งหมดให้เป็นคำสั่ง 32 บิตที่เทียบเท่า *ก่อนนำไปประมวลผล* รูป 7.6 ถึง 7.8 รายการรูปแบบคำสั่ง และรหัสดำเนินการของ RV32C เป็นสิ่งที่ตัวถอดรหัสต้องแปลง โดยการรองรับส่วนนี้ใช้เพียงประมาณ 400 เกตเท่านั้น เมื่อเทียบกับโปรเซสเซอร์ 32 บิตที่เล็กที่สุด—ที่ไม่มีส่วนขยาย RISC-V—ใช้ประมาณ 8000 เกต มีขนาดเพียงประมาณ 5% ของดีไซน์ขนาดเล็กมาก ดังนั้น ในโปรเซสเซอร์ระดับปานกลางที่มีแคช (ประมาณ 100,000 เกต) ส่วนของตัวถอดรหัสนี้แทบไม่มีนัยสำคัญเลย

อะไรที่แตกต่าง RV32C ไม่มีคำสั่งแบบไบต์หรือครึ่งเวิร์ด เพราะคำสั่งประเภทอื่นมีผลต่อการลดขนาดโค้ดมากกว่า ข้อได้เปรียบด้านขนาดโค้ดของ Thumb-2 เหนือ RV32C ในรูป 1.5 หน้าที่ 11 เกิดจากคำสั่งโหลดและเก็บหลายรายการ **Load and Store Multiple** ที่ช่วยลดขนาดโค้ดขั้นตอนเข้าและออกจากฟังก์ชัน แต่ RV32C ไม่รวมคำสั่งเหล่านี้เพื่อรักษาการแมปแบบหนึ่งต่อหนึ่งกับ RV32G และลดความซับซ้อนของการออกแบบฮาร์ดแวร์ระดับสูง เนื่องจาก Thumb-2 เป็น ISA แยกจาก ARM-32 แต่โปรเซสเซอร์สามารถสลับกันได้ ฮาร์ดแวร์จึงต้องมีตัวถอดรหัสคำสั่ง 2 ตัว สำหรับ ARM-32 และอีกหนึ่งตัวสำหรับ Thumb-2 แต่ RV32GC เป็น ISA เดียว ดังนั้นโปรเซสเซอร์ RISC-V ต้องการตัวถอดรหัสเพียงตัวเดียว

■ รายละเอียดเพิ่มเติม: ทำไมสถาปนิกถึงข้าม RV32C?

การถอดรหัสคำสั่งอาจเป็นคอขวด (bottleneck) ในโปรเซสเซอร์ซูเปอร์สเกลาร์ ที่ต้องดึงคำสั่งหลายคำสั่งต่อรอบสัญญาณนาฬิกา อีกเทคนิคหนึ่งคือ มาโครฟิวชัน *macrofusion* ซึ่งตัวถอดรหัสคำสั่งจะรวมหลายคำสั่ง RISC-V เข้าด้วยกันเพื่อสร้างคำสั่งที่มีประสิทธิภาพมากกว่าก่อนนำไปประมวลผล (ดูบท 1) การผสมกันระหว่างคำสั่ง RV32C 16 บิต และคำสั่ง RV32I 32 บิต อาจทำให้การถอดรหัสที่ซับซ้อนทำได้ภายใน 1 รอบสัญญาณนาฬิกา ในโปรเซสเซอร์ประสิทธิภาพสูง

7.2 การเปรียบเทียบ RV32GC, Thumb-2, microMIPS และ x86-32

รูปที่ 7.2 ตารางสรุปขนาดของ Insertion Sort และ DAXPY สำหรับ ISA ทั้ง 4 แบบ จากคำสั่ง RV32I เดิม 19 คำสั่ง ใน Insertion Sort มี 12 คำสั่ง ที่กลายเป็น RV32C



ค่าใช้จ่าย

ทำให้ขนาดโค้ดลดลงจาก $19 \times 4 = 76$ ไบต์เหลือ $12 \times 2 + 7 \times 4 = 52$ ไบต์ ประหยัดได้ $24/76 = 32\%$. สำหรับ DAXPY ขนาดโค้ดลดลงจาก $11 \times 4 = 44$ ไบต์เหลือ $8 \times 2 + 3 \times 4 = 28$ ไบต์ หรือประหยัดได้ $16/44 = 36\%$

ผลลัพธ์จากตัวอย่างขนาดเล็กทั้งสองตัวอย่างนี้ สอดคล้องกันอย่างน่าประหลาดใจกับรูป 1.5 ในหน้า 11 บทที่ 1 ซึ่งแสดงว่าโค้ด RV32G มีขนาดใหญ่กว่าโค้ด RV32GC ประมาณ 37% เมื่อพิจารณาจากชุดโปรแกรมที่มีขนาดใหญ่ และจำนวนมากกว่า เพื่อให้ได้การประหยัดขนาดโค้ดในระดับนี้ มากกว่าครึ่งหนึ่งของคำสั่งในโปรแกรมต้องเป็นคำสั่งแบบ RV32C

■ รายละเอียดเพิ่มเติม: RV32C มีความเป็นเอกลักษณ์จริงหรือไม่?

คำสั่ง RV32I ไม่สามารถแยกออกได้จาก RV32IC กล่าวคือ การใช้ RV32C ไม่เปลี่ยนความหมายของคำสั่งเดิม เปรียบเทียบกับ Thumb-2 เป็น ISA แยกต่างหาก ประกอบด้วยคำสั่ง 16 บิต และคำสั่งส่วนใหญ่ (แต่ไม่ทั้งหมด) ของ ARMv7 ตัวอย่างเช่น คำสั่ง *Compare and Branch on Zero* มีใน Thumb-2 แต่ไม่มีใน ARMv7 ในขณะที่คำสั่ง *Reverse Subtract with Carry* มีใน ARMv7 แต่ไม่มีใน Thumb-2 เปรียบเทียบกับ microMIPS32 ก็ไม่ใช่ซูเปอร์เซตของ MIPS32 ตัวอย่างเช่น microMIPS คุณระยการกระโดดด้วย 2 แต่ใน MIPS32 จะคูณด้วย 4 ใน RISC-V จะคูณระยการกระโดดด้วย 2 เสมอ

7.3 ข้อสังเกตสรุป

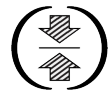
ฉันจะเขียนจดหมายที่สั้นกว่านี้ได้ แต่ฉันไม่มีเวลามากพอ

—แบลส ปาสกาล, 1656.

เขาเป็นนักคณิตศาสตร์ผู้สร้างเครื่องคำนวณเชิงกลรุ่นแรก ซึ่งเป็นแรงบันดาลใจให้ Niklaus Wirth ผู้ได้รับรางวัล Turing Award นำชื่อของเขามาตั้งเป็นชื่อภาษาโปรแกรม

RV32C ทำให้ RISC-V มีขนาดโค้ดเล็กที่สุดในบรรดาสถาปัตยกรรมปัจจุบัน เกือบจะถือได้ว่า RV32C เป็นเสมือนคำสั่งเทียมที่มีฮาร์ดแวร์ช่วยรองรับ อย่างไรก็ตาม แอสเซมเบลอร์จะซ่อนคำสั่งเหล่านี้ จากโปรแกรมเมอร์ภาษาแอสเซมบลี และนักเขียนคอมไพเลอร์ ซึ่งแตกต่างจากใน Chapter 3 ที่คำสั่งเทียมถูกใช้เพื่อขยายชุดคำสั่งจริง ด้วยการดำเนินการยอदनนิยมที่ทำให้โค้ด RISC-V ใช้งานง่ายและอ่านเข้าใจได้ดีขึ้น ซึ่งทั้งสองแนวทางนี้ ล้วนช่วยเพิ่มประสิทธิภาพการทำงานของโปรแกรมเมอร์

เราถือว่า RV32C เป็นหนึ่งในตัวอย่างที่ดีที่สุดของ RISC-V ในการออกแบบที่ “เรียบง่ายแต่ทรงพลัง” ซึ่งช่วยเพิ่มประสิทธิภาพต่อต้นทุน



ขนาดโปรแกรม



ส่างาน

7.4 เรียนรู้เพิ่มเติม

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


```

# RV32C (19 คำสั่ง, 52 bytes)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
0: 00450693 addi a3,a0,4 # a3 เป็นตัวชี้ไปที่ a[i]
4: 4705 c.li a4,1 # (ขยายเป็น addi a4,x0,1) i = 1
Outer Loop:
6: 00b76363 bltu a4,a1,c # ถ้า i < n กระโดดไปที่ Continue Outer loop
a: 8082 c.ret # (ขยายเป็น jalr x0,ra,0) ออกจากฟังก์ชัน
Continue Outer Loop:
c: 0006a803 lw a6,0(a3) # x = a[i]
10: 8636 c.mv a2,a3 # (ขยายเป็น add a2,x0,a3) a2 เป็นตัวชี้ไปที่ a[j]
12: 87ba c.mv a5,a4 # (ขยายเป็น add a5,x0,a4) j = i
InnerLoop:
14: ffc62883 lw a7,-4(a2) # a7 = a[j-1]
18: 01185763 ble a7,a6,26 # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit InnerLoop
1c: 01162023 sw a7,0(a2) # a[j] = a[j-1]
20: 17fd c.addi a5,-1 # (ขยายเป็น addi a5,a5,-1) j--
22: 1671 c.addi a2,-4 # (ขยายเป็น addi a2,a2,-4) ลด a2 ชี้ไปที่ a[j]
24: fbe5 c.bnez a5,14 # (ขยายเป็น bne a5,x0,14) ถ้า j != 0 กระโดดไปที่ InnerLoop
Exit InnerLoop:
26: 078a c.slli a5,0x2 # (ขยายเป็น slli a5,a5,0x2) คูณ a5 by 4
28: 97aa c.add a5,a0 # (ขยายเป็น add a5,a5,a0) a5 = byte address of a[j]
2a: 0107a023 sw a6,0(a5) # a[j] = x
2e: 0705 c.addi a4,1 # (ขยายเป็น addi a4,a4,1) i++
30: 0691 c.addi a3,4 # (ขยายเป็น addi a3,a3,4) เพิ่ม a3 ชี้ไปที่ a[i]
32: bfd1 c.j 6 # (ขยายเป็น jal x0,6) กระโดดไปที่ Outer Loop

```

รูปที่ 7.3: โค้ด RV32C สำหรับการเรียงลำดับการแทรก ที่รูป 2.5 ใช้คำสั่ง 16 บิต จำนวน 12 คำสั่ง ทำให้โค้ดเล็กลง 32% ขนาดของคำสั่งแต่ละคำสั่ง ดูได้จากจำนวนหลักเลขฐานสิบหกในคอลัมน์ที่สอง คำสั่ง RV32C (ที่เริ่มต้นด้วย c.) ถูกแสดงให้เห็นอย่างชัดเจน แต่โดยปกติแล้ว โปรแกรมเมอร์และคอมไพเลอร์จะไม่มองเห็นคำสั่งเหล่านี้

```

# RV32DC (11 คำสั่ง, 28 bytes)
# a0 คือ n, a1 เป็นตัวชี้ไปที่ x[0], a2 เป็นตัวชี้ไปที่ y[0], fa0 คือ a
0: cd09 c.beqz a0,1a # (ขยายเป็น beq a0,x0,1a) ถ้า n==0, กระโดดไปที่ Exit
2: 050e c.slli a0,a0,0x3 # (ขยายเป็น slli a0,a0,0x3) a0 = n*8
4: 9532 c.add a0,a2 # (ขยายเป็น add a0,a0,a2) a0 = ที่อยู่ของ y[n]
Loop:
6: 2218 c.fld fa4,0(a2) # (ขยายเป็น fld fa4,0(a2)) fa5 = x[]
8: 219c c.fld fa5,0(a1) # (ขยายเป็น fld fa5,0(a1)) fa4 = y[]
a: 0621 c.addi a2,8 # (ขยายเป็น addi a2,a2,8) a2++ (เพิ่มตัวชี้ไปที่ y)
c: 05a1 c.addi a1,8 # (ขยายเป็น addi a1,a1,8) a1++ (เพิ่มตัวชี้ไปที่ 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 # ถ้า i != n กระโดดไปที่ Loop
Exit:
1a: 8082 ret # (ขยายเป็น jalr x0,ra,0) ออกจากฟังก์ชัน

```

รูปที่ 7.4: โค้ด RV32DC สำหรับ DAXPY ที่รูป 2.5 คำสั่งขนาด 16 บิต จำนวน 8 คำสั่ง ช่วยลดขนาดโค้ดลงประมาณ 36% ความกว้างของคำสั่งแต่ละคำ: สามารถสังเกตได้จากจำนวนหลักของเลขฐานสิบหก ในคอลัมน์ที่สอง คำสั่ง RV32C (เริ่มต้นด้วย c.) ถูกแสดงให้เห็นอย่างชัดเจนในตัวอย่างนี้ แต่โดยปกติโปรแกรมเมอร์และคอมไพเลอร์ภาษาแอสเซมบลีจะไม่มองเห็นคำสั่งเหล่านี้

15	13	12	11	7	6	5	4	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

รูปที่ 7.5: แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 01) แสดงโครงสร้าง opcodes

รูปแบบคำสั่งและชื่อคำสั่ง rd', rs1' และ rs2' หมายถึง 10 รีจิสเตอร์ยอดนียม

a0-a5, s0-s1, sp, และ ra (ตารางที่ 12.5 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

15	13	12	10	9	7	6	5	4	2	1	0	
000	0						0		00		CIW <i>Illegal instruction</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	

รูปที่ 7.6: แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 00) แสดงโครงสร้าง opcodes

รูปแบบคำสั่งและชื่อคำสั่ง rd', rs1' และ rs2' อ้างถึง 10 รีจิสเตอร์ยอดนียม a0-a5, s0-s1, sp และ ra (ตารางที่ 12.4 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

15	13	12	11	7	6	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

รูปที่ 7.7: แผนผังรหัสดำเนินการ RV32C (bits[1 : 0] = 10) แสดงรายการโครงสร้าง opcodes รูปแบบและชื่อ (ตารางที่ 12.6 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

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

รูปที่ 7.8: รูปแบบคำสั่งแบบบีบอัด RVC 16 บิต rd', rs1' และ rs2' หมายถึง 10 รีจิสเตอร์ยอดนิยาม a0-a5, s0-s1, sp และ ra (ตารางที่ 12.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

ซีมัวร์ เครย์ (1925–1996) เป็นสถาปนิกผู้ออกแบบ Cray-1 ในปี 1976 ซึ่งเป็นซูเปอร์คอมพิวเตอร์เชิงพาณิชย์เครื่องแรก que ประสบความสำเร็จโดยใช้สถาปัตยกรรมแบบเวกเตอร์ Cray-1 ถือเป็นผลงานชิ้นเอก โดยเป็นคอมพิวเตอร์ที่เร็วที่สุดในโลก แม้กระทั่งในกรณีที่ไม่ต้องใช้คำสั่งเวกเตอร์เลยก็ตาม



ประสิทธิภาพ

ส่วนขยายมัลติมีเดียของ Intel (MMX) ในปี 1997 ทำให้ SIMD ได้รับความนิยม มันถูกยอมรับและขยายต่อผ่าน Streaming SIMD Extensions (SSE) ในปี 1999 และ Advanced Vector Extensions (AVX) ในปี 2010 ความสำเร็จของ MMX ได้รับแรงหนุนจากแคมเปญโฆษณาของ Intel ที่แสดงภาพพนักงานสายการผลิตเขมิกอนดักเตอร์เดินดิสก์ไนซุตสะอาดสีสันสดใส (<https://www.youtube.com/watch?v=5zyjSBSvqPc>)

ฉันชอบความเรียบง่ายมาก ถ้ามันซับซ้อนมาก ฉันก็ไม่สามารถเข้าใจมันได้

—ซีมัวร์ เครย์

8.1 บทนำ

บทนี้มุ่งเน้นที่ *การประมวลผลแบบขนานในระดับข้อมูล* (data-level parallelism) ซึ่งเป็นรูปแบบของการประมวลผลที่มีข้อมูลจำนวนมาก ให้โปรแกรมสามารถคำนวณได้พร้อมกัน (concurrently) อาร์เรย์เป็นตัวอย่างที่พบได้บ่อย แม้แนวคิดนี้จะเป็นพื้นฐานสำคัญในงานด้านวิทยาศาสตร์ แต่โปรแกรมมัลติมีเดียก็ใช้อาร์เรย์เช่นกัน โดย งานทางวิทยาศาสตร์มักใช้ข้อมูลแบบจำนวนลอยตัว ทั้งแบบแบบความแม่นยำเดียวและความแม่นยำสองเท่า ส่วนงานมัลติมีเดียมักใช้ข้อมูลจำนวนเต็มขนาด 8 บิต และ 16 บิต สถาปัตยกรรมที่เป็นที่รู้จักมากที่สุดสำหรับการประมวลผลแบบขนานในระดับข้อมูล คือ SIMD (Single Instruction, Multiple Data) SIMD เริ่มได้รับความนิยมจากการแบ่งรีจิสเตอร์ขนาด 64 บิต ออกเป็นส่วนย่อยหลายส่วน เช่น 8 บิต, 16 บิต หรือ 32 บิต แล้วทำการคำนวณแต่ละส่วนพร้อมกันแบบขนาน รหัสดำเนินการ (opcode) จะระบุทั้งขนาดของข้อมูลและชนิดของการดำเนินการ การถ่ายโอนข้อมูลทำได้โดยการโหลดและเก็บข้อมูลของรีจิสเตอร์ SIMD แบบกว้างเพียงตัวเดียว

ขั้นตอนแรกของการแบ่งรีจิสเตอร์ 64 บิตเดิมออกเป็นส่วนย่อยนั้นน่าสนใจ เพราะทำได้ตรงไปตรงมา แต่เพื่อให้ SIMD ทำงานได้เร็วขึ้น นักออกแบบสถาปัตยกรรมจึงขยายขนาดรีจิสเตอร์ให้กว้างขึ้น เพื่อให้สามารถประมวลผลส่วนย่อยได้มากขึ้นพร้อมกัน เนื่องจาก ISA แบบ SIMD อยู่ในแนวทางการออกแบบแบบเพิ่มขยาย (incremental design) และ opcode เป็นตัวกำหนดขนาดข้อมูล การขยายขนาดรีจิสเตอร์จึงทำให้ต้องขยายชุดคำสั่ง SIMD ไปด้วย ทุกครั้งที่มีการเพิ่มขนาดรีจิสเตอร์เป็นสองเท่า รวมถึงเพิ่มจำนวนคำสั่ง SIMD ทำให้สถาปัตยกรรมชุดคำสั่งมีความซับซ้อนมากขึ้นอย่างต่อเนื่อง ซึ่งภาระความซับซ้อนนี้ตกอยู่กับผู้ออกแบบโปรเซสเซอร์ นักพัฒนาคอมไพเลอร์ และโปรแกรมเมอร์แอสเซมบลี

อีกแนวทางหนึ่งที่เก่ากว่าและ (ตามความเห็นของผู้เขียน) สง่างามกว่าในการใช้ประโยชน์จากการประมวลผลแบบขนานระดับข้อมูล คือ *สถาปัตยกรรมเวกเตอร์* บทนี้จะอธิบายเหตุผลว่าทำไม RISC-V จึงเลือกใช้เวกเตอร์แทน SIMD

คอมพิวเตอร์แบบเวกเตอร์จะรวบรวมข้อมูลจากหน่วยความจำหลัก (main memory) แล้วนำมาเก็บไว้ในรีจิสเตอร์เวกเตอร์ที่มีความยาวและต่อเนื่อง จากนั้นหน่วยประมวลผลแบบ pipeline ที่ทำงานขนานกันจะคำนวณบนรีจิสเตอร์เวกเตอร์เหล่านี้ได้อย่างมีประสิทธิภาพสูง แล้วจึงกระจาย (scatter) ผลลัพธ์กลับจากรีจิสเตอร์เวกเตอร์ไปยังหน่วยความจำหลักอีกครั้ง ขนาดของรีจิสเตอร์เวกเตอร์ถูกกำหนดโดยการออกแบบฮาร์ดแวร์ (implementation) ไม่ได้ถูกกำหนดตายตัวในรหัสดำเนินการ (opcode) เหมือนกับ SIMD แนวคิดสำคัญของสถาปัตยกรรมเวกเตอร์คือ การแยก *ความยาวเวกเตอร์และจำนวนการดำเนินการสูงสุดต่อรอบนาฬิกา* ออกจากรูปแบบของการเข้ารหัสคำสั่ง ผู้ออกแบบไมโครสถาปัตยกรรมสามารถออกแบบฮาร์ดแวร์แบบขนานได้อย่างยืดหยุ่น โดยไม่กระทบต่อโปรแกรมเมอร์ และโปรแกรมเมอร์สามารถใช้เวกเตอร์ที่ยาวขึ้นได้ โดยไม่ต้องเขียนโค้ดใหม่ ดังนั้น โค้ดไบนารีสามารถทำงานระหว่างการออกแบบเวกเตอร์ที่ต่างกันได้ และโปรแกรมสามารถปรับตัวตามขนาดเวกเตอร์โดยอัตโนมัติ ซึ่งแตกต่างจาก SIMD นอกจากนี้ สถาปัตยกรรมเวกเตอร์ยังมีคำสั่งน้อยกว่าสถาปัตยกรรมแบบ SIMD มาก และรองรับรูปแบบการคอมไพล์ที่เรียบง่ายและสวยงามกว่า

บทนี้มีการเปลี่ยนแปลงมากที่สุดเมื่อเทียบกับบทอื่น ๆ ในการพิมพ์ฉบับที่สอง เนื่องจาก RV32V ยังอยู่ระหว่างการพัฒนาในปี 2017 ข้อดีอย่างหนึ่งของชุดคำสั่งแบบเปิด คือ นักออกแบบสถาปัตยกรรมจำนวนมากสามารถมีส่วนร่วมในการออกแบบ และนำมุมมองที่แตกต่างกันเกี่ยวกับกลยุทธ์การออกแบบฮาร์ดแวร์มารวมอภิปราย ซึ่งมุมมองที่หลากหลายเหล่านี้นำไปสู่การปรับปรุงสำคัญ ทำให้ vector ISA ใช้งานได้ง่ายขึ้นและมีประสิทธิภาพมากขึ้น รูปที่ 8.1 แสดงภาพรวมของคำสั่งหลักใน RV32V

สถาปัตยกรรมแบบเวกเตอร์พบได้น้อยกว่าสถาปัตยกรรม SIMD ดังนั้นผู้อ่านจำนวนมากจึงอาจไม่คุ้นเคยกับสถาปัตยกรรมเวกเตอร์มากนัก (หากต้องการศึกษาเพิ่มเติมเกี่ยวกับสถาปัตยกรรมเวกเตอร์ แนะนำให้อ่านบทที่ 4 และภาคผนวก G ของ [Hennessy and Patterson 2019]) ด้วยเหตุนี้ บทนี้จึงมีลักษณะเป็นเชิงการสอน มากกว่าบทก่อนหน้า RV32V มีคุณสมบัติใหม่ที่จะช่วยลดจำนวนคำสั่งใน ISA ซึ่งต้องอธิบายเพิ่มเติม แม้ว่าผู้อ่านจะคุ้นเคยกับสถาปัตยกรรมเวกเตอร์อยู่แล้วก็ตาม

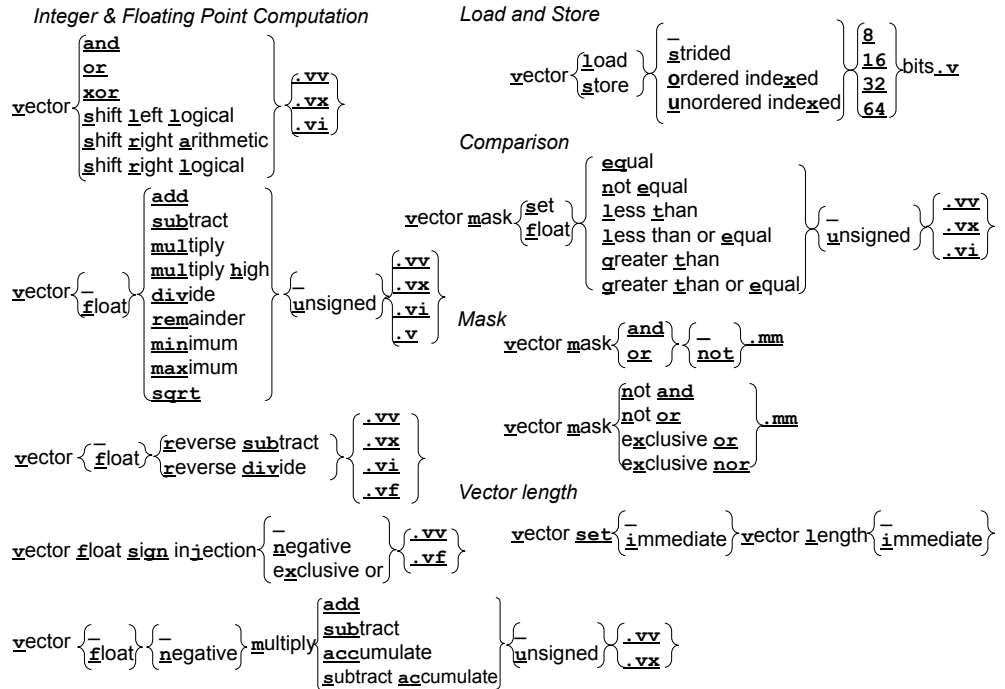
จากเป้าหมายเหล่านี้ ทำให้บทนี้มีความยาวมากที่สุด จึงมีการสรุปภาพรวมของบทเพื่อเป็นแนวทางให้กับผู้อ่าน ดังนี้

- หัวข้อ 8.2 ทบทวนคำสั่งพื้นฐานสำหรับการคำนวณแบบเวกเตอร์ สำหรับข้อมูลจำนวนเต็มและจำนวนลอยตัว
- หัวข้อ 8.3 อธิบายรีจิสเตอร์เวกเตอร์และแนวคิดใหม่ของ dynamic typing ความกว้าง (width) ไม่ได้ถูกกำหนดโดยรหัสดำเนินการ เหมือนในสถาปัตยกรรมเวกเตอร์ส่วนใหญ่ โดยมีคุณสมบัติสำคัญของ RISC-V vector 2 อย่างได้แก่ การกำหนดความยาวเวกเตอร์ในแต่ละลูปด้วยคำสั่ง `vsetvli`—เพื่อลดภาระงานในการจัดการ (overhead)—และการรวมรีจิสเตอร์เวกเตอร์ด้วยคำสั่ง `lmul`—เพื่อนำรีจิสเตอร์หลายตัวมารวมกันเป็นรีจิสเตอร์ที่ยาวขึ้นได้ ในกรณีที่โปรแกรมไม่จำเป็นต้องใช้รีจิส



ความเรียบง่าย

RV32V



รูปที่ 8.1: แผนภาพของคำสั่งเวกเตอร์ RV32V ที่ได้รับความนิยมมากที่สุด แม้ว่าโค้ดส่วนใหญ่จะใช้เพียงคำสั่งหลักไม่กี่แบบ แต่รูปที่ 8.5 ถึง รูปที่ 8.8 แสดงรูปแบบคำสั่งทั้งหมดเช่นเดียววก่อนหน้า รูปนี้แสดงชุดคำสั่งที่ครอบคลุมของ RV32V โดยที่ไม่รวมทุกการผสมของคำสั่งที่เป็นไปได้ ตัวอย่างเช่น มีคำสั่ง `vfadd` และ `vaddu` แต่ไม่มีคำสั่ง `vfaddu` เพราะการคำนวณจำนวนลอยตัวแบบไม่มีเครื่องหมายนั้นไม่มีความหมาย เนื่องจากใช้แนวคิดการกำหนดประเภทข้อมูลแบบไดนามิก (dynamic typing) แผนภาพคำสั่งนี้จึงสามารถใช้กับ RV64V ได้โดยไม่ต้องเปลี่ยนแปลง ในบท 10

เตอร์ทั้ง 32 ตัว เราจะเห็นข้อดีของแนวคิดเหล่านี้จากตัวอย่าง DAXPY ในหัวข้อ 8.8

- หัวข้อ 8.4 แนะนำคำสั่งหลักสำหรับการย้ายข้อมูล ซึ่งรองรับการเข้าถึงแบบต่อเนื่อง (unit stride), แบบมีระยะห่าง (strided) และแบบอ้างอิงตามดัชนี indexed เช่นเดียวกับสถาปัตยกรรมเวกเตอร์ทั่วไป โดยมีนวัตกรรมคือ การอนุญาตให้การดำเนินการแบบอ้างอิงดัชนีที่ทำงานได้ทั้งแบบมีลำดับ (ordered) และไม่มีลำดับ (un-ordered)
- หัวข้อ 8.5 ชี้ให้เห็นว่า สถาปัตยกรรมเวกเตอร์สามารถรองรับการคำนวณข้อมูลหลายขนาด และขยายความยาวเวกเตอร์ได้ โดยไม่ต้องเปลี่ยนการเข้ารหัสคำสั่ง ซึ่งแตกต่างจากสถาปัตยกรรม SIMD
- หัวข้อ 8.6 แสดงวิธีที่ สถาปัตยกรรมแบบเวกเตอร์จัดการเงื่อนไขโดยใช้รีจิสเตอร์มาส์ก
- หัวข้อ 8.7 อธิบายว่า สถาปัตยกรรมคำสั่งเวกเตอร์ RISC-V ช่วยลดภาระงานส่วนเกินจากการแบ่งข้อมูลออกเป็นช่วง (stripmining) ซึ่งเป็นปัญหาที่พบได้บ่อยในสถาปัตยกรรมเวกเตอร์และ SIMD
- หัวข้อ 8.8 แสดงตัวอย่างโค้ด DAXPY แบบเวกเตอร์ ซึ่งใช้จำนวนคำสั่งน้อยกว่าแบบสเกลาร์ (scalar) มาก
- ผู้อ่านอาจเลือกข้ามรายละเอียดในหลายหน้าถัดไป แล้วไปอ่านสรุปในตอนท้ายในหัวข้อ 8.13 โดยตรง สำหรับผู้อ่านที่สนใจศึกษาสถาปัตยกรรมชุดคำสั่งเวกเตอร์ในเชิงลึกมากขึ้น จะมีหัวข้อ 8.9 อธิบายคำสั่งการขยาย (widening) และการย่อขนาด (narrowing) สำหรับกรณีที่ขนาดของข้อมูลต้นทางและปลายทางไม่เท่ากัน
- หัวข้อ 8.10 อธิบายรูปแบบต่างๆ ของคำสั่งการย้ายข้อมูลในรีจิสเตอร์เวกเตอร์, การขยายจำนวนเต็ม และคำสั่งแปลงค่าจำนวนทศนิยม
- หัวข้อ 8.11 หัวข้อสุดท้ายจะกล่าวถึงคำสั่งอื่น ๆ ที่ใช้งานไม่บ่อย ซึ่งมักใช้ในไลบรารีที่เขียนด้วยมือ หรือช่วยแก้ปัญหายาก ๆ ให้กับคอมไพเลอร์ เช่น ปัญหาที่ซับซ้อน: การรวมค่าหลายค่าเป็นค่าเดียว (reductions), การจัดการมาส์ก, การจัดเรียง/เข้าถึงข้อมูลในรีจิสเตอร์เวกเตอร์, การคำนวณแบบจุดคงตัว และคำสั่งย้ายข้อมูลพิเศษ
- สำหรับผู้ชื่นชอบ ISA หัวข้อ 8.12 เปรียบเทียบผลลัพธ์โค้ด DAXPY ระหว่าง x86 AVX2 กับ SIMD 128 บิต, x86 AVX-512 กับ SIMD 512 บิต, สถาปัตยกรรมเวกเตอร์ ARMv8 SVE และ RV64V
- หัวข้อ 8.13 สรุปด้วยการสะท้อนถึงวิวัฒนาการจาก SIMD ไปสู่สถาปัตยกรรมชุดคำสั่งเวกเตอร์

V ใน RISC-V ยังหมายถึงเวกเตอร์ (Vector) อย่างไม่เป็นทางการสถาปนิกของ RISC-V มีประสบการณ์ในเชิงบวกอย่างกว้างขวางกับสถาปัตยกรรมแบบเวกเตอร์ และรู้สึกผิดหวังที่ SIMD ครอบงำไม่โครโปรเซสเซอร์ ดังนั้น V จึงย่อมาจากโครงการ RISC ที่ทำของเบิร์กลีย์ และเพราะสถาปัตยกรรมชุดคำสั่ง RISC-V จะเน้นที่เวกเตอร์

8.2 คำสั่งคำนวณเวกเตอร์

คำสั่งเวกเตอร์ทั้งหมดจะขึ้นต้นด้วยตัวอักษร `v` แทบทุกคำสั่งคำนวณทั้งแบบจำนวนเต็มและแบบจุดลอยตัว (floating-point) จากบทก่อนหน้านี้ มีเวอร์ชันแบบเวกเตอร์ให้ใช้งาน โดยรูป 8.1 ได้รวมการดำเนินการจาก RV32I, RV32M, RV32H, RV32F และ RV32D คำสั่งเวกเตอร์แต่ละคำสั่งมีหลายรูปแบบ ขึ้นอยู่กับชนิดของตัวถูกดำเนินการต้นทาง ได้แก่ ใช้รีจิสเตอร์เวกเตอร์ทั้งหมด (`.vv` ต่อท้ายคำสั่ง) ใช้รีจิสเตอร์เวกเตอร์และรีจิสเตอร์จำนวนเต็มแบบสเกลาร์ (`.vx` ต่อท้าย) ใช้รีจิสเตอร์เวกเตอร์และค่าคงที่ (`.vi` ต่อท้าย) ใช้รีจิสเตอร์เวกเตอร์และรีจิสเตอร์จุดลอยตัวแบบสเกลาร์ (`.vf` ต่อท้าย) ตัวอย่างเช่น ในโปรแกรม DAXPY (รูป 5.5 หน้าที่ 63 ในบท 5) แสดงการคำนวณสมการ $Y = a \times X + Y$ โดยที่ X และ Y เป็นเวกเตอร์ และ a เป็นค่าสเกลาร์ ดังนั้นจึงใช้คำสั่ง `vfmacc.vf` สำหรับการดำเนินการแบบเวกเตอร์และรีจิสเตอร์สเกลาร์ ฟิลด์ `rs1` จะใช้ระบุรีจิสเตอร์สเกลาร์ที่ต้องการใช้งาน ส่วนในกรณีการดำเนินการแบบเวกเตอร์กับค่าคงที่ ฟิลด์ `rs1` จะเข้ารหัสเป็นค่าคงที่ (immediate) ขนาด 5 บิต

ในรูปแบบคำสั่งของคำสั่งเวกเตอร์ส่วนใหญ่ ตำแหน่งของรีจิสเตอร์ต้นทางตัวที่หนึ่ง และตัวที่สองจะถูกสลับกัน การกำหนดให้ที่อยู่ (address) อยู่ในรีจิสเตอร์สเกลาร์ ที่ระบุผ่านฟิลด์ `rs1` ช่วยทำให้การออกแบบ pipeline ของหน่วยความจำในฮาร์ดแวร์ทำได้ง่ายขึ้น สำหรับการดำเนินการแบบเวกเตอร์กับสเกลาร์นั้น ต้องการให้ ตัวถูกดำเนินการตัวที่สองเป็นรีจิสเตอร์สเกลาร์ ดังนั้นฟิลด์ `rs1` จะกำหนดให้เป็นตัวถูกดำเนินการตัวที่สอง และฟิลด์ `rs2` จะระบุตัวถูกดำเนินการตัวที่หนึ่ง

แม้ว่าค่าคงที่ (immediate) ที่มีขนาดใหญ่กว่านี้จะมีประโยชน์ แต่รูปแบบคำสั่งเวกเตอร์ที่มีพื้นที่จำกัด จึงทำให้ค่าคงที่ถูกจำกัดไว้เพียง 5 บิต ซึ่งใช้พื้นที่เดียวกับฟิลด์ระบุรีจิสเตอร์ โดยทั่วไป ค่าคงที่จะถูกขยายเครื่องหมาย (sign-extended) ยกเว้นในกรณีที่ใช้แทนค่าจำนวนสำหรับการเลื่อนบิต (shift amounts) หรือระบุตำแหน่งขององค์ประกอบ ซึ่งจะใช้การขยายศูนย์ (zero-extended) ในทุกกรณี ฟิลด์ `rs2` จะเป็นรีจิสเตอร์เวกเตอร์ต้นทาง และ `rd` เป็นรีจิสเตอร์เวกเตอร์ปลายทาง เพื่อประหยัดพื้นที่ของ opcode คำสั่งการคูณแล้วบวกสะสม (multiply-accumulate) ใน RVV ถูกออกแบบให้เป็นแบบเขียนทับค่าเดิม (destructive) ดังนั้น `rd` จะทำหน้าที่เป็นตัวถูกดำเนินการต้นทางตัวที่สองไปด้วยในตัว

การดำเนินการที่ไม่สมมาตรบางอย่าง เช่น การลบและการหาร มีรูปแบบคำสั่งที่ตัวถูกดำเนินการแรกเป็นสเกลาร์และตัวที่สองเป็นเวกเตอร์ คำสั่งเหล่านี้มีคำนำหน้า `vr` หรือ `vfr` ซึ่งบ่งบอกถึงการดำเนินการแบบย้อนกลับ

ผู้อ่านอาจสงสัยได้ว่า คำสั่งส่วนใหญ่ในรูป 8.1 ไม่ได้ระบุขนาดขององค์ประกอบที่ใช้ในการคำนวณ หัวข้อถัดไปจะอธิบายเหตุผลของเรื่องนี้

■ รายละเอียดเพิ่มเติม: คำสั่งเวกเตอร์สามารถมองได้ว่า เทียบเท่ากับการลูป (loop) ในภาษา C

ตัวอย่างเช่น โค้ดต่อไปนี้มีคําศัพท์เทียบเท่ากับคำสั่ง `vadd.vv v1, v2, v3`

```
for (i=0; i<v1; i++)
    v1[i] = v2[i] + v3[i]
```

8.3 รีจิสเตอร์เวกเตอร์และการกำหนดชนิดข้อมูลแบบไดนามิก

RV32V เพิ่มรีจิสเตอร์เวกเตอร์ 32 ตัว ซึ่งชื่อของรีจิสเตอร์เวกเตอร์จะขึ้นต้นด้วยตัวอักษร `v` จำนวนบิตของรีจิสเตอร์เวกเตอร์ เรียกว่า `VLEN` (Vector Length) จะแตกต่างกันไป

ในแต่ละเครื่อง โดยสำหรับคอมพิวเตอร์ RISC-V ทั่วไปในปัจจุบัน ค่า VLEN มักอยู่ในช่วงประมาณ 128 ถึง 512 บิต ความกว้างของรีจิสเตอร์เวกเตอร์จะเป็นตัวกำหนดขีดจำกัดสูงสุดของจำนวนการดำเนินการ ที่สามารถทำได้ต่อหนึ่งคำสั่งเวกเตอร์ ดังนั้นผู้ออกแบบโปรเซสเซอร์จะเลือกค่า VLEN โดยพิจารณาจากพื้นที่ของชิป ที่สามารถจัดสรรให้กับรีจิสเตอร์เวกเตอร์ และเป้าหมายด้านประสิทธิภาพของระบบ ตัวอย่างเช่น หากโปรเซสเซอร์จัดสรรพื้นที่ 1 KiB สำหรับรีจิสเตอร์เวกเตอร์ จะได้ว่า VLEN จะเท่ากับ 256 และรีจิสเตอร์เวกเตอร์ทั้ง 32 ตัวจะมีความกว้างเพียงพอสำหรับเก็บ ข้อมูลขนาด 64 บิต ได้ 4 ค่า, ข้อมูลขนาด 32 บิต ได้ 8 ค่า, ข้อมูลขนาด 16 บิต ได้ 16 ค่า, ข้อมูลขนาด 8 บิต ได้ 32 ค่า

เพื่อให้ความกว้างของรีจิสเตอร์เวกเตอร์มีความยืดหยุ่นใน vector ISA โปรเซสเซอร์เวกเตอร์จะคำนวณ *ความยาวเวกเตอร์สูงสุด* (maximum vector length: VLMAX) ที่โปรแกรมสามารถใช้เพื่อให้ทำงานได้อย่างถูกต้องบนโปรเซสเซอร์ ที่มีความกว้างรีจิสเตอร์เวกเตอร์แตกต่างกัน ความยาวนี้ถูกกำหนดในรูปของ จำนวนองค์ประกอบ (elements) รีจิสเตอร์ความยาวเวกเตอร์ (v1) ใช้กำหนดจำนวนองค์ประกอบของเวกเตอร์สำหรับการดำเนินการแต่ละครั้ง ซึ่งช่วยให้โปรแกรมจัดการกรณีที่ขนาดของอาร์เรย์ไม่เป็นจำนวนเท่าของ VLMAX ได้อย่างถูกต้อง v1 เป็นรีจิสเตอร์เชิงสถาปัตยกรรม (architectural register) แยกต่างหาก ที่ใช้ควบคุมการทำงานของคำสั่งเวกเตอร์ เราจะอธิบายรายละเอียดเกี่ยวกับ VLMAX และ v1 เพิ่มเติมในหัวข้อถัดไป

RV32V ใช้แนวทางใหม่ โดยเลือกขนาดขององค์ประกอบเวกเตอร์แบบไดนามิก แทนการกำหนดไว้คงที่ในรหัสดำเนินการ (opcode) ก่อนการคำนวณเวกเตอร์ โปรแกรมจะตั้งค่า SEW (selected element width: SEW) ซึ่งระบุว่าแต่ละองค์ประกอบมีขนาดเท่าไร ตัวอย่างที่ VLEN เท่ากับ 256 ถ้า SEW เท่ากับ 8 จะได้องค์ประกอบเวกเตอร์ เท่ากับ 32 องค์ประกอบ แต่ถ้า SEW เป็น 64 องค์ประกอบเวกเตอร์ เท่ากับ 4 องค์ประกอบ การกำหนดชนิดข้อมูลแบบไดนามิก (dynamically typing) ช่วยลดจำนวนคำสั่งเวกเตอร์ลงมาก ซึ่งสำคัญมาก เพราะเครื่อง RV32V รองรับจำนวนเต็ม 3 ขนาดและจำนวนลอยตัว 4 ขนาด ดังที่จะได้เห็นในหัวข้อ 8.12 เมื่อเปรียบเทียบกับคำสั่ง SIMD ที่มีจำนวนมาก สถาปัตยกรรมเวกเตอร์ที่ใช้การกำหนดประเภทข้อมูลแบบไดนามิก ช่วยลดภาระด้านความเข้าใจของโปรแกรมเมอร์ภาษาแอสเซมบลี

นอกจาก การอนุญาตให้โปรแกรมเมอร์ เลือก ขนาด ของ องค์ ประกอบ เวก เทอร์ แบบไดนามิกแล้ว RV32V ยังรองรับการจัดกลุ่มรีจิสเตอร์เวกเตอร์ (grouping vector registers) ซึ่งช่วยให้โปรแกรมเมอร์ลดจำนวนรีจิสเตอร์เวกเตอร์ ที่มีผลในทางปฏิบัติเพื่อแลกกับการเพิ่มความยาวของรีจิสเตอร์ ตัวคูณความยาวเวกเตอร์ *vector length multiplier: LMUL* ช่วยให้สามารถใช้พื้นที่รีจิสเตอร์เวกเตอร์ที่มีจำกัดได้อย่างมีประสิทธิภาพมากขึ้นในการคำนวณที่ไม่ต้องการรีจิสเตอร์เวกเตอร์จำนวนมาก เวกเตอร์ที่ยาวขึ้นสามารถเพิ่มประสิทธิภาพได้อย่างมาก โดยเฉพาะในระบบที่มีความหน่วงของหน่วยความจำสูง ค่า LMUL สามารถตั้งได้เป็น 1, 2, 4 หรือ 8 เมื่อ LMUL เท่ากับ 1 จะสามารถใช้รีจิสเตอร์เวกเตอร์ได้ครบทั้ง 32 ตัว และแต่ละตัวมีขนาดเท่ากับ VLEN บิต เมื่อ LMUL เท่ากับ 8 จะ

องค์ประกอบ(elements) สามารถมีความกว้างได้สูงสุดถึง ELEN บิต ในมาตรฐาน RVV กำหนดให้ ELEN = 64 เสมอ แต่ในระบบฝังตัว (embedded systems) สามารถเลือกใช้ ISA เพียงบางส่วน (subset) โดยรองรับเพียง ELEN = 32 ได้

เครื่อง CRAY-1 มีรีจิสเตอร์ความยาวเวกเตอร์ แต่ระบบ IBM System/370-XA เป็นผู้ที่น่าแนวคิด การเขียนโปรแกรมแบบไม่ขึ้นกับ VLMAX มาใช้

ooc
การเขียนโปรแกรม


ประสิทธิภาพ

31	30			8	7	6	5	32	0
vill	Reserved				vma	vta	vsew	vlmul	
1	23				1	1	3	3	

รูปที่ 8.2: รีจิสเตอร์ควบคุมชนิดเวกเตอร์ของ RV32V, `vtype` บิต 3 บิตด้านขวาสุดใช้เข้ารหัสตัวคูณความยาวเวกเตอร์ (LMUL) ในรูปแบบเลขชี้กำลังส่วนเติมเต็มฐานสอง (two's complement exponent) โดยมีความสัมพันธ์ว่า $LMUL = 2^{v_{lmul}}$ บิต 3 บิตถัดมาใช้เข้ารหัสความกว้างขององค์ประกอบที่เลือก (SEW) โดยมีค่าเป็น $SEW = 8 \times 2^{v_{sew}}$ บิต ฟิลด์ถัดไป 2 ฟิลด์ (ฟิลด์ละ 1 บิต) ใช้ระบุว่า องค์ประกอบส่วนท้าย (tail elements `vta`) หรือองค์ประกอบที่ถูกมาสก์ (masked elements `vma`) จะถูกเก็บค่าเดิมไว้ (0) หรือถูกเขียนทับ (1) เมื่อมีการเขียนค่าใหม่ บิตซ้ายสุด คือ `vill` ใช้ระบุว่า `vtype` ถูกตั้งค่าอย่างไม่ถูกต้องหรือไม่ หาก `vill = 1` การพยายามรันคำสั่งเวกเตอร์จะทำให้เกิดข้อยกเว้น (exception)

ค่า LMUL ยังสามารถตั้งเป็นค่าเศษส่วนได้ ระหว่าง $\frac{1}{8}$ ถึง $\frac{1}{2}$ ซึ่งจะทำให้ค่า VLMAX ลดลงตามสัดส่วน แต่ไม่ได้เพิ่มจำนวนรีจิสเตอร์เวกเตอร์ที่ใช้งานได้

สามารถใช้ชื่อรีจิสเตอร์ได้เฉพาะทุกๆ ตัวที่ 8 เช่น (`v0`, `v8`, `v16` และ `v24`) โดยแต่ละตัวจะมีขนาด $8 \times VLEN$ บิต โดยทั่วไป รีจิสเตอร์เวกเตอร์จะมีขนาด $LMUL \times VLEN$ บิต ดังนั้นจำนวนองค์ประกอบสูงสุด $VLMAX = LMUL \times VLEN / SEW$ LMUL เป็นคุณสมบัติเฉพาะของ RVV และดังที่เราจะเห็นในหัวข้อ 8.12 มันเป็นข้อได้เปรียบที่สำคัญเหนือสถาปัตยกรรมเวกเตอร์และ SIMD อื่นๆ

รีจิสเตอร์ `vtype` เพียงตัวเดียวจะเข้ารหัสค่า SEW และ LMUL สำหรับรีจิสเตอร์เวกเตอร์ทั้ง 32 ตัว รูปที่ 8.2 แสดงโครงสร้างของ `vtype` คำสั่ง `vsetvl`, `vsetvli` และ `vsetivli` ซึ่งใช้กำหนดความยาวเวกเตอร์ `v1` จะตั้งค่า `vtype` ไปพร้อมกันด้วย รูป 8.3 สรุปตัวบ่งชี้รีจิสเตอร์แบบไดนามิกที่กล่าวถึงในหัวข้อนี้

8.4 คำสั่งโหลดและเก็บข้อมูลแบบเวกเตอร์

กรณีที่ง่ายที่สุดสำหรับคำสั่งโหลดและเก็บข้อมูลแบบเวกเตอร์ คือการจัดการกับอาร์เรย์หนึ่งมิติที่ถูกจัดเก็บแบบต่อเนื่องในหน่วยความจำ คำสั่งโหลดเวกเตอร์แบบต่อเนื่อง (unit-stride) เช่น `vle8.v`, `vle16.v`, `vle32.v` และ `vle64.v` จะเดิมข้อมูลลงในรีจิสเตอร์จากตำแหน่งหน่วยความจำที่เรียงต่อเนื่อง โดยเริ่มจาก ตำแหน่งหน่วยความจำที่เก็บอยู่ในรีจิสเตอร์จำนวนเต็ม `rs1` ชื่อจริงของ `vle8.v` คือ คำสั่งโหลดเวกเตอร์ องค์ประกอบขนาด 8 บิต (Vector Load, 8-bit Elements) ส่วน `vle16.v` คือ คำสั่งโหลดเวกเตอร์ องค์ประกอบขนาด 16 บิต (Vector Load, 16-bit Elements) และอื่นๆ ก็มีความหมายในลักษณะเดียวกัน รีจิสเตอร์ความยาวเวกเตอร์ `v1` จะกำหนดจำนวนองค์ประกอบที่จะต้องโหลด

เช่น หาก `a0` เก็บค่า 1024 คำสั่ง `vle32.v v8, (a0)` จะโหลดองค์ประกอบขนาด 32 บิตจากตำแหน่งหน่วยความจำ 1024, 1028, 1032, ..., $1024 + 4 \times (v1-1)$ นำไปเก็บในรีจิสเตอร์เวกเตอร์ `v8` ในทำนองเดียวกัน คำสั่งเก็บข้อมูลแบบต่อเนื่อง (unit-stride)

Name	Meaning	Type	Book section
v1	Current Vector Length in elements	Register	8.3
vttype	Vector TYPE	Register	8.3
vsew	Selected Element Width	Field of vttype register	8.3
v1mul	Length MULTiplier	Field of vttype register	8.3
vta	Tail elements preserved or not	Field of vttype register	8.6
vma	Mask elements preserved or not	Field of vttype register	8.6
VLEN	Size of one vector register in bits	Per-machine constant	8.3
SEW	Selected Element Width	Value represented by vsew	8.3
LMUL	Length MULTiplier	Value represented by v1mul	8.3
ELEN	Element LENgth: maximum vector element size	Per machine constant	8.3
EEW	Effective Element Width (usually EEW=SEW)	Derived value	8.9
EMUL	Effective register length MULTiplier	Derived value	8.9

รูปที่ 8.3: ชื่อและความหมายของตัวชี้ความยาวและความกว้างของเวกเตอร์ ชื่อตัวพิมพ์เล็ก (lower case names) ระบุเป็นรีจิสเตอร์หรือฟิลด์ของรีจิสเตอร์ และชื่อตัวพิมพ์ใหญ่ (upper case names) โดยทั่วไปเป็นค่าคงที่ที่คอมไพเลอร์ใช้งาน ค่าของตัวบ่งชี้ตัวแบบตัวพิมพ์ใหญ่จะแสดงในรูปแบบที่โปรแกรมเมอร์เข้าใจ ขณะที่ค่าของตัวบ่งชี้แบบตัวพิมพ์เล็กคือรูปแบบการเข้ารหัสของค่าในรีจิสเตอร์ ตัวอย่างเช่น ถ้า $vsew = 0$ แล้ว $SEW = 8$ และถ้า $v1mul = 0$ ดังนั้น $LMUL = 1$ ตัวบ่งชี้บางตัวจะอธิบายเพิ่มเติมในหัวข้อ 8.9

เช่น $vse8.v$, $vse16.v$, $vse32.v$ และ $vse64.v$ จะการคัดลอกข้อมูลจากรีจิสเตอร์เวกเตอร์ไปยังตำแหน่งหน่วยความจำที่ต่อเนื่องกัน ในหน่วยความจำ

สังเกตว่า ต่างจากคำสั่งคำนวณ คำสั่งเหล่านี้จะเข้ารหัสความกว้างขององค์ประกอบไว้ใน opcode โดยตรง แทนที่จะใช้ค่า SEW ด้วยเหตุผลที่จะอธิบายในหัวข้อ 8.9 ในที่นี้ให้ถือว่า ความกว้างที่เข้ารหัสตรงกับค่า SEW

สำหรับอาร์เรย์หลายมิติ การเข้าถึงข้อมูลบางกรณีจะไม่ต่อเนื่อง หากข้อมูลถูกจัดเก็บแบบเรียงตามแถว (row-major order) การเข้าถึงข้อมูลตามคอลัมน์จะต้องอ่านข้อมูลที่มีระยะห่างเท่ากับขนาดของหนึ่งแถว สถาปัตยกรรมเวกเตอร์รองรับการเข้าถึงลักษณะนี้ด้วย คำสั่งโหลดแบบเว้นระยะคงที่ (stride) ระหว่างแต่ละองค์ประกอบ ได้แก่ คำสั่ง $vlse8.v$, $vlse16.v$, $vlse32.v$ และ $vlse64.v$ และคำสั่งจัดเก็บแบบเว้นระยะคงที่ได้แก่ ($vsse8.v$, $vsse16.v$, $vsse32.v$, และ $vsse64.v$) โดยคำสั่ง $vlse8.v$ หมายถึง คำสั่งโหลดเวกเตอร์แบบเว้นระยะคงที่ องค์ประกอบขนาด 8 บิต *Vector Load Strided, 8-bit Elements* คำสั่งเหล่านี้ใช้รีจิสเตอร์ต้นทางสองตัว ได้แก่ $rs1$ ตำแหน่งหน่วยความจำเริ่มต้น และ $rs2$ ค่าเว้นระยะหน่วยเป็นไบต์

ตัวอย่างเช่น ถ้า $a0$ มีตำแหน่งหน่วยความจำเริ่มต้น 1024 และ $a1$ (ขนาดหนึ่งแถว) เท่ากับ 64 ไบต์ คำสั่ง $vlse32.v v8, (a0), a1$ จะเข้าถึงตำแหน่งหน่วยความจำ: 1024, 1088, 1152, ..., $1024 + 64 \times (v1-1)$ ข้อมูลที่โหลดจะถูกเขียนลงในองค์ประกอบขนาด 32 บิต ที่เรียงติดกันต่อเนื่องภายในรีจิสเตอร์เวกเตอร์ปลายทาง

การเข้าถึงหน่วยความจำเว้นระยะคงที่ (strided accesses) ถือเป็นการขยายแนวคิดของการเข้าถึงหน่วยความจำแบบต่อเนื่อง unit-stride กล่าวคือ เราสามารถใช้คำสั่ง

vlse32.v โดยตั้งระยะห่าง 4 ไบต์ เพื่อให้ได้ผลลัพธ์เหมือนคำสั่ง vle32.v แต่การใช้คำสั่ง unit-stride โดยตรงจะช่วยให้โปรเซสเซอร์รู้ว่าการเข้าถึงเป็นแบบต่อเนื่อง ทำให้สามารถให้แบนด์วิดท์หน่วยความจำได้สูงกว่า และยังช่วยลดขนาดโค้ดเพราะไม่ต้องโหลดค่าเว้นระยะ (stride) ลงในรีจิสเตอร์ rs2



จนถึงตอนนี้เราสมมติว่าโปรแกรมทำงานกับอาร์เรย์แบบหนาแน่น (dense arrays) แต่เพื่อรองรับอาร์เรย์แบบเบาบาง (sparse arrays) สถาปัตยกรรมเวกเตอร์จึงมีคำสั่งโหลดและเก็บข้อมูลแบบจัดทำดัชนี (indexed) ได้แก่คำสั่งโหลด (vloxei8.v, vloxei16.v, vloxei32.v และ vloxei64.v) และคำสั่งเก็บข้อมูล (vsxei8.v, vsxei16.v, vsxei32.v และ vsxei64.v) คำสั่งแบบจัดทำดัชนีใช้รีจิสเตอร์ต้นทาง 2 ตัว คือโดยตัวหนึ่งอ้างอิงถึงรีจิสเตอร์เวกเตอร์ rs2 และอีกตัวหนึ่งเป็นรีจิสเตอร์รีจิสเตอร์สเกลาร์ rs1 กล่าวคือ รีจิสเตอร์สเกลาร์ rs1 จะเก็บตำแหน่งที่อยู่เริ่มต้นของอาร์เรย์แบบเบาบาง ส่วนแต่ละองค์ประกอบของรีจิสเตอร์เวกเตอร์ rs2 จะเก็บค่าดัชนี (หน่วยไบต์) ของตำแหน่งข้อมูลที่ไม่เป็นศูนย์ในอาร์เรย์นั้น สำหรับคำสั่งโหลดและเก็บแบบจัดทำดัชนีนี้รหัสดำเนินการ (opcode) จะเข้ารหัสความกว้างขององค์ประกอบดัชนี ขณะที่ค่า SEW จะกำหนดความกว้างขององค์ประกอบข้อมูลที่ถูกโหลดหรือเก็บ วิธีนี้ทำให้ เวกเตอร์ดัชนีและเวกเตอร์ของข้อมูลสามารถใช้ขนาดองค์ประกอบที่แตกต่างกันได้ ซึ่งเป็นกรณีที่พบได้บ่อยในทางปฏิบัติ

สมมติว่าตำแหน่งหน่วยความจำเริ่มต้นในรีจิสเตอร์ a0 คือ 1024 สมมติว่าแอดเดรสเริ่มต้นในรีจิสเตอร์ a0 คือ 1024 และรีจิสเตอร์เวกเตอร์ v4 มีค่าดัชนี (หน่วยเป็นไบต์) ใน 4 องค์ประกอบแรกเป็น: 16, 48, 80, 160 คำสั่ง vloxei32.v v0, a0, v4 จะส่งลำดับตำแหน่งหน่วยความจำเหล่านี้: 1040 (1024 + 16), 1072 (1024 + 48), 1104 (1024 + 80), 1184 (1024 + 160) ไปยังหน่วยความจำ ข้อมูลที่โหลดกลับมาจะถูกจัดเก็บลงในรีจิสเตอร์เวกเตอร์ปลายทาง โดยเรียงเป็นองค์ประกอบที่ต่อเนื่องกัน แต่ละองค์ประกอบมีขนาดเท่ากับ SEW บิต

ใน RISC-V นั้น จริงๆ แล้วมีคำสั่งของถ่ายโอนข้อมูลแบบเวกเตอร์แบบจัดทำดัชนี 2 แบบ คือ แบบเรียงลำดับ (ordered) และ แบบไม่เรียงลำดับ (unordered) คำสั่งแบบไม่เรียงลำดับจะบอกโปรเซสเซอร์ว่า การโหลดและการจัดเก็บข้อมูลของแต่ละองค์ประกอบสามารถเกิดขึ้นในลำดับใดก็ได้ โดยยังคงให้ผลลัพธ์ที่ถูกต้อง ซึ่งในบางสถาปัตยกรรมอาจช่วยเพิ่มแบนด์วิดท์หน่วยความจำได้ เช่น ในกรณีที่เกิดการพลาดแคช (cache misses) ทำให้การเข้าถึงหน่วยความจำบางตำแหน่งเกิดช้ากว่าตำแหน่งอื่น ในทางกลับกัน คำสั่งแบบเรียงลำดับ (ordered) จะบอกให้ตัวประมวลผลทำการเข้าถึงข้อมูลตามลำดับขององค์ประกอบ เพื่อให้ได้ผลลัพธ์ที่ถูกต้อง เนื่องจากเป็นไปได้อันที่ค่าดัชนีเดียวกันจะปรากฏซ้ำหลายครั้ง ตัวอย่างเช่น ในคำสั่งเก็บข้อมูลชนิดจัดทำดัชนีแบบเรียงลำดับ ค่าในหน่วยความจำควรถูกกำหนดโดยองค์ประกอบสุดท้ายที่มีค่าดัชนี ซ้ำกันในรีจิสเตอร์เวกเตอร์โดยทั่วไป คอมไพเลอร์จะเลือกใช้คำสั่งแบบเรียงลำดับ เว้นแต่จะสามารถพิสูจน์ได้ว่าดัชนีทั้งหมดไม่ซ้ำกัน ในระดับแอสเซมบลีจะใช้ตัวอักษร o สำหรับคำสั่งแบบเรียงลำดับ—เช่น

การโหลดแบบจัดทำดัชนี (indexed load) เรียกอีกชื่อว่า การรวบรวมข้อมูล (gather) และการเก็บข้อมูลแบบจัดทำดัชนี มักเรียกว่า การกระจายข้อมูล (scatter)



ประสิทธิภาพ

vloxei32.v หมายถึง คำสั่งโหลดข้อมูลแบบเวกเตอร์ชนิดจัดทำดัชนีแบบเรียงลำดับองค์ประกอบ 32 บิต (Vector Load Ordered indexed, 32-bit Elements) —และใช้ตัวอักษร *u* สำหรับคำสั่งแบบไม่เรียงลำดับ—ดังนั้น vluxei32.v หมายถึง คำสั่งโหลดข้อมูลแบบเวกเตอร์ชนิดจัดทำดัชนีแบบไม่เรียงลำดับ องค์ประกอบ 32 บิต (Vector Load Unordered indexed, 32-bit Elements)

อาเรย์แบบกระจาย (Sparse arrays) เป็นแรงจูงใจหลักของการใช้คำสั่งโหลดและเก็บข้อมูลชนิดจัดทำดัชนี อย่างไรก็ตาม ยังมีอัลกอริธึมอื่นๆ อีกมากที่เข้าถึงข้อมูลแบบอ้อมผ่านตารางของดัชนี

8.5 ความขนานระหว่างการประมวลผลแบบเวกเตอร์

แม้ว่าโปรเซสเซอร์เวกเตอร์แบบง่ายอาจประมวลผลทีละหนึ่งองค์ประกอบ (element) แต่โดยนิยามแล้ว การคำนวณของแต่ละองค์ประกอบเป็นอิสระต่อกัน ดังนั้น ในทางทฤษฎี โปรเซสเซอร์สามารถคำนวณทุกองค์ประกอบ พร้อมกันได้ทั้งหมด สำหรับ RV32G ขนาดข้อมูลที่กว้างที่สุดคือ 64 บิต และโปรเซสเซอร์เวกเตอร์ในปัจจุบันมักประมวลผลข้อมูลขนาด 64 บิตได้ครั้งละ 2, 4 หรือ 8 องค์ประกอบต่อหนึ่งรอบสัญญาณนาฬิกา (clock cycle) ฮาร์ดแวร์จะเป็นผู้จัดการกรณีขอบ (fringe cases) เช่น เมื่อความยาวเวกเตอร์ไม่เป็นพหุคูณของจำนวนองค์ประกอบ ที่ประมวลผลได้ต่อหนึ่งรอบ

เช่นเดียวกับ SIMD จำนวนการคำนวณแบบขนานจะเพิ่มขึ้น เมื่อขนาดขององค์ประกอบเล็กลง กล่าวคือ หากโปรเซสเซอร์เวกเตอร์ สามารถคำนวณได้ 4 ค่าแบบ 64 บิตต่อหนึ่งรอบ ก็จะสามารถคำนวณได้ประมาณ 8 ค่าแบบ 32 บิต, 16 ค่าแบบ 16 บิต และ 32 ค่าแบบ 8 บิตต่อหนึ่งรอบ

ใน SIMD ผู้ออกแบบ ISA จะเป็นผู้กำหนดจำนวนการประมวลผลแบบขนานสูงสุดต่อหนึ่งคำสั่ง และจำนวนองค์ประกอบต่อหนึ่งรีจิสเตอร์ แต่ในทางตรงกันข้าม สำหรับ RV32V ผู้ออกแบบโปรเซสเซอร์สามารถกำหนดทั้งสองอย่างนี้ได้โดยอิสระ โดยไม่จำเป็นต้องเปลี่ยน ISA คอมไพเลอร์ หรือแม้แต่โค้ดไบนารีเลย ในขณะที่การเพิ่มความกว้างของรีจิสเตอร์ใน SIMD (เช่น เพิ่มเป็นสองเท่า) มักทำให้จำนวนคำสั่ง SIMD เพิ่มมากกว่าสองเท่า และต้องแก้ไขคอมไพเลอร์รวมถึงคอมไพล์โปรแกรมใหม่ ความยืดหยุ่นที่ซ่อนอยู่นี้ทำให้โปรแกรม RV32V เดียวกัน สามารถทำงานได้อย่างมีประสิทธิภาพบนโปรเซสเซอร์เวกเตอร์ที่มีตั้งแต่ขนาดเล็ก ไปจนถึงระดับสมรรถนะสูง โดยไม่ต้องแก้ไขโปรแกรมเลย

8.6 การดำเนินการเวกเตอร์แบบมีเงื่อนไข

โค้ดเวกเตอร์บางส่วนมีคำสั่ง `if` อยู่ด้วย โปรแกรมเมอร์สามารถใช้การกระโดดแบบมีเงื่อนไข (branch) เพื่อกำหนดค่าขององค์ประกอบแต่ละตัวในเวกเตอร์ได้ แต่แนวทางนี้จะไม่มีประสิทธิภาพ ตั้งแต่ยุคของ Cray-1 เป็นต้นมา สถาปัตยกรรมเวกเตอร์จึงใช้ มาสก์ (mask) เพื่อยับยั้งการดำเนินการกับองค์ประกอบบางส่วนของเวกเตอร์ คำสั่งเปรียบเทียบ



ประสิทธิภาพ



การเขียนโปรแกรม



ประสิทธิภาพ

ใน RV32V รูป 8.1 จะทำการตรวจสอบเงื่อนไขระหว่างเวกเตอร์กับเวกเตอร์ หรือระหว่างเวกเตอร์กับสเกลาร์ และเขียนผลลัพธ์ลงในแต่ละบิตของรีจิสเตอร์ปลายทาง โดยให้ค่าเป็น 1 หากเงื่อนไขเป็นจริง และเป็น 0 หากไม่เป็นจริง คำสั่งเวกเตอร์ถัดไปสามารถใช้มาสก์นี้ไปใช้ได้ โดย บิตที่ $i = 1$ หมายความว่า องค์ประกอบที่ i จะถูกดำเนินการ ถ้าบิตที่ $i = 0$ หมายความว่า องค์ประกอบที่ i จะไม่ถูกเปลี่ยนแปลง เพื่อระบุว่าคำสั่งเป็นแบบมีมาสก์ โปรแกรมเมอร์จะเติม .t ต่อท้ายคำสั่งแอสเซมบลี

ใน RV32V คำสั่งสามารถทำงานได้สองแบบ คือ แบบไม่ใช้มาสก์ (unmasked) หมายถึง ดำเนินการกับทุกองค์ประกอบ และแบบใช้มาสก์ (masked) โดยใช้รีจิสเตอร์เวกเตอร์ตัวแรก ($v0$) เป็นตัวกำหนด เมื่อใช้มาสก์ บิตที่มีนัยสำคัญน้อยสุดของแต่ละองค์ประกอบใน $v0$ จะระบุว่าองค์ประกอบนั้นถูกดำเนินการหรือไม่ นอกจากนี้ คำสั่ง `vmand`, `vmnand`, `vmandn`, `vmxor`, `vmor`, `vmnor`, `vmorn` และ `vmxnor` ใช้สำหรับการดำเนินการเชิงตรรกะกับมาสก์ที่เก็บอยู่ในรีจิสเตอร์เวกเตอร์ เพื่อให้สามารถประมวลผลเงื่อนไขซ้อน (nested conditional statements) ได้อย่างมีประสิทธิภาพ

ตัวอย่างเช่น สมมติว่าองค์ประกอบที่มีดัชนีเป็นเลขคู่ทั้งหมด ในรีจิสเตอร์เวกเตอร์ $v3$ เป็นจำนวนเต็มลบ และองค์ประกอบที่มีดัชนีเป็นเลขคี่ทั้งหมดเป็นจำนวนเต็มบวก ผลลัพธ์ของโค้ดต่อไปนี้ คือ

```
vslt.vx v0, v3, x0 # ตั้งค่าบิตของ v0 ถึง 1 เมื่อองค์ประกอบของ v3 < 0
vadd.vv v4, v1, v2, v0.t # องค์ประกอบของ v4 ถูกแทนด้วยผลลัพธ์ v1+v2 เฉพาะองค์ประกอบที่ถูกมาสก์ใน v0
```

ผลลัพธ์ของโค้ดนี้จะทำให้บิตตำแหน่งเลขคู่ทั้งหมดในรีจิสเตอร์เวกเตอร์ $v0$ มีค่าเป็น 1 และบิตตำแหน่งเลขคี่ทั้งหมดมีค่าเป็น 0 และจะเขียนทับองค์ประกอบตำแหน่งเลขคู่ทั้งหมดของ $v4$ ด้วยผลรวมขององค์ประกอบที่สอดคล้องกันใน $v1$ และ $v2$

แล้วองค์ประกอบตำแหน่งเลขคี่ของ $v4$ จะเป็นอย่างไร? โปรแกรมเมอร์สามารถกำหนดได้ว่าองค์ประกอบเหล่านี้ควร เก็บค่าเดิมไว้ (undisturbed) หรือไม่สนใจค่าเดิม (agnostic) และให้โปรเซสเซอร์เปลี่ยนค่าได้ตามต้องการ โดยควบคุมผ่านฟิลด์ `vma` ในรีจิสเตอร์ `vtype` ถ้า `vma = 0` คงค่าเดิม หรือถ้า `vma = 1` ไม่รับประกันค่า (โปรเซสเซอร์อาจเขียนทับได้)

รีจิสเตอร์ `vtype` ยังควบคุม องค์ประกอบส่วนท้าย (tail elements) เมื่อ $v1 < VL-MAX$ ผ่านฟิลด์ `vta` ถ้าเป็น 0 องค์ประกอบส่วนท้ายคงค่าเดิม และถ้าเป็น 1 ค่าไม่ถูกกำหนดแน่ชัด (สามารถถูกเปลี่ยนได้) สถาปัตยกรรม RVV รองรับทั้งแบบ “ไม่สนใจค่า” (agnostic) และ “คงค่าเดิม” (undisturbed) สำหรับองค์ประกอบที่ถูกมาสก์และองค์ประกอบส่วนท้าย เนื่องจาก เพราะตัวเลือกแรกช่วยเพิ่มประสิทธิภาพของโปรเซสเซอร์ในบางกรณี ในขณะที่ตัวเลือกหลังช่วยให้การแปลงโปรแกรมเป็นเวกเตอร์ทำได้ง่ายขึ้น (vectorization) ทำได้ง่ายขึ้น

แม้ว่าการเลือกใช้รีจิสเตอร์หลายตัวสำหรับมาสก์จะช่วยให้โปรแกรมเขียนได้ง่ายขึ้น และลดภาระของรีจิสเตอร์มาสก์ แต่การกำหนดให้ใช้เพียงรีจิสเตอร์ตัวเดียว ช่วยให้การเข้ารหัสคำสั่งมีความกระชับ ทำให้สามารถเข้ารหัสคำสั่งเวกเตอร์ทั้งหมดได้ภายใน 32 บิต

โปรแกรมจะถูกเรียกว่า **vectorizable** (สามารถแปลงเป็นเวกเตอร์ได้) หากการดำเนินการส่วนใหญ่สามารถทำได้ด้วยคำสั่งเวกเตอร์ คำสั่งโหลดแบบรวบรวมข้อมูล (gather), การเก็บแบบกระจายข้อมูล (scatter) และคำสั่งที่ใช้มาสก์ จะช่วยเพิ่มจำนวนโปรแกรมที่สามารถแปลงเป็นเวกเตอร์ได้



ประสิทธิภาพ

doc

การเขียนโปรแกรม

8.7 การควบคุมความยาวเวกเตอร์

ดังที่กล่าวไปก่อนหน้านี้ คำสั่ง `vsetvl`, `vsetvli` และ `vsetivli` จะทำหน้าที่กำหนดทั้ง จีจิสเตอร์ความยาวเวกเตอร์ (`vl`) และจีจิสเตอร์ชนิดเวกเตอร์ `vtype` ภายใน `vtype` จะประกอบด้วย ความกว้างองค์ประกอบที่เลือก (`SEW`) ตัวคูณความยาวเวกเตอร์ (`LMUL`) และการตั้งค่าส่วนท้ายแบบไม่สนใจค่า (`vta`) และมาสก์ไม่สนใจค่า `vma`) คำอธิบายในโค้ดด้านล่างจะแสดงการทำงานของคำสั่ง `vset` แต่ละแบบ

```
vsetvli rd, rs1, vtypei # rd = vl ใหม่, rs1 = AVL, vtypei = ค่า vtype ใหม่
vsetivli rd, uimm, vtypei # rd = vl ใหม่, uimm = AVL, vtypei = ค่า vtype ใหม่
vsetvl rd, rs1, rs2 # rd = vl ใหม่, rs1 = AVL, rs2 = ค่า vtype ใหม่
```

คำสั่ง `vsetvl` และ `vsetvli` จะรับค่า ความยาวเวกเตอร์ที่โปรแกรมต้องการ (`AVL`) จากจีจิสเตอร์สเกลาร์ `rs1` ส่วน `vsetivli` จะรับค่า `AVL` จากค่าคงที่ ค่า `AVL` จาก `rs1` หรือค่าคงที่ ร่วมกับค่า `VLMAX` จะถูกใช้ในการกำหนดค่าใหม่ของ `vl` โดย ตัวประมวลผลสามารถเลือกค่า `vl` ได้อย่างยืดหยุ่น แต่ต้องไม่เกินทั้ง `AVL` และ `VLMAX` (โดยทั่วไปแล้ว มักตั้งค่า `vl` เป็นค่าที่เล็กกว่าระหว่าง `AVL` และ `VLMAX` นั่นคือ $vl = \min(AVL, VLMAX)$) ค่า `vl` ใหม่จะถูกเขียนกลับไปยังจีจิสเตอร์สเกลาร์ `rd` ด้วย ขณะที่ค่าของ `vtype` ใหม่จะเป็นตัวกำหนดค่า `VLMAX` ใหม่

การใช้คำสั่ง `vsetvl` ช่วยให้ซอฟต์แวร์สามารถประมวลผลอาร์เรย์เป็นช่วง ๆ ขนาด `VLMAX` ได้ โดย ไม่จำเป็นต้องรู้ค่า `VLMAX` ล่วงหน้าในขั้นตอนคอมไพล์ และไม่จำเป็นต้องให้อาร์เรย์มีขนาดหารด้วย `VLMAX` ลงตัว ในแต่ละรอบของลูป ถ้ายังมีข้อมูลเหลืออยู่จำนวน `n` องค์ประกอบ โปรแกรมเรียกคำสั่ง `vsetvl` เพื่อร้องขอเวกเตอร์ขนาด `n` องค์ประกอบ เครื่องจะกำหนดค่า `vl` เป็น `n` หรือค่าที่น้อยกว่า `n` (แต่ไม่เกิน `VLMAX`) จากนั้นดำเนินการคำสั่งเวกเตอร์ในลูป ลบค่า `vl` ออกจาก `n` ทำซ้ำจนกว่า `n` จะเป็นศูนย์ เมื่อสิ้นสุดลูป อาร์เรย์ทั้งหมดจะถูกประมวลผลครบ การแบ่งอาร์เรย์ออกเป็นช่วงตามขนาดที่เครื่องรองรับ แล้วประมวลผลวนซ้ำแบบนี้เรียกว่า การแบ่งเวกเตอร์เป็นช่วง (*strip-mining*) เราจะเห็นตัวอย่างของเทคนิคสำคัญนี้ในหัวข้อต่อไป

■ **รายละเอียดเพิ่มเติม:** คำสั่ง `vsetvl`, `vsetvli` มีกรณีพิเศษเมื่อจีจิสเตอร์ `rs1` เท่ากับ `x0`

พฤติกรรมข้างต้นใช้กับกรณีทั่วไปที่ `rs1` ไม่เท่ากับ `x0` แต่มีกรณีพิเศษสองรูปแบบ ที่เพิ่มความเร็วในสถานการณ์ที่พบไม่บ่อย ดังนี้:

- ถ้า `rs1` เท่ากับ `x0` แต่ `rd` ไม่เท่ากับ `x0` คำสั่งปรับค่า `vtype` ตามปกติ จากนั้นตั้งค่า `vl` และ เขียนค่า `VLMAX` ลงใน `rd`
 - ถ้า `rs1` และ `rd` เท่ากับ `x0` ทั้งคู่ คำสั่งจะปรับค่า `vtype` ตามปกติ แต่คงค่า `vl` เดิมไว้ รูปแบบนี้ใช้ได้เฉพาะกรณีที่ค่า `vl` เดิมไม่เกิน `VLMAX` ใหม่ เพื่อให้เงื่อนไขเป็นจริง สามารถทำได้โดยรักษาอัตราส่วนของ `SEW` ต่อ `LMUL` ให้คงที่ เพราะจะทำให้ค่า `VLMAX` คงที่
-

8.8 ตัวอย่างเวกเตอร์ การคำนวณ DAXPY ใน RV32V

แม้ว่า RVV จะมีคุณสมบัติมากกว่านี้ แต่จากที่อธิบายมาถือว่าเพียงพอที่จะพาไปสู่ตัวอย่าง


```

# a0 คือ n, a1 ตัวชี้ไปที่ x[0], a2 ตัวชี้ไปที่ y[0], fa0 คือ a
loop:
    0: 0db576d7 vsetvli a3, a0, e64,m8,ta,ma # avl=n; SEW=64; LMUL=8
    4: 0205f007 vle64.v v0, (a1)             # load vector x
    8: 02067407 vle64.v v8, (a2)             # load vector y
    c: b2055457 vfmacc.vf v8, fa0, v0        # v8 += fa0 * v0
    10: 02067427 vse64.v v8, (a2)            # store vector y
    14: 8d15 sub a0, a0, a3                  # n -= vl
    16: 068e slli a3, a3, 3                  # vl * 8 (bytes)
    18: 95b6 add a1, a1, a3                  # เพิ่ม x pointer
    1a: 9636 add a2, a2, a3                  # เพิ่ม y pointer
    1c: f175 bnez a0, loop                   # repeat if n != 0
    1e: 8082 ret                             # return

```

รูปที่ 8.4: โค้ด RV32V สำหรับ DAXPY ในรูปที่ 5.5 คำสั่งสเกลาร์ ในรูปที่ 5.7 หน้าที่ 65 มีความคล้ายกันมาก ทั้งจำนวนและชนิดของคำสั่ง โดยความแตกต่างหลักคือ `vsetvli` เข้ามาแทนที่การกระโดดแบบมีเงื่อนไขตัวแรก

แบบครบขั้นตอนแล้ว รูปที่ 8.4 แสดงภาษาแอสเซมบลี RV32V สำหรับ DAXPY (รูป 5.5 ในหน้าที่ 63 บทที่ 5) ซึ่งเราจะอธิบายทีละขั้นตอน

นอกจากนี้ โค้ด DAXPY แบบ RV32V ไม่มีส่วนโพรล็อก (prologue) โดยจะเริ่มเข้าสู่ลูปลหลักทันที ขั้นตอนแรกคือ ใช้คำสั่ง `vsetvli` เพื่อกำหนดค่า `vl` และ `vtype` โดยเราจะเน้นอธิบาย `vtype` ก่อน `vtype` ประกอบด้วย ความกว้างขององค์ประกอบที่เลือก (SEW), ตัวคูณความยาวเวกเตอร์สูงสุด (LMUL), การตั้งค่าส่วนท้ายแบบไม่กำหนดค่า (`vta` = 1) และการตั้งค่ามาสก์แบบไม่สนใจค่า (`vma` = 1) เราตั้งค่า SEW เป็น 64 บิต เพราะการคำนวณ DAXPY ใช้จำนวนลอยตัวแบบความแม่นยำสองเท่า เนื่องจาก DAXPY ใช้รีจิสเตอร์เวกเตอร์จำนวน 2 ตัว—เพื่อเก็บส่วนของอาร์เรย์ `x` และ `y`—จึงตั้งค่า LMUL = 8 เพื่อให้เวกเตอร์ยาวขึ้น (ประสิทธิภาพดีขึ้น) โดยจะใช้ `v0` เพื่อเก็บ `x` และ `v8` เพื่อเก็บข้อมูล `y` โปรแกรมนี้ใช้รูปแบบการตั้งค่าแบบไม่สนใจองค์ประกอบส่วนท้ายและมาสก์ไม่สนใจค่า (agnostic) เมื่อรวมเข้าด้วยกัน รูปแบบการตั้งค่า คือ `e64,m8,ta,ma` หมายถึง SEW=64, LMUL=8, Tail-Agnostic (`ta` = 1) และ Mask-Agnostic (`ma` = 1) ตามลำดับ

สมมติว่าโปรเซสเซอร์ RV32V มีพื้นที่รีจิสเตอร์เวกเตอร์รวม 2048 ไบต์ (เราเลือกขนาดนี้เพื่อให้เทียบกับสถาปัตยกรรม SIMD ของ x86 ที่เปรียบเทียบใน หัวข้อ 8.12) เนื่องจากมีรีจิสเตอร์เวกเตอร์ทั้งหมด 32 ตัว แต่ละตัวมีขนาด 64 ไบต์ แต่เมื่อกำหนดค่า LMUL = 8 รีจิสเตอร์เวกเตอร์จะถูกจัดกลุ่ม ทำให้เหลือรีจิสเตอร์ที่ใช้งานได้จริง 4 กลุ่ม โดยแต่ละกลุ่มมีขนาด 512 ไบต์ หรือเทียบเท่ากับองค์ประกอบแบบเลขจุดลอยตัวชนิดความแม่นยำสองเท่า จำนวน 64 ค่า ดังนั้นค่า VLMAX สำหรับรูปนี้จึงเท่ากับ 64

นอกจากการตั้งค่า `vtype` แล้ว คำสั่ง `vsetvli` ที่ต้นลูปลยังใช้กำหนดค่าความยาวเวกเตอร์ (`vl`) สำหรับคำสั่งเวกเตอร์ในรอบนั้น โดยในแต่ละรอบของลูปล โปรแกรมจะร้องขอความยาวเวกเตอร์ ให้เท่ากับจำนวนองค์ประกอบที่เหลืออยู่ในอาร์เรย์ คำสั่ง `vsetvli` จะเป็นตัวกำหนดจำนวนองค์ประกอบที่จะถูกประมวลผลจริงในรอบนั้น—ซึ่ง

สถาปัตยกรรมเวกเตอร์ที่ไม่มี `v1` ต้องเขียนโค้ดเพิ่มเพื่อจัดการกับองค์ประกอบชุดสุดท้ายของลูปล และเพื่อตรวจสอบว่า `n = 0` หรือไม่

โปรเซสเซอร์เวกเตอร์ CRAY X1 มีการปรับปรุง

จะอยู่ระหว่าง n และ V_{LMAX} ถ้า n (จำนวนข้อมูลที่เหลือ) มีค่ามาก ในรอบส่วนใหญ่ของ ลูป $v1$ จะเท่ากับ V_{LMAX} แต่เมื่อเข้าสู่ช่วงท้ายของลูป หาก n ไม่สามารถหารด้วย V_{LMAX} ลงตัว ค่า $v1$ จะลดลง เพื่อป้องกันไม่ให้เกิดการอ่านหรือเขียนข้อมูลเกินขอบเขตของอาร์เรย์ x และ y

คำสั่ง `vle64.v` ที่ตำแหน่งหน่วยความจำ 4 โหลดข้อมูลแบบเวกเตอร์ จากตำแหน่ง หน่วยความจำของอาร์เรย์ x (ที่ชี้โดยรีจิสเตอร์สเกลาร์ $a1$) จำนวน $v1$ องค์ประกอบ เข้าสู่ รีจิสเตอร์เวกเตอร์ $v0$

คำสั่ง `vle64.v` ที่ตำแหน่งหน่วยความจำ 8 จะโหลดข้อมูล $v1$ องค์ประกอบของ y จากหน่วยความจำเข้าสู่ $v8$ คำสั่ง `(add)` จะเพิ่มค่าตัวชี้ตำแหน่งของ x

คำสั่งที่ตำแหน่งหน่วยความจำ c คือจุดสำคัญที่สุด คำสั่ง `vfmacc.vf` จะคูณองค์ประกอบของ x (ใน $v0$) กับค่าคงที่ a ($fa0$) แล้วนำผลลัพธ์ไปบวกกับองค์ประกอบ ของ y ($v8$) และเก็บผลลัพธ์กลับลงใน y ($v8$)

สิ่งที่ต้องดำเนินการต่อไปคือ การเก็บผลลัพธ์ไปยังหน่วยความจำ และการจัดการงานใน ส่วนของลูป คำสั่งเก็บข้อมูลที่ตำแหน่งหน่วยความจำ10 (`vse64.v`) จะเก็บผลลัพธ์จำนวน $v1$ องค์ประกอบของ y ลงในหน่วยความจำ คำสั่งที่ตำแหน่งหน่วยความจำ 14 (`sub`) จะ ลดจำนวนองค์ประกอบที่เหลือ n (ใน $a0$) ลงด้วย $v1$ เพื่อเก็บจำนวนข้อมูลที่ประมวลผล เสร็จแล้วในรอบของลูป คำสั่งเลื่อนบิต (`slli`) จะคูณค่าความยาวเวกเตอร์ $Tv1$ ด้วยขนาด ข้อมูลในหน่วยไบต์ ด้วยขนาดข้อมูล (8 ไบต์) เพื่อใช้ปรับ pointer ค่าความยาวเวกเตอร์ ด้วยขนาดข้อมูลในหน่วยไบต์ (8 ไบต์) เพื่อใช้ในการปรับตัวชี้ตำแหน่ง (pointer) x และ y ในขั้นตอนถัดไป คำสั่ง `add` ที่ตำแหน่งหน่วยความจำ 16 และ 18 จะเพิ่มค่าตัวชี้ตำแหน่ง ของ x และ y ตามลำดับ คำสั่งกระโดด (branch) ที่ตำแหน่งหน่วยความจำ 1a จะทำให้ ลูปทำงานซ้ำ ถ้า n (ใน $a0$) ไม่เป็นศูนย์ ถ้า n เป็นศูนย์แล้ว คำสั่งสุดท้าย `ret` จะคืนการ ควบคุมกลับไปยังจุดที่เรียกฟังก์ชัน

จุดเด่นของสถาปัตยกรรมเวกเตอร์ คือ ในแต่ละรอบของลูปที่มีเพียง 10 คำสั่งนี้ สามารถดำเนินการเข้าถึงหน่วยความจำได้ $3 \times 64 = 192$ ครั้ง และทำการคำนวณแบบ จำนวนลอยตัวได้ $2 \times 64 = 128$ ครั้ง (โดยสมมติว่า n มีค่ามากกว่าหรือเท่ากับ V_{LMAX}) โดยเฉลี่ยแล้วต่อ 1 คำสั่ง ประมาณ 19 ครั้ง ในการเข้าถึงหน่วยความจำ และ 13 ครั้ง ในการดำเนินการจำนวนลอยตัวต่อ 1 คำสั่ง ดังที่เราจะเห็นในหัวข้อ 8.12 ค่าอัตราส่วน เหล่านี้สำหรับสถาปัตยกรรม SIMD ของ x86 แม้จะมีขนาดรีจิสเตอร์เวกเตอร์เท่ากันแต่มี ประสิทธิภาพแยกว่าประมาณ 5 เท่า

ตามที่กล่าวไว้ในบทนำ ผู้อ่านที่สนใจในคำอธิบายของคุณลักษณะหลักของ RV32V สามารถข้ามเนื้อหาถัดไป เพื่อไปยังหัวข้อบทสรุปในส่วน 8.13 ส่วนผู้อ่านที่สนใจภาพรวม ควรดำเนินการต่อไป

8.9 คำสั่งขยายขนาดและลดขนาด

จนถึงตอนนี้ เราสมมติกรณีทั่วไปที่ ตัวถูกดำเนินการต้นทางและปลายทางมีขนาดเท่ากัน



ประสิทธิภาพ

ทั้งหมด เนื่องจาก RVV ใช้แนวความคิดแบบไดนามิก (dynamic typing) เพื่อลดจำนวนคำสั่ง แล้ว RVV จัดการกับการคำนวณที่ตัวถูกดำเนินการต้นทางและปลายทางมีขนาดต่างกันได้อย่างไร? คำตอบคือ การเพิ่มคำสั่งอีกสามประเภท:

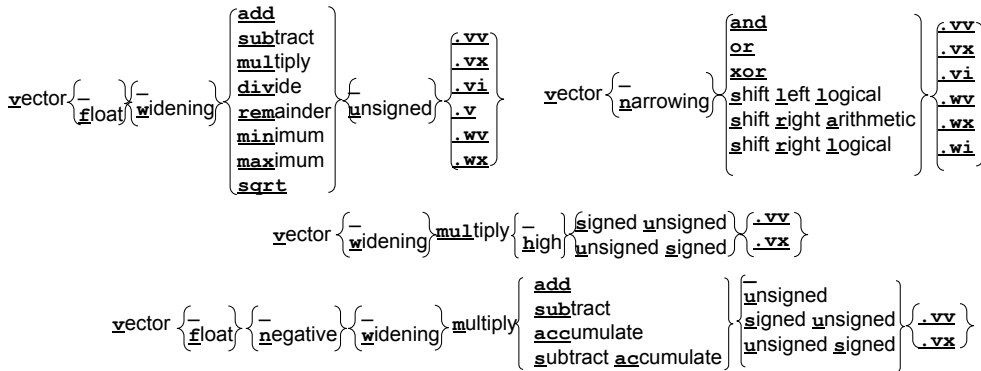
1. คำสั่งคำนวณแบบขยายขนาด (Widening) โดยที่ขนาดของ EEW (Effective Element Width) = $2 \times \text{SEW}$ บิต และ EEW ของตัวถูกดำเนินการต้นทางทั้งสอง มีขนาด SEW บิต
2. คำสั่งคำนวณขยายขนาดแบบผสมขนาด ที่ EEW ของตัวถูกดำเนินการปลายทางและตัวถูกดำเนินการต้นทางตัวแรก (v_{s2}) มีขนาด $2 \times \text{SEW}$ บิต และ EEW ของตัวถูกดำเนินการต้นทางที่สอง (v_{s1}) มีขนาด SEW บิต
3. คำสั่งคำนวณแบบลดขนาด (Narrowing) ที่ EEW ของตัวถูกดำเนินการปลายทางและตัวถูกดำเนินการต้นทางที่สอง (v_{s1}) มีขนาด SEW บิต และ EEW ของตัวถูกดำเนินการต้นทางแรก (v_{s2}) มีขนาด $2 \times \text{SEW}$ บิต

สถาปนิกพบว่ามีความจำเป็นน้อยมากสำหรับการดำเนินการแบบลดขนาด ที่ตัวถูกดำเนินการทั้งสองมีขนาด $2 \times \text{SEW}$ bits รูปที่ 8.5 แสดงการดำเนินการทั้งหมดที่มีคำสั่งขยายขนาด (widening) และคำสั่งลดขนาด (narrowing) ที่ไม่แสดงใน รูปที่ 8.1 เพื่อไม่ให้ผู้อ่านรู้สึกหนักเกินไปหัวข้อของ RV32V

หากตัวดำเนินการบางตัวมีองค์ประกอบที่กว้างกว่าตัวอื่น องค์ประกอบที่กว้างกว่านั้นจะจัดวางในรีจิสเตอร์เวกเตอร์อย่างไร? คำตอบคือ ตัวถูกดำเนินการเหล่านี้จะใช้พื้นที่หลายรีจิสเตอร์เวกเตอร์ เสมือนว่าค่า LMUL ถูกเพิ่มขึ้น กล่าวอย่างแม่นยำ ค่าตัวคูณความยาวเวกเตอร์เชิงผล (EMUL) ของแต่ละตัวถูกกำหนดโดย: $\text{LMUL} \times \text{EEW} / \text{SEW}$ ดังนั้น หาก $\text{LMUL} = 1$ และมีการดำเนินการแบบขยายขนาด (widening) ซึ่งมี $\text{EEW} = 2 \times \text{SEW}$ ตัวถูกดำเนินการปลายทางจะใช้รีจิสเตอร์เวกเตอร์ 2 ตัว

ในภาษาแอสเซมบลี คำสั่งที่มีตัวดำเนินการมีขนาดต่างกัน (mixed-width คำสั่งแบบขยายขนาด (จำนวนเต็ม) ใช้คำนำหน้า v คำสั่งแบบขยายขนาด (เลขจุดลอยตัว) ใช้ vf คำสั่งแบบลดขนาด (จำนวนเต็ม) ใช้ v คำสั่งแบบลดขนาด (เลขจุดลอยตัว) ใช้ vf ตามลำดับ ส่วนต่อท้าย $.vv$ และ $.vx$ หมายถึงตัวถูกดำเนินการแรกเป็นขนาดปกติ (narrow) หากตัวถูกดำเนินการตัวแรกมีขนาดกว้าง (wide) จะใช้ $.wv$ และ $.wx$ โดยตัวอักษร w ใน suffix ใช้ระบุตัวถูกดำเนินการมีขนาดกว้างกว่าเดิม (ใช้ w แทน v)

RV32V



รูปที่ 8.5: แผนภาพของคำสั่งคำนวณและลอจิคอลแบบขยายขนาดและลดขนาด ซึ่งไม่แสดงในรูปที่ 8.1 เนื่องจากการกำหนดชนิดข้อมูลแบบไดนามิก แผนภาพคำสั่งนี้จึงสามารถใช้ได้กับ RV64V โดยไม่ต้องเปลี่ยนแปลงในบทที่ 10

ตัวอย่างเช่น หากตั้งค่า SEW สำหรับตัวถูกดำเนินการขนาด 32 บิต คำสั่งต่อไปนี้จะสามารถสร้างผลคูณ 64 บิตได้ ทั้งจากตัวถูกดำเนินการ 32 บิตสองตัว หรือจากตัวถูกดำเนินการ 64 บิตหนึ่งตัวร่วมกับตัวถูกดำเนินการ 32 บิตอีกหนึ่งตัว

```
vwmul.vv v0, v2, v1 # v0 มีความกว้าง เป็น 2*SEW v2 เป็น SEW และ v1 เป็น SEW
vwmul.wv v0, v2, v1 # v0 มีความกว้าง เป็น 2*SEW v2 เป็น 2*SEW และ v1 เป็น SEW
```

คำสั่งต่อไปนี้จะเลื่อนบิตทางขวาของตัวถูกดำเนินการขนาด 64 และเก็บผลลัพธ์ลงในรีจิสเตอร์ปลายทาง 32 บิต โดยใช้ค่าการเลื่อนบิตจากตัวถูกดำเนินการขนาด 32 บิต

```
vnsrl.vv v0, v2, v1 # v0 มีความกว้าง เป็น SEW, v2 เป็น SEW และ v1 เป็น 2*SEW
```

คำสั่งโหลดและเก็บข้อมูลก็ใช้แนวคิดของ EEW และ EMUL เช่นเดียวกัน ในหัวข้อ 8.4 ได้กล่าวว่า คำสั่งโหลดและเก็บข้อมูลจะเข้ารหัสความกว้างขององค์ประกอบไว้ในรหัสดำเนินการโดยตรง ตัวอย่างเช่น `vse16.v` หมายความว่า ข้อมูลที่ถูกเก็บมีค่า EEW = 16 บิต เช่นเดียวกับการดำเนินการทางคณิตศาสตร์ ค่า EMUL ของข้อมูลที่เก็บเท่ากับ `LMUL×EEW/SEW`

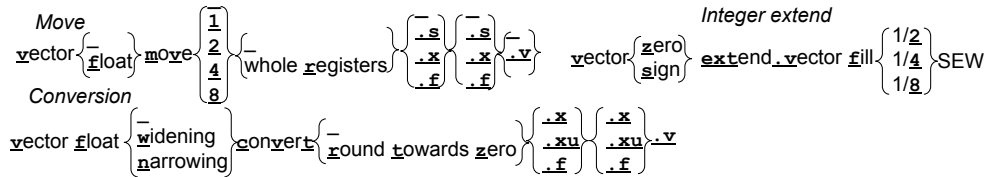
ต่อไปนี้เป็นตัวอย่างสั้นๆ ที่โหลดเวกเตอร์สองตัวของจำนวนเต็ม 8 บิต ใช้การคูณแบบขยายขนาดเพื่อสร้างเวกเตอร์จำนวนเต็ม 16 บิต จากนั้น เก็บผลลัพธ์ไว้ในหน่วยความจำ

```
vsetivli x0, 4, e8, m1, ta, ma # เวกเตอร์ขนาด 4 องค์ประกอบ แต่ละองค์ประกอบกว้าง 8 บิต
vle8.v v0, (a0) # โหลดข้อมูล 8 บิต จำนวน 4 องค์ประกอบ
vle8.v v1, (a1) # โหลดข้อมูล 8 บิต จำนวน 4 องค์ประกอบ
vwmul.vv v2, v1, v0 # คูณได้ผลลัพธ์เป็นข้อมูล 16 บิต
vse16.v v2, (a2) # เก็บผลลัพธ์ 16 บิต จำนวน 4 องค์ประกอบ
```

สังเกตว่าไม่จำเป็นต้องเปลี่ยนค่า `vtype` หรือ `v1` ก่อนเก็บข้อมูลกลับสู่หน่วยความจำ แม้ว่าคำสั่งเก็บข้อมูลจะเข้าถึงข้อมูลในหน่วยความจำ มากกว่าคำสั่งโหลดถึงสองเท่าก็ตาม

โปรแกรมสามารถรักษาอัตราส่วน SEW/LMUL ให้คงที่ (และทำให้ VLMAX คงที่) เมื่อต้องดำเนินการกับเวกเตอร์ที่มีความแม่นยำต่างกัน โดยการปรับค่า `vtype` แบบไดนามิก

RV32V



รูปที่ 8.6: แผนภาพของคำสั่งย้าย การขยายจำนวนเต็ม และการแปลงของ RV32V เนื่องจากการกำหนดชนิดข้อมูลแบบไดนามิก แผนภาพคำสั่งชุดนี้จึงสามารถใช้ได้กับ RV64V โดยไม่ต้องเปลี่ยนแปลง ในบท 10

8.10 คำสั่งย้ายข้อมูล ขยายจำนวนเต็มและแปลงชนิดข้อมูลของเวกเตอร์

คำสั่งกลุ่มนี้มีหลายรูปแบบสำหรับ การย้ายข้อมูล การขยายจำนวนเต็มและการแปลงค่า รูปที่ 8.6 แสดงภาพกราฟิกของคำสั่งเหล่านี้

คำสั่งอ่าน/เขียนสเกลาร์แบบจำนวนเต็ม ใช้สำหรับถ่ายโอนค่าหนึ่งค่าระหว่าง รีจิสเตอร์สเกลาร์ x กับองค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ ถ้ารีจิสเตอร์ปลายทางกว้างกว่ารีจิสเตอร์ต้นทาง จะขยายเครื่องหมายอัตโนมัติ

```
vmv.x.s rd, vs2 # x[rd] = vs2[0] (vs1=0)
vmv.s.x vd, rs1 # vd[0] = x[rs1] (vs2=0)
```

สำหรับรีจิสเตอร์จำนวนลอยตัว ก็มีการคำสั่งย้ายข้อมูลในลักษณะเดียวกัน คือ คำสั่ง `vfmv.fs` และ `vfmv.sf`

คำสั่งย้ายรีจิสเตอร์ทั้งชุด (whole-register move) สามารถคัดลอกกลุ่มรีจิสเตอร์เวกเตอร์ได้โดยไม่ต้องสนใจค่า `v1` หรือ `vtype` ซึ่งเพิ่มประสิทธิภาพได้ คำสั่ง `vmv<nr>r.v` สามารถคัดลอกรีจิสเตอร์เวกเตอร์เป็นกลุ่มขนาด 1, 2, 4, หรือ 8 รีจิสเตอร์ ตัวอย่างเช่น:

```
vmv1r.v v1, v2 # คัดลอก v1=v2
vmv2r.v v10, v12 # คัดลอก v10=v12; v11=v13
vmv4r.v v4, v8 # คัดลอก v4=v8; v5=v9; v6=v10; v7=v11
vmv8r.v v0, v8 # คัดลอก v0=v8; v1=v9; ...; v7=v15
```

คำสั่งขยายจำนวนเต็ม ใช้สำหรับขยายค่าแบบขยายศูนย์ (zero-extend) หรือแบบขยายเครื่องหมาย (sign-extend) จากข้อมูลที่มีขนาดครึ่งหนึ่ง (SEW/2), หนึ่งในสี่ (SEW/4) หรือหนึ่งในแปด (SEW/8) ของรีจิสเตอร์ให้มีขนาดเต็มตามค่า SEW ในขณะเดียวกัน ค่า EEW และ EMUL ของรีจิสเตอร์ต้นทางจะถูกปรับลดลงให้สอดคล้องกัน ตัวอย่างเช่น คำสั่ง `vsext.vf4` จะมี EEW=SEW/4 และ EMUL=LMUL/4

```
vzext.vf2 vd, vs2, vm # ขยายศูนย์จาก SEW/2 ถึง SEW
vsext.vf2 vd, vs2, vm # ขยายเครื่องหมายจาก SEW/2 ถึง SEW
vzext.vf4 vd, vs2, vm # ขยายศูนย์จาก SEW/4 ถึง SEW
vsext.vf4 vd, vs2, vm # ขยายเครื่องหมายจาก SEW/4 ถึง SEW
vzext.vf8 vd, vs2, vm # ขยายศูนย์จาก SEW/8 ถึง SEW
vsext.vf8 vd, vs2, vm # ขยายเครื่องหมายจาก SEW/8 ถึง SEW
```



ประสิทธิภาพ

ตามที่คาดไว้ RVV มีคำสั่งแปลงค่าจำนวนมากสำหรับแปลงค่าระหว่าง จำนวนลอยตัวหลายขนาด กับจำนวนเต็มแบบมีเครื่องหมาย หรือไม่มีเครื่องหมาย คำสั่งเหล่านี้มีจำนวนมากแต่ไม่ได้ซับซ้อนเกินไป (ดูรายละเอียดเพิ่มเติม เกี่ยวกับการแปลงค่าในหัวข้อ 5.4) ด้านล่างนี้คือตัวอย่างบางส่วน ซึ่งเข้าใจได้จากคอมเมนต์ สังเกตการใช้ w หมายถึง ขยายขนาด และ n หมายถึง ย่อขนาด

<code>vfcvt.xu.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มไม่มีเครื่องหมาย
<code>vfcvt.x.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มมีเครื่องหมาย
<code>vfcvt.rtz.xu.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มไม่มีเครื่องหมายพร้อมตัดทศนิยม
<code>vfcvt.rtz.x.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มมีเครื่องหมาย พร้อมตัดทศนิยม
<code>vfcvt.f.xu.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนเต็มไม่มีเครื่องหมายเป็นจำนวนลอยตัว
<code>vfcvt.f.x.v</code>	<code>vd, vs2, vm</code>	# แปลงจำนวนเต็มมีเครื่องหมายเป็นจำนวนลอยตัว
<code>vfwcvt.xu.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มไม่มีเครื่องหมายขนาด 2 เท่า
<code>vfwcvt.x.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มมีเครื่องหมายขนาด 2 เท่า
<code>vfwcvt.rtz.xu.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มไม่มีเครื่องหมายขนาด 2 เท่า พร้อมตัดทศนิยม
<code>vfwcvt.rtz.x.f.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวเป็นจำนวนเต็มมีเครื่องหมายขนาด 2 เท่า พร้อมตัดทศนิยม
<code>vfwcvt.f.xu.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนเต็มไม่มีเครื่องหมายเป็นจำนวนลอยตัวขนาด 2 เท่า
<code>vfwcvt.f.x.v</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนเต็มมีเครื่องหมายเป็นจำนวนลอยตัวขนาด 2 เท่า
<code>vfwcvt.f.f.v</code>	<code>vd, vs2, vm</code>	# แปลงจำนวนลอยตัวเป็นจำนวนลอยตัวขนาด 2 เท่า
<code>vfnvcvt.xu.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย
<code>vfnvcvt.x.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนเต็มแบบมีเครื่องหมาย
<code>vfnvcvt.rtz.xu.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย พร้อมตัดทศนิยม
<code>vfnvcvt.rtz.x.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนเต็มแบบมีเครื่องหมาย พร้อมตัดทศนิยม
<code>vfnvcvt.f.xu.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนเต็มไม่มีเครื่องหมายขนาด 2 เท่าเป็นจำนวนลอยตัวขนาดปกติ
<code>vfnvcvt.f.x.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนเต็มมีเครื่องหมายขนาด 2 เท่าเป็นจำนวนลอยตัวขนาดปกติ
<code>vfnvcvt.f.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนลอยตัวขนาดปกติ
<code>vfnvcvt.rod.f.f.w</code>	<code>vd, vs2, vm</code>	# แปลงค่าจำนวนลอยตัวขนาด 2 เท่าเป็นจำนวนลอยตัวขนาดปกติ ใช้การปัดเศษไปยังจำนวนเต็ม

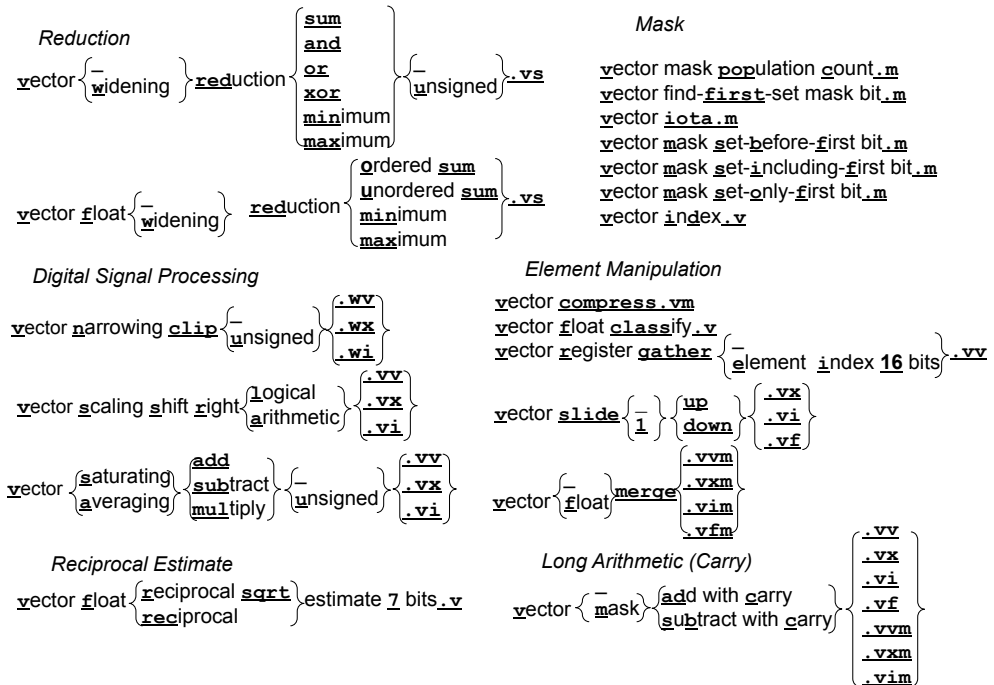
คำสั่งแปลงค่าแบบลดขนาด มีค่า EEW และ EMUL ของรีจิสเตอร์ต้นทาง เป็นสองเท่าของ SEW และ LMUL คำสั่งแปลงค่าแบบขยายขนาด มีค่า EEW และ EMUL ของรีจิสเตอร์ปลายทาง เป็นสองเท่า SEW และ LMUL

8.11 คำสั่งเวกเตอร์สำหรับเร่งประสิทธิภาพไลบรารี

หัวข้อก่อนหน้านี้ ได้อธิบายคำสั่งเวกเตอร์ที่ คอมไพเลอร์สามารถสร้างได้โดยตรงจากซอร์สโค้ดภาษา HLL (High-Level Language) ส่วนหัวข้อนี้จะแนะนำคำสั่งเวกเตอร์ ที่ออกแบบมาเพื่อใช้กับไลบรารีที่ได้รับการปรับแต่งอย่างพิถีพิถันเป็นหลัก—ซึ่งอาจถูกสร้างผ่านคำสั่งคอมไพเลอร์หรือเขียนด้วยภาษาแอสเซมบลีโดยตรง—แม้ว่าบางคำสั่งจะถูกคอมไพเลอร์สร้างขึ้นได้ในบางกรณีพิเศษ คำสั่งเหล่านี้อาจไม่ได้มีลักษณะทั่วไปมากนัก แต่ในสถานการณ์ที่เหมาะสมสามารถช่วยเพิ่มประสิทธิภาพได้อย่างมาก แนวคิดนี้เองก็เป็นเหตุผลที่ทำให้สถาปัตยกรรม x86 SIMD มีคำสั่งจำนวนหลายร้อยคำสั่ง อย่างไรก็ตาม จุดเด่นของ RVV คือ ความกว้างของเวกเตอร์และความกว้างของชนิดข้อมูลไม่ได้ถูกเข้ารหัสไว้ใน opcode จึงไม่จำเป็นต้องมีจำนวนคำสั่งมากเท่า SIMD แบบดั้งเดิม

เราแบ่งคำสั่งเหล่านี้ออกเป็น 8 หมวด ได้แก่:

RV32V



รูปที่ 8.7: แสดงแผนภาพของคำสั่งรีดักชัน, คำสั่งดำเนินการกับบิตของรีจิสเตอร์มาสก์, คำสั่งประมวลผลสัญญาณดิจิทัล, คำสั่งจัดการองค์ประกอบ และคำสั่งเลขคณิตมัลติเวิร์ดที่รองรับตัวทอด (carry) เนื่องจากใช้แนวคิดการแบ่งประเภทข้อมูลแบบไดนามิก แผนภาพคำสั่งนี้จึงสามารถใช้กับ RV64V ในบท 10 ได้ โดยไม่ต้องเปลี่ยนแปลง

- การทำรีดักชัน (Reductions) กับทุกองค์ประกอบภายในรีจิสเตอร์เวกเตอร์เดียว
- การดำเนินการกับบิตของรีจิสเตอร์มาสก์
- การจัดการองค์ประกอบภายในรีจิสเตอร์เวกเตอร์เดียว
- การคำนวณเลขจุดคงที่ (Fixed-point arithmetic) สำหรับงานประมวลผลสัญญาณดิจิทัล (DSP)
- การประมาณค่าส่วนกลับ (Reciprocal estimation)
- การเรียงสับเปลี่ยนข้อมูล (Permutations)
- การคำนวณข้อมูลมัลติเวิร์ด
- คำสั่งย้ายข้อมูลที่เพิ่มประสิทธิภาพหรือหลีกเลี่ยงข้อบกพร่องที่ไม่ต้องการ

รูปที่ 8.7 แสดงคำสั่งสำหรับห้าหมวดหมู่แรก

หนึ่งในการดำเนินการเวกเตอร์ที่มีประโยชน์คือ **รีดักชัน** (Reduction) ซึ่งเป็นการดำเนินการกับทุกองค์ประกอบภายในรีจิสเตอร์เวกเตอร์ เพื่อสรุปผลลัพธ์ออกมาเป็นค่าเดียว RVV รองรับรีดักชันหลายรูปแบบ เช่น ผลรวม (sum), ค่าน้อยสุด (min), และค่ามากที่สุด (max) สำหรับจำนวนเต็มที่มีเครื่องหมาย, จำนวนเต็มที่ไม่มีเครื่องหมาย, และ



จำนวนลอยตัว นอกจากนี้ RVV ยังมีคำสั่งขยายขนาดสำหรับหาผลรวม (widening sum) สำหรับข้อมูลจำนวนเต็ม รองรับรีดักชันของการดำเนินการเชิงตรรกะ AND, OR และ XOR เนื่องจากการบวกจำนวนลอยตัวไม่มีคุณสมบัติการสลับที่ (associative) RVV จึงมีทั้ง การหาผลรวมจำนวนลอยตัวแบบเรียงลำดับ (ordered) และการหาผลรวมจำนวนลอยตัวแบบไม่เรียงลำดับ (unordered) เพื่อให้สามารถเลือกได้ระหว่างประสิทธิภาพที่สูงกว่า เมื่อโปรแกรมไม่สนใจความแตกต่างเล็กน้อยของผลลัพธ์ หรือเลือกผลลัพธ์ที่ทำซ้ำได้แน่นอน เมื่อโปรแกรมต้องการความสม่ำเสมอของผลคำนวณ อินพุตของการดำเนินการรีดักชันประกอบด้วย ทุกองค์ประกอบของรีจิสเตอร์เวกเตอร์ต้นทางแรก และองค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ต้นทางที่สอง โดยผลลัพธ์จะถูกเก็บที่องค์ประกอบที่ 0 ของรีจิสเตอร์เวกเตอร์ปลายทาง ส่วนองค์ประกอบที่เหลือในรีจิสเตอร์เวกเตอร์ปลายทาง ถือเป็นองค์ประกอบส่วนเกินของเวกเตอร์ (tail) ซึ่งจะคงค่าเดิมไว้หรือไม่ขึ้นอยู่กับฟิลด์ `vta` ถ้า `vta = 0` เก็บค่าเดิม แต่ถ้า `vta = 1` ไม่สนใจค่า (ดูรูปที่ 8.3) ตัวอย่างบางส่วนมีดังนี้:

```
# การรีดักชันจำนวนเต็มอย่างง่าย โดยที่ [*] หมายถึงองค์ประกอบที่ใช้งานทั้งหมด
vredsum.vs vd, vs2, vs1, vm # vd[0] = sum (vs1[0], vs2[*])
vredmaxu.vs vd, vs2, vs1, vm # vd[0] = maxu (vs1[0], vs2[*])
vredmax.vs vd, vs2, vs1, vm # vd[0] = max (vs1[0], vs2[*])
vredminu.vs vd, vs2, vs1, vm # vd[0] = minu (vs1[0], vs2[*])
vredmin.vs vd, vs2, vs1, vm # vd[0] = min (vs1[0], vs2[*])

# การรีดักชันผลรวมจำนวนเต็มแบบขยาย
vwredsumu.vs vd, vs2, vs1, vm # vd[0] = ผลรวม (vs1[0], ขยายศูนย์ (vs2[*]))
vwredsum.vs vd, vs2, vs1, vm # vd[0] = ผลรวม (vs1[0], ขยายศูนย์ (vs2[*]))

# การรีดักชันทศนิยมอย่างง่าย
vfredosum.vs vd, vs2, vs1, vm # ผลรวมแบบเรียงลำดับ
vfredusum.vs vd, vs2, vs1, vm # ผลรวมแบบไม่เรียงลำดับ

# การรีดักชันทศนิยมแบบขยาย
vfwredosum.vs vd, vs2, vs1, vm # ผลรวมแบบเรียงลำดับ
vfwredusum.vs vd, vs2, vs1, vm # ผลรวมแบบไม่เรียงลำดับ
```

การดำเนินการกับรีจิสเตอร์มาสก์ จุดประสงค์หลักของรีจิสเตอร์มาสก์ คือเพิ่มสัดส่วนของโค้ดที่สามารถทำให้เป็นเวกเตอร์ได้ โดยช่วยรองรับการดำเนินการแบบมีเงื่อนไข (conditional operations) ก่อนหน้านี้นี้เราได้กล่าวถึงคำสั่งมาสก์ที่ใช้สนับสนุนคำสั่งเงื่อนไขแบบซ้อนกัน ได้แก่คำสั่ง: `vmand`, `vmnand`, `vmandn`, `vmxor`, `vmor`, `vmnor`, `vmorn` และ `vmxnor`

คำสั่งดำเนินการกับรีจิสเตอร์มาสก์ที่เหลือมีประโยชน์มากสำหรับอัลกอริธึมเฉพาะ ตัวอย่างแรกมีที่มาจากสถาปัตยกรรมเวกเตอร์รุ่นแรก:

- `vcpop.m` คำสั่งนับจำนวนบิต 1 นับจำนวนองค์ประกอบของมาสก์ในรีจิสเตอร์เวกเตอร์มาสก์ต้นทางที่มีค่าเป็น 1 แล้วเขียนผลลัพธ์ลงในรีจิสเตอร์สเกลาร์ `x`
- `vfirst.m` คำสั่งค้นหาบิตมาสก์ตัวแรกที่เป็น 1 ค้นหาองค์ประกอบลำดับต่ำสุดของเวกเตอร์มาสก์ต้นทางที่มีค่าเป็น 1 แล้วเขียนดัชนีขององค์ประกอบนั้นลงในรีจิสเตอร์สเกลาร์ `x` หากไม่มีองค์ประกอบที่มีค่าเป็น 1 จะเขียนค่า -1 แทน

เมื่อโปรแกรมเมอร์ใช้รีจิสเตอร์เวกเตอร์เป็นมาสก์ EEW ของแต่ละองค์ประกอบจะเท่ากับ 1 บิต

- `viota.m` เขียนผลรวมสะสมแบบขนานของค่ามาสก์ ลงในรีจิสเตอร์เวกเตอร์ปลายทาง โดยอ่านรีจิสเตอร์มาสก์เวกเตอร์ต้นทาง แล้วเขียนค่าไปยังแต่ละองค์ประกอบของรีจิสเตอร์เวกเตอร์ปลายทาง โดยค่าที่เขียนคือผลรวมของบิตทั้งหมดในรีจิสเตอร์มาสก์ ที่มีดัชนีน้อยกว่าองค์ประกอบปัจจุบัน

คำสั่งมาสก์ต่อไปนี้จะสร้างรูปแบบบิต (bit pattern) ซึ่งโดยทั่วไปจะเป็นบิต 1 จำนวนมากตามด้วยบิต 0 จำนวนมาก—ซึ่งรูปแบบนี้มักถูกเรียกว่า มาสก์—โดยอ้างอิงจากตำแหน่งของบิตแรกที่มีค่าเป็น 1

- `vmsbif.m set-before-first mask bit` เป็นคำสั่งอ่านและเขียนรีจิสเตอร์มาสก์ คำสั่งนี้จะเขียนค่า 1 ไปยังทุกองค์ประกอบของมาสก์ที่อยู่ *ก่อนหน้า* องค์ประกอบต้นทางแรกที่มีค่าเป็น 1 (โดยพิจารณาจากขวาไปซ้าย) จากนั้นเขียนค่า 0 ให้กับทุกองค์ประกอบถัดไป หากไม่มีบิตใดในเวกเตอร์ต้นทางที่มีค่าเป็น 1 ทุกบิตมาสก์ปลายทางจะถูกตั้งค่าเป็น 1
- `vmsif.m set-including-first mask bit` ทำงานคล้ายกับคำสั่ง `vmsbif.m` แตกต่างเพียงอย่างเดียวคือ คำสั่งนี้จะเขียนค่า 1 ให้กับตำแหน่งที่พบค่า 1 ตัวแรกด้วย รวมทั้งบิตก่อนหน้าทั้งหมด
- `vmsof.m set-only-first mask bit` ทำงานคล้ายกับคำสั่ง `vmsbif.m` แต่จะตั้งค่าเป็น 1 เฉพาะองค์ประกอบแรกที่ตรงเงื่อนไขเท่านั้น ส่วนองค์ประกอบอื่นทั้งหมดจะถูกตั้งค่าเป็น 0

สุดท้าย `vid.v, vector element index` เขียนดัชนีของแต่ละองค์ประกอบลงในรีจิสเตอร์เวกเตอร์ปลายทาง โดยมีค่าตั้งแต่ 0 ถึง `v1-1` ตัวอย่างการใช้งานคำสั่ง `vid.v` ใช้ในการคำนวณฟังก์ชันทางคณิตศาสตร์บนเวกเตอร์ที่มีค่ากระจายแบบเชิงเส้น (linearly spaced) ในกรณีนี้ ค่าของตัวแปร (argument) จะเป็นฟังก์ชันเชิงเส้น (affine function) ของหมายเลขขององค์ประกอบ ตัวอย่างอีกหนึ่งคือ ใช้รองรับการเขียนโปรแกรมแบบ CUDA-like ซึ่งโปรแกรมเมอร์ระบุการคำนวณโดยอิงกับหมายเลขขององค์ประกอบ

การคำนวณแบบจุดคงที่ การดำเนินการเหล่านี้ พบได้ทั่วไปในการประมวลผลสัญญาณดิจิทัล (Digital Signal Processing: DSP) คำสั่ง `vnclip` และคำสั่ง `vnclipu` ใช้บีบค่าข้อมูลจุดคงที่ แบบมีเครื่องหมาย หรือไม่มีเครื่องหมาย ให้มีขนาดแคบลงในรีจิสเตอร์ปลายทาง โดยใช้กระบวนการ: การปัดค่า (rounding) การปรับสเกล (scaling) และการอิมตัวของค่า (saturation) ระหว่างการลดขนาดข้อมูล คำสั่งเลื่อนบิตเพื่อปรับสเกล `vssra` และคำสั่ง `vssrl` จะเลื่อนบิตไปทางขวา (right shift) และปัดค่าบิตที่เลื่อนออกตามโหมดการปัดที่กำหนดในรีจิสเตอร์สถานะ `vxrm` คำสั่งบวกและลบแบบเฉลี่ย ได้แก่ `vaadd`, `vaaddu`, `vasub`, และ `vasubu` จะเลื่อนผลลัพธ์ไปทางขวาหนึ่งบิต (เท่ากับ หาร 2) แล้วปัดค่าตามรีจิสเตอร์ `vxrm` คำสั่งบวกและลบแบบอิมตัวของค่า ได้แก่คำสั่ง `vsadd`, `vsaddu`, `vssub` และ `vssubu` จะไม่เกิดการล้นวนกลับเมื่อเกิดโอเวอร์โฟลว์ แต่จะให้ผลลัพธ์เป็นค่าต่ำสุดหรือค่าสูงสุดที่ชนิดข้อมูลนั้นสามารถแทนได้แทน คำสั่งคูณแบบอิมตัว

ของค่า `vsmul` ทำงานคล้ายกัน แต่มีขั้นตอนเพิ่มคือ เลื่อนผลคูณขนาด $2 \times \text{SEW}$ ไปทางขวา SEW-1 บิตก่อน จากนั้นปัดค่าผลลัพธ์ตามโหมดใน `vxrm` ก่อน จากนั้นจึงจำกัดค่าให้อยู่ในช่วงที่ข้อมูลสามารถแทนได้ขนาด SEW บิต

การประมาณค่าส่วนกลับ RV32V มีคำสั่งสองคำสั่งสำหรับประมาณค่า $1/x$ หรือ $1/\sqrt{x}$ ด้วยความแม่นยำ 7 บิต ได้แก่คำสั่ง `vfrec7.v` และ `vfrsqrt7.v` โดยทั่วไปคำสั่งเหล่านี้ทำงานเร็วกว่าคำสั่งหาร `vfdi.vv` หรือคำสั่งรากที่สอง `vfsqrt.v` มาก บางครั้งโปรแกรมเมอร์ใช้คำสั่งเหล่านี้ร่วมกับขั้นตอนการทำซ้ำแบบ Newton-Raphson เพื่อเพิ่มความแม่นยำให้สูงขึ้นประมาณสองเท่า

การจัดเรียงสับเปลี่ยนข้อมูล RV32V มีคำสั่งหลายคำสั่งที่เรียงสับเปลี่ยนองค์ประกอบภายในรีจิสเตอร์เวกเตอร์ คำสั่งการรวบรวมรีจิสเตอร์เวกเตอร์ (`vrgather.vv`) ใช้สร้างเวกเตอร์ผลลัพธ์ใหม่ โดยการรวบรวมองค์ประกอบจากรีจิสเตอร์เวกเตอร์ข้อมูลตัวหนึ่งตามตำแหน่งองค์ประกอบที่ระบุโดยรีจิสเตอร์เวกเตอร์ดัชนีต้นทางตัวอีกตัวหนึ่ง



ประสิทธิภาพ

```
# อาร์กิวเมนต์สุดท้ายเก็บค่าดัชนีตั้งแต่ 0..VLMAX-1
# ซึ่งใช้เลือกองค์ประกอบจากรีจิสเตอร์ vs2
vrgather.vv    vd, vs2, vs1, vm # รีจิสเตอร์เวกเตอร์ดัชนีใน vs1
vrgather.vx    vd, vs2, rs1, vm # ดัชนีเดียวใน x[rs1]
vrgather.vi    vd, vs2, rs1, vm # ดัชนีค่าคงที่เดียว
vrgatherei16.vv vd, vs2, vs1, vm # เวกเตอร์ของดัชนี EEW=16 ใน vs1
```

ตัวอย่างเช่น หากองค์ประกอบสี่ตัวแรกของ `v2` มีค่าเป็น 8, 0, 4, 2 คำสั่ง `vrgather.vv v0, v1, v2` จะทำการกำหนดค่าใหม่ให้กับ `v0` ดังนี้ องค์ประกอบที่ 0 ของ `v0` รับค่าจากองค์ประกอบที่ 8 ของ `v1` องค์ประกอบที่ 1 ของ `v0` รับค่าจาก องค์ประกอบที่ 0 ของ `v1` องค์ประกอบที่ 2 ของ `v0` รับค่าจาก องค์ประกอบที่ 4 ของ `v1` และองค์ประกอบที่ 3 ของ `v0` รับค่าจาก ด้วยองค์ประกอบที่ 2 ของ `v1` ถ้าดัชนีใดมีค่าเกิน VLMAX-1 องค์ประกอบในรีจิสเตอร์เวกเตอร์ปลายทางจะถูกกำหนดให้เป็น 0

คำสั่ง `vrgather` ยังมีรูปแบบ `.vx` และ `.vi` ซึ่งใช้ดึงองค์ประกอบหนึ่งตัวจากรีจิสเตอร์เวกเตอร์ต้นทาง แล้วทำซ้ำองค์ประกอบนั้นไปยังทุกองค์ประกอบรีจิสเตอร์เวกเตอร์ปลายทาง รูปแบบเหล่านี้ทำงานได้เร็วกว่าคำสั่ง `vrgather.vv` ในหลายสถาปัตยกรรม

คำสั่ง `vrgatherei16.vv` ทำงานคล้ายกับคำสั่ง `vrgather.vv` แต่เวกเตอร์ดัชนีมีค่า EEW = 16 ทำให้ข้อมูลและดัชนีสามารถมีความกว้างขององค์ประกอบต่างกันได้

คำสั่ง **รวมเวกเตอร์** (`vmerge.vvm`) ใช้รวมข้อมูลจากรีจิสเตอร์เวกเตอร์สองตัวเข้าด้วยกัน โดยเลือกแต่ละองค์ประกอบจากรีจิสเตอร์ต้นทางตัวใดตัวหนึ่งตามค่าในรีจิสเตอร์มาสก์ ถ้าบิตมาสก์มีค่าเป็น 0 องค์ประกอบใหม่จะมาจาก `vs2` หรือถ้าบิตมาสก์มีค่าเป็น 1 องค์ประกอบใหม่จะมาจาก `vs1`

```
# องค์ประกอบที่ i ของ v0 กำหนดว่าองค์ประกอบที่ i ใหม่ของ vd จะมาจาก
# vs2[i] หรือ vs1[i]
% ถ้าบิตมาสก์ i ของ v0 เป็น 0 หรือมาจาก
vmerge.vvm vd, vs2, vs1, v0 # vs1[i] ถ้าบิตมาสก์ i ของ v0 เป็น 1
vmerge.vxm vd, vs2, rs1, v0 # x[rs1] ถ้าบิตมาสก์ i ของ v0 เป็น 1
vmerge.vim vd, vs2, imm, v0 # imm ถ้าบิตมาสก์ i ของ v0 เป็น 1
```

ตัวอย่างเช่น หากองค์ประกอบสี่ตัวแรกของ $v0$ มีค่าเป็น 1, 0, 0, 1, องค์ประกอบสี่ตัวแรกของ $v2$ มีค่าเป็น 1, 2, 3, 4 และองค์ประกอบสี่ตัวแรกของ $v1$ มีค่าเป็น 10, 20, 30, 40 คำสั่ง `vmerge.vvm v3, v2, v1, v0` จะทำให้องค์ประกอบสี่ตัวแรกของ $v3$ มีค่า 10, 2, 3, 40

คำสั่ง `vmerge` ยังมีรูปแบบ `.vxm` และ `.vim` ซึ่งจะคัดลอกค่าจากรีจิสเตอร์จำนวนเต็มหรือค่าคงที่ทันทีที่ขยายเครื่องหมาย เมื่อ-bitmask มีค่าเป็น 1

คำสั่ง *การเลื่อนลงของเวกเตอร์* `vslidedown` ใช้เลื่อนองค์ประกอบภายในเวกเตอร์จากช่วงกลางหรือท้ายของเวกเตอร์ต้นทาง มาไว้ที่ช่วงต้นของเวกเตอร์ปลายทาง

```
vslidedown.vx vd, vs2, rs1, vm # เลื่อนลงตามค่า x[rs1]
vslidedown.vi vd, vs2, uimm, vm # เลื่อนลงตามค่า uimm
```

ตัวอย่างเช่น ถ้าความยาวเวกเตอร์ $v1$ เป็น 64 และรีจิสเตอร์ $a0$ มีค่าเท่ากับ 16 คำสั่ง `vslidedown.vx v0, v1, a0` จะคัดลอก 48 องค์ประกอบสุดท้ายของ $v1$ ไปยัง 48 องค์ประกอบแรกของ $v0$

คำสั่ง `vslidedown` มีประโยชน์มากสำหรับการรีดักชัน ที่ใช้วิธีการแบ่งครึ่งซ้ำแบบเวียนกลับ (recursive-halving approach) กับตัวดำเนินการแบบสองโอเปอเรนด์ที่สามารถเปลี่ยนกลุ่มการคำนวณได้ (binary associative operator) โดยผลลัพธ์ยังคงเดิม ตัวอย่างการคำนวณผลคูณของทุกองค์ประกอบในรีจิสเตอร์เวกเตอร์ ใช้คำสั่ง `vslidedown` เพื่อย้ายครึ่งหลังของเวกเตอร์ มายังครึ่งหน้าของอีกรีจิสเตอร์เวกเตอร์หนึ่ง แล้วลดค่าความยาวเวกเตอร์ลงครึ่งหนึ่ง จากนั้น คูณรีจิสเตอร์เวกเตอร์ทั้งสองนี้เข้าด้วยกัน และทำซ้ำการแบ่งครึ่งแบบวนซ้ำจนความยาวเวกเตอร์จะเท่ากับ 1 เมื่อจบกระบวนการ ค่าในองค์ประกอบที่ 0 จะเป็นผลคูณขององค์ประกอบทั้งหมดเมื่อจบกระบวนการ ค่าในองค์ประกอบที่ 0 จะเป็นผลคูณขององค์ประกอบทั้งหมดในเวกเตอร์เดิม คำสั่ง `vslideup` ทำงานตรงกันข้ามกับ `vslidedown` โดยเลื่อนองค์ประกอบไปในทิศทางกลับกัน

คำสั่งเลื่อนอื่นๆ อีกสี่คำสั่ง เลื่อนองค์ประกอบเพียงหนึ่งตำแหน่ง แต่แทรกค่าแบบสเกลาร์ลงในช่องว่าง คำสั่ง `vslide1up` คัดลอกองค์ประกอบตั้งแต่ตำแหน่ง 0 ถึง $v1-2$ จากรีจิสเตอร์เวกเตอร์ต้นทางไปยังตำแหน่ง 1 ถึง $v1-1$ ในรีจิสเตอร์เวกเตอร์ปลายทาง จากนั้นวางรีจิสเตอร์ x แบบสเกลาร์ลงในตำแหน่ง 0 ของรีจิสเตอร์ปลายทาง คำสั่ง `vslide1down` คัดลอกองค์ประกอบตั้งแต่ตำแหน่ง 1 ถึง $v1-1$ จากรีจิสเตอร์เวกเตอร์ต้นทาง ไปยังตั้งแต่ตำแหน่ง 0 ถึง $v1-2$ ในรีจิสเตอร์เวกเตอร์ปลายทาง จากนั้นวางรีจิสเตอร์ x แบบสเกลาร์ลงในตำแหน่ง $v1-1$ ของรีจิสเตอร์ปลายทาง `vfslide1up` และ `vfslide1down` ใช้รีจิสเตอร์ f แทนรีจิสเตอร์ x

```
vslide1up.vx vd, vs2, rs1, vm # vd[0] = x[rs1], vd[i+1] = vs2[i]
vslide1up.vf vd, vs2, rs1, vm # vd[0] = f[rs1], vd[i+1] = vs2[i]
vslide1down.vx vd, vs2, rs1, vm # vd[i] = vs2[i+1], vd[v1-1] = x[rs1]
vslide1down.vf vd, vs2, rs1, vm # vd[i] = vs2[i+1], vd[v1-1] = f[rs1]
```

คำสั่งจัดการองค์ประกอบเวกเตอร์คำสั่งสุดท้าย คือ `vcompress` ซึ่งทำหน้าที่บีบอัดองค์ประกอบที่ถูกเลือกโดยรีจิสเตอร์มาสก์เวกเตอร์ต้นทาง ให้ถูกจัดเรียงต่อเนื่องกันไปยังองค์ประกอบช่วงต้นของรีจิสเตอร์เวกเตอร์ปลายทาง:



```
vcompress.vvm vd, vs2, vs1 # เลือกเฉพาะองค์ประกอบของ vs2
                             # ที่ตั้งค่าใน vs1 แล้วบีบอัดเรียงติดกันลงใน vd
```

การคำนวณมัลติเวิร์ด คำสั่งบวกพร้อมตัวทด (add with carry) และคำสั่งการลบพร้อมตัวยืม (subtract with borrow) สำหรับจำนวนเต็ม ช่วยเร่งการคำนวณเลขคณิตมัลติเวิร์ด คำสั่งรูปแบบแรก ได้แก่ `vadc.vvm` และ `vsbc.vvm` ดำเนินการบวกหรือการลบข้อมูลขนาด SEW บิตตามปกติ และเพิ่มเติมการนำค่าตัวทดบิตเข้า (carry-in) หรือตัวยืมบิตเข้า (borrow-in) จากรีจิสเตอร์มาสก์ `v0` มารวมคำนวณด้วย โดยให้ผลลัพธ์ขนาด SEW บิต คำสั่งรูปแบบที่สอง `vadc.vvm` และ `vmsbc.vvm` จะดำเนินการคำนวณแบบเดียวกันกับรูปแบบแรก แต่แทนที่จะให้ผลลัพธ์ข้อมูล จะสร้างค่าตัวทดบิตออก (carry-out) หรือตัวยืมบิตออก (borrow-out) ไปเก็บในรีจิสเตอร์มาสก์ คำสั่งรูปแบบสุดท้าย `vadc.vv` และ `vmsbc.vv` มีลักษณะคล้ายกัน แต่ไม่รับค่าตัวทดบิตเข้า (carry-in) หรือตัวยืมบิตเข้า (borrow-in)—จึงไม่มี `m` ในส่วนต่อท้าย (suffix)—ทำให้เหมาะสำหรับใช้สร้างค่าตัวทดจากค่าที่มีนัยสำคัญต่ำสุด (least-significant word) ของการคำนวณมัลติเวิร์ด

```
# สร้างผลบวกหรือผลลบแบบมีการทดบิตเข้าหรือยืมบิตเข้า
vadc.vvm vd, vs2, vs1, v0 # vd[i] = vs2[i] + vs1[i] + v0[i]
vsbc.vvm vd, vs2, vs1, v0 # vd[i] = vs2[i] - vs1[i] - v0[i]

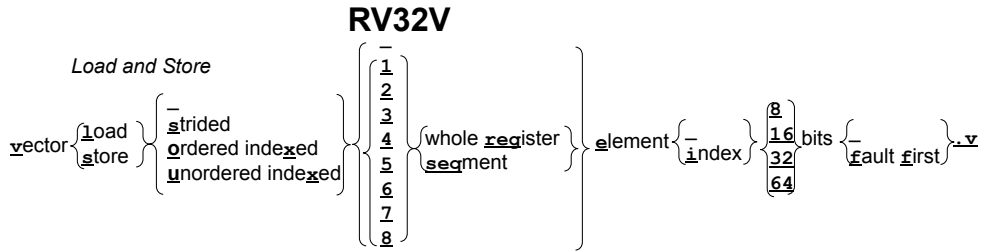
# สร้างผลลัพธ์แบบมีการทดบิตออกหรือยืมบิตเข้า
vadc.vvm vd, vs2, vs1, v0 # vd[i]=carry( vs2[i] + vs1[i] + v0[i] )
vmsbc.vvm vd, vs2, vs1, v0 # vd[i]=borrow( vs2[i] - vs1[i] - v0[i] )

# สร้างผลลัพธ์แบบไม่มีการทดบิตออกหรือการยืมบิตเข้า
vadc.vv vd, vs2, vs1 # vd[i]=carry( vs2[i] + vs1[i] )
vmsbc.vv vd, vs2, vs1 # vd[i]=borrow( vs2[i] - vs1[i] )
```

สำหรับการบวก RVV มีรูปแบบที่รับค่าจากรีจิสเตอร์สเกลาร์ `x` (`.vx` และ `.vxm`) หรือค่าคงที่แบบทันที (immediate) ที่มีการขยายเครื่องหมาย (`.vi` และ `.vim`) สำหรับการลบ มีเฉพาะรูปแบบที่ใช้รีจิสเตอร์ `x` เท่านั้น แต่ไม่มีรูปแบบที่ใช้ค่าคงที่ทันที

การรับส่งข้อมูล (Data Transfer) หมวดสุดท้ายของคำสั่งไลบรารี ช่วยในการสลับบริบท การหลีกเลี่ยงข้อบกพร่องที่ไม่ตั้งใจระหว่างการรับส่งข้อมูลเวกเตอร์ และการทำเวกเตอร์ไรซ์อาร์เรย์ของโครงสร้างข้อมูล (array of structures) โดยแผนภาพคำสั่งอยู่ในรูป 8.8

คำสั่ง *Vector Whole-Register Loads and Stores* ใช้สำหรับบันทึกและกู้คืนรีจิสเตอร์เวกเตอร์ ในกรณีที่ไม่ทราบชนิดข้อมูล หรือความยาวเวกเตอร์ของข้อมูลปัจจุบันภายในรีจิสเตอร์เวกเตอร์ หรือในกรณีที่การเปลี่ยนแปลงค่า `v1` และ `vtype` มีต้นทุนสูงเกินไป ตัวอย่างการใช้งานได้แก่ การ spill รีจิสเตอร์โดยคอมไพเลอร์ การเรียกฟังก์ชันเวกเตอร์ที่มีการส่งค่าผ่านรีจิสเตอร์เวกเตอร์ ตัวจัดการอินเทอร์รัปต์ (interrupt handler) และการสลับบริบทของระบบปฏิบัติการ (OS context switch) ตัวอย่างด้านล่างใช้ข้อมูลขนาดเวิร์ด (32 บิต) แต่ยังมีรูปแบบของคำสั่งสำหรับไบต์ (8 บิต) ฮาล์ฟเวิร์ด (16 บิต) และดับเบิลเวิร์ด (64 บิต) ด้วย



รูปที่ 8.8: แผนภาพของของคำสั่งรับส่งข้อมูลแบบสมบูรณ์ ซึ่งไม่ได้แสดงไว้ใน 8.1 เนื่องจากการกำหนดประเภทข้อมูลแบบไดนามิก แผนภาพคำสั่งนี้จึงสามารถใช้กับ RV64V ในบท 10 ได้โดยไม่ต้องเปลี่ยนแปลง

```

v11re32.v v3, (a0) # โหลด v3 ด้วย 1*VLEN/32 เวิร์ดจากที่อยู่ a0
v12re32.v v2, (a0) # โหลด v2-v3 ด้วย 2*VLEN/32 เวิร์ดจากที่อยู่ a0
v14re32.v v4, (a0) # โหลด v4-v7 ด้วย 4*VLEN/32 เวิร์ดจากที่อยู่ a0
v18re32.v v8, (a0) # โหลด v8-v15 ด้วย 8*VLEN/32 เวิร์ดจากที่อยู่ a0
vs1r.v v3, (a0) # บันทึก v3 ไปยังที่อยู่ a0
vs2r.v v2, (a0) # บันทึก v2-v3 ไปยังที่อยู่ a0
vs4r.v v4, (a0) # บันทึก v4-v7 ไปยังที่อยู่ a0
vs8r.v v8, (a0) # บันทึก v8-v15 ไปยังที่อยู่ a0

```

คำสั่ง `v11r` มีการเข้ารหัสความกว้างขององค์ประกอบ (element width) อยู่ในรหัสคำสั่ง แม้ว่าการใช้คำสั่ง `v11re8` เพียงรูปแบบเดียวจะยังคง ถูกต้องเชิงฟังก์ชัน (functionally correct) แต่รูปแบบอื่นๆ ช่วยให้ฮาร์ดแวร์ทราบลักษณะการใช้งานข้อมูลที่โหลดมา ซึ่งช่วยให้ฮาร์ดแวร์ปรับโครงสร้างภายในชุดรีจิสเตอร์ (register file) แบบไดนามิกตามความกว้างขององค์ประกอบได้ แนวคิดเดียวกันนี้ไม่จำเป็นสำหรับคำสั่งเก็บข้อมูล `vs1r` เพราะฮาร์ดแวร์ทราบชนิดข้อมูลที่ใช้งานล่าสุดอยู่แล้ว ดังนั้นคำสั่ง `vs1r` จึงไม่จำเป็นต้องเข้ารหัสความกว้างขององค์ประกอบไว้ในคำสั่ง

โหลดเวกเตอร์แบบต่อเนื่องที่ตรวจข้อผิดพลาดเฉพาะบิตแรก (Unit-stride Fault-Only-First Loads) ช่วยให้สามารถทำเวกเตอร์ไรซ์ลูปที่ไม่ทราบจำนวนรอบการทำงาน (trip count) ตั้งแต่ก่อนเข้าสู่ลูปได้ ตัวอย่างเช่น การคำนวณความยาวของสตริงที่จบด้วยค่า null (null-terminated) ปัญหาในการทำเวกเตอร์ไรซ์ลูปประเภทนี้คือ คำสั่งโหลดเวกเตอร์อาจโหลดสตริงไปเรื่อย ๆ แล้วอ่านเลยขอบเขตของโครงสร้างข้อมูล ส่งผลให้เกิดข้อยกเว้น (exception) จากการเข้าถึงหน่วยความจำระหว่างโหลดข้อมูลเวกเตอร์โดยไม่ตั้งใจ การโหลดแบบ fault-only-first ถูกออกแบบมาเพื่อหลีกเลี่ยงปัญหานี้ คำสั่งเหล่านี้ทำงานคล้ายกับคำสั่งโหลดแบบต่อเนื่อง (unit-stride) ปกติ แต่จะแจ้งการดักจับ (trap) เฉพาะเมื่อเกิดข้อยกเว้นที่องค์ประกอบที่ 0 เท่านั้น หากตรวจพบข้อยกเว้นที่องค์ประกอบ 0 การดักจับจะเกิดขึ้นและค่า `v1` จะไม่ถูกเปลี่ยนแปลง หากตรวจพบข้อยกเว้นที่องค์ประกอบถัดไป การดักจับที่สอดคล้องกันจะไม่ถูกเรียกใช้งาน แทนที่จะเป็นเช่นนั้น ความยาวเวกเตอร์ `v1` จะถูกลดลงให้เท่ากับดัชนีขององค์ประกอบที่ควรจะทำให้เกิดข้อยกเว้น กลไกนี้เปิดโอกาสซอฟต์แวร์ตรวจสอบได้ว่าข้อยกเว้นนั้นควรถูกยกเว้นนั้นหรือไม่ (เช่น เพราะพบ

อักขระ null ในองค์ประกอบก่อนหน้า) ในทางกลับกัน หากยังจำเป็นต้องเกิด ข้อยกเว้นจริง การวนลูปรอบถัดไปจะตรวจพบข้อยกเว้นเดิมที่องค์ประกอบที่ 0 แล้วจึงเกิดการดักจับในเวลานั้น ต่อไปนี้คือคำสั่งที่เกี่ยวข้อง:

```
# vd รีจิสเตอร์ปลายทาง rs1 ที่อยู่ฐาน vm มาสก์
vle8ff.v vd, (rs1), vm # โหลดแบบต่อเนื่อง ตรวจข้อผิดพลาดเฉพาะบิตแรก ขนาด 8 บิต
vle16ff.v vd, (rs1), vm # โหลดแบบต่อเนื่อง ตรวจข้อผิดพลาดเฉพาะบิตแรก ขนาด 16 บิต
vle32ff.v vd, (rs1), vm # โหลดแบบต่อเนื่อง ตรวจข้อผิดพลาดเฉพาะบิตแรก ขนาด 32 บิต
vle64ff.v vd, (rs1), vm # โหลดแบบต่อเนื่อง ตรวจข้อผิดพลาดเฉพาะบิตแรก ขนาด 64 บิต
```

ชุดสุดท้ายของคำสั่งรับส่งข้อมูล คือ *Vector Segment Loads and Stores* ซึ่งช่วยให้สามารถทำเวกเตอร์ไรซ์อาร์เรย์ของโครงสร้างข้อมูล (arrays of structures) ได้อย่างมีประสิทธิภาพ แต่ละฟิลด์ของโครงสร้างข้อมูลสามารถถูกโหลดเข้าสู่รีจิสเตอร์เวกเตอร์แยกกันได้ โดยคำสั่งรองรับจำนวนฟิลด์ตั้งแต่ 1 ถึง 8 ฟิลด์ เราจะแสดงการทำงานของคำสั่งเหล่านี้ผ่านตัวอย่างสองกรณี พิจารณาโครงสร้างที่เก็บพิกเซล RGB แบบ packed (แดง, เขียว, น้ำเงิน) โดยใช้ข้อมูลสีละ 8 บิต รวมเป็นโครงสร้างข้อมูลขนาด 24 บิต โดยการตั้งค่าจำนวนฟิลด์เป็น 3 เมื่อกำหนดจำนวนฟิลด์เป็น 3 และเริ่มต้นที่รีจิสเตอร์เวกเตอร์ v8 คำสั่งจะโหลด ค่าสีแดง (red pixels) ลงใน v8, ค่าสีเขียว (green pixels) ลงใน v9, และค่าน้ำเงิน (blue pixels) ลงใน v10

```
vlseg3e8.v v8, (a0)
```

อีกตัวอย่างการใช้งานที่ได้รับความนิยม คือ จำนวนเชิงซ้อน (complex numbers) ตัวอย่างนี้ใช้คำสั่ง vlseg เพื่อแยกโครงสร้างของจำนวนเชิงซ้อน ออกเป็นส่วนจริง (real part) ขนาด 32 บิต ในรีจิสเตอร์ v8 และส่วนจินตภาพ (imaginary part) ขนาด 32 บิต ในรีจิสเตอร์ v9 จากนั้นสร้างคอนจูเกตเชิงซ้อน (complex conjugates) โดยทำให้ส่วนจินตภาพมีเครื่องหมายตรงข้าม และสุดท้ายใช้คำสั่ง vsseg จัดเก็บข้อมูลกลับลงหน่วยความจำในรูปแบบโครงสร้างข้อมูลแบบ packed

```
vlseg2e32.v v8, (a0) # โหลดส่วนจริงลง v8 และส่วนจินตภาพลง v9
vfsgnfn.vv v9, v9, v9 # กลับเครื่องหมายของส่วนจินตภาพ
vsseg2e32.v v8, (a0) # จัดเก็บกลับลงหน่วยความจำในรูปแบบ packed
```

นอกจากคำสั่งโหลดและเก็บข้อมูลเป็นชุดแบบต่อเนื่อง (unit stride) แล้ว RVV ยังมีรูปแบบของการโหลดและเก็บเป็นชุดแบบเว้นระยะคงที่ (strided) และแบบจัดทำดัชนี (indexed) อีกด้วย โดยแบบจัดทำดัชนี ยังแบ่งออกเป็นแบบเรียงลำดับ และไม่เรียงลำดับ ตัวอย่างเช่น:

```
vlseg3e8.v v4, (x5), x6 # โหลดไบต์ จาก (x5 + i*x6) ลงใน v4[i]
# และโหลดไบต์ จาก (x5 + i*x6 + 1) ลงใน v5[i]
# และโหลดไบต์ จาก (x5 + i*x6 + 2) ลงใน v6[i]
vsuxseg2ei32.v v7, (x5), v3 # จัดเก็บแบบไม่เรียงลำดับโดยใช้
# องค์ประกอบขนาด SEW บิต และใช้ดัชนีขนาด 32 บิต
# v7[i] จัดเก็บที่ (x5 + v3[i])
# v8[i] จัดเก็บที่ (x5 + v3[i] + (SEW/8))
```

ในขั้นตอนสุดท้ายของการศึกษาเชิงลึกนี้ รูปที่ 8.13 ถึง 8.17 ส่วนท้ายบทจะแสดงรูปแบบคำสั่ง และการเข้ารหัส ของคำสั่งเวกเตอร์ที่นิยมใช้ ซึ่งแสดงไว้ใน 8.1

```

# rdi คือ n, rsi ตัวชี้ไปที่ x[0], rdx ตัวชี้ไปที่ y[0], xmm0 คือ a
0: c4 e2 7d 19 d0    vbroadcastsd ymm2, xmm0 # ymm2 = 4 copies of a
5: 31 c0             xor     eax, eax      # i = 0
7: 48 89 f9          mov     rcx, rdi      #
a: 48 83 e1 fc        and     rcx, -4        # rcx = floor(n / 4) * 4
e: 74 19             je      after_avx_loop # skip main loop if n < 4

avx_loop:
10: c5 fd 10 04 c6    vmovupd ymm0, [rsi+rax*8] # load x
15: c4 e2 ed a8 04 c2 vfmadd213pd ymm0, ymm2, [rdx+rax*8] # 4 mul-adds
1b: c5 fd 11 04 c2    vmovupd [rdx+rax*8], ymm0 # store y
20: 48 83 c0 04        add     rax, 4        # i += 4
24: 48 39 c1          cmp     rcx, rax      # i < floor(n / 4) * 4?
27: 75 e7             jne     avx_loop      # Repeat if so

after_avx_loop:
29: 48 39 c7          cmp     rdi, rax      # Any fringe elements?
2c: 76 18             jbe     after_scalar_loop # Skip if not

scalar_loop:
2e: c5 fb 10 04 c6    vmovsd xmm0, [rsi+rax*8] # load x
33: c4 e2 e9 a9 04 c2 vfmadd213sd xmm0, xmm2, [rdx+rax*8] # 1 mul-add
39: c5 fb 11 04 c2    vmovsd [rdx+rax*8], xmm0 # store y
3e: 48 ff c0          inc     rax        # i++
41: 48 39 c7          cmp     rdi, rax      # i < n?
44: 75 e8             jne     scalar_loop      # Repeat if so

after_scalar_loop:
46: c3               ret                # Return

```

รูปที่ 8.9: โค้ด x86-64 AVX2 สำหรับ DAXPY ในรูปที่ 5.5 คำสั่ง AVX `vbroadcastsd` ที่ตำแหน่งที่อยู่ 0 สร้างสำเนาของ `a` จำนวน 4 ชุดไปไว้ใน `ymm2` การตรวจสอบ `n < 4` ที่ตำแหน่ง `e` ใช้เพื่อตัดสินว่าจะข้ามลูปหลักหรือไม่ คำสั่ง AVX ทั้งสามคำสั่งที่ตำแหน่งที่อยู่ 2e ถึง 39 (`vmovsd`, `vfmadd213sd`, `vmovsd`) ใช้จัดการกรณีที่ $\text{mod}(n,4) \neq 0$ โดยคำสั่งเหล่านี้จะทำการคำนวณ DAXPY ที่ละหนึ่งองค์ประกอบ และวนลูปซ้ำจนกระทั่งดำเนินการคูณแล้วบวก ครบจำนวน $\text{mod}(n,4)$ ครั้ง

8.12 การเปรียบเทียบ RV64V กับ ARMv8 SVE และ x86-64 AVX SIMD

เราจะเปรียบเทียบการดำเนินการ DAXPY บนสถาปัตยกรรมเวกเตอร์ของ RISC-V กับ x86 Advanced Vector eXtension (AVX) ทั้งแบบ SIMD ขนาด 128 บิต และ 512 บิต จากนั้นจึงเปรียบเทียบกับการดำเนินการบน ARMv8 Scalable Vector Extension (SVE)

คำว่า (pd ในคำสั่ง `vfmadd213pd` ที่ตำแหน่ง 15 หมายถึงการทำการคูณแล้วบวกแบบขนาน 4 ค่า ในขณะที่ sd ในคำสั่ง `vfmadd213sd` ที่ตำแหน่ง 33 จะทำการคูณแล้วบวกเพียงค่าเดียว ภาระงานส่วนเกินในการควบคุม (bookkeeping overhead) ของ SIMD ที่ตำแหน่ง a ถึง e และ 29 ถึง 44 เห็นได้ชัดเมื่อเปรียบเทียบกับโค้ดนี้กับโค้ด RVV ในรูป 8.4 โค้ดลักษณะนี้ไม่จำเป็นสำหรับ RVV เนื่องจากรีจิสเตอร์ความยาวเวกเตอร์ `v1` และคำสั่ง `vsetv1` ทำให้ลูปสามารถทำงานได้กับบางค่าของ `n`

x86-64 AVX2 SIMD Intel ได้พัฒนา SIMD หลายรุ่น ซึ่งสามารถเห็นได้จากโค้ดใน รูปที่ 8.9 การขยายชุดคำสั่ง Streaming SIMD Extensions (SSE) ไปเป็น SIMD ขนาด 128 บิต นำไปสู่การมีรีจิสเตอร์ `xmm` จำนวน 8 ตัว และคำสั่งใหม่ที่สามารถใช้งานรีจิสเตอร์เหล่านี้

นี้ได้ การขยายต่อไปเป็น SIMD ขนาด 256 บิต ภายใต้ชื่อ Advanced Vector Extensions (AVX) ทำให้เกิดรีจิสเตอร์ `ymm` จำนวน 16 ตัว และคำสั่งใหม่จำนวนมาก ที่สามารถใช้งานรีจิสเตอร์เหล่านี้ได้

กลุ่มคำสั่งแรกที่ตำแหน่งที่อยู่ 0 ถึง e ในรูปที่ 8.9 โหลดตัวแปรจากหน่วยความจำ ทำสำเนา `a` 4 ชุดในรีจิสเตอร์ `ymm` ขนาด 128 บิต และตรวจสอบเพื่อให้แน่ใจว่า `n` มีค่าน้อยกว่า 4 ก่อนเข้าสู่ลูปลหลัก โค้ดเตรียมก่อนลูปล (preamble) นี้ใช้คำสั่ง AVX หนึ่งคำสั่ง (คำบรรยายของ รูปที่ 8.9 อธิบายรายละเอียดเพิ่มเติม)

ลูปลหลักดำเนินการคำนวณส่วนสำคัญของ DAXPY คำสั่ง AVX `vmovupd` ตำแหน่งที่อยู่ 10 โหลด 4 องค์ประกอบของ `x` ลงใน `ymm0` คำสั่ง AVX `vfmadd213pd` ที่ตำแหน่งที่อยู่ 15 คูณสำเนา 4 ชุดของ `a` (`ymm2`) กับ 4 องค์ประกอบของ `x` (`ymm0`) บวกกับ 4 องค์ประกอบของ `y` ในหน่วยความจำตำแหน่งที่อยู่ `ecx+edx*8`) และใส่ผลรวม 4 ตัวลงใน `ymm0` คำสั่ง AVX ถัดไปนี้ที่ตำแหน่งที่อยู่ 1b คำสั่ง `vmovupd` เก็บผลลัพธ์ 4 ตัวลงใน `y` คำสั่งสามคำสั่งถัดไปเพิ่มตัวนับและทำซ้ำลูปลหากจำเป็น

ต่างจาก RVV เนื่องจากไม่มีรีจิสเตอร์ความยาวเวกเตอร์ AVX จึงต้องมีคำสั่งจัดการงานควบคุมเพิ่มเติมเพื่อตรวจสอบค่าที่มีปัญหาของ `n` โค้ด "จัดการข้อมูลส่วนที่เหลือ" (fringe) ระหว่างที่อยู่ 29 และ 44 จัดการกรณีเมื่อ `n` ไม่ใช่ทวีคูณของ 4 โดยใช้คำสั่ง AVX สามคำสั่ง

คำสั่ง 6 คำสั่งของลูปลหลักในโค้ด DAXPY ของ x86-64 AVX2 ทำการเข้าถึงหน่วยความจำแบบแมนย่าสองเท่า 12 ครั้ง และการดำเนินการจำนวนลอยตัว 8 ครั้ง โดยเฉลี่ย 2 การเข้าถึงหน่วยความจำ ประมาณ 1.3 การดำเนินการต่อคำสั่ง

x86-64 AVX-512 SIMD ขั้นตอนต่อไปในวิวัฒนาการของ x86 เป็น 512-บิต SIMD เรียกว่า AVX-512 นอกเหนือจากคำสั่ง SIMD 512 บิตใหม่ มันยังขยายเป็น 32 `zmm` รีจิสเตอร์ และเพิ่มรีจิสเตอร์มาสก์ ซึ่งเป็นส่วนประกอบมาตรฐาน ของสถาปัตยกรรมเวกเตอร์แบบดั้งเดิม นอกจากนี้จะช่วยในการใช้คำสั่ง SIMD เมื่อลูปลมีคำสั่งตามเงื่อนไขแล้ว มาสก์เหล่านี้ยังลดภาระงานควบคุมการแยกส่วนของ DAXPY ถ้า `n` ไม่ใช่ทวีคูณของ 8 โค้ดที่ตำแหน่ง 4a จะตั้งมาสก์ แล้วใช้มาสก์นั้นเพื่อดำเนินการวนลูปลครั้งสุดท้ายหนึ่งรอบ ซึ่งมีระหว่าง 1 ถึง 7 การดำเนินการ ขึ้นอยู่กับค่าของ $\text{mod}(n,8)$

รูปที่ 8.10 หน้าที่ 110 แสดงคำสั่ง AVX-512 ของ DAXPY คำสั่ง 7 คำสั่งของลูปลหลักในโค้ด x86-64 AVX-512 DAXPY ทำการเข้าถึงหน่วยความจำแบบแมนย่าสองเท่า 24 ครั้ง และการดำเนินการจำนวนลอยตัว 16 ครั้ง โดยเฉลี่ย 3.4 การเข้าถึงหน่วยความจำและ 2.3 การดำเนินการต่อคำสั่ง

ARM64 SVE Vector รูปที่ 8.11 หน้าที่ 110 แสดงคำสั่ง ARMv8 SVE ของ DAXPY เนื่องจากการกล่าวถึงคำสั่ง ARMv8 ครั้งแรกของเรา เราจะอธิบายทั้ง 11 คำสั่ง

1. คำสั่ง `mov` ที่ตำแหน่ง 0 เริ่มต้นด้วยการกำหนดค่าเริ่มต้นให้ตัวแปร `i` (`x3`) เป็น 0
2. คำสั่ง `mov` ถัดไปจะคัดลอกค่าในตัวแปร `a` ซึ่งแสดงด้วย รีจิสเตอร์จำนวนลอยตัวสเกลาร์ความแมนย่าสองเท่า `d0` ไปยังทุกองค์ประกอบของรีจิสเตอร์เวกเตอร์ความ

โค้ดสำหรับงานควบคุมลักษณะนี้ (bookkeeping code) ถือเป็นส่วนหนึ่งของ stripmining ตามคำอธิบายในรูป 8.9 ได้กล่าวไว้ รีจิสเตอร์ความยาวเวกเตอร์ `v1` ทำให้โค้ดควบคุมแบบ SIMD ไม่จำเป็นสำหรับ RVV สถาปัตยกรรมเวกเตอร์แบบดั้งเดิมต้องมีโค้ดเพิ่มเติมเพื่อจัดการกรณีพิเศษเมื่อ `n = 0` แต่ใน RVV เพียงแค่ทำให้คำสั่งเวกเตอร์ทำงานเสมือนเป็นคำสั่งว่าง (nop) เมื่อ `v1 = 0`


```

# rdi คือ n, rsi ตัวชี้ไปที่ x[0], rdx ตัวชี้ไปที่ y[0], xmm0 คือ a
0: 89 f9          mov     ecx, edi      #
2: 83 e1 07        and     ecx, 7          # Form mask for fringe
5: 41 ba 01 00 00 00 mov     r10d, 1      # cases by computing
b: 41 d3 e2        shl     r10d, cl       # 2^(n mod 8) - 1
e: 41 ff ca        dec     r10d      #
11: 62 f2 fd 48 19 d0 vbroadcastsd zmm2, xmm0 # zmm2 = 8 copies of a
17: 48 89 f9        mov     rcx, rdi      #
1a: 48 83 e1 f8      and     rcx, -8         # rcx = floor(n / 8) * 8
1e: c5 f5 46 c9      kxnorb  k1, k1, k1    # mask = all 1s
22: 31 c0           xor     eax, eax       # i = 0
24: eb 1f          jmp     tail          # กระโดดไปที่ loop tail
loop:
26: 62 f1 fd c9 10 04 c6 vmovupd zmm0{k1}{z}, [rsi+rax*8] # load x
2d: 62 f1 fd c9 10 0c c2 vmovupd zmm1{k1}{z}, [rdx+rax*8] # load y
34: 62 f2 f5 c9 98 c2  vfmadd132pd zmm0{k1}{z}, zmm1, zmm2 # 8 mul-adds
3a: 62 f1 fd 49 11 04 c2 vmovupd [rdx+rax*8]{k1}, zmm0      # store y
41: 48 83 c0 08      add     rax, 8          # i += 8
tail:
45: 48 39 c1         cmp     rcx, rax      # i < floor(n / 8) * 8?
48: 77 dc          ja     loop       # Repeat if so
4a: c4 c1 7b 92 ca   kmovd  k1, r10d      # Install mask for fringe cases
4f: 48 39 c7         cmp     rdi, rax      # i < n?
52: 77 d2          ja     loop       # Perform final iteration if so
54: c3            ret              # Return

```

รูปที่ 8.10: โค้ด x86-64 AVX-512 สำหรับ DAXPY ในรูปที่ 5.5 ลูปหลักที่ตำแหน่ง 26 ถึง 41 คล้ายกับรูป 8.9 มาก เพียงว่าโค้ดนี้ใช้การโหลดสองครั้ง และการคูณแล้วบวก ในขณะที่โค้ดก่อนหน้านี้ใช้การโหลดครั้งเดียว และมีตัวถูกดำเนินการหน่วยความจำในคำสั่งคูณแล้วบวก ความแตกต่างใหญ่คือการใช้มาสก์สำหรับการแบ่งข้อมูลออกเป็นช่วง (stripmining) คำสั่งที่ตำแหน่ง 2 ถึง e เริ่มต้นมาสก์ และคำสั่งที่ตำแหน่ง 45 ถึง 52 ปรับหน้ากาสำหรับเมื่อ $\text{mod}(n,8) \neq 0$ การดำเนินการมีมาสก์คำนวณองค์ประกอบที่เหลือทั้งหมดในขั้นตอนเดียว แทนที่จะคำนวณทีละองค์ประกอบอย่างในรูปที่ 8.9

```

# x0 คือ n, x1 ตัวชี้ไปที่ x[0], x2 ตัวชี้ไปที่ y[0], d0 คือ a
0: d2800003 mov     x3, 0          # i = 0
4: 05282000 mov     z0.d, d0      # broadcast a across vector z0
loop:
8: 25e01c60 whilelo p0.d, x3, x0      # form predicates from i, n
c: a5e34021 ld1d   z1.d, p0/z, [x1, x3, lsl 3] # load vector x
10: a5e34042 ld1d   z2.d, p0/z, [x2, x3, lsl 3] # load vector y
14: 65e28001 fmad   z1.d, p0/m, z0.d, z2.d    # z1 = z1 * z0 + z2
18: e5e34041 st1d   z1.d, p0, [x2, x3, lsl 3] # store vector y
1c: 04f0e3e3 incd   x3              # i += number of doubles in a vector
20: eb03001f cmp     x0, x3              # n > i?
24: 54ffff28 b.hi  loop          # branch if so
28: d65f03c0 ret              # return

```

รูปที่ 8.11: โค้ด ARMv8 SVE สำหรับ DAXPY ในรูปที่ 5.5 เปรียบเทียบโค้ดนี้กับ รูปที่ 8.4 ทั้งคู่ใช้ 11 คำสั่ง แม้ว่า SVE จะมีคำสั่งในลูปน้อยกว่า (ดูรูปที่ 8.12) การแบ่งส่วนดำเนินการโดยใช้มาสก์ เช่นเดียวกับ AVX-512 ในรูปที่ 8.10

แม่นยำสองเท่า $z0.d$ ซึ่งใช้ในขั้นตอนที่ 6

3. คำสั่งที่ตำแหน่ง 8 เป็นคำสั่งแรกของลูป ที่มีบทบาทเดียวกับคำสั่ง `vsetvli` ในโค้ด RVV DAXPY ในรูปที่ 8.4 โดยใช้มาสก์เหมือน AVX-512 แทนการปรับ `v1` คำสั่ง `whilelo` ตั้งค่าบิตมาสก์ `p0/z` เป็น 1 สำหรับทุกองค์ประกอบที่ $i(x3)$ ต่ำกว่า $n(x0)$ ซึ่งให้ผลตามที่ต้องการ
4. การโหลดเวกเตอร์ x ของ DAXPY เป็นขั้นตอนต่อไป คำสั่ง `LD1D z1.d` เป็นการโหลดแบบต่อเนื่องของดับเบิลเวิร์ดไปยังรีจิสเตอร์เวกเตอร์ดับเบิลเวิร์ด $z1.d$ ฟลัดถัดไป `p0/z` หมายถึงมันทำงานภายใต้มาสก์ `p0` ที่กำหนดโดย `whilelo` ส่วนตำแหน่งที่อยู่ `[x1, x3, ls1 3]` หมายถึงที่อยู่ฐานใน $x1$ (ที่อยู่ของเวกเตอร์ x) ถูกบวกกับตัวแปร i คูณด้วย 8 (โดยการเลื่อนซ้าย 3 บิต) เพื่อเปลี่ยนดัชนีดับเบิลเวิร์ดเป็นออฟเซตไบต์ การเลื่อนนี้เป็นคำสั่งแยกต่างหากที่ตำแหน่ง 16 ในโค้ด RVV DAXPY ใน รูปที่ 8.4
5. การโหลดที่ตำแหน่ง 10 ดำเนินการโหลดแบบเดียวกัน สำหรับเวกเตอร์ y ของ DAXPY ลงในรีจิสเตอร์เวกเตอร์ดับเบิลเวิร์ด $z2.d$
6. คำสั่ง `fmad z1.d, p0/m, z0.d, z2.d` คูณ $z1.d(x)$ ด้วย $z0.d(a)$ ที่ตั้งในขั้นตอนที่ 2) บวกกับ $z2.d(y)$ และใส่ผลรวมกลับลงใน $z1.d$ การดำเนินการนี้ทำภายใต้มาสก์ `p0` เดียวกัน ARM SVE ไม่มีคำสั่งคูณ-บวกแบบเวกเตอร์-สเกลาร์ ซึ่งเป็นเหตุผลที่โค้ด SVE ต้องการคำสั่งในขั้นตอนที่ 2 เพื่อทำซ้ำ a ในรีจิสเตอร์เวกเตอร์ แต่ RVV ไม่ต้องการ
7. คำสั่งเก็บ `st1d` ที่ตำแหน่ง 18 คล้ายกับการโหลดในขั้นตอนที่ 5 เพราะมันเก็บผลลัพธ์ลงในที่อยู่เดียวกันของเวกเตอร์ y แทนการโหลด
8. คำสั่ง `incd x3` เพิ่มค่า $i(x3)$ ด้วยจำนวนดับเบิลเวิร์ดในรีจิสเตอร์เวกเตอร์โปรเซสเซอร์คำนวณจำนวนนั้นจากความยาวของรีจิสเตอร์เวกเตอร์ RVV แทนที่จะลด i ด้วยความยาวของเวกเตอร์โดยใช้คำสั่งสเกลาร์ `sub` มาตรฐาน
9. คำสั่ง `cmp` ที่ตำแหน่ง 20 เปรียบเทียบ $n(x0)$ กับ $i(x3)$ และตั้งรหัสเงื่อนไข
10. คำสั่งกระโดดตามเงื่อนไข `b.hi` กระโดดไปที่จุดเริ่มต้นของลูปหาก $n > i$
11. หากคำสั่งกระโดดแบบมีเงื่อนไขไม่ถูกดำเนินการ คำสั่ง `ret` จะกลับไปยังคืนการควบคุมไปยังผู้เรียก (callee)

คุณอาจสงสัยว่าโค้ด SVE จะมีปัญหาเมื่อ n เป็น 0 แต่ไม่มี เพราะเมื่อ $n = 0$ SVE จะตั้งบิตมาสก์ทั้งหมดเป็น 0 ดังนั้นจึงไม่มีอะไรเปลี่ยนแปลง เช่นเดียวกับคำสั่งเวกเตอร์ RVV จะไม่เปลี่ยนแปลงเมื่อ $v1 = 0$

การอภิปราย รูปที่ 8.12 สรุปจำนวนคำสั่งและจำนวนไบต์ใน DAXPY ของโปรแกรมสำหรับ x86-64 AVX2, x86-64 AVX-512, ARMv8 SVE และ RV64GCV

ในขณะที่สถาปัตยกรรม ARM-32 รุ่นก่อนหน้ามี Thumb และ Thumb2 เพื่อบีบอัดขนาดโค้ด แต่ผู้ออกแบบ ARMv8 ได้ตัดความสามารถในการบีบอัดโค้ดออกไป เพื่อให้การออกแบบหน่วยประมวล

ISA	x86-64 AVX2	x86-64 AVX-512	ARMv8 SVE	RV64GCV
Instructions (static)	20	22	11	11
Bytes (static)	70	84	44	32
Instructions per Main Loop	6	7	8	10
Results per Main Loop (512B state)	4	–	2	16
Instructions (dynamic, n=1000)	1508	–	4003	631
Results per Main Loop (2048B state)	–	8	8	64
Instructions (dynamic, n=1000)	–	892	1003	161

รูปที่ 8.12: จำนวนคำสั่งและขนาดโค้ดของ DAXPY สำหรับ SIMD และ ISA แบบเวกเตอร์ ตารางนี้ระบุจำนวนคำสั่งทั้งหมด (แบบคงที่), ขนาดโค้ด จำนวนคำสั่ง แถวที่ 5 และ 6 แสดงผลลัพธ์ต่อลูปและจำนวนคำสั่งที่ดำเนินการโดยสมมติว่า $n = 1000$ และสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์มีขนาด 512 ไบต์ ซึ่งเท่ากับขนาดสถานะรีจิสเตอร์ SIMD ของ AVX2 แถวที่ 7 และ 8 แสดงตัวชี้วัดเดียวกันแต่สถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์มีขนาด 2048 ไบต์ เท่ากับขนาดสถานะรีจิสเตอร์ SIMD ของ AVX-512 SVE ดำเนินการคำสั่งมากที่สุดสำหรับแต่ละขนาดสถานะ และ LMUL ทำให้ RV64V ดำเนินการคำสั่งน้อยที่สุด โปรดทราบว่าโค้ดเวกเตอร์ทั้งหมดไม่เปลี่ยนแปลง ไม่ว่าความยาวเวกเตอร์จะเป็นเท่าใด แต่ต้องคอมไพล์ใหม่เพื่อเพิ่มขนาด SIMD จาก 128 เป็น 512 บิต เนื่องจากใช้คำสั่งที่แตกต่างกัน

โค้ดสำหรับการคำนวณแบบ SIMD มีขนาดเล็กมากเมื่อเทียบกับโค้ดควบคุม (book-keeping code) โค้ดส่วนใหญ่ของ SIMD เป็นโค้ดภาระงานส่วนเกิน—ประมาณ 70% สำหรับ AVX2 และ 68% สำหรับ AVX-512—ซึ่งใช้สำหรับเตรียมข้อมูลสำหรับลูป SIMD หลัก หรือสำหรับการแบ่งข้อมูลเป็นช่วงใน SIMD โค้ด RVV ในรูปที่ 8.4 และโค้ด SVE ใน รูปที่ 8.11 ไม่ต้องการโค้ดส่วนควบคุมเช่นนี้ ต่างจาก SIMD ทั้ง RVV และ SVE มีกลไกควบคุมความยาวเวกเตอร์ (`v1` สำหรับ RVV และ `while10` สำหรับ SVE) ซึ่งทำให้คำสั่งเวกเตอร์ทำงานได้กับทุกค่า n ใดๆ โค้ดของสถาปัตยกรรมเวกเตอร์ทั้งสองมีขนาดเล็กกว่ารุ่น x86 ขนาดโค้ด RVV เล็กกว่า SVE เพราะ RVV สามารถรวมกับคำสั่งบีบอัดของ RVC ได้ ในขณะที่ ARMv8 นั้นกำหนดให้ทุกคำสั่งกว้าง 32 บิต โค้ด RV64GCV เล็กกว่า SVE 1.4 เท่า เล็กกว่า AVX2 2.2 เท่า และเล็กกว่า AVX-512 2.6 เท่า แม้จะมีคำสั่งขนาดตัวแปรไบต์ แนวทางการเพิ่มขึ้นของ ISA ส่งผลเสียต่อขนาดคำสั่ง: คำสั่ง AVX2 ยาว 6 ไบต์ และ 7 ไบต์สำหรับ AVX-512 ใน DAXPY ความยาวคำสั่งเฉลี่ยเพิ่มจาก 3.5 สำหรับ AVX2 เป็น 3.8 สำหรับ AVX-512 เทียบกับ 2.9 ไบต์สำหรับ RV64CV และ 4.0 สำหรับ SVE

นอกเหนือจากโค้ดภาษาเครื่องที่เล็กลง โค้ดเวกเตอร์ยังเข้าใจง่ายกว่า SIMD มาก เมื่อเปรียบเทียบรุ่นสเกลาร์และเวกเตอร์ของ RISC-V DAXPY ในจากรูปที่ 5.7 หน้า 65 และรูปที่ 8.4 หน้า 94 ทั้งคู่มีจำนวนคำสั่งเท่ากัน ประกอบด้วย 2 การโหลด, 1 การคูณ-บวก, 1 การเลื่อนบิต, 1 การเก็บ, และ 1 การคืนค่า แบบแรก มี 3 คำสั่งบวกและคำสั่งกระโดดแบบมีเงื่อนไข 2 คำสั่ง และแบบหลัง มี 2 คำสั่งบวก, 1 คำสั่งลบ, คำสั่งกระโดดมีเงื่อนไข 1 คำสั่ง และ 1 คำสั่ง `vsetvli` รุ่น SIMD ของ x86 มีคำสั่งมากกว่ารุ่นสเกลาร์สองเท่า และยากที่



ความเรียบง่าย



ความเรียบง่าย

จะเข้าใจว่าโค้ดทำอะไรหากไม่บอกล่วงหน้า

อย่างไรก็ตาม ความแตกต่างที่สำคัญที่สุดระหว่าง SIMD และการประมวลผลแบบเวกเตอร์ไม่ใช่ขนาดโค้ดแบบคงที่ แม้ว่า RVV และ AVX ใช้ขนาดสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์เท่ากันที่ 512 ไบต์ SIMD ดำเนินการคำสั่งมากกว่า RVV 2.4 เท่า เพราะแต่ละลูบ SIMD จำนวนเพียง 4 องค์ประกอบ แทนที่จะเป็น 16 สำหรับ RVV การดึงคำสั่งและถอดรหัสคำสั่งเพิ่มเติมหมายถึง พลังงานที่สูงขึ้นในการทำงานเดียวกัน

แถวถัดไป 2 สองของรูปที่ 8.12 แสดงข้อดีอีกประการของสถาปัตยกรรมแบบเวกเตอร์ โค้ดเดียวกันทำงานได้โดยไม่เปลี่ยนแปลงบนสถาปัตยกรรมแบบเวกเตอร์ที่มีสถานะสถาปัตยกรรมรีจิสเตอร์เวกเตอร์ 2048 ไบต์ โดยการเพิ่มสถานะ 4 เท่าลดจำนวนคำสั่งลงเกือบ 4 เท่า เมื่อเทียบกับ AVX-512 RVV ดำเนินการคำสั่งน้อยลงเกือบ 5.5 เท่า

เนื่องจาก SVE ไม่มีสิ่งที่เทียบเท่ากับ LMUL ทำให้ RVV ดำเนินการคำสั่งน้อยลง 6.3 เท่าโดยไม่ขึ้นกับขนาดของสถานะสถาปัตยกรรมเวกเตอร์ ในขณะที่ RVV ดำเนินการคำสั่งน้อยกว่า AVX2 หรือ AVX-512 สำหรับทุกขนาดสถานะเวกเตอร์ในตารางนี้ SVE ดำเนินการคำสั่งมากกว่า AVX2 หรือ AVX-512 1.3 เท่า เมื่อขนาดสถานะเวกเตอร์เท่ากับขนาดสถานะ AVX ความแตกต่างที่สำคัญที่สุดระหว่างสถาปัตยกรรม RVV และ SVE คือ:

- SVE ไม่มีสิ่งที่เทียบเท่ากับตัวคูณความยาวเวกเตอร์ (LMUL) ดังนั้นรีจิสเตอร์เวกเตอร์ทั้งหมดจึงมีความยาวเท่ากัน (1/32 ของสถานะสถาปัตยกรรมเวกเตอร์) ไม่ว่าส่วนของโค้ดจะต้องการรีจิสเตอร์จำนวนเท่าใด LMUL ลดจำนวนรอบการวนลูบของ DAXPY สำหรับ RVV ลง 8 เท่า
- การดำเนินการของ SVE มีขนาดของข้อมูลคงที่ ต่างจาก RVV ที่มีการกำหนดประเภทข้อมูลแบบไดนามิก ดังนั้นสถาปัตยกรรม SVE จึงมีจำนวนคำสั่งมากกว่า
- SVE ลดค่าใช้จ่ายจากการแบ่งส่วนข้อมูล (strip-mining) โดยการตั้งคำรีจิสเตอร์มาส์ (predicate) ด้วยคำสั่ง `while` แทนการควบคุมความยาวเวกเตอร์ ด้วยคำสั่ง `vsetvli` อย่างที่ RVV ทำ การพึ่งพามาส์สำหรับการแบ่งข้อมูลออกเป็นช่วงอาจส่งผลต่อประสิทธิภาพ ในลูปที่มีคำสั่งเงื่อนไข และยังทำให้การออกแบบโปรเซสเซอร์เวกเตอร์แบบชั่วคราว (temporal vector processors) ซึ่งประมวลผลองค์ประกอบของเวกเตอร์ตามลำดับเวลา แทนการประมวลผลพร้อมกัน มีความซับซ้อนมากขึ้น
- SVE ไม่รองรับการดำเนินการแบบเวกเตอร์-สเกลาร์โดยตรง ดังนั้นจึงต้องใช้คำสั่งพิเศษเพื่อทำซ้ำรีจิสเตอร์สเกลาร์ไปยังรีจิสเตอร์เวกเตอร์ (ดูตำแหน่งที่ 4 ใน รูปที่ 8.11)
- โหมดการกำหนดที่อยู่แบบปรับขนาดตามดัชนีของ ARMv8 SVE ลดจำนวนคำสั่งที่คำนวณที่อยู่ในลูบหลักของ DAXPY (ตำแหน่ง c, 10, และ 18)



ประสิทธิภาพ

หากค่าใช้จ่ายการดำเนินการสเกลาร์จำกัด ประสิทธิภาพของโค้ด x86 SIMD และ ARM SVE โดยพื้นฐาน สามารถทำสิ่งต่าง ๆ เช่น การคลายลูบ, การเลื่อนตัวแปรเหนี่ยวนำ หรือการพับค่าคงที่, และการรวมตัวแปรการลุด เพื่อลดค่าใช้จ่ายเหล่านั้น ขนาดโค้ดแบบคงที่ที่ได้จะใหญ่ขึ้น แต่จำนวนคำสั่งเฉลี่ยต่อการวนและจำนวนคำสั่งแบบไดนามิกจะลดลง คุณจะต้องเขียนไลบรารีที่ซับซ้อนมากขึ้นด้วยโค้ดที่มากขึ้น ซึ่งทำให้การบำรุงรักษาไลบรารีและการออกแบบคอมไพเลอร์ซับซ้อนขึ้น

■ **รายละเอียดเพิ่มเติม: Illiac IV เป็นคอมพิวเตอร์เครื่องแรกๆ ที่แสดงให้เห็นถึงความยากลำบากในการคอมไพล์สำหรับ SIMD)**

ด้วยหน่วยประมวลผลจำนวนลอยตัว (FPU) ขนาด 64 บิตจำนวน 64 หน่วยที่ทำงานแบบขนาน ILLIAC IV มีแผนที่จะใช้เกิดตรรกะมากกว่า 1 ล้านเกต แม้แต่ก่อนที่มัวร์จะเผยแพร่กฎของเขา สถาปนิกของเครื่องนี้เดิมทีคาดการณ์ว่าจะมีความสามารถประมวลผล 1,000 ล้านการดำเนินการจำนวนลอยตัวต่อวินาที (MFLOPS) David Kuck เป็นสถาปนิกซอฟต์แวร์เพียงคนเดียวสำหรับ Illiac IV (ประมาณปี 1975) คำสะท้อนของเขา [Kuc 1991] คือ “สิ่งที่ฉันรู้สึกหงุดหงิดจริง ๆ คือการเขียนโปรแกรมสำหรับเครื่องนี้ยากมาก และสถาปัตยกรรมอาจไม่เหมาะกับบางแอปพลิเคชันที่เราพยายามรัน แนวคิดหลักคือ ฉันไม่คิดว่าเรามีความสอดคล้องที่ดีใน ILLIAC IV ระหว่างแอปพลิเคชันและสถาปัตยกรรม” ประสิทธิภาพที่แท้จริงของมันอยู่ที่ 15 MFLOPS สูงสุด ค่าใช้จ่ายเพิ่มขึ้นจากประมาณการ 8 ล้านดอลลาร์ในปี 1966 เป็น 31 ล้านดอลลาร์ในปี 1972 แม้ว่าจะมีการสร้าง FPU เพียง 64 ตัวจากที่วางแผนไว้ 256 ตัว โครงการเริ่มต้นในปี 1965 แต่ต้องใช้เวลานานถึงปี 1976 จึงจะรันแอปพลิเคชันจริงตัวแรกได้ ซึ่งเป็นปีที่ Cray-1 ถูกเปิดตัว อาจเป็นซูเปอร์คอมพิวเตอร์ที่โอ้อวดที่สุด มันติดอันดับ 10 ของภัยพิบัติทางวิศวกรรม [Falk 1976]

8.13 ข้อสังเกตสรุป

หากได้สามารถทำเวกเตอร์ไรซ์ได้ สถาปัตยกรรมที่ดีที่สุดคือสถาปัตยกรรมเวกเตอร์

—จิม สมิธ ปาฐกถาพิเศษ, การประชุมนานาชาติด้วยสถาปัตยกรรมคอมพิวเตอร์, 1994

หากคุณมองในมุมที่ต่างออกไป คุณสามารถมอง SIMD เป็นสถาปัตยกรรมเวกเตอร์ที่มีข้อจำกัด โดยมีรีจิสเตอร์เวกเตอร์ขนาดสั้น ผลลัพธ์จาก DAXPY เผยให้เห็นถึงความก้าวหน้าของสถาปัตยกรรม SIMD โดยใช้ AVX2 เป็นฐานเปรียบเทียบ AVX-512 เพิ่มการรองรับการใช้มาสก์ (masking) ทำให้สามารถทำเวกเตอร์ไรซ์ขั้นของเงื่อนไขและการแบ่งส่วนข้อมูล (strip-mining) ได้โดยไม่ต้องทำซ้ำส่วนของลูปลำดับสำหรับกรณีขอบเขต (fringe case) ขั้นตอนต่อไปของ SIMD คือการแยกความยาวของเวกเตอร์ออกจาก opcode เช่นเดียวกับใน SVE ตัวคูณความยาว (LMUL) ของ RVV อยู่ในระดับที่ก้าวหน้ากว่าบนเส้นทางการพัฒนา โดยลดปริมาณการรับส่งคำสั่ง (instruction traffic) สำหรับการคำนวณเดียวกัน

เมื่อเปรียบเทียบผลลัพธ์ในรูปที่ 8.12 กับรุ่นสเกลาร์ของ DAXPY ในรูปที่ 5.6 หน้าที่ 63 SIMD เพิ่มขนาดของโค้ดทั้งในจำนวนคำสั่งและขนาดไบต์เป็นสองเท่าเมื่อเทียบกับโค้ดสเกลาร์ x86 แต่ลูปลำดับยังคงประมวลผลคำสั่งในจำนวนเท่าเดิม

ที่ถูกประมวลผลมีค่าเกือบ 7 เท่าสำหรับ AVX-512 เมื่อเทียบกับโค้ดสเกลาร์ x86 ขนาดโค้ดเวกเตอร์ RV64GCV เพิ่มขึ้น 1.1 เท่าเมื่อเทียบกับ RV64C แบบสเกลาร์ แต่ด้วยสถานะสถาปัตยกรรมเวกเตอร์ขนาด 2048 ไบต์ จำนวนคำสั่งแบบไดนามิกของ RV64CV น้อยกว่า RV64C ถึง 40 เท่า!

วิธีหนึ่งในการประเมินสถาปัตยกรรมการประมวลผลแบบขนานข้อมูล คือการหาค่า N_p ซึ่งในที่นี้หมายถึงค่าของ n ที่ทำให้โค้ดแบบขนานข้อมูลทำงานได้เร็วกว่าโค้ดแบบสเกลาร์ เพื่อให้การคำนวณได้ง่ายขึ้น สมมติว่า ไม่มีการพลาดแคช (cache misses) และทุกคำสั่งในโปรแกรมสมมติทั้งสามแบบใช้เวลา 1 รอบสัญญาณนาฬิกาต่อคำสั่ง ข้อยกเว้นอย่างเดียวคือคำสั่งเวกเตอร์ ซึ่งเราจะสมมติว่าสามารถประมวลผลได้ 4 องค์กรประกอบต่อรอบสัญญาณนาฬิกา แม้ว่าจำนวนคำสั่งที่ถูกดำเนินการจริงระหว่างรันไทม์ (dynamic instruc-



ประสิทธิภาพ



ความเรียบง่าย

tion count) จะเป็นความแตกต่างที่สำคัญมาก แต่ในมุมมองของเรา นั้นยังเป็นเพียงความแตกต่างที่สำคัญอันดับสองระหว่าง SIMD กับสถาปัตยกรรมเวกเตอร์เท่านั้น ปัญหาที่สำคัญกว่าคือ การไม่มีโมเดลการเขียนโปรแกรมแบบไม่ยึดติดกับความยาวเวกเตอร์ (vector-length-agnostic programming model) ซึ่งทำให้จำนวนคำสั่งเพิ่มขึ้นอย่างมหาศาล ISA อย่าง x86-64 ที่ยึดตามแนวความคิดการพัฒนาแบบเพิ่มขยาย (incrementalist doctrine)—แทนที่จะใช้แนวคิดแบบโมดูลาร์ของ RISC-V —จำเป็นต้องสร้างสำเนาของคำสั่ง SIMD เดิมทั้งหมดที่ออกแบบมาสำหรับรีจิสเตอร์ SIMD ขนาดแคบกว่าใหม่อีกครั้ง ทุกครั้งที่มีการเพิ่มความกว้างของ SIMD เป็นสองเท่า แน่นอนว่ามีคำสั่ง x86 จำนวนหลายร้อยคำสั่งได้ถูกสร้างขึ้นตลอดวิวัฒนาการหลายยุคของ SIMD ISA และในอนาคตก็ย่อมจะมีเพิ่มขึ้นอีกหลายร้อยคำสั่ง ภาระทางความคิด (cognitive load) ที่แนวทางการพัฒนา ISA แบบ “เพิ่มคำสั่งเข้าไปเรื่อย ๆ” (brute-force approach to ISA evolution) สร้างให้กับโปรแกรมเมอร์ภาษาแอสเซมบลีนั้น ย่อมหนักหนามหาศาล ใครกันจะสามารถจดจำได้ว่าคำสั่ง `vfmadd213pd` หมายถึงอะไร และควรใช้งานเมื่อใด?

เมื่อเปรียบเทียบกันแล้ว โค้ดแบบเวกเตอร์ไม่ได้รับผลกระทบจากขนาดของรีจิสเตอร์เวกเตอร์เลย ไม่เพียงแต่โค้ดเวกเตอร์จะไม่ต้องเปลี่ยนแปลงเมื่อมีการเพิ่มขนาดหน่วยความจำสถานะเวกเตอร์เท่านั้น แต่ยังไม่ต้องคอมไพล์ใหม่อีกด้วย เนื่องจากโปรเซสเซอร์เป็นตัวกำหนดค่าความยาวเวกเตอร์สูงสุดให้กับซอฟต์แวร์เอง โค้ดในรูปที่ 8.4 จึงยังคงใช้งานได้เหมือนเดิม แม้ว่าตัวประมวลผลเพิ่มขนาดพื้นที่ฮาร์ดแวร์ที่เก็บข้อมูลและสถานะของระบบเวกเตอร์ จาก 512 บิตเป็น 2048 บิตก็ตาม

แตกต่างจาก SIMD ที่ ISA เป็นตัวกำหนดฮาร์ดแวร์ที่ต้องมีอย่างตายตัว—และเมื่อ ISA เปลี่ยน ตัวคอมไพเลอร์ก็ต้องเปลี่ยนตาม—สถาปัตยกรรมชุดคำสั่งเวกเตอร์ เปิดโอกาสให้นักออกแบบตัวประมวลผลสามารถเลือกปริมาณทรัพยากรสำหรับการประมวลผลแบบขนานข้อมูล (data parallelism) ให้เหมาะกับงานที่ต้องการได้ โดยไม่กระทบต่อโปรแกรมเมอร์หรือตัวคอมไพเลอร์อาจกล่าวได้ว่า SIMD ขัดกับหลักการออกแบบ ISA ที่กล่าวไว้ใน 1 ที่ต้องการแยกสถาปัตยกรรมออกจากรายละเอียดการการนำฮาร์ดแวร์ไปใช้งาน [Patterson and Waterman 2017]

เราเห็นว่า ความแตกต่างอย่างชัดเจนในด้านต้นทุน พลังงานต่อประสิทธิภาพ ความซับซ้อน และความง่ายในการเขียนโปรแกรม ระหว่าง แนวทางสถาปัตยกรรมเวกเตอร์แบบโมดูลาร์ของ RISC-V กับสถาปัตยกรรม SIMD แบบค่อยๆ เพิ่มขยายของ x86-64 และสถาปัตยกรรมเวกเตอร์ของ SVE ที่มีประสิทธิภาพต่ำกว่า อาจเป็นหนึ่งในเหตุผลที่น่าเชื่อถือและทรงพลังที่สุดในการสนับสนุน RISC-V



สร้างงาน

■ รายละเอียดเพิ่มเติม: RV128V

ตามที่อธิบายเพิ่มเติมไว้ตอนท้ายของบท 10 นั้น RISC-V มีทั้งรุ่นที่มีเวอร์ชันที่อยู่หน่วยความจำ 64 บิต และเวอร์ชันที่อยู่หน่วยความจำ 128 บิตของ แม้ในปัจจุบันจะยังไม่มีคำสั่งที่เป็นแรงจูงใจสำหรับตัวประมวลผลเวกเตอร์ ที่มีพื้นที่ที่อยู่หน่วยความจำขนาดใหญ่ถึงระดับนั้น แต่ตลอดอายุการใช้งานที่คาดว่าจะยาวนานของ RISC-V ก็อาจมีความต้องการดังกล่าวเกิดขึ้นได้ในอนาคต แนวคิดการกำหนดชนิดข้อมูลแบบไดนามิก ทำให้ชุดคำสั่งที่อธิบายในบทนี้ สามารถรองรับได้ทั้ง RV32V, RV64V และแม้กระทั่ง RV128V

8.14 เรียนรู้เพิ่มเติม

Oral-history:David Kuck, 1991. URL https://ethw.org/Oral-History:David_Kuck.

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, sixth edition*. Elsevier, 2019.

D. Patterson and A. Waterman. SIMD instructions considered harmful, 2017. URL <https://www.sigarch.org/simd-instructions-considered-harmful/>.

7. Vector Loads and Stores

7.4. Vector Unit-Stride Instructions

31	29	28	27	26	25	24	20	19	15	14	12	11	7	6	0	
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	0	0	0	VLE8.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	0	1	VLE16.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	1	0	VLE32.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	1	1	VLE64.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	0	0	0	VSE8.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	0	1	VSE16.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	1	0	VSE32.V
0	0	0	0	0	0	vm	0	0	0	0	0	rs1	1	1	1	VSE64.V
0	0	0	0	0	0	1	0	1	0	1	1	rs1	0	0	0	VLM.V
0	0	0	0	0	0	1	0	1	0	1	1	rs1	0	0	0	VSM.V

7.5. Vector Strided Instructions

31	29	28	27	26	25	24	20	19	15	14	12	11	7	6	0	
0	0	0	0	1	0	vm	rs2	rs1	0	0	0	vd	0	0	0	VLSE8.V
0	0	0	0	1	0	vm	rs2	rs1	1	0	1	vd	0	0	0	VLSE16.V
0	0	0	0	1	0	vm	rs2	rs1	1	1	0	vd	0	0	0	VLSE32.V
0	0	0	0	1	0	vm	rs2	rs1	1	1	1	vd	0	0	0	VLSE64.V
0	0	0	0	1	0	vm	rs2	rs1	0	0	0	vs3	0	1	0	VSSE8.V
0	0	0	0	1	0	vm	rs2	rs1	1	0	1	vs3	0	1	0	VSSE16.V
0	0	0	0	1	0	vm	rs2	rs1	1	1	0	vs3	0	1	0	VSSE32.V
0	0	0	0	1	0	vm	rs2	rs1	1	1	1	vs3	0	1	0	VSSE64.V

7.6. Vector Indexed Instructions

31	29	28	27	26	25	24	20	19	15	14	12	11	7	6	0	
0	0	0	0	0	1	vm	vs2	rs1	0	0	0	vd	0	0	0	VLUXEI8.V
0	0	0	0	0	1	vm	vs2	rs1	1	0	1	vd	0	0	0	VLUXEI16.V
0	0	0	0	0	1	vm	vs2	rs1	1	1	0	vd	0	0	0	VLUXEI32.V
0	0	0	0	0	1	vm	vs2	rs1	1	1	1	vd	0	0	0	VLUXEI64.V
0	0	0	0	0	1	vm	vs2	rs1	0	0	0	vs3	0	1	0	VSUXEI8.V
0	0	0	0	0	1	vm	vs2	rs1	1	0	1	vs3	0	1	0	VSUXEI16.V
0	0	0	0	0	1	vm	vs2	rs1	1	1	0	vs3	0	1	0	VSUXEI32.V
0	0	0	0	0	1	vm	vs2	rs1	1	1	1	vs3	0	1	0	VSUXEI64.V
0	0	0	0	1	1	vm	vs2	rs1	0	0	0	vd	0	0	0	VLOXEI8.V
0	0	0	0	1	1	vm	vs2	rs1	1	0	1	vd	0	0	0	VLOXEI16.V
0	0	0	0	1	1	vm	vs2	rs1	1	1	0	vd	0	0	0	VLOXEI32.V
0	0	0	0	1	1	vm	vs2	rs1	1	1	1	vd	0	0	0	VLOXEI64.V
0	0	0	0	1	1	vm	vs2	rs1	0	0	0	vs3	0	1	0	VSOXEI8.V
0	0	0	0	1	1	vm	vs2	rs1	1	0	1	vs3	0	1	0	VSOXEI16.V
0	0	0	0	1	1	vm	vs2	rs1	1	1	0	vs3	0	1	0	VSOXEI32.V
0	0	0	0	1	1	vm	vs2	rs1	1	1	1	vs3	0	1	0	VSOXEI64.V

รูปที่ 8.13: แผนผังรหัสดำเนินการ ของ RV32V สำหรับการโหลดและจัดเก็บ มีโครงสร้างคำสั่ง รหัสดำเนินการ ชนิดของรูปแบบคำสั่ง และชื่อคำสั่ง (ไม่แสดงหมายเลขบิตในรูปเนื่องจากเป็นภาพตัวอย่างจำลอง)

11.4. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0	
0	1	0	0	0	0	0							VADC.VVM
0	1	0	0	0	0	0							VADC.VXM
0	1	0	0	0	0	0							VADC.VIM
0	1	0	0	0	1	0							VMADC.VVM
0	1	0	0	0	1	0							VMADC.VXM
0	1	0	0	0	1	0							VMADC.VIM
0	1	0	0	1	0	0							VSBC.VVM
0	1	0	0	1	0	0							VSBC.VXM
0	1	0	0	1	1	0							VMSBC.VVM
0	1	0	0	1	1	0							VMSBC.VXM

11.5. Vector Bitwise Logical Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0	
0	0	1	0	0	1	vm							VAND.VV
0	0	1	0	0	1	vm							VAND.VX
0	0	1	0	0	1	vm							VAND.VI
0	0	1	0	1	0	vm							VOR.VV
0	0	1	0	1	0	vm							VOR.VX
0	0	1	0	1	0	vm							VOR.VI
0	0	1	0	1	1	vm							VXOR.VV
0	0	1	0	1	1	vm							VXOR.VX
0	0	1	0	1	1	vm							VXOR.VI

11.6. Vector Single-Width Bit Shift Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0	
1	0	0	1	0	1	vm							VSLL.VV
1	0	0	1	0	1	vm							VSLL.VX
1	0	0	1	0	1	vm							VSLL.VI
1	0	1	0	0	0	vm							VSRL.VV
1	0	1	0	0	0	vm							VSRL.VX
1	0	1	0	0	0	vm							VSRL.VI
1	0	1	0	0	1	vm							VSRA.VV
1	0	1	0	0	1	vm							VSRA.VX
1	0	1	0	0	1	vm							VSRA.VI

รูปที่ 8.14: แผนผังรหัสดำเนินการ ของ RV32V สำหรับการดำเนินการทางเลขคณิตและการดำเนินการเชิงตรรกะบางส่วน แสดงโครงสร้างคำสั่ง รหัสดำเนินการ ชนิดของรูปแบบคำสั่ง และชื่อคำสั่ง (ไม่แสดงหมายเลขบิตในรูป เนื่องจากเป็นภาพตัวอย่างจำลอง)

11.1. Vector Single-Width Integer Add and Subtract

31	26	25	24	20	19	15	14	12	11	7	6	0									
0	0	0	0	0	0	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VADD.VV
0	0	0	0	0	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VADD.VX
0	0	0	0	0	0	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VADD.VI
0	0	0	0	0	1	0	vm	vs2	vs1	0	0	0	vd	1	0	1	0	1	1	1	VSUB.VV
0	0	0	0	0	1	0	vm	vs2	rs1	1	0	1	vd	1	0	1	0	1	1	1	VSUB.VX
0	0	0	0	0	1	1	vm	vs2	rs1	1	0	1	vd	1	0	1	0	1	1	1	VRSUB.VX
0	0	0	0	0	1	1	vm	vs2	simm5	0	1	1	vd	1	0	1	0	1	1	1	VRSUB.VI

11.8. Vector Integer Comparison Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0									
0	1	1	0	0	0	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSEQ.VV
0	1	1	0	0	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSEQ.VX
0	1	1	0	0	0	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSEQ.VI
0	1	1	0	0	1	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSNE.VV
0	1	1	0	0	1	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSNE.VX
0	1	1	0	0	1	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSNE.VI
0	1	1	0	1	0	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSLTU.VV
0	1	1	0	1	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSLTU.VX
0	1	1	0	1	1	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSLT.VV
0	1	1	0	1	1	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSLT.VX
0	1	1	1	0	0	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSLEU.VV
0	1	1	1	0	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSLEU.VX
0	1	1	1	0	0	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSLEU.VI
0	1	1	1	0	1	vm	vs2		vs1	0	0	0	vd	1	0	1	0	1	1	1	VMSLE.VV
0	1	1	1	0	1	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSLE.VX
0	1	1	1	0	1	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSLE.VI
0	1	1	1	1	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSGTU.VX
0	1	1	1	1	0	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSGTU.VI
0	1	1	1	1	1	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VMSGT.VX
0	1	1	1	1	1	vm	vs2		simm5	0	1	1	vd	1	0	1	0	1	1	1	VMSGT.VI

11.13. Vector Single-Width Integer Multiply-Add Instructions

31	26					25	24	20				19	15			14	12		11	7			6	0				
1	0	1	1	0	1	vm	vs2				vs1		0	1	0	vd / vs3		1	0	1	0	1	1	1	VMACC.VV			
1	0	1	1	0	1	vm	vs2				rs1		1	0	1	vd / vs3		1	0	1	0	1	1	1	VMACC.VX			
1	0	1	1	1	1	vm	vs2				vs1		0	1	0	vd / vs3		1	0	1	0	1	1	1	VNMSAC.VV			
1	0	1	1	1	1	vm	vs2				rs1		1	0	1	vd / vs3		1	0	1	0	1	1	1	VNMSAC.VX			
1	0	1	0	0	1	vm	vs2				vs1		0	1	0	vd / vs3		1	0	1	0	1	1	1	VMADD.VV			
1	0	1	0	0	1	vm	vs2				rs1		1	0	1	vd / vs3		1	0	1	0	1	1	1	VMADD.VX			
1	0	1	0	0	1	vm	vs2				vs1		0	1	0	vd / vs3		1	0	1	0	1	1	1	VNMSUB.VV			
1	0	1	0	0	1	vm	vs2				rs1		1	0	1	vd / vs3		1	0	1	0	1	1	1	VNMSUB.VX			

รูปที่ 8.15: แผนผังรหัสดำเนินการ RV32V สำหรับการดำเนินการเลขคณิตบางส่วน มีโครงสร้างคำสั่ง รหัสดำเนินการ ชนิดของรูปแบบคำสั่ง และชื่อคำสั่ง (ไม่แสดงหมายเลขบิตในรูป เนื่องจากเป็นภาพตัวอย่างจำลอง)

13.11. Vector Floating-Point MIN/MAX Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0										
0	0	0	1	0	0	vm		vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFMIN.VV
0	0	0	0	1	0	vm		vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFMIN.VF
0	0	0	0	1	1	vm		vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFMAX.VV
0	0	0	1	1	0	vm		vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFMAX.VF

13.12. Vector Floating-Point Sign-Injection Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0										
0	0	1	0	0	0	vm		vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFSGNJ.VV
0	0	1	0	0	0	vm		vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFSGNJ.VF
0	0	1	0	0	1	vm		vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFSGNJN.VV
0	0	1	0	0	1	vm		vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFSGNJN.VF
0	0	1	0	1	0	vm		vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFSGNJX.VV
0	0	1	0	1	0	vm		vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFSGNJX.VF

13.13. Vector Floating-Point Compare Instructions

31	26					25	24	20			19	15		14	12		11	7			6	0				
0	1	1	0	0	0	vm		vs2			vs1		0	0	1		vd		1	0	1	0	1	1	1	VMFEQ.VV
0	1	1	0	0	0	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFEQ.VF
0	1	1	1	0	0	vm		vs2			vs1		0	0	1		vd		1	0	1	0	1	1	1	VMFNE.VV
0	1	1	1	0	0	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFNE.VF
0	1	1	0	1	1	vm		vs2			vs1		0	0	1		vd		1	0	1	0	1	1	1	VMFLT.VV
0	1	1	0	1	1	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFLT.VF
0	1	1	0	0	1	vm		vs2			vs1		0	0	1		vd		1	0	1	0	1	1	1	VMFLE.VV
0	1	1	0	0	1	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFLE.VF
0	1	1	1	0	1	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFGT.VF
0	1	1	1	1	1	vm		vs2			rs1		1	0	1		vd		1	0	1	0	1	1	1	VMFGE.VF

รูปที่ 8.16: รหัสดำเนินการ RV32V สำหรับการดำเนินการจำนวนลอยตัวบางส่วน มีโครงสร้างคำสั่ง รหัสดำเนินการ ชนิดของรูปแบบคำสั่ง และชื่อคำสั่ง (ไม่แสดงหมายเลขบิตในรูป เนื่องจากเป็นภาพตัวอย่างจำลอง)

13. Vector Floating-Point Instructions

13.2. Vector Single-Width Floating-Point Add/Subtract Instructions

31	26 25 24				20 19			15 14		12 11		7 6		0									
0	0	0	0	0	0	vm	vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFADD.VV		
0	0	0	0	0	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	VFADD.VF		
0	0	0	0	0	1	0	vm	vs2		vs1	0	0	1	vd	1	0	1	0	1	1	1	VFSUB.VV	
0	0	0	0	1	0	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	1	VFSUB.VF	
1	0	0	0	1	1	1	vm	vs2		rs1	1	0	1	vd	1	0	1	0	1	1	1	1	VFRSUB.VF

13.4. Vector Single-Width Floating-Point Multiply/Divide Instructions

31	26 25 24					20 19					15 14			12 11		7 6		0					
1	0	0	1	0	0	vm	vs2			vs1		0	0	1	vd		1	0	1	0	1	1	VFMUL.VV
1	0	0	1	0	0		vs2			rs1		1	0	1	vd		1	0	1	0	1	1	VFMUL.VF
1	0	0	0	0	0	vm	vs2			vs1		0	0	1	vd		1	0	1	0	1	1	VFDIV.VV
1	0	0	0	0	0		vs2			rs1		1	0	1	vd		1	0	1	0	1	1	VFDIV.VF
1	0	0	0	0	1	vm	vs2			rs1		1	0	1	vd		1	0	1	0	1	1	VFRDIV.VF

13.6. Vector Single-Width Floating-Point Fused Multiply-Add Instructions

31	26	25	24	20			19	15	14	12	11	7	6	0										
1	0	1	1	0	0	vm	vs2		vs1	0	0	1	vd / vs3	1	0	1	0	1	1	1	VFMACC.VV			
1	0	1	1	1	0	vm	vs2		rs1	1	0	1	vd / vs3	1	0	1	0	1	1	1	VFMACC.VF			
1	0	1	1	1	0	1	vm		vs2		vs1	0	0	1		vd / vs3	1	0	1	0	1	1	VFNMACC.VV	
1	0	1	1	1	0	1	vm		vs2		rs1	1	0	1		vd / vs3	1	0	1	0	1	1	1	VFNMACC.VF
1	0	1	1	1	1	0	vm		vs2		vs1	0	0	1		vd / vs3	1	0	1	0	1	1	1	VFMSAC.VV
1	0	1	1	1	1	0	vm		vs2		rs1	1	0	1		vd / vs3	1	0	1	0	1	1	1	VFMSAC.VF
1	0	1	1	1	1	1	vm		vs2		vs1	0	0	1		vd / vs3	1	0	1	0	1	1	1	VFNMSAC.VV
1	0	1	1	1	1	1	vm		vs2		rs1	1	0	1		vd / vs3	1	0	1	0	1	1	1	VFNMSAC.VF
1	0	1	0	0	0	vm	vs2		vs1	0	0	1	vd / vs3	1	0	1	0	1	1	1	1	VFMADD.VV		
1	0	1	0	0	0	vm	vs2		rs1	1	0	1	vd / vs3	1	0	1	0	1	1	1	1	VFMADD.VF		
1	0	1	0	0	1	vm	vs2		vs1	0	0	1	vd / vs3	1	0	1	0	1	1	1	1	VFNMADD.VV		
1	0	1	0	0	1	vm	vs2		rs1	1	0	1	vd / vs3	1	0	1	0	1	1	1	1	VFNMADD.VF		
1	0	1	0	1	0	vm	vs2		vs1	0	0	1	vd / vs3	1	0	1	0	1	0	1	1	1	VFMSUB.VV	
1	0	1	0	1	0	vm	vs2		rs1	1	0	1	vd / vs3	1	0	1	0	1	0	1	1	1	VFMSUB.VF	
1	0	1	0	1	1	vm	vs2		vs1	0	0	1	vd / vs3	1	0	1	0	1	0	1	1	1	VFNMSUB.VV	
1	0	1	0	1	1	vm	vs2		rs1	1	0	1	vd / vs3	1	0	1	0	1	0	1	1	1	VFNMSUB.VF	

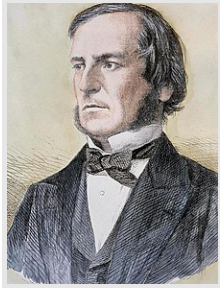
15.1. Vector Mask-Register Logical Instructions

31	26	25	24	20	19	15	14	12	11	7	6	0									
0	1	1	0	0	1	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMAND.MM
0	1	1	1	0	1	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMNAND.MM
0	1	1	0	0	0	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMANDNOT.MM
0	1	1	0	1	1	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMXOR.MM
0	1	1	0	1	0	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMOR.MM
0	1	1	1	1	0	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMINOR.MM
0	1	1	1	0	0	vm	vs2		vs1	0	1	0	vd	1	0	1	0	1	1	1	VMORNOT.MM

รูปที่ 8.17: รหัสดำเนินการ ของ RV32V สำหรับการดำเนินการของจำนวนลอยตัวและการดำเนินการกับมาส์ก แสดงโครงสร้างคำสั่ง รหัสดำเนินการ ชนิดของรูปแบบคำสั่งและชื่อคำสั่ง (ไม่แสดงหมายเลขบิตในรูป เนื่องจากเป็นภาพตัวอย่างจำลอง)

RV32B: การจัดการบิต

จอร์จ บูล (1815–1864)
เป็นนักคณิตศาสตร์ชาว
อังกฤษที่ทำคุณูปการต่อ
ตรรกศาสตร์เชิงพีชคณิต
และตรรกศาสตร์บูลถูกตั้งชื่อ
ตามชื่อของเขา



ไม่ว่าทฤษฎีคณิตศาสตร์จะถูกคิดค้นเพียงใด เราก็ไม่ควรพอใจว่ามันสมบูรณ์แบบจนกว่าจะรู้สึกว่ามัน
สวยงามด้วย

—จอร์จ บูล, ca.1854

9.1 บทนำ

RV32B เพิ่มคำสั่งการจัดการบิตหลากหลายแบบให้กับ RV32I [Marshall 2021] เช่นเดียวกับ RV32V คำสั่งเหล่านี้ครอบคลุมตั้งแต่คำสั่งที่ปกติถูกสร้างโดยคอมไพเลอร์ ไปจนถึงคำสั่งที่พบเฉพาะในรูทีนไลบรารีที่เขียนด้วยมือ การจัดการบิตยังเป็นส่วนขยายคำสั่งแรกที่ไม่เป็นแบบรวม (monolithic) โดยนักออกแบบโปรเซสเซอร์สามารถเลือกนำชุดย่อยของ RV32B ไปใช้งานได้อย่างอิสระ ณ ปี 2022 ชุดย่อยอย่างเป็นทางการทั้ง 4 คือ:

- การคำนวณที่อยู่ มีชื่อย่อว่า Zba
- การจัดการบิตทั่วไป, มีชื่อย่อว่า Zbb
- การคูณแบบไม่มีแครี่, มีชื่อย่อว่า Zbc
- การจัดการบิตเดียว, มีชื่อย่อว่า Zbs

RV32B

Address Calculation (Zba)	General Bit Manipulation (Zbb)
$\text{shift} \left\{ \begin{array}{l} \underline{1} \\ \underline{2} \\ \underline{3} \end{array} \right\} \text{ bits and } \underline{\text{add}}$	$\left\{ \begin{array}{l} \underline{\text{and}} \\ \underline{\text{or}} \\ \underline{\text{exclusive}} \\ \underline{\text{nor}} \end{array} \right\} \underline{\text{negate}}$
	$\left\{ \begin{array}{l} \underline{\text{minimum}} \\ \underline{\text{maximum}} \end{array} \right\} \underline{\text{unsigned}}$
Carryless Multiply (Zbc)	$\underline{\text{rotate}} \left\{ \begin{array}{l} \underline{\text{left}} \\ \underline{\text{right}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{immediate}} \end{array} \right\}$
$\underline{\text{carryless}} \underline{\text{multiply}} \left\{ \begin{array}{l} \underline{\text{upper}} \underline{\text{half}} \\ \underline{\text{reversed}} \end{array} \right\}$	$\underline{\text{sign}} \underline{\text{extend}} \left\{ \begin{array}{l} \underline{\text{byte}} \\ \underline{\text{half}} \end{array} \right\}$
Single Bit Manipulation (Zbs)	$\underline{\text{count}} \left\{ \begin{array}{l} \underline{\text{leading}} \\ \underline{\text{trailing}} \end{array} \right\} \underline{\text{zeros}}$
$\underline{\text{bit}} \left\{ \begin{array}{l} \underline{\text{set}} \\ \underline{\text{clear}} \\ \underline{\text{invert}} \\ \underline{\text{extract}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{immediate}} \end{array} \right\}$	$\underline{\text{count}} \underline{\text{population}} \text{ of } 1 \text{ bits}$
	$\underline{\text{reverse}} \underline{\text{g}} \text{ bits}$
	$\underline{\text{cyclic}} \underline{\text{redundancy}} \underline{\text{check}} \underline{32} \text{ bits } \underline{\text{byte}}$

รูปที่ 9.1: แผนผังคำสั่ง RV32B

รูปที่ 9.1 เป็นการแสดงกราฟิกของชุดคำสั่งส่วนขยาย RV32B พร้อมระบุชุดย่อย และรูปที่ 9.2 แสดงรายการรหัสดำเนินการของคำสั่งเหล่านี้ ส่วนที่เหลือของบทนี้จะทบทวนชุดย่อยทั้งสิ้น

9.2 Zba: การคำนวณที่อยู่

คำสั่ง `sh1add`, `sh2add` และ `sh3add` จะเลื่อนบิตในรีจิสเตอร์ `rs1` ไปทางซ้าย 1, 2 หรือ 3 บิต แล้วบวกค่าที่เลื่อนเข้ากับรีจิสเตอร์ `rs2` การใช้งานที่ได้รับความนิยมมากที่สุดคือ การแปลงดัชนีของฮาล์ฟเวิร์ด (halfwords), เวิร์ด (words) และ ดับเบิลเวิร์ด (doublewords) เป็นที่อยู่หน่วยความจำแบบไบต์

9.3 Zbb: การจัดการบิตทั่วไป

ชุดย่อย RV32B นี้มีคำสั่งมากที่สุด คำสั่งเหล่านี้ครอบคลุมตั้งแต่คำสั่งที่พบทั่วไปในคอมพิวเตอร์ ซึ่งสร้างโดยคอมไพเลอร์ ไปจนถึงคำสั่งที่พบเฉพาะในไลบรารีที่เขียนด้วยมือ และไม่มีในสถาปัตยกรรมอื่น

การดำเนินการตรรกะแบบกลับค่า คำสั่ง `andn`, `orn` และ `xnor` ดำเนินการตรรกะทั่วไป เช่น AND, OR และ XOR โดยตัวถูกดำเนินการทั้งสองจะถูกกลับค่า

การหาค่ามากที่สุดหรือน้อยสุด คำสั่ง `min`, `minu`, `max` และ `maxu` หาค่าน้อยสุดและมากที่สุดระหว่างเรจิสเตอร์ต้นทางสองตัว โดยข้อมูลอาจเป็นจำนวนเต็มที่มีเครื่องหมายหรือไม่มีเครื่องหมาย

การหมุนบิต คำสั่ง `rol` และ `ror` ทำการหมุนบิตไปทางซ้ายหรือขวาในรีจิสเตอร์ต้นทาง โดยจำนวนการหมุนระบุในรีจิสเตอร์อื่นหรือในฟิลด์ค่าทันที (immediate field)

การขยายเครื่องหมายและขยายศูนย์ คำสั่ง `sext.b` และ `sext.h` ทำการขยายเครื่องหมายของไบต์ หรือฮาล์ฟเวิร์ดที่มีนัยสำคัญน้อยที่สุดในรีจิสเตอร์ให้เต็มความกว้างของมัน คำสั่ง `zext.h` ทำการขยายศูนย์ของไบต์ที่มีนัยสำคัญน้อยที่สุดในรีจิสเตอร์ให้เต็มความกว้างของมัน

การนับบิตศูนย์ คำสั่ง `clz` จะนับจำนวนศูนย์นำหน้าที่อยู่ก่อนบิตแรกซึ่งเป็นหนึ่ง โดยเริ่มต้นจากบิตที่มีค่านัยสำคัญสูงสุด (most significant bit) ผลลัพธ์คือจำนวนศูนย์ตามหลังที่อยู่ก่อนบิตแรกซึ่งเป็นหนึ่ง โดยเริ่มต้นจากบิตที่มีค่านัยสำคัญต่ำสุด (least significant bit) คำสั่ง `ctz` จะนับจำนวนศูนย์จากส่วนท้าย (trailing zeros) ทั้งสองกรณีหากค่าที่ถูกประเมินเป็น 0 ผลลัพธ์จะเท่ากับความกว้างของเรจิสเตอร์ และหากค่าเป็น -1 ผลลัพธ์จะเป็น 0 คำสั่งเหล่านี้ถูกใช้สำหรับการสแกนบิตแมป —ซึ่งถูกใช้โดย `malloc`— สำหรับการหาตัวหารร่วมมาก (greatest common divisor) คิวลำดับความสำคัญ (priority queues) และไลบรารีซอฟต์แวร์ที่เลียนแบบการคำนวณจุดลอยตัว (floating point arithmetic) **การนับบิตหนึ่ง** คำสั่ง `popc` นับจำนวนบิต 1 ในรีจิสเตอร์ (population of 1s) ใช้สำหรับการคำนวณน้ำหนักแฮมมิง (Hamming weight) และอัลกอริทึมอื่นๆ

การกลับลำดับและ CRC คำสั่ง `rev8` กลับลำดับของไบต์ในรีจิสเตอร์ จึงทำการสลับระหว่างการแทน `big endian` และ `little endian` คำสั่ง `CRC32.b` ดำเนินการลดพหุนามของรีจิสเตอร์ที่ตีความเป็นสถานะ CRC32 และเลื่อน 8 บิต ใช้สำหรับการคำนวณ CRC ในโปรเซสเซอร์ RISC-V ที่ไม่รวมคำสั่งคูณแบบไม่มีแครี่ ซึ่งซึ่งจะกล่าวถึงต่อไป

9.4 Zbc: การคูณแบบไม่มีแครี่

การคูณแบบไม่มีแครี่ (Carryless multiplication) คือการคูณในวงแหวนพหุนามเหนือ $GF(2)$ (polynomial ring over $GF(2)$) ตัวอย่างแอปพลิเคชันรวมถึง hashing การสร้างตัวเลขสุ่มเทียม (pseudo-random number generation) การตรวจสอบ Cyclic Redundancy Check (CRC) โหมด Galois/Counter (GCM) รหัสแก้ไขข้อผิดพลาด (erasure codes), และรหัสเกรย์ (gray codes) การคูณแบบนี้คล้ายการคูณปกติยกเว้นว่าแครี่ออกถูกทิ้ง (carry is discarded) หรือเรียกว่า *การคูณ XOR* เพราะการบวกแบบทิ้งแครี่เทียบเท่ากับการดำเนินการ XOR

คำสั่ง `clmul` สร้างครึ่งล่างของผลคูณแบบไม่มีแครี่ และ `clmulh` สร้างครึ่งบน หากความกว้างของรีจิสเตอร์คือ $XLEN$ แล้วคำสั่ง `clmulr` สร้างบิตตั้งแต่ $2 \times XLEN - 2$ ถึง $XLEN - 1$ ของผลคูณแบบไม่มีแครี่ขนาด $2 \times XLEN$ (หมายเหตุ: `clmulh = clmulr` ที่เลื่อนขวา 1 บิต)

9.5 Zbs: การจัดการบิตเดียว

คำสั่ง `bset`, `bclr`, `binv`, และ `bext` ตั้งค่าบิต ล้างบิต สลับค่าบิต หรือดึงบิตเดียวในรีจิสเตอร์ตามดัชนีบิต (bit index) ที่ระบุในรีจิสเตอร์หรือในฟิลด์ค่าทันที

9.6 ข้อสังเกตสรุป

หากคุณไม่สามารถแก้ปัญหาวิศวกรรมที่ยากๆ ให้เหลือแค่กระดาษขนาด $8\frac{1}{2} \times 11$ นิ้วแผ่นเดียวได้
คุณคงไม่เข้าใจปัญหานั้น

—ราล์ฟ บราเซลตัน เพ็ค, วิศวกรโยธาที่มีชื่อเสียง



ประสิทธิ์ภาพ

เพื่อสนับสนุนการดำเนินการตรรกะทั้งแบบเรียบง่ายและซับซ้อน RV32B มีชุดคำสั่งหลากหลายให้สถาปนิก RISC-V เลือกใช้งาน

(ตารางที่ 2.1 ของ [Marshall 2021] เป็นที่มาของรูปนี้)

9.7 เรียนรู้เพิ่มเติม

B. Marshall, editor. *RISC-V Bit-Manipulation ISA-extensions Version 1.0.0*. June 2021. URL <https://github.com/riscv/riscv-bitmanip>.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
0010000		rs2			rs1		010		rd		0110011			R sh1add
0010000		rs2			rs1		100		rd		0110011			R sh2add
0010000		rs2			rs1		110		rd		0110011			R sh3add
0000101		rs2			rs1		001		rd		0110011			R clmul
0000101		rs2			rs1		010		rd		0110011			R clmulr
0000101		rs2			rs1		011		rd		0110011			R clmulh
0100100		rs2			rs1		001		rd		0110011			R bclr
0010100		rs2			rs1		001		rd		0110011			R bset
0110100		rs2			rs1		001		rd		0110011			R binv
0100100		rs2			rs1		101		rd		0110011			R bext
01001		shamt			rs1		001		rd		0010011			I bclri
00101		shamt			rs1		001		rd		0010011			I bseti
01101		shamt			rs1		001		rd		0010011			I binvi
01001		shamt			rs1		101		rd		0010011			I bexti
0100000		rs2			rs1		111		rd		0110011			R andn
0100000		rs2			rs1		110		rd		0110011			R orn
0100000		rs2			rs1		100		rd		0110011			R xnor
0000101		rs2			rs1		100		rd		0110011			R min
0000101		rs2			rs1		101		rd		0110011			R minu
0000101		rs2			rs1		110		rd		0110011			R max
0000101		rs2			rs1		111		rd		0110011			R maxu
0110000		rs2			rs1		001		rd		0110011			R rol
0110000		rs2			rs1		101		rd		0110011			R ror
01100		shamt			rs1		101		rd		0010011			I rori
0110000		00100			rs1		001		rd		0010011			I sext.b
0110000		00101			rs1		001		rd		0010011			I sext.h
0000100		00000			rs1		100		rd		0110011			I zext.h
0110000		00000			rs1		001		rd		0010011			I clz
0110000		00001			rs1		001		rd		0010011			I ctz
0110000		00010			rs1		001		rd		0010011			I cpop
01101		0011000			rs1		101		rd		0010011			I rev8
not part of spec anymore (now Zbr)?														
0110000		10000			rs1		001		rd		0010011			I crc32.b

รูปที่ 9.2: แผนผังรหัสดำเนินการของ RV32B ประกอบด้วยโครงสร้างคำสั่ง, รหัสดำเนินการ (opcodes), ประเภทรูปแบบ (format type), และชื่อคำสั่ง (names)

RV64: ชุดคำสั่งแอดเดรสแบบ 64 บิต

ซี. กอร์ดอน เบลล์ (เกิด ค.ศ.1934-) เป็นหนึ่งในสถาปนิกหลักของสถาปัตยกรรมมินิคอมพิวเตอร์ ที่ได้รับความนิยมมากที่สุดในยุคนั้น สองระบบ: PDP-11 ของบริษัท Digital Equipment Corporation (แอดเดรส 16 บิต) เปิดตัวในปี 1970 และระบบรุ่นถัดมาในอีกเจ็ดปีต่อมา คือ VAX-11 (แอดเดรส 32 บิต) ซึ่งย่อมาจาก Virtual Address eXtension



มีความผิดพลาดเพียงอย่างเดียวในการออกแบบคอมพิวเตอร์ ที่แก้ไขได้ยากมากภายหลัง คือ—การมีจำนวนบิตของแอดเดรสไม่เพียงพอ สำหรับการอ้างอิงหน่วยความจำและการจัดการหน่วยความจำ
—ซี. กอร์ดอนเบลล์ ,1976

10.1 บทนำ

รูปที่ 10.1 ถึง 10.4 แสดงภาพรวมของชุดคำสั่ง RV64G ซึ่งเป็นรุ่น 64 บิตของคำสั่ง RV32G ภาพรวมเหล่านี้แสดงให้เห็นถึงการเปลี่ยนไปใช้ ISA แบบ 64 บิตใน RISC-V ต้องเพิ่มจำนวนคำสั่งเพียงเล็กน้อยเท่านั้น โดยทั่วไป ISA จะเพิ่มคำสั่งแบบ word, doubleword หรือ long จากคำสั่งเดิม 32 บิต และขยายรีจิสเตอร์ทั้งหมด รวมถึงขยายตัวนับโปรแกรม (Program Counter: PC) ให้เป็น 64 บิต ดังนั้นคำสั่ง sub ใน RV64I จะทำการลบจำนวน 64 บิต แทนที่จะเป็นการลบจำนวน 32 บิตสองจำนวนอย่างใน RV32I แม้ว่า RV64 เป็น ISA ที่มีความใกล้เคียงกับ RV32 มาก แต่ไม่เหมือนกันทั้งหมด มีการเพิ่มคำสั่งบางส่วน และคำสั่งพื้นฐานบางคำสั่งมีพฤติกรรมเปลี่ยนไปเล็กน้อย

ตัวอย่างเช่น การเรียงลำดับแบบแทรก (Insertion Sort) สำหรับ RV64I ในรูปที่ 10.10 มีความใกล้เคียงกับโค้ดสำหรับ RV32I ในรูปที่ 2.8 ในหน้า 34 ในบทที่ 2 โดยมีจำนวนคำสั่งและจำนวนไบต์เท่ากัน การเปลี่ยนแปลงเพียงอย่างเดียว คือคำสั่งโหลดและจัดเก็บทีกเวิร์ด (word) กลายเป็นโหลดและจัดเก็บแบบดับเบิลเวิร์ด (doublewords) และการเพิ่มที่อยู่เปลี่ยนจาก 4 ไบต์สำหรับเวิร์ด เป็น 8 ไบต์สำหรับดับเบิลเวิร์ด รูปที่ 10.5 แสดงรายการรหัสดำเนินการของคำสั่ง RV64GC ในรูปที่ 10.1 ถึง 10.4

แม้ว่า RV64I จะมีแอดเดรส 64 บิต และขนาดข้อมูลเริ่มต้นเป็น 64 บิต แต่ข้อมูลแบบ 32 บิต (word) ยังคงเป็นชนิดข้อมูลที่ใช้งานในโปรแกรม ดังนั้น RV64I ต้องรองรับเวิร์ด (word) เช่นเดียวกับที่ RV32I ต้องรองรับไบต์และฮาร์ฟเวิร์ด (half word) เนื่องจากรีจิสเตอร์มีความกว้างเป็น 64 บิตแล้ว RV64I จึงเพิ่มคำสั่งแบบเวิร์ดของคำสั่ง บวกและลบ: addw, addiw, subw โดยผลลัพธ์จะถูกตัดให้เหลือ 32 บิต (truncate) แล้วขยายเครื่องหมาย (sign-extend) กลับเป็น 64 บิต ก่อนเก็บในรีจิสเตอร์ปลายทาง RV64I ยังมีคำสั่งเลื่อนบิตแบบเวิร์ด เพื่อได้ผลลัพธ์แบบ 32 บิต sllw, slliw, srlw, srliw, srav, sraiw สำหรับการย้ายโอนข้อมูล 64 บิต ใช้คำสั่งโหลดและจัดเก็บแบบดับเบิล

RV64I

Integer Computation

add { immediate } { word }

subtract { word }

{ and
or
exclusive or } { immediate }

{ shift left logical
shift right arithmetic
shift right logical } { immediate } { word }

load upper immediate

add upper immediate to pc

set less than { immediate } { unsigned }

Control transfer

branch { equal
not equal }

branch { greater than or equal
less than } { unsigned }

jump and link { register }

Loads and Stores

{ load
store } { byte
halfword
word
doubleword }

load { byte
halfword
word } { unsigned }

Miscellaneous instructions

fence loads & stores

fence instruction & data

environment { break
call }

control status register { read & clear bit
read & set bit
read & write } { immediate }

รูปที่ 10.1: แผนภาพคำสั่ง RV64I ตัวอักษรที่ขีดเส้นใต้ นำมาต่อกันจากซ้ายไปขวา เพื่อสร้างชื่อคำสั่ง RV64I ส่วนที่แสดงแบบจาง คือคำสั่งเดิมของ RV32I ซึ่งถูกขยายให้ทำงานกับรีจิสเตอร์ 64 บิต ส่วนที่แสดงสีเข้ม (แดง) คือคำสั่งใหม่ที่เพิ่มเข้ามาใน RV64I

RV64M

multiply { word }

multiply high { unsigned
signed unsigned }

divide
remainder } { unsigned } { word }

atomic memory operation

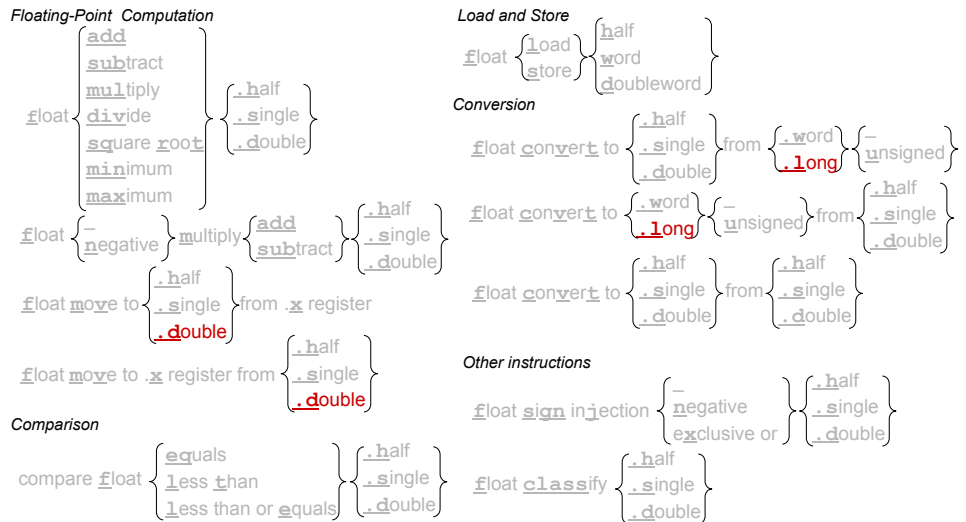
RV64A

{ add
and
or
swap
xor
maximum
maximum unsigned
minimum
minimum unsigned } { .word
.doubleword }

{ load reserved
store conditional } { .word
.doubleword }

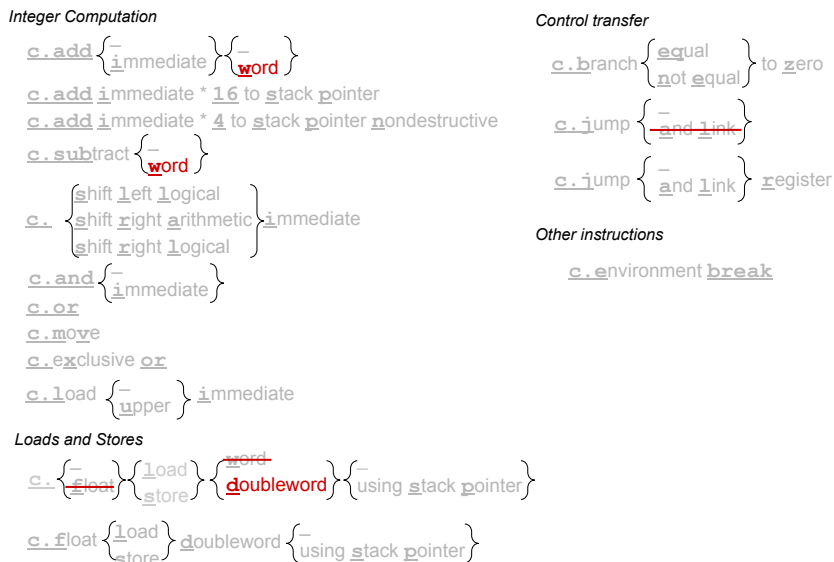
รูปที่ 10.2: แผนภาพของคำสั่ง RV64M และ RV64A

Zfh, RV64F, and RV64D



รูปที่ 10.3: แผนภาพของคำสั่ง Zfh, RV64F, และ RV64D

RV64C



รูปที่ 10.4: แผนภาพของคำสั่ง RV64C

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]						rs1	110		rd		0000011			I lwu
imm[11:0]						rs1	011		rd		0000011			I ld
imm[11:5]					rs2	rs1	011		imm[4:0]		0100011			S sd
000000					shamt	rs1	001		rd		0010011			I slli
000000					shamt	rs1	101		rd		0010011			I srlr
010000					shamt	rs1	101		rd		0010011			I srar
imm[11:0]						rs1	000		rd		0011011			I addiw
0000000					shamt	rs1	001		rd		0011011			I slliw
0000000					shamt	rs1	101		rd		0011011			I srliw
0100000					shamt	rs1	101		rd		0011011			I srarw
0000000					rs2	rs1	000		rd		0111011			R addw
0100000					rs2	rs1	000		rd		0111011			R subw
0000000					rs2	rs1	001		rd		0111011			R sllw
0000000					rs2	rs1	101		rd		0111011			R srlw
0100000					rs2	rs1	101		rd		0111011			R srarw

RV64M Standard Extension (in addition to RV32M)

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

RV64A Standard Extension (in addition to RV32A)

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

RV64F Standard Extension (in addition to RV32F)

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

RV64D Standard Extension (in addition to RV32D)

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

รูปที่ 10.5: แผนผังภาพรหัสดำเนินการ ของ RV64 ครอบคลุมทั้ง คำสั่งพื้นฐานและส่วนขยายแบบเลือกได้ แสดงโครงสร้างของคำสั่ง รหัสดำเนินการ ประเภทของรูปแบบคำสั่ง และชื่อคำสั่ง (อ้างอิงจากตารางที่ 19.2 ของ [Waterman and Asanović 2017])

เวิร์ด (doubleword) `ld` และ `sd` สู้ตาย เช่นเดียวกับ RV32I ที่มีคำสั่งโหลดไบต์แบบไม่มีเครื่องหมาย และฮาร์ฟเวิร์ดแบบไม่มีเครื่องหมาย ใน RV64I จึงมีคำสั่งโหลดเวิร์ดแบบไม่มีเครื่องหมาย: `lduw`

ด้วยเหตุผลที่คล้ายกัน RV64M จำเป็นต้องเพิ่มคำสั่งแบบเวิร์ด (word) สำหรับการคูณ การหาร และเศษ: `mulw`, `divw`, `divuw`, `remw`, `remuw` เพื่อรองรับการชิงโครโนซ์ ทั้งเวิร์ด (word) และดับเบิลเวิร์ด (double word) RV64A จึงเพิ่มคำสั่งแบบดับเบิลเวิร์ด (double word) ของคำสั่งทั้ง 11 คำสั่ง

`Zfh`, RV64F และ RV64D เพิ่มการแปลงข้อมูลที่เกี่ยวข้องกับจำนวนเต็มดับเบิลเวิร์ด (convert instructions) โดยเรียกว่า *longs* เพื่อป้องกันความสับสนกับข้อมูลความแม่นยำสองเท่า: แปลงเป็น *long* แบบมีเครื่องหมาย (`fcvt.lh`, `fcvt.ls`, `fcvt.ld`) แปลงเป็น *long* แบบไม่มีเครื่องหมาย (`fcvt.lu.h`, `fcvt.lu.s`, `fcvt.lu.d`) และแปลงจาก *longs* แบบมีเครื่องหมาย (`fcvt.h.l`, `fcvt.s.l`, `fcvt.d.l`) และแปลงจาก *long* แบบไม่มีเครื่องหมาย (`fcvt.h.lu`, `fcvt.s.lu`, `fcvt.d.lu`) เนื่องจากรีจิสเตอร์จำนวนเต็ม $\times$ มีความกว้างเป็น 64 บิตแล้ว จึงสามารถเก็บค่าจำนวนลอยตัวความแม่นยำสองเท่าได้ ดังนั้น RV64D จึงเพิ่มคำสั่งการย้ายจุดทศนิยม 2 คำสั่งคือ `fmv.x.d` และ `fmv.d.x`

ข้อยกเว้นเดียวสำหรับความสัมพันธ์แบบซูเปอร์เซต (superset) ระหว่าง RV64 และ RV32 คือ คำสั่งแบบบีบอัด (RV32C) RV64C ได้แทนที่คำสั่งบางส่วนของ RV32C เนื่องจากมีคำสั่งอื่นช่วยลดขนาดโค้ดได้มากกว่าคำสั่งหน่วยความจำที่อยู่ 64 บิต RV64C จึงตัดคำสั่ง กระโดดและเชื่อมโยงแบบบีบอัด (`c.jal`) และคำสั่งโหลดและจัดเก็บเวิร์ดสำหรับทั้งจำนวนเต็มและจุดทศนิยม (`c.lw`, `c.sw`, `c.lwsp`, `c.swsp`, `c.flw`, `c.fsw`, `c.flwsp`, `c.fswsp`) ออกไป และเพิ่มคำสั่งการบวกและลบแบบเวิร์ด ที่นิยมใช้งานมากกว่าแทน (`c.addw`, `c.addiw`, `c.subw`) และเพิ่มคำสั่งโหลดและจัดเก็บแบบดับเบิลเวิร์ด (`c.ld`, `c.sd`, `c.ldsp`, `c.sdsp`)

รูปที่ 10.7 แผนภาพของ RV64Zba, RV64Zbb และ RV64Zbs แสดงลักษณะคล้ายกับแผนภาพก่อนหน้านี้ แต่เพิ่มคำสั่งแบบเวิร์ด (word) ของคำสั่งเดิม โดยเพิ่มคำสั่งที่ใช้ในการคำนวณค่านอนแอดเดรสด้วย offset แบบไม่มีเครื่องหมาย (unsigned offsets) ได้แก่ คำสั่ง (`sh1add.uw`, `sh2add.uw`, `sh3add.uw`, `slli.uw`), คำสั่งหมุนบิต ได้แก่ คำสั่ง (`rolw`, `ror[i]w`), คำสั่งนับบิต ได้แก่ คำสั่ง (`clzw`, `ctzw`, `cpopw`) และคำสั่งจัดการบิตเดี่ยว ได้แก่ คำสั่ง (`bclr[i]w`, `bset[i]w`, `binv[i]w`, `bextw`) ส่วน RV64Zbc ไม่มีคำสั่งใหม่เพิ่มจาก RV32Zbc รูปที่ 10.6 รายการรหัสดำเนินการ RV64B

■ รายละเอียดเพิ่มเติม: ABI ของ RV64 มีดังนี้ *lp64*, *lp64f*, และ *lp64d*

lp64 หมายความว่า ชนิดข้อมูล *long* และ *pointer* ในภาษา C มีขนาด 64 บิต ส่วน *int* ยังคงเป็น 32 บิต ตัวอักษรต่อท้าย *f* และ *d* ใช้ระบุวิธีการส่งอาร์กิวเมนต์แบบจุดลอยตัว ซึ่งเหมือนกับที่ใช้ใน RV32 (ดูบทที่ 3)



ขนาดโปรแกรม

RV64B Standard Extension (in addition to RV32B)

31	25	24	20	19	15	14	12	11	7	6	0
0010000	rs2			rs1		010		rd		0111011	R sh1add.uw
0010000	rs2			rs1		100		rd		0111011	R sh2add.uw
0010000	rs2			rs1		110		rd		0111011	R sh3add.uw
0110000	rs2			rs1		001		rd		0111011	R rolw
0110000	rs2			rs1		101		rd		0111011	R rorw
0110000	shamt			rs1		101		rd		0011011	I roriw
0110000	00000			rs1		001		rd		0011011	I clzw
0110000	00001			rs1		001		rd		0011011	I ctzw
0110000	00010			rs1		001		rd		0011011	I cpopw
0000100	rs2			rs1		000		rd		0111011	R add.uw
0000010	shamt			rs1		001		rd		0011011	I slli.uw

รูปที่ 10.6: แผนผังรหัสดำเนินการของการจัดการบิตของ แสดงโครงสร้างคำสั่ง, รหัสดำเนินการ, รูปแบบคำสั่ง และชื่อคำสั่ง (อ้างอิงจากตาราง 19.2 ของ [Waterman and Asanović 2017])

RV64B

Address Calculation (Zba)

$\text{shift} \left\{ \begin{array}{l} \underline{1} \\ \underline{2} \\ \underline{3} \end{array} \right\} \text{ bits and } \text{add} \left\{ \begin{array}{l} - \\ \text{word} \end{array} \right\}$

$\text{add, unsigned word}$
 $\text{shift logical left immediate, unsigned word}$

Carry-less Multiply (Zbc)

$\text{carry-less multiply} \left\{ \begin{array}{l} - \\ \text{upper half} \\ \text{reversed} \end{array} \right\}$

Single Bit Manipulation (Zbs)

$\text{bit} \left\{ \begin{array}{l} \text{set} \\ \text{clear} \\ \text{invert} \\ \text{extract} \end{array} \right\} \left\{ \begin{array}{l} - \\ \text{immediate} \end{array} \right\}$

General Bit Manipulation (Zbb)

$\left\{ \begin{array}{l} \text{and} \\ \text{or} \\ \text{exclusive or} \end{array} \right\} \text{negate}$
 $\left\{ \begin{array}{l} \text{minimum} \\ \text{maximum} \end{array} \right\} \left\{ \begin{array}{l} - \\ \text{unsigned} \end{array} \right\}$
 $\text{rotate} \left\{ \begin{array}{l} \text{left} \\ \text{right} \end{array} \right\} \left\{ \begin{array}{l} - \\ \text{immediate} \end{array} \right\} \left\{ \begin{array}{l} - \\ \text{word} \end{array} \right\}$

$\text{sign extend} \left\{ \begin{array}{l} \text{.byte} \\ \text{.half} \end{array} \right\}$

$\text{count} \left\{ \begin{array}{l} \text{leading} \\ \text{trailing} \end{array} \right\} \text{zeros} \left\{ \begin{array}{l} - \\ \text{word} \end{array} \right\}$

$\text{count population of 1 bits} \left\{ \begin{array}{l} - \\ \text{word} \end{array} \right\}$

$\text{reverse 8 bits} \left\{ \begin{array}{l} - \\ \text{word} \end{array} \right\}$

$\text{cyclic redundancy check 32 bits, byte}$

รูปที่ 10.7: แผนผังของคำสั่ง RV64B คำสั่ง add.uw และ slli.uw มีลักษณะเหมือนกับคำสั่ง add และ slli แต่จะล้างครั้งบนของตัวถูกดำเนินการ 64 บิตก่อนแล้วจึงทำการบวกหรือเลื่อนบิต

■ รายละเอียดเพิ่มเติม: ไม่มีแผนภาพคำสั่งสำหรับ RV64V

เนื่องจาก มีรูปแบบเหมือนกับ RV32V ทุกประการ เพราะใช้กลไกการเลือกขนาดข้อมูล(element width) แบบไดนามิก ความแตกต่างเพียงอย่างเดียวคือ ฟิลด์ `vill` ในรีจิสเตอร์ `vtype` ในรูปที่ 8.2 ถูกย้ายไปอยู่ที่ บิตที่ 63

10.2 การเปรียบเทียบกับ ISA 64 บิตอื่นโดยใช้การเรียงลำดับแบบแทรก

ตามที่ กอร์ ดอน เบลล์ กล่าว ไว้ใน ตอน ต้น ของ บท นี้ ความ ผิด พลาด ร้าย แรง ของ สถาปัตยกรรมคือ การมีจำนวนบิตของแอดเดรสไม่เพียงพอ เมื่อโปรแกรมเริ่มถึงขีดจำกัด ของ 32 บิต สถาปนิกจึงพัฒนา ISA แบบ 64 บิตขึ้นมา [Mashey 2009]

สถาปัตยกรรมที่เริ่มใช้ก่อนคือ MIPS ในปี ค.ศ. 1991 โดยได้ขยายรีจิสเตอร์ทั้งหมดและ ตัวนับโปรแกรม (program counter) จาก 32 บิตเป็น 64 บิต และเพิ่มคำสั่ง 64 บิตของ MIPS-32 คำสั่งภาษาแอสเซมบลีของ MIPS-64: มักขึ้นต้นด้วยตัวอักษร “d” เช่น `daddu` หรือ `dsll` (ดูที่ 10.12) โปรแกรมเมอร์สามารถใช้คำสั่ง MIPS-32 และ MIPS-64 รวมกัน ในโปรแกรมเดียวได้นอกจากนี้ MIPS-64 ยังยกเลิกช่องว่างของคำสั่งโหลด (load delay slot) ที่มีอยู่ใน MIPS-32 (เนื่องจากไปป์ไลน์จะเกิดการหยุดชะงัก (stall) เมื่อคำสั่งถัดไป ต้อง “อ่านค่า” ที่คำสั่งก่อนหน้านี้ “ยังเขียนไม่เสร็จ” (read-after-write dependence))

หนึ่งทศวรรษต่อมา ก็ถึงเวลาของสถาปัตยกรรมรุ่นถัดไปของ x86-32 เมื่อสถาปนิกเพิ่ม ขนาดที่อยู่ พวกเขาใช้โอกาสนี้ปรับปรุงสถาปัตยกรรม x86-64 เพิ่มเติม ดังนี้:

- เพิ่มจำนวนรีจิสเตอร์จำนวนเต็มจาก 8 เป็น 16 (`r8-r15`);
- เพิ่มจำนวนรีจิสเตอร์ SIMD จาก 8 เป็น 16 (`xmm8-xmm15`);
- เพิ่มการอ้างอิงที่อยู่ข้อมูลสัมพันธ์กับ PC (PC-relative addressing) เพื่อรองรับโค้ด ที่ไม่ขึ้นกับตำแหน่ง (position-independent code) ได้ดีขึ้น

การปรับปรุงเหล่านี้ช่วยขจัดข้อบกพร่องบางประการของ x86-32

คุณสามารถเห็นข้อดีได้จากการเปรียบเทียบการเรียงลำดับแบบแทรก บน x86-32 ใน รูปที่ 2.11 หน้าที่ 37 บทที่ 2 กับ x86-64 ในรูปที่ 10.13 ใน ISA รุ่นใหม่ ตัวแปรทั้งหมดถูก เก็บไว้ในรีจิสเตอร์ แทนที่จะเก็บบางส่วนในหน่วยความจำ ส่งผลให้จำนวนคำสั่งลดลงจาก 20 คำสั่งเป็น 15 คำสั่ง แต่ขนาดโค้ดกลับใหญ่ขึ้น 1 ไบต์ เป็น 46 ไบต์ จากเดิม 45 ไบต์ เนื่องจาก x86-64 ต้องรองรับรีจิสเตอร์ที่เพิ่มขึ้น จึงต้องเพิ่มไบต์หน้าคำสั่ง (prefix byte) เพื่อระบุรหัสดำเนินการใหม่ของคำสั่ง ทำให้ความยาวเฉลี่ยของคำสั่งใน x86-64 มากกว่า x86-32

ARM เผชิญปัญหาขนาดที่อยู่ในอีกหนึ่งทศวรรษต่อมา แทนที่จะพัฒนา ISA เดิมให้ รองรับที่อยู่ 64 บิต เหมือนกับ x86-64 พวกเขาใช้โอกาสนี้สร้าง ISA ใหม่ทั้งหมด ด้วยการ เริ่มต้นใหม่ ARM สามารถแก้ข้อจำกัดหลายอย่างของ ARM-32 และออกแบบให้เป็น ISA ที่ ทันสมัยมากขึ้น

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

รูปที่ 10.8: จำนวนคำสั่งและขนาดโค้ดของการเรียงลำดับแบบแทรก สำหรับ ISA ทั้งสี่แบบ โดย ARM Thumb-2 และ microMIPS เป็น ISA แบบที่อยู่ 32 บิต จึงไม่สามารถใช้กับ ARM-64 และ MIPS-64 ได้

- เพิ่มจำนวนรีจิสเตอร์จำนวนเต็มจาก 15 เป็น 31
- สบต่วนับโปรแกรม (PC) ออกจากชุดรีจิสเตอร์
- จัดให้มีรีจิสเตอร์ที่ตั้งค่าเป็นศูนย์อย่างถาวรสำหรับคำสั่งส่วนใหญ่ (r31);
- ต่างจาก ARM-32 โหมดการกำหนดที่อยู่ข้อมูลใน ARM-64 ทำงานได้กับทุกขนาดและประเภทของข้อมูล
- ARM-64 ยกเลิกคำสั่งโหลดและบันทึกลายรายการของ ARM-32
- ARM-64 ไม่รวมตัวเลือกการทำงานแบบมีเงื่อนไขของคำสั่ง ARM-32

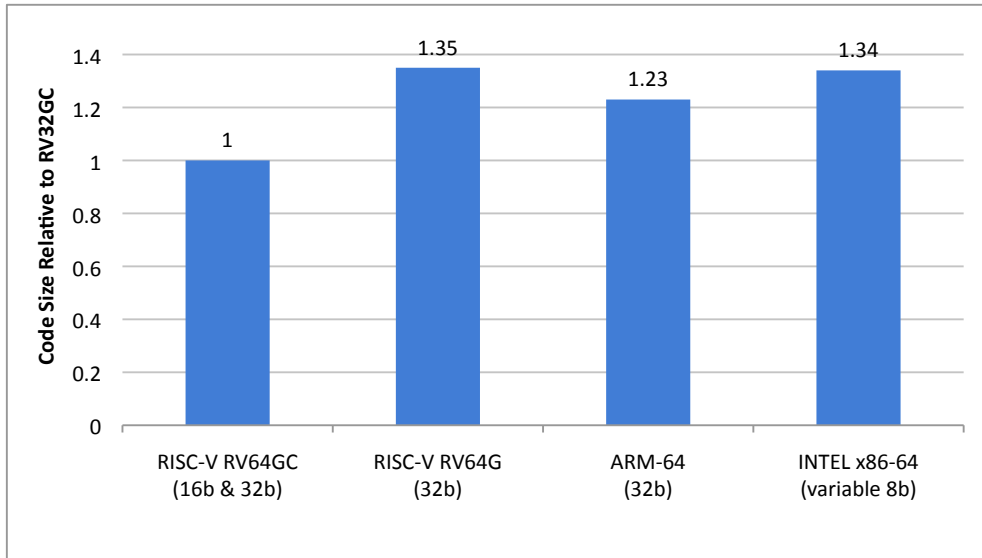
อย่างไรก็ตาม ARM-64 ยังคงมีข้อจำกัดบางประการที่สืบทอดมาจาก ARM-32 ได้แก่ การใช้รหัสเงื่อนไขสำหรับคำสั่งกระโดด (branch), ฟลัดรีจิสเตอร์ต้นทางและปลายทางมีตำแหน่งเปลี่ยนไปในรูปแบบคำสั่ง, คำสั่งย้ายข้อมูลแบบมีเงื่อนไข (conditional move), โหมดการอ้างอิงที่อยู่ที่ซับซ้อน, ต่วนับประสิทธิภาพที่ไม่สม่ำเสมอ, และคำสั่งที่มีความยาวคงที่เพียง 32 บิต นอกจากนี้ ARM-64 ไม่สามารถสลับไปใช้ ISA แบบ Thumb-2 ได้ เนื่องจาก Thumb-2 รองรับเฉพาะที่อยู่แบบ 32 บิตเท่านั้น

แตกต่างจาก RISC-V แล้ว ARM เลือกใช้แนวทางการออกแบบ ISA แบบสูงสุด (maximalist) หมายถึงรวมความสามารถไว้ให้มากที่สุดในตัวสถาปัตยกรรมเอง แม้ว่า ARM-64 จะเป็น ISA ที่ดีกว่า ARM-32 อย่างชัดเจน แต่ก็มีความซับซ้อนกว่า ตัวอย่างเช่น มีคำสั่งมากกว่า 1,000 คำสั่ง และคู่มือ ARM-64 มีความยาวถึง 3,185 หน้า [ARM 2015] นอกจากนี้ ISA นี้ยังคงขยายตัวอย่างต่อเนื่อง โดยมีการเพิ่มความสามารถใหม่ๆ เข้ามาหลายครั้งนับตั้งแต่เปิดตัวในปี 2011

โค้ดของ ARM-64 สำหรับการเรียงลำดับแบบแทรก ในรูป 10.11 มีลักษณะคล้ายกับโค้ดของ RV64I หรือ x86-64 มากกว่าโค้ดของ ARM-32 ตัวอย่างเช่น การมีรีจิสเตอร์ 31 ตัว ไม่จำเป็นต้องจัดเก็บและเรียกคืนรีจิสเตอร์จากสแต็ก และเนื่องจาก PC ไม่ใช่รีจิสเตอร์อีกต่อไป ARM-64 จึงใช้คำสั่งค้นหาแตกต่างหาก

รูปที่ 10.8 เป็นตารางที่สรุปจำนวนคำสั่งและจำนวนไบต์ในโค้ดการเรียงลำดับการแทรก สำหรับ ISA ทั้งสี่แบบ รูปที่ 10.10 ถึง 10.13 แสดงโค้ดที่คอมไพล์สำหรับ RV64I, ARM-64, MIPS-64 และ x86-64 นอกจากนี้ ข้อความในวงเล็บภายในคอมเมนต์ของโปรแกรมทั้งสี่นี้ ยังใช้เพื่อชี้ให้เห็นความแตกต่าง ระหว่างคำสั่ง RV32I ในบท 2 และคำสั่ง RV64I เหล่านี้

Intel ไม่ได้เป็นผู้คิดค้น ISA แบบ x86-64 เมื่อเปลี่ยนไปใช้ที่อยู่ 64 บิต Intel ได้ออกแบบ ISA ใหม่ชื่อว่า Itanium ซึ่งไม่เข้ากันกับ x86-32 คู่แข่งของ Intel ในตลาด x86-32 ไม่สามารถใช้ Itanium ได้ ดังนั้น บริษัท AMD จึงพัฒนา ISA 64 บิตของ x86-32 ขึ้นมาเอง เรียกว่า AMD64 ต่อมา Itanium ไม่ประสบความสำเร็จ ทำให้ Intel จำเป็นต้องยอมรับสถาปัตยกรรม AMD64 มาใช้เป็นรุ่นต่อ ยอดแบบ 64 บิตของ x86-32 ซึ่งปัจจุบันเรียกว่า x86-64 [Kerner and Padgett 2007]



รูปที่ 10.9: ขนาดโปรแกรมสัมพันธ์ของ RV64G, ARM-64 และ x86-64 เมื่อเทียบกับ RV64GC การเปรียบเทียบนี้ใช้โปรแกรมขนาดใหญ่กว่ามาก เมื่อเทียบกับในรูป 10.8 และ กราฟนี้เป็นรูปแบบที่อยู่ 64 บิตที่เทียบเท่ากับกราฟของ ISA แบบ 32 บิต ในรูป 1.5 ใน หน้า 11 บทที่ 1 ขนาดโค้ดของ RV32C ใกล้เคียงกับ RV64C มาก โดยเล็กกว่าประมาณ 1% เนื่องจาก ARM Thumb-2 ไม่มีเวอร์ชันสำหรับแอดเดรส 64 บิต ดังนั้น ISA แบบ 64 บิตอื่น ๆ จึงมีขนาดโค้ดโดยรวมใหญ่กว่า RV64GC อย่างมีนัยสำคัญ โปรแกรมที่ใช้ในการ วัดคือ ชุดการทดสอบ SPEC CPU2006 โดยใช้คอมไพเลอร์ GCC [Waterman 2016]

ในบรรดา ISA ทั้งหมด MIPS-64 ต้องใช้จำนวนคำสั่งมากที่สุด สาเหตุหลักมาจากคำสั่ง nop ที่ต้องใส่ในช่องหน้าต่างการกระโดด (delayed branch slots) ที่ไม่ได้ถูกใช้งาน ส่วน RV64I ใช้จำนวนคำสั่งน้อยกว่า เนื่องจากมีคำสั่งเปรียบเทียบและกระโดด (compare-and-branch) รวมอยู่ในคำสั่งเดียว และไม่มีการหน้าต่างการกระโดด แม้ว่า ARM-64 และ x86-64 จะต้องใช้คำสั่งเปรียบเทียบถึงสองคำสั่ง (ซึ่ง RV64I ไม่จำเป็นต้องใช้) แต่ทั้งสองมี โหมดการอ้างอิงที่อยู่แบบปรับสเกล (scaling addressing modes) ช่วยลดจำนวนคำสั่งที่ คำนวณที่อยู่ RV64I ต้องใช้ ทำให้โดยรวมแล้วมีคำสั่งน้อยที่สุด อย่างไรก็ตาม RV64C ที่ ใช้ร่วมกับ RV64I (คือ RV64I+RV64C) จะมีขนาดโค้ดเล็กกว่ามาก ดังที่จะอธิบายในหัวข้อ ถัดไป

■ รายละเอียดเพิ่มเติม: ชื่อ ARM-64, MIPS-64 และ x86-64 ไม่ใช่ชื่อเป็นทางการ

โดยชื่ออย่างเป็นทางการของ ARM-64 คือ ARMv9 [ARM 2021] ส่วนของ MIPS-64 คือ MIPS-IV และ x86-64 คือ AMD64 (ดูแถบด้านข้างหน้าก่อนหน้าสำหรับประวัติของ x86-64)

10.3 ขนาดโปรแกรม

รูปที่ 10.9 แสดงการเปรียบเทียบขนาดโค้ดเฉลี่ยสัมพันธ์ของ RV64, ARM-64 และ x86-64



และเมื่อเปรียบเทียบกับรูปที่ 1.5 ในหน้าที่ 11 ในบท 1 จะพบว่า ขนาดโค้ดของ RV32GC มีขนาดเกือบเท่ากับ RV64GC โดยเล็กกว่าประมาณ 1% เท่านั้น และความใกล้เคียงนี้เกิดขึ้นกับ RV32I และ RV64I เช่นกัน แม้ว่าโค้ด ARM-64 จะเล็กกว่าโค้ด ARM-32 อยู่ประมาณ 8% แต่เนื่องจากไม่มีรุ่นที่อยู่ 64 บิตของ Thumb-2 คำสั่งทั้งหมดจึงยังคงมีความยาว 32 บิต ส่งผลให้โค้ดของ ARM-64 มีขนาดใหญ่กว่า ARM Thumb-2 ถึงประมาณ 25% โค้ด x86-64 ใหญ่กว่าโค้ด x86-32 อยู่ประมาณ 7% เนื่องจากการเพิ่มรหัสดำเนินการ (prefix opcodes) ในคำสั่ง x86-64 เพื่อรองรับการดำเนินการใหม่และชุดเรจิสเตอร์ที่เพิ่มขึ้น ในขณะที่ RV64GC มีขนาดเล็กที่สุด โดยโค้ด ARM-64 มีขนาดใหญ่กว่า RV64GC ถึง 23% และโค้ดของ x86-64 ใหญ่กว่า RV64GC ประมาณ 34% ความแตกต่างนี้มากพอที่จะเพิ่มประสิทธิภาพจากอัตราการพลาดแคชของคำสั่งที่ลดลง หรือลดต้นทุนโดยใช้แคชคำสั่งที่เล็กลง แต่ยังคงมีอัตราการพลาดแคชที่ยอมรับได้

10.4 ข้อสรุปท้ายบท

ปัญหาหนึ่งของการเป็นผู้บุกเบิกคือ คุณมักจะทำผิดพลาด และฉันไม่อยากเป็นผู้บุกเบิกเลย เป็นการดีกว่าที่จะเป็นอันดับที่สอง เพราะเราสามารถเรียนรู้จากความผิดพลาดของผู้บุกเบิกได้

—Seymour Cray สถาปนิกของซูเปอร์คอมพิวเตอร์เครื่องแรก, 1976

การมีจำนวน บิตของ แอดเดรสไม่เพียง พอถือเป็น จุดอ่อนร้ายแรง ของสถาปัตยกรรมคอมพิวเตอร์ หลายสถาปัตยกรรมต้องล้มเหลวจากข้อจำกัดนี้ ARM-32 และ Thumb-2 ยังคงเป็นสถาปัตยกรรม 32 บิต จึงไม่เหมาะสมสำหรับโปรแกรมขนาดใหญ่ ISA บางตัว เช่น MIPS-64 และ x86-64 สามารถผ่านการเปลี่ยนผ่านไปสู่ 64 บิตได้สำเร็จ แต่ x86-64 ก็ไม่ได้ถือเป็นตัวอย่างที่ดีเลิศของการออกแบบ ISA และอนาคตของ MIPS-64 ก็ยังไม่แน่นอน ในเวลานี้ ARM-64 เป็น ISA ขนาดใหญ่ที่ค่อนข้างตัวใหม่ที่ใช้กันอย่างแพร่หลายในตลาดสมาร์ทโฟน และเริ่มประสบความสำเร็จในระบบคลาวด์ เมื่อมองไปข้างหน้า ISA แบบ 64 บิตที่ได้รับความนิยมมากที่สุดน่าจะเป็น ARM-64, RV64 และ x86-64

RISC-V ได้รับประโยชน์จากการออกแบบสถาปัตยกรรมทั้งแบบ 32 บิตและ 64 บิตไปพร้อมกัน ในขณะที่ ISA รุ่นเก่ามักต้องออกแบบตามลำดับขั้น (ทำ 32 บิตก่อน แล้วค่อยขยายเป็น 64 บิตในภายหลัง) จึงไม่น่าแปลกใจที่การเปลี่ยนผ่านระหว่าง 32 บิตไปเป็น 64 บิตนั้นง่ายที่สุดสำหรับโปรแกรมเมอร์และพัฒนาคอมไพเลอร์ของ RISC-V โดย RV64I มีคำสั่งเกือบทั้งหมดเหมือนกับ RV32I นั่นคือเหตุผลที่เราสามารถสรุปทั้ง RV32GCV และ RV64GCV ได้ในเพียงสองหน้าของการ์ดอ้างอิง (Reference Card)

ที่สำคัญยิ่งกว่านั้น การออกแบบพร้อมกันทำให้สถาปัตยกรรม 64 บิตไม่จำเป็นต้องถูกบีบอัดลงในพื้นที่รหัสคำสั่ง (opcode space) แบบ 32 บิตที่มีข้อจำกัด จึงมีพื้นที่เพียงพอสำหรับส่วนขยายเพิ่มเติม โดยเฉพาะ RV64C ซึ่งช่วยให้ RISC-V เป็นผู้นำด้านขนาดโค้ด เรามองว่าสถาปัตยกรรม 64 บิตเป็นหลักฐานเพิ่มเติมของการออกแบบที่ดีของ RISC-V แม้ว่า จะยอมรับได้ว่าการเริ่มต้นช้ากว่าประมาณ 20 ปี ทำให้สามารถนำแนวคิดที่ดีของผู้บุกเบิกมาใช้ และหลีกเลี่ยงข้อผิดพลาดของพวกเขาได้



ค่าใช้จ่าย

MIPS มีเจ้าของมาแล้วถึงห้าราย โดย Tallwood Venture Capital ได้เข้าซื้อ MIPS ในปี 2017 ด้วยมูลค่า 65 ล้านดอลลาร์ และกลายเป็นเจ้าของรายที่สาม ก่อนจะโอนให้บริษัท Wave Computing ในปี 2018 เมื่อ Wave Computing ยื่นล้มละลาย ในปี 2020 Tallwood Venture Capital ได้ซื้อ MIPS คืนในราคา 61 ล้านดอลลาร์



การเขียนโปรแกรม



ที่ว่างสำหรับส่วนขยาย



ขนาดโปรแกรม



สง่างาม

■ รายละเอียดเพิ่มเติม: RV64E ส่วนขยายมาตรฐานสำหรับระบบฝังตัว

เช่นเดียวกับที่ RV32E ช่วยลดต้นทุนของคอร์แบบ 32 บิตระดับล่าง RV64E เป็นชุดคำสั่งพื้นฐานที่ลดจำนวนรีจิสเตอร์ลง 16 ตัวสำหรับคอร์ 64 บิตระดับล่าง

■ รายละเอียดเพิ่มเติม: RV128

RV128 เริ่มต้นจากมุกภายในของทีมสถาปนิก RISC-V เพื่อแสดงให้เห็นว่าสามารถสร้าง ISA แบบที่อยู่ 128 บิตได้ อย่างไรก็ตาม ในอนาคต ระบบคอมพิวเตอร์ขนาดระดับดาต้าเซ็นเตอร์ (warehouse-scale computers) อาจมีหน่วยความจำมากกว่า 2^{64} (เช่น DRAM และ Flash) ซึ่งโปรแกรมเมอร์อาจต้องการเข้าถึงผ่าน address โดยตรง นอกจากนี้ยังมีแนวความคิดใช้ที่อยู่ 128 บิตเพื่อเพิ่มความปลอดภัยของระบบอีกด้วย [Woodruff et al. 2014] เอกสารของ RISC-V ได้กำหนด ISA แบบ 128 บิตเต็มรูปแบบไว้แล้วคือ RV128G [Waterman and Asanović 2017] คำสั่งเพิ่มเติมใน RV128 โดยพื้นฐานแล้วเหมือนกับการขยายจาก RV32 เป็น RV64 ซึ่ง แสดงในรูปแบบที่ 10.1 ถึง 10.4 แสดงให้เห็นว่า รีจิสเตอร์ทั้งหมดขยายเป็น 128 บิต และมีคำสั่ง RV128 ใหม่แบบ 128 บิต (ใช้ตัวอักษร Q สำหรับ quadword) และคำสั่งเวิร์ชัน 64 บิต (ใช้ตัวอักษร D สำหรับ doubleword)

10.5 เรียนรู้เพิ่มเติม

I. ARM. ARMv8-A architecture reference manual. 2015.

I. ARM. Arm architecture reference manual supplement armv9, for armv9-a architecture profile. 2021.

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 คำสั่ง, 76 bytes, or 52 bytes กับ RV64C)
# a1 คือ n, a3 ชี้ไปที่ a[0], a4 คือ i, a5 คือ j, a6 คือ x
0: 00850693 addi a3,a0,8 # (8 vs 4) a3 เป็นตัวชี้ไปที่ a[i]
4: 00100713 li a4,1 # i = 1
Outer Loop:
8: 00b76463 bltu a4,a1,10 # ถ้า i < n กระโดดไปที่ Continue Outer loop
Exit Outer Loop:
c: 00008067 ret # ออกจากฟังก์ชัน
Continue Outer Loop:
10: 0006b803 ld a6,0(a3) # (ld vs lw) x = a[i]
14: 00068613 mv a2,a3 # a2 เป็นตัวชี้ไปที่ 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 # ถ้า a[j-1] <= a[i] กระโดดไปที่ 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) ลด a2 ชี้ไปที่ a[j]
30: fe0796e3 bnez a5,1c # ถ้า j != 0 กระโดดไปที่ Inner Loop
Exit Inner Loop:
34: 00379793 slli a5,a5,0x3 # (8 vs 4) คูณ a5 by 8
38: 00f507b3 add a5,a0,a5 # a5 is now byte address of 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 # เพิ่ม a3 ชี้ไปที่ a[i]
48: fc1ff06f j 8 # กระโดดไปที่ Outer Loop
```

รูปที่ 10.10: โค้ด RV64I สำหรับการเรียงลำดับการแทรก ในรูป 2.5 โปรแกรมแอสเซมบลี RV64I มีความคล้ายกับโปรแกรมแอสเซมบลีของ RV32I ในรูป 2.8 หน้าที่ 34 ในบท 2 อย่างมาก โดยเราระบุความแตกต่างไว้ในวงเล็บภายในคอมเมนต์ ขนาดข้อมูลเปลี่ยนจาก 4 ไบต์เป็น 8 ไบต์ ทำให้มีคำสั่ง 3 คำสั่ง ที่ต้องเปลี่ยนค่าคงที่จาก 4 เป็น 8 นอกจากนี้ ความกว้างของข้อมูลที่เพิ่มขึ้นนี้ยังทำให้คำสั่งโหลดแบบเวิร์ด (lw) เป็นโหลดแบบดับเบิลเวิร์ด (ld) และคำสั่งจัดเก็บแบบเวิร์ด (sw) เป็นจัดเก็บแบบดับเบิลเวิร์ด(sd)

```

# ARM-64 (16 คำสั่ง, 64 bytes)
# x0 ชี้ไปที่ a[0], x1 คือ n, x2 คือ j, x3 คือ i, x4 คือ x
0: d2800023  mov  x3, #0x1          # i = 1
Outer Loop:
4: eb01007f  cmp  x3, x1          # เปรียบเทียบ i vs n
8: 54000043  b.cc 10             # ถ้า i < n กระโดดไปที่ Continue Outer loop
Exit Outer Loop:
c: d65f03c0  ret                # ออกจากฟังก์ชัน
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 ตัวชี้ไปที่ a[j]
1c: f85f80a5  ldur x5, [x5, #-8]    # x5 = a[j]
20: eb0400bf  cmp  x5, x4          # เปรียบเทียบ a[j-1] vs. x
24: 5400008d  b.le 34             # ถ้า a[j-1] <= a[i] กระโดดไปที่ 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             # ถ้า j != 0 กระโดดไปที่ 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             # กระโดดไปที่ Outer Loop

```

รูปที่ 10.11: โค้ด ARM-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 โปรแกรมแอสเซมบลี ARM-64 แตกต่างจากภาษาแอสเซมบลี ARM-32 ในรูปที่ 2.5

ในหน้าที่ 37 บทที่ 2 เนื่องจากเป็นชุดคำสั่งใหม่ทั้งหมด รีจิสเตอร์ใช้ชื่อเริ่มต้นด้วย x แทน a โหมดการอ้างอิงที่อยู่ข้อมูล (data addressing modes) สามารถเลื่อนรีจิสเตอร์ได้ 3 บิต เพื่อปรับสเกลดัชนีให้เป็นที่อยู่แบบไบต์ (byte address) มีรีจิสเตอร์ 31 ตัว จึงไม่จำเป็นต้องเก็บและกู้คืนรีจิสเตอร์จากสแต็ก (stack) เนื่องจากตัวนับโปรแกรม (PC) ไม่ใช่หนึ่งในรีจิสเตอร์ จึงใช้คำสั่ง return แยกต่างหาก โดยรวมแล้ว โค้ดของ ARM-64 มีลักษณะใกล้เคียงกับ RV64I หรือ x86-64 มากกว่า ARM-32

```
# MIPS-64 (24 คำสั่ง, 96 bytes)
# a1 คือ n, a3 เป็นตัวชี้ไปที่ a[0], v0 คือ j, v1 คือ i, t0 คือ x
0: 64860008 daddiu a2,a0,8 # (daddiu vs addiu, 8 vs 4) a2 เป็นตัวชี้ไปที่ a[i]
4: 24030001 li v1,1 # i = 1
Outer Loop:
8: 0065102b sltu v0,v1,a1 # set on i < n
c: 14400003 bnez v0,1c # ถ้า i < n กระโดดไปที่ Continue Outer Loop
10: 00c03825 move a3,a2 # a3 เป็นตัวชี้ไปที่ a[j] (slot filled)
14: 03e00008 jr ra # ออกจากฟังก์ชัน
18: 00000000 nop # branch delay slot unfilled
Continue Outer Loop:
1c: dcc80000 ld a4,0(a2) # (ld vs lw) x = a[i]
20: 00601025 move v0,v1 # j = i
Inner Loop:
24: dce9fff8 ld a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
28: 0109502a slt a6,a4,a5 # (no load delay slot) set a[i] < a[j-1]
2c: 11400005 beqz a6,44 # ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit Inner Loop
30: 00000000 nop # branch delay slot unfilled
34: 6442ffff daddiu v0,v0,-1 # (daddiu vs addiu) j--
38: fce90000 sd a5,0(a3) # (sd vs sw, a5 vs t1) a[j] = a[j-1]
3c: 1440ffff bnez v0,24 # ถ้า j != 0 กระโดดไปที่ Inner Loop (next slot filled)
40: 64e7fff8 daddiu a3,a3,-8 # (daddiu vs addiu, 8 vs 4) ลด a3 ตัวชี้ไปที่ a[j]
Exit Inner Loop:
44: 000210f8 dsll v0,v0,0x3 # (dsll vs sll)
48: 0082102d daddu v0,a0,v0 # (daddu vs addu) v0 now byte address of 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 # กระโดดไปที่ Outer Loop (next delay slot filled)
58: 64c60008 daddiu a2,a2,8 # (daddiu vs addiu, 8 vs 4) เพิ่ม a2 ตัวชี้ไปที่ a[i]
5c: 00000000 nop # Unnecessary(?)
```

รูปที่ 10.12: โค้ด MIPS-64 สำหรับการเรียงลำดับการแทรกในรูปที่ 2.5 โปรแกรมแอสเซมบลีของ MIPS-64 มีความแตกต่างจาก MIPS-32 หลายประการ ในรูปที่ 2.10 หน้าที่ 36 บทที่ 2 ประการแรก คำสั่งสำหรับข้อมูล 64 บิตส่วนใหญ่จะเติมตัวอักษร “d” นำหน้าชื่อ เช่น daddiu, daddu, dsll เช่นเดียวกับ รูป 10.10 มีคำสั่ง 3 คำสั่งที่เปลี่ยนค่าคงที่จาก 4 เป็น 8 เนื่องจากขนาดของข้อมูลเพิ่มจาก 4 เป็น 8 ไบต์ เช่นเดียวกับ RV64I ความกว้างที่เพิ่มขึ้นเปลี่ยนคำสั่งโหลดแบบเวิร์ด (lw) เป็นโหลดแบบดับเบิลเวิร์ด (ld) และคำสั่งจัดเก็บแบบเวิร์ด (sw) เป็นจัดเก็บแบบเบิ้ลเวิร์ด (sd). สุดท้าย MIPS-64 ไม่มีช่องหน่วงการโหลด เหมือนใน MIPS-32 โดย pipeline จะหยุด (stall) เมื่อเกิดการพึ่งพาข้อมูลแบบ read-after-write

```

# x86-64 (15 คำสั่ง, 46 bytes)
# rax คือ j, rcx คือ x, rdx คือ i, rsi คือ n, rdi เป็นตัวชี้ไปที่ a[0]
0: ba 01 00 00 00 mov edx,0x1
Outer Loop:
5: 48 39 f2          cmp rdx,rsi          # เปรียบเทียบ i vs. n
8: 73 23            jae 2d <Exit Loop>  # ถ้า i >= n กระโดดไปที่ 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          # เปรียบเทียบ a[j-1] vs. x
19: 7e 09            jle 24 <Exit Loop>  # ถ้า a[j-1] <= a[i] กระโดดไปที่ 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>  # ถ้า j != 0 กระโดดไปที่ 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>   # กระโดดไปที่ Outer Loop
Exit Outer Loop:
2d: c3              ret                  # ออกจากฟังก์ชัน

```

รูปที่ 10.13: โค้ด x86-64 สำหรับการเรียงลำดับการแทรก ในรูปที่ 2.5 โปรแกรมแอสเซมบลี x86-64 แตกต่างจากโปรแกรมแอสเซมบลี x86-32 อย่างมาก ในรูปที่ 2.11 หน้าที 37 บทที่ 2 ประการแรก มีรีจิสเตอร์ที่มีความกว้างมากขึ้นมีชื่อใหม่ เช่น rax, rcx, rdx, rsi, rdi, r8 ประการที่สอง เนื่องจาก x86-64 มีรีจิสเตอร์เพิ่มอีก 8 ตัว ทำให้สามารถเก็บตัวแปรทั้งหมดไว้ในรีจิสเตอร์แทนที่จะเก็บในหน่วยความจำ ประการที่สาม คำสั่ง x86-64 มีความยาวมากขึ้น เนื่องจากต้องเติมบิตข้างหน้าคำสั่ง ขนาด 8 บิต หรือ 16 บิตเพื่อรองรับคำสั่งใหม่ลงในพื้นที่รหัสดำเนินการ (opcode space) ตัวอย่างเช่น คำสั่งเพิ่มหรือลดรีจิสเตอร์ (inc, dec) ใช้ 1 ไบต์ใน x86-32 แต่ใช้ 3 ไบต์ใน x86-64 ดังนั้น แม้ว่าจะใช้คำสั่งจะลดลง แต่ขนาดโค้ดของการเรียงลำดับการแทรก ใน x86-64 กลับแทบไม่ต่างจาก x86-32 (45 ไบต์ เทียบกับ 46 ไบต์)

RV32/64 สถาปัตยกรรมสิทธิพิเศษ

เอ็ดเกอร์ ไวปี ไจค์สตรา
(ค.ศ. 1930–2020) ผู้ได้รับ
รางวัล Turing Award ปี
1972 สำหรับการมีส่วนร่วม
พื้นฐานในการพัฒนาภาษา
โปรแกรม



ความเรียบง่ายเป็นสิ่งจำเป็นสำหรับความน่าเชื่อถือ

—เอ็ดเกอร์ ไวปี ไจค์สตรา

11.1 บทนำ

เนื้อหาของหนังสือนี้มุ่งเน้นการสนับสนุนของ RISC-V สำหรับการประมวลผลทั่วไป โดยคำสั่งทั้งหมดที่เราได้แนะนำนั้นสามารถใช้งานได้โมดผู้ใช้ (user mode) ซึ่งเป็นโมดที่โปรแกรมแอปพลิเคชันทำงานอยู่โดยปกติ สำหรับบทนี้จะแนะนำ โมดสิทธิพิเศษ (privilege modes) ใหม่อีกสองโมด ได้แก่ โมดเครื่อง (machine mode) ซึ่งใช้รันโค้ดที่เชื่อถือได้สูงสุด และ โมดผู้ดูแลระบบ (supervisor mode) ซึ่งรองรับสำหรับระบบปฏิบัติการ เช่น Linux, FreeBSD และ Windows ทั้งสองโมดใหม่นี้มีสิทธิ์สูงกว่าโมดผู้ใช้ จึงเป็นที่มาของชื่อบทนี้

โดยทั่วไปแล้ว โมดที่มีสิทธิ์สูงกว่าจะสามารถเข้าถึงความสามารถทั้งหมดของโมดที่มีสิทธิ์ต่ำกว่าได้ และยังสามารถเพิ่มความสามารถพิเศษที่โมดที่มีสิทธิ์ต่ำกว่าไม่สามารถทำได้ เช่น ความสามารถในการจัดการอินเทอร์รัปต์ และการดำเนินการ I/O โดยทั่วไป โปรเซสเซอร์จะใช้เวลาส่วนใหญ่ในการทำงานอยู่ในโมดที่มีสิทธิ์ต่ำที่สุด และเมื่อเกิดอินเทอร์รัปต์หรือข้อบกพร่อง การควบคุมจะถูกโอนไปยังโมดที่มีสิทธิ์สูงกว่า

ระบบรันไทม์ของระบบฝังตัวและระบบปฏิบัติการจะใช้คุณสมบัติของโมดใหม่เหล่านี้ เพื่อตอบสนองต่อเหตุการณ์ภายนอก เช่น การมาถึงของแฟกต์เกิดเครือข่าย, เพื่อรองรับการทำงานหลายงานพร้อมกัน (multitasking) และการป้องกันระหว่างงาน และเพื่อ

RV32/64 Privileged Instructions

$\left\{ \begin{array}{l} \text{machine-mode} \\ \text{supervisor-mode} \end{array} \right\}$ trap return

supervisor-mode fence virtual memory address
wait for interrupt

รูปที่ 11.1: ไดอะแกรมของคำสั่งสิทธิพิเศษของ RISC-V

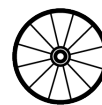
31	27	26	25	24	20	19	15	14	12	11	7	6	0	
0001000	00010					00000	000			00000		1110011		R sret
0011000	00010					00000	000			00000		1110011		R mret
0001000	00101					00000	000			00000		1110011		R wfi
0001001	rs2					rs1	000			00000		1110011		R sfence.vma

รูปที่ 11.2: รูปแบบคำสั่งสิทธิพิเศษของ RISC-V, รหัสดำเนินการ, ประเภทรูปแบบคำสั่ง และชื่อคำสั่ง (ตารางที่ 6.1 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

นามธรรมและจำลองคุณสมบัติของฮาร์ดแวร์ ด้วยความกว้างของหัวข้อเหล่านี้ คู่มือสำหรับโปรแกรมเมอร์แบบละเอียดจะต้องเป็นหนังสืออีกเล่มหนึ่งโดยเฉพาะ ดังนั้นบทนี้จึงมุ่งเน้นเฉพาะประเด็นสำคัญของคุณสมบัติใน RISC-V เท่านั้น โปรแกรมเมอร์ที่ไม่สนใจระบบฝังตัวหรือระบบปฏิบัติการ สามารถข้ามหรืออ่านแบบคร่าวๆ ในบทนี้ได้

รูปที่ 11.1 เป็นการนำเสนอแบบกราฟิกของคำสั่งสิทธิพิเศษของ RISC-V และ รูปที่ 11.2 ระบุรหัสดำเนินการของคำสั่งเหล่านี้ จะเห็นได้ว่า สถาปัตยกรรมสิทธิพิเศษเพิ่มคำสั่งเพียงไม่กี่คำสั่ง แต่เพิ่มรีจิสเตอร์ควบคุมและสถานะ (CSRs) ใหม่หลายตัวเพื่อเปิดฟังก์ชันการทำงานเพิ่มเติม

บทนี้จะอธิบายเกี่ยวกับสถาปัตยกรรมสิทธิพิเศษของ RV32 และ RV64 พร้อมกัน แนวคิดบางอย่างแตกต่างกันเพียงขนาดของรีจิสเตอร์จำนวนเต็ม ดังนั้นเพื่อให้คำอธิบายกระชับ เราใช้คำว่า XLEN แทนขนาดของรีจิสเตอร์จำนวนเต็มในหน่วยบิต โดย XLEN มีค่าเป็น 32 สำหรับ RV32 หรือ 64 สำหรับ RV64



ความเรียบง่าย

11.2 โหมดเครื่องสำหรับระบบฝังตัวอย่างง่าย

โหมดเครื่อง หรือย่อว่า M-mode เป็นโหมดที่มีสิทธิสูงสุดที่ hart (hardware thread) ของ RISC-V สามารถทำงานได้ Harts ที่ทำงานใน M-mode จะสามารถเข้าถึงหน่วยความจำทั้งหมด, อินพุต/เอาต์พุต (I/O), และฟีเจอร์ระดับต่ำของระบบ (low-level system features) ซึ่งเป็นสิ่งจำเป็นสำหรับการเริ่มต้นระบบ (boot) และการกำหนดค่าระบบ (configure) ดังนั้น M-mode จึงเป็นโหมดสิทธิพิเศษโหมดเดียวที่โปรเซสเซอร์ RISC-V มาตรฐานทุกตัวต้องรองรับ และในความเป็นจริง ไม่ใคร่คอนโทรลเลอร์ RISC-V แบบง่ายจำนวนมากรองรับ เฉพาะ M-mode เท่านั้น ระบบลักษณะนี้คือหัวข้อหลักของส่วนนี้

คุณสมบัติที่สำคัญที่สุดของโหมดเครื่อง คือความสามารถในการดักจับ (intercept) และจัดการข้อยกเว้น (exceptions): ซึ่งเป็นเหตุการณ์ผิดปกติที่เกิดขึ้นระหว่างการทำงาน (runtime) RISC-V แบ่งข้อยกเว้นออกเป็น 2 ประเภท: *ข้อยกเว้นแบบซิงโครนัส* (synchronous exceptions) เกิดขึ้นจากการทำงานของคำสั่ง เช่น การเข้าถึงหน่วยความจำที่ไม่ถูกต้อง หรือการทำงานของคำสั่งที่มีรหัสดำเนินการไม่ถูกต้อง *อินเทอร์รัปต์* (Interrupts) เป็นเหตุการณ์ภายนอกที่ไม่สัมพันธ์กับลำดับคำสั่ง เช่น การคลิกเมาส์ ข้อยกเว้น (exceptions) ใน RISC-V เป็นแบบ *precise* กล่าวคือ คำสั่งทั้งหมดก่อนหน้าที่เกิดข้อยกเว้นจะถูกดำเนินการเสร็จสมบูรณ์แล้ว และคำสั่งที่อยู่ถัดไปจะยังไม่เริ่มทำงาน (ไม่

คำว่า *Hart* มาจากคำย่อของคำว่า *hardware thread* เราใช้คำนี้เพื่อแยกความแตกต่างจากซอฟต์แวร์เธรด (software threads) ซึ่งโปรแกรมเมอร์ส่วนใหญ่คุ้นเคย ซอฟต์แวร์เธรดจะถูกนำมาสลับใช้งานบน hart ตามเวลา (time-multiplexed) และโดยทั่วไปแล้ว หน่วยประมวลผลส่วนใหญ่มีเพียงหนึ่ง hart



แยกสถาปัตยกรรม กับนำไปใช้

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

รูปที่ 11.3: สาเหตุของข้อบกพร่องและอินเทอร์รัปต์ใน RISC-V บิตที่มีนัยสำคัญที่สุดของรีจิสเตอร์ mcause จะถูกตั้งค่าเป็น 1 สำหรับอินเทอร์รัปต์ หรือ 0 สำหรับข้อบกพร่องเชิงโครงสร้าง และบิตที่เหลือที่มีนัยสำคัญน้อยที่สุดจะใช้ระบุชนิดของอินเทอร์รัปต์หรือข้อบกพร่องนั้น และอินเทอร์รัปต์ของโหมดผู้ดูแลระบบ และข้อบกพร่องการเข้าถึงเพจผิดพลาด (page-fault exceptions) จะเกิดขึ้นได้เฉพาะเมื่อมีการใช้งานโหมดผู้ดูแลระบบ (ดูหัวข้อ 11.5) (ตารางที่ 3.6 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้)

คำสั่งที่ตามมา) รูปที่ 11.3 แสดงรายการสาเหตุของข้อบกพร่องมาตรฐาน

มีข้อบกพร่องแบบซิงโครนัส (synchronous exceptions) อยู่ 5 ประเภท ที่สามารถเกิดขึ้นได้ระหว่างการดำเนินงานในโหมดเครื่อง (M-mode):

- ข้อบกพร่องการเข้าถึงผิดพลาด (Access fault exceptions) เกิดขึ้นเมื่อที่อยู่หน่วยความจำกายภาพ (physical memory address) ไม่รองรับประเภทการเข้าถึงนั้น —เช่น การพยายามเขียนข้อมูลลงใน ROM
- ข้อบกพร่องจากจุดหยุด (Breakpoint Exceptions) เกิดจากการทำงานคำสั่ง ebreak หรือเมื่อที่อยู่หรือข้อมูลตรงกับเงื่อนไขของฮาร์ดแวร์ตรวจจับอัตโนมัติ
- ข้อบกพร่องจากการเรียกสภาพแวดล้อม (Environment Call Exceptions) เกิดจากการทำงานคำสั่ง ecall (ใช้สำหรับเรียกระบบ เช่น system call)
- ข้อบกพร่องคำสั่งที่ไม่ถูกต้อง (Illegal Instruction Exceptions) เกิดจากการถอดรหัสคำ (decode) สิ่งที่มีรหัสดำเนินการที่ไม่ถูกต้อง
- ข้อบกพร่องที่อยู่ที่ไม่จัดแนว เกิดขึ้นเมื่อที่อยู่ที่มีค่านวนได้ไม่สามารถหารลงตัวด้วยขนาดการเข้าถึง (Access size)—เช่นคำสั่ง amoadd.w ที่ใช้ตำแหน่งที่อยู่เป็น 0x12

ข้อบกพร่องจากคำสั่งที่อยู่ไม่จัดแนว (misaligned instruction address) ไม่สามารถเกิดขึ้นได้เมื่อมีการใช้ส่วนขยาย C เนื่องจากไม่สามารถกระโดดไปยังที่อยู่เลขคู่ได้ โดย: คำสั่งของคำสั่งการกระโดดแบบมีเงื่อนไขและ JAL จะเป็นเลขคู่เสมอ และคำสั่ง JALR จะลบ (mask) บิตที่มีนัยสำคัญน้อยที่สุดของที่อยู่ที่มีค่านวนได้ (effective address) ออก หากไม่มีส่วนขยาย C ข้อบกพร่องนี้เกิดขึ้นเมื่อกระโดดไปที่ที่อยู่ที่มีค่าเท่ากับ 2 mod 4 (คือไม่หารด้วย 4 ลงตัว)

หากคุณจำได้ในบทที่ 2 ที่กล่าวว่า การโหลดและการจัดเก็บที่ไม่จัดแนว สามารถทำได้ คุณอาจสงสัยว่าทำไม ข้อบกพร่องของการเข้าถึงตำแหน่งที่อยู่แบบไม่จัดแนวสำหรับการโหลดและการจัดเก็บ จึงยังคงระบุไว้ในรูปที่ 11.3 มีเหตุผลอยู่สองประการ ประการแรก คำสั่งการดำเนินการหน่วยความจำจะตอมิก ในบทที่ 6 จำเป็นต้องใช้ ที่อยู่ที่จัดแนวตามขนาดข้อมูล (naturally aligned) เท่านั้น ประการที่สอง ผู้พัฒนาบางรายไม่รองรับการโหลดและการจัดเก็บที่ไม่จัดแนวในฮาร์ดแวร์ เนื่องจากเป็นคุณสมบัติที่ยากต่อการออกแบบและไม่ค่อยถูกใช้งานบ่อย โปรเซสเซอร์ที่ไม่มีการรองรับนี้ในฮาร์ดแวร์จะอาศัย ตัวจัดการข้อบกพร่อง (exception handler) แทน เพื่อดักจับ (trap) และจำลอง (emulate) การทำงานของการโหลดและบันทึกที่ไม่จัดแนวด้วยซอฟต์แวร์ โดยใช้ลำดับของคำสั่งการโหลดและบันทึกขนาดเล็กที่มีการจัดแนวถูกต้อง ในมุมมองของโปรแกรมแอปพลิเคชันจะไม่รับรู้ถึงความแตกต่างนี้เลย กล่าวคือ การเข้าถึงหน่วยความจำแบบไม่จัดแนวยังคงทำงานได้ตามปกติ เพียงแต่จะทำงานได้ช้าลง ในขณะที่ฮาร์ดแวร์ยังคงมีความเรียบง่าย หรืออีกทางเลือกหนึ่ง โปรเซสเซอร์ที่มีประสิทธิภาพสูงกว่า สามารถรองรับการโหลดและบันทึกที่ไม่จัดแนวในฮาร์ดแวร์ได้โดยตรง ความยืดหยุ่นในการออกแบบนี้ เกิดจากการตัดสินใจของ RISC-V ที่อนุญาตให้มีการโหลดและบันทึกที่ไม่จัดแนว โดยใช้คำสั่ง โหลดและจัดเก็บ ปกติ ซึ่งสอดคล้องกับแนวคิดในการ “แยกสถาปัตยกรรมออกจากการนำไปใช้งานจริง” ตามหลักการที่กล่าวไว้ในบท 1

มีแหล่งที่มาของอินเทอร์รัปต์ มาตรฐาน 3 ประเภท ได้แก่ อินเทอร์รัปต์ซอฟต์แวร์ (Software interrupts): เกิดจากการเขียนค่าไปยังรีจิสเตอร์ที่แมปกับหน่วยความจำ (memory-mapped register) โดยทั่วไปใช้เพื่อให้ hart หนึ่งสามารถอินเทอร์รัปต์การทำงานของอีก hart หนึ่ง ซึ่งในสถาปัตยกรรมอื่นเรียกว่า อินเทอร์รัปต์ระหว่างโปรเซสเซอร์



แยกสถาปัตยกรรม กับนำไปใช้

XLEN-1	XLEN-2				23	22	21	20	19	18	17	
SD	Reserved				TSR	TW	TVM	MXR	SUM	MPRV		
1	XLEN-24				1	1	1	1	1	1	1	
16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	Res.	SPP	MPIE	Res.	SPIE	Res.	MIE	Res.	SIE	Res.
2	2	2	2	1	1	1	1	1	1	1	1	1

รูปที่ 11.4: รีจิสเตอร์ CSR `mstatus` ในโปรเซสเซอร์แบบง่ายที่มีเฉพาะโหมดเครื่องและไม่มีส่วนขยาย F (Floating-Point) และ V (Vector) ฟิลด์ที่มีอยู่จะมีเฉพาะ MIE (global interrupt enable) และ MPIE (Machine Previous Interrupt Enable) โดยหลังจากเกิดข้อยกเว้น ค่าเดิมของ MIE จะเก็บไว้ใน MPIE ค่า XLEN จะเป็น 32 สำหรับ RV32 หรือ 64 สำหรับ RV64 รูปที่ 3.7 ในเอกสาร [Waterman and Asanović 2017] และสามารถดูคำอธิบายของฟิลด์อื่นๆ ได้ในหัวข้อ 3.1 ของเอกสาร

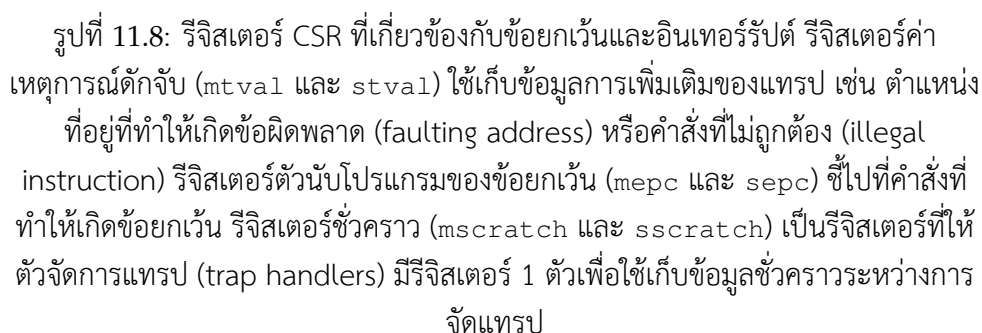
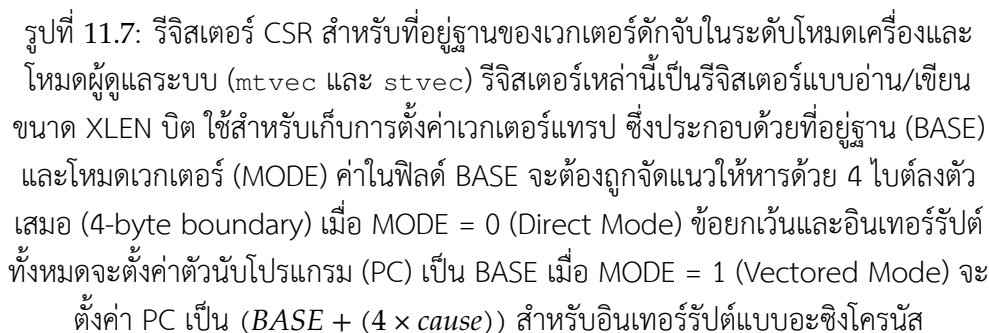
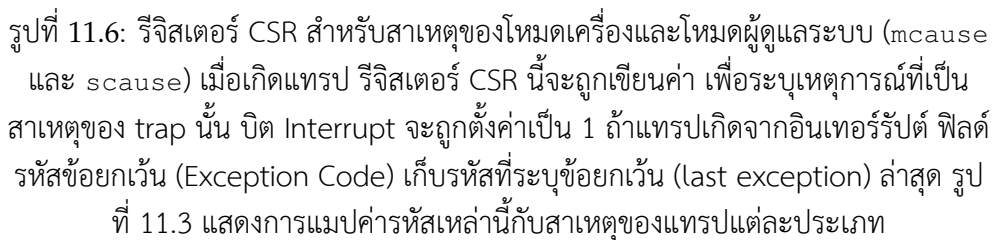
XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	MEIP	Res.	SEIP	Res.	MTIP	Res.	STIP	Res.	MSIP	Res.	SSIP	Res.	
Reserved	MEIE	Res.	SEIE	Res.	MTIE	Res.	STIE	Res.	MSIE	Res.	SSIE	Res.	
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	1

รูปที่ 11.5: รีจิสเตอร์ CSRs สำหรับอินเทอร์รัปต์ในระดับเครื่อง รีจิสเตอร์เหล่านี้เป็นรีจิสเตอร์แบบอ่าน/เขียน (read/write) ขนาด XLEN บิต ใช้สำหรับเก็บอินเทอร์รัปต์ที่รอดำเนินการ (pending interrupts) และบิตเปิดใช้งานอินเทอร์รัปต์ (pending interrupts) ในรีจิสเตอร์ `mie` จะมีเพียงบิตที่เกี่ยวข้องกับอินเทอร์รัปต์ของโหมดสิทธิต่ำกว่าเท่านั้นที่สามารถเขียนได้ผ่าน CSR นี้ ได้แก่ ซอฟต์แวร์อินเทอร์รัปต์ (SSIP) อินเทอร์รัปต์ตัวจับเวลา (STIP) และอินเทอร์รัปต์ภายนอก (SEIP) ส่วนบิตอื่น ๆ ที่เหลือใน `mip` จะเป็นแบบ อ่านอย่างเดียว (read-only)

(interprocessor interrupt) อินเทอร์รัปต์ตัวจับเวลา (Timer interrupts) เกิดขึ้นเมื่อตัวเปรียบเทียบเวลาของ hart ซึ่งเป็นรีจิสเตอร์ที่แมปกับหน่วยความจำ ชื่อ `mtimecmp` มีค่าเท่ากับหรือมากกว่าตัวนับเวลาจริง ชื่อ `mtime` อินเทอร์รัปต์ภายนอก (External interrupts) เกิดจากตัวควบคุมอินเทอร์รัปต์ระดับแพลตฟอร์ม ซึ่งเป็นอุปกรณ์ที่เชื่อมต่อกับอุปกรณ์ภายนอกส่วนใหญ่ส่งค่าเข้ามา เนื่องจากแพลตฟอร์มฮาร์ดแวร์แต่ละแบบมีแผนผังหน่วยความจำ (memory maps) และต้องการคุณสมบัติของตัวควบคุมอินเทอร์รัปต์ที่แตกต่างกัน กลไกในการกระตุ้น (raise) และยกเลิก (clear) อินเทอร์รัปต์ จึงแตกต่างกันไปในแต่ละแพลตฟอร์ม อย่างไรก็ตาม สิ่งที่ “เหมือนกัน” ในทุกระบบ RISC-V คือ: วิธีการจัดการข้อยกเว้นและการปิดกั้นอินเทอร์รัปต์ (masking interrupts) ซึ่งเป็นหัวข้อของส่วนถัดไป

11.3 การจัดการข้อยกเว้นในโหมดเครื่อง

มีรีจิสเตอร์ควบคุมและสถานะ (CSRs) จำนวน 8 ตัว ที่มีบทบาทสำคัญต่อการจัดการข้อยกเว้นในโหมดเครื่อง ได้แก่



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

รูปที่ 11.9: ระดับสิทธิพิเศษของ RISC-V และการเข้ารหัส

- *mstatus*, *Machine Status* สถานะเครื่อง เก็บสถานะโดยรวมของระบบ เช่น การเปิดใช้งานอินเทอร์รัปต์ทั่วไป รวมถึงสถานะอื่นๆ อีกจำนวนมาก ตามที่แสดงในรูปที่ 11.4
- *mcause*, *Machine Exception Cause* สาเหตุข้อยกเว้นเครื่อง ระบุว่ามีการข้อยกเว้นใดเกิดขึ้น (ดูรายละเอียดในรูป 11.6)
- *mie*, *Machine Interrupt Enable* การเปิดใช้งานอินเทอร์รัปต์ของเครื่อง ระบุว่าอินเทอร์รัปต์ใดอนุญาตให้โปรเซสเซอร์รับได้หรือถูกละเว้น (ดูรายละเอียดในรูป 11.5)
- *mip*, *Machine Interrupt Pending* สถานะอินเทอร์รัปต์ของเครื่อง แสดงรายการอินเทอร์รัปต์ที่กำลังรอการดำเนินการอยู่ในขณะนั้น (ดูรายละเอียดในรูป 11.5)
- *mtvec*, *Machine Trap Vector* เวกเตอร์การแทรกของเครื่อง เก็บที่อยู่ทีโปรเซสเซอร์จะกระโดดไปเมื่อเกิดข้อยกเว้น ดูรายละเอียดในรูป 11.7)
- *mtval*, *Machine Trap Value* ค่าแทรกเครื่อง เก็บข้อมูลเพิ่มเติมของแทรก เช่น ที่อยู่ที่ทำให้เกิดข้อผิดพลาด (สำหรับข้อยกเว้นที่เกี่ยวข้องกับที่อยู่), คำสั่งที่ผิด (สำหรับข้อยกเว้นคำสั่งไม่ถูกต้อง), และค่าศูนย์ สำหรับข้อยกเว้นอื่นๆ (ดูรายละเอียดในรูป 11.8)
- *mepc*, *Machine Exception PC* ตัวนับโปรแกรมข้อยกเว้นในเครื่อง ชี้ไปยังคำสั่งที่เกิดข้อยกเว้น (ดูรายละเอียดในรูป 11.8)
- *mscratch*, *Machine Scratch* รีจิสเตอร์ชั่วคราวของเครื่อง ใช้เก็บข้อมูลชั่วคราวหนึ่งเวิร์ดสำหรับตัวจัดการแทรก (ดูรายละเอียดในรูป 11.8)

เมื่อโปรเซสเซอร์ทำงานในโหมดเครื่อง (M-mode) อินเทอร์รัปต์จะเกิดขึ้นได้ก็ต่อเมื่อเปิดใช้งานอินเทอร์รัปต์แบบทั่วไป คือ *mstatus.MIE* ถูกตั้งค่าเป็น 1 นอกจากนี้ อินเทอร์รัปต์แต่ละประเภทยังมีเปิดใช้งานของตัวเองในรีจิสเตอร์ CSR *mie* ตำแหน่งของบิตใน *mie* จะสอดคล้องกับรหัสอินเทอร์รัปต์ ในรูป 11.3 ตัวอย่างเช่น *mie[7]* สอดคล้องกับอินเทอร์รัปต์ตัวจับเวลาของเครื่อง รีจิสเตอร์ *mip* จะมีรูปแบบบิตเหมือนกับ *mie* และใช้ระบุว่าอินเทอร์รัปต์ใดที่กำลังรอการจัดการ การนำ CSR ทั้งสามตัวมาพิจารณาร่วมกัน (*mstatus*, *mie*, *mip*) อินเทอร์รัปต์ตัวจับเวลาในโหมดเครื่องจะเกิดขึ้นได้ก็ต่อเมื่อ *mstatus.MIE=1*, *mie[7]=1* และ *mip[7]=1*

เมื่อ hart เกิดข้อยกเว้น ฮาร์ดแวร์จะทำการเปลี่ยนสถานะหลายอย่างพร้อมกัน (แบบอะตอมมิก) ดังนี้:

RISC-V รองรับอินเทอร์รัปต์แบบเวกเตอร์ซึ่งเป็นรูปแบบที่โปรเซสเซอร์จะกระโดดไปยังที่อยู่เฉพาะตามชนิดของอินเทอร์รัปต์นั้นๆ แทนที่จะไปยังจุดเริ่มต้นเดียว (single entry point) การใช้วิธีนี้ช่วย

- ค่า PC ของคำสั่งที่ทำให้เกิดข้อยกเว้นถูกเก็บรักษาไว้ใน `mepc` และ PC จะถูกตั้งค่าเป็น `mtvec` (สำหรับข้อยกเว้นแบบซิงโครนัส `mepc` ซ้ำไปยังคำสั่งที่ทำให้เกิดข้อยกเว้น สำหรับอินเทอร์รัปต์ `mepc` จะชี้ไปยังตำแหน่งที่การดำเนินการควรกลับมาทำงานต่อหลังจัดการอินเทอร์รัปต์เสร็จ)
- `mcause` จะถูกตั้งค่าเป็นรหัสของข้อยกเว้น ตามรูป 11.3 และ `mtval` จะถูกตั้งค่าเป็นข้อมูลที่ผิดพลาดหรือข้อมูลอื่นที่เกี่ยวข้องกับข้อยกเว้นนั้น
- อินเทอร์รัปต์จะถูกปิดใช้งานโดยตั้งค่า `MIE=0` ใน `mstatus` และค่าก่อนหน้าของ `MIE` จะถูกเก็บไว้ใน `MPIE`
- โหมดสิทธิพิเศษก่อนเกิดข้อยกเว้น จะถูกเก็บไว้ในฟิลด์ `MPP` ของ `mstatus` จากนั้นโหมดจะถูกเปลี่ยนเป็นโหมดเครื่อง (M-mode) แสดงการเข้ารหัสของ `MPP` ในรูป 11.9 (หมายเหตุ: หากโปรเซสเซอร์รองรับเฉพาะโหมด M ขั้นตอนนี้จะถูกข้ามไปโดยปริยาย)

เพื่อหลีกเลี่ยงการเขียนทับค่าที่อยู่ในรีจิสเตอร์จำนวนเต็ม ส่วนต้น (prologue) ของตัวจัดการอินเทอร์รัปต์ มักเริ่มต้นด้วยการสลับค่า (swap) ระหว่างรีจิสเตอร์จำนวนเต็มหนึ่งตัว (เช่น `a0`) กับรีจิสเตอร์ `CSR mscratch` โดยทั่วไป ซอฟต์แวร์จะจัดเตรียมให้ `mscratch` เก็บค่าตัวชี้ที่ชี้ไปยังพื้นที่เก็บข้อมูลชั่วคราวในหน่วยความจำ จากนั้น ตัวจัดการจะใช้พื้นที่นี้เพื่อบันทึกรีจิสเตอร์จำนวนเต็มทั้งหมดที่จำเป็นต้องใช้ภายใน ตัวจัดการ หลังจากส่วนหลัก (body) ของตัวจัดการทำงานเสร็จ ส่วนท้าย (epilogue) จะทำการกู้คืนค่ารีจิสเตอร์ที่บันทึกไว้ในหน่วยความจำ จากนั้นสลับค่า `a0` กับ `mscratch` อีกครั้ง เพื่อให้รีจิสเตอร์ทั้งสองกลับสู่ค่าก่อนเกิดข้อยกเว้น สุดท้าย ตัวจัดการจะจบการทำงานด้วยคำสั่ง `mret` ซึ่งเป็นคำสั่งเฉพาะของโหมด M คำสั่ง `mret` จะตั้งค่า PC เป็น `mepc` กู้คืนสถานะอินเทอร์รัปต์โดยคัดลอกค่า `MPIE` ของ `mstatus` ไปยัง `MIE` เพื่อกู้คืนการตั้งค่าการเปิดใช้งานอินเทอร์รัปต์ก่อนหน้า เปลี่ยนโหมดสิทธิกลับเป็นค่าที่เก็บไว้ในฟิลด์ `MPP` ของ `mstatus` ซึ่งโดยรวมแล้ว เป็นการย้อนกลับการเปลี่ยนแปลงที่เกิดขึ้นตอนเกิดข้อยกเว้น

รูปที่ 11.10 แสดงโค้ดแอสเซมบลี RISC-V สำหรับตัวจัดการอินเทอร์รัปต์ของตัวจับเวลาแบบพื้นฐาน ซึ่งทำงานตามรูปแบบที่กล่าวไว้ก่อนหน้านี้ โดยตัวอย่างนี้ทำเพียง เพิ่มค่าตัวเปรียบเทียบเวลาแล้วกลับไปยังงานก่อนหน้า ในทางปฏิบัติ ตัวจัดการอินเทอร์รัปต์ของตัวจับเวลาที่สมจริงมากขึ้น อาจเรียกใช้งานตัวจัดตารางงาน (scheduler) เพื่อสลับการทำงานระหว่าง task ต่าง ๆ ตัวจัดการอินเทอร์รัปต์นี้ไม่สามารถถูกแทรกได้ (non-preemptible) ดังนั้นจึงคงสถานะปิดอินเทอร์รัปต์ตลอดการทำงานของตัวจัดการ (handler) อย่างไรก็ตาม หากไม่นับข้อจำกัดนี้ ตัวอย่างนี้ถือเป็นตัวอย่างที่สมบูรณ์ของตัวจัดการอินเทอร์รัปต์ ใน RISC-V ที่สามารถแสดงอยู่ในหน้าเดียวได้!

บางครั้งอาจต้องการให้อินเทอร์รัปต์ที่มีลำดับความสำคัญสูงกว่า สามารถแทรกเข้ามาได้ระหว่างการจัดการข้อยกเว้นที่มีลำดับต่ำกว่า แต่มีปัญหาคือมีเพียงสำเนาเดียวของรีจิสเตอร์ `CSR mepc, mcause, mtval` และ `mstatus` หากมีอินเทอร์รัปต์ใหม่เข้ามา ค่า




```

# บันทึก รีจิสเตอร์
csrrw a0, mscratch, a0 # บันทึก a0; set a0 = &ที่จัดเก็บชั่วคราว
sw a1, 0(a0)           # บันทึก a1
sw a2, 4(a0)           # บันทึก a2
sw a3, 8(a0)           # บันทึก a3
sw a4, 12(a0)          # บันทึก a4

# ถอดรหัสสาเหตุขัดจังหวะ
csrr a1, mcause         # อ่านสาเหตุขัดจังหวะ
bgez a1, exception     # กระโดดถ้าไม่ขัดจังหวะ
andi a1, a1, 0x3f       # แยกสาเหตุขัดจังหวะ
li a2, 7                # a2 = ตรวจสอบเวลาเหตุขัดจังหวะ
bne a1, a2, otherInt    # กระโดดถ้าไม่มีเวลาขัดจังหวะ

# การจัดการเวลาในการขัดจังหวะโดยการเพิ่มเวลาในการเปรียบเทียบ
la a1, mtimecmp         # a1 = &เปรียบเทียบเวลา
lw a2, 0(a1)            # ดึง 32 bits ล่าง ของตัวเปรียบเทียบ
lw a3, 4(a1)            # ดึง 32 bits บน ของตัวเปรียบเทียบ
addi a4, a2, 1000        # เพิ่ม bits ล่าง 1000 รอบ
sltu a2, a4, a2          # สร้าง ตัวทดออก (carry-out)
add a3, a3, a2           # เพิ่ม bits บน
sw a3, 4(a1)            # จัดเก็บ 32 bits บน
sw a4, 0(a1)            # จัดเก็บ 32 bits ล่าง

# คืนค่ารีจิสเตอร์และส่งคืน
lw a4, 12(a0)           # กู้คืน a4
lw a3, 8(a0)            # กู้คืน a3
lw a2, 4(a0)            # กู้คืน a2
lw a1, 0(a0)            # กู้คืน a1
csrrw a0, mscratch, a0 # กู้คืน a0; mscratch = &ที่จัดเก็บชั่วคราว
mret                   # ออกจาก handler

```

รูปที่ 11.10: โค้ด RISC-V สำหรับตัวจัดการอินเทอร์รัปต์ของตัวจับเวลาแบบง่าย โค้ดนี้ตั้งสมมติว่า ได้เปิดใช้งานอินเทอร์รัปต์แบบควบคุมทั้งหมด (global) แล้ว โดยตั้งค่า `mstatus.MIE`; ได้เปิดใช้งานอินเทอร์รัปต์ตัวจับเวลาแล้ว โดยตั้งค่า `mie[7]`; รีจิสเตอร์ CSR `mtvec` ถูกตั้งค่าให้ชี้ไปยังที่อยู่ของตัวจัดการนี้; และรีจิสเตอร์ CSR `mscratch` ถูกตั้งค่าให้ชี้ไปยังบัฟเฟอร์ในหน่วยความจำชั่วคราว ขนาด 16 ไบต์ เพื่อเก็บรีจิสเตอร์ ส่วนเริ่มต้น (Prologue) ทำการบันทึกรีจิสเตอร์ 5 ตัว โดยเก็บ `a0` ไว้ใน `mscratch` และ `a1-a4` ในหน่วยความจำ จากนั้นตรวจสอบสาเหตุของแทรป โดยตรวจสอบ `mcause`: ถ้า `mcause < 0` เป็นอินเทอร์รัปต์ หรือถ้า `mcause ≥ 0` เป็นข้อบกพร่องแบบซิงโครนัส ถ้าเป็นอินเทอร์รัปต์ จะตรวจสอบว่าบิตล่างของ `mcause` เท่ากับ 7 หรือไม่ ซึ่งหมายถึงเป็นอินเทอร์รัปต์ตัวจับเวลาของเครื่อง ถ้าเป็นอินเทอร์รัปต์ตัวจับเวลา จะเพิ่มค่า 1,000 รอบให้กับตัวเปรียบเทียบเวลา เพื่อให้อินเทอร์รัปต์ตัวจับเวลาครั้งถัดไปไปอีกประมาณ 1,000 รอบ ส่วนท้าย (epilogue) กู้คืนค่ารีจิสเตอร์ `a0-a4` และ `mscratch` จากนั้นกลับไปยังตำแหน่งเดิมด้วยคำสั่ง `mret`

เดิมจะถูกเขียนทับ ทำให้ข้อมูลสูญหายหากไม่มีซอฟต์แวร์ช่วยเหลือเพิ่มเติม ตัวจัดการอินเทอร์รัปต์แบบอินเทอร์รัปต์ล่วงหน้า (preemptible) สามารถบันทึกบริจิสเตอร์เหล่านี้ลงในสแต็กในหน่วยความจำ จากนั้น เปิดใช้งานอินเทอร์รัปต์อีกครั้ง ก่อนออกจากตัวจัดการให้ปิดอินเทอร์รัปต์ และกู้คืนบริจิสเตอร์จากสแต็ก

นอกจากคำสั่ง `mret` ที่ได้กล่าวไปก่อนหน้านี้แล้ว โหมดเครื่องยังมีคำสั่งเพิ่มเติมเพียงคำสั่งเดียวคือ คำสั่ง `wfi` (Wait For Interrupt) คำสั่ง `wfi` ใช้แจ้งให้โปรเซสเซอร์ทราบว่าจะไม่มีงานที่มีประโยชน์ให้ทำในขณะนั้น ดังนั้นโปรเซสเซอร์ควรเข้าสู่โหมดประหยัดพลังงาน (lower-power mode) จนกว่าอินเทอร์รัปต์ที่เปิดใช้งานเกิดขึ้น กล่าวคือ $(mie \& mip) \neq 0$ โปรเซสเซอร์ RISC-V อาจมีวิธีการ implement คำสั่งนี้แตกต่างกันไป เช่น การหยุดนาฬิกาจนกว่าจะมีอินเทอร์รัปต์ หรือบางระบบอาจทำเหมือนคำสั่ง `nop` ดังนั้น โดยทั่วไปคำสั่ง `wfi` มักถูกใช้ภายใน loop

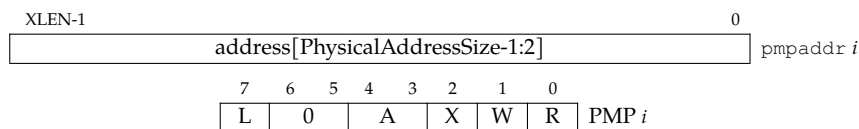
■ **รายละเอียดเพิ่มเติม:** คำสั่ง `wfi` สามารถทำงานได้ ไม่ว่าจะมีการเปิดใช้งานอินเทอร์รัปต์แบบทั่วทั้งระบบ (global) หรือไม่

ถ้า `wfi` ถูกดำเนินการเมื่ออินเทอร์รัปต์แบบทั่วทั้งระบบถูกเปิดอยู่ (`mstatus.MIE=1`) และมีอินเทอร์รัปต์ที่เปิดใช้งานเกิดขึ้น โปรเซสเซอร์จะกระโดดไปยังตัวจัดการข้อยกเว้น ในทางกลับกัน หาก `wfi` ถูกดำเนินการเมื่ออินเทอร์รัปต์ถูกปิดใช้งานทั่วทั้งระบบ (`mstatus.MIE=0`) และมีอินเทอร์รัปต์ที่เปิดใช้งานเกิดขึ้น โปรเซสเซอร์จะไม่กระโดดไปตัวจัดการ แต่จะทำงานต่อจากคำสั่งถัดไปหลัง `wfi` ในกรณีนี้ โค้ดที่ตามหลัง `wfi` มักจะตรวจสอบค่าในรีจิสเตอร์ควบคุมและสถานะ `mip` เพื่อตัดสินใจว่าจะดำเนินการอย่างไรต่อไป แนวทางนี้สามารถลดระยะเวลาหน่วงของอินเทอร์รัปต์ เมื่อเทียบกับการกระโดดไปตัวจัดการข้อยกเว้นโดยตรง เพราะไม่จำเป็นต้องบันทึกและกู้คืนรีจิสเตอร์จำนวนเต็ม

11.4 โหมดผู้ใช้และการแยกกระบวนการในระบบฝังตัว

แม้ว่าโหมดเครื่องจะเพียงพอสำหรับระบบฝังตัวแบบง่าย แต่เหมาะสมเฉพาะเมื่อโค้ดทั้งหมดในระบบเป็นโค้ดที่เชื่อถือได้เท่านั้น เนื่องจาก M-mode มีสิทธิ์เข้าถึงฮาร์ดแวร์ทั้งหมดโดยไม่มีข้อจำกัด อย่างไรก็ตาม ในทางปฏิบัติ มักไม่สามารถเชื่อถือโค้ดของแอปพลิเคชันทั้งหมดได้ เพราะไม่ทราบล่วงหน้าว่าโค้ดคืออะไร หรือมีขนาดใหญ่เกินกว่าที่จะพิสูจน์ความถูกต้องได้ ดังนั้น RISC-V จึงมีกลไกเพื่อปกป้องระบบจากโค้ดที่ไม่น่าเชื่อถือ (untrusted code) และปกป้องโปรเซสที่ไม่น่าเชื่อถือไม่ให้กระทบกันเอง

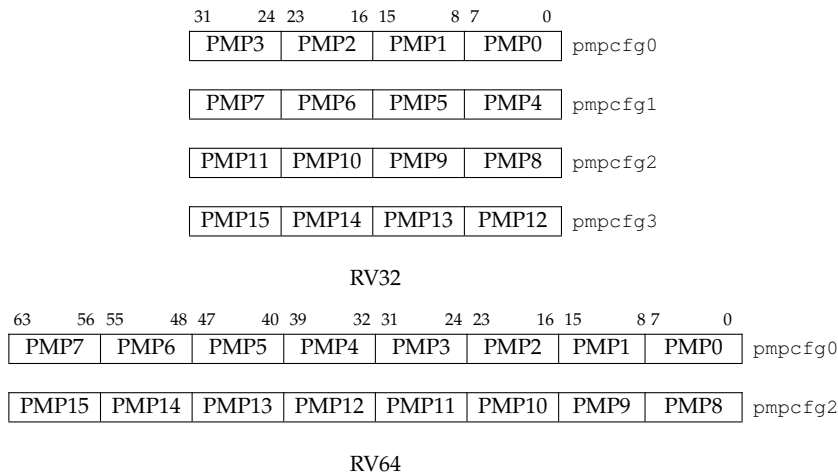
โค้ดที่ไม่น่าเชื่อถือ จะต้องถูกห้ามไม่ให้ใช้คำสั่งที่มีสิทธิพิเศษ เช่น `mret` และเข้าถึงรีจิสเตอร์ CSR ที่มีสิทธิพิเศษ เช่น `mstatus` เนื่องจากสิ่งเหล่านี้จะทำให้โปรแกรมสามารถควบคุมระบบทั้งหมดได้ ข้อจำกัดนี้ทำได้ โดยการเพิ่มโหมดสิทธิพิเศษเพิ่มอีกระดับหนึ่งคือ โหมดผู้ใช้ *User mode* (U-mode) โดย U-mode ไม่อนุญาตให้ใช้คำสั่งหรือ CSR ของ M-mode หากพยายามใช้ จะเกิดข้อยกเว้นคำสั่งไม่ถูกต้อง (illegal instruction exception) นอกเหนือจากข้อจำกัดนี้แล้ว U-mode และ M-mode มีพฤติกรรมคล้ายกันมากซอฟต์แวร์ใน M-mode สามารถเข้าสู่ U-mode ได้โดยตั้งค่าฟิลด์ `mstatus.MPP` เป็น U (ซึ่งมีค่า `encoding = 0` ตามที่ระบุที่ 11.9) จากนั้นดำเนินการคำสั่ง `mret` หากเกิดข้อยกเว้นใน U-mode การควบคุมจะถูกส่งกลับไปยัง M-mode



รูปที่ 11.11: รีจิสเตอร์ที่อยู่และรีจิสเตอร์กำหนดค่าของ PMP รีจิสเตอร์ที่อยู่คือ CSR `pmpaddr` ซึ่งค่าที่เก็บอยู่จะถูกเลื่อนบิตไปทางขวา 2 บิต (right-shifted by 2) และหาขนาดของที่อยู่กายภาพ (physical addresses) มีขนาดน้อยกว่า `XLEN-2` บิต บิตด้านบนที่เหลือจะถูกเติมเป็นศูนย์ รีจิสเตอร์กำหนดค่าของ PMP ลำดับที่ i ประกอบด้วยฟิลด์บิตที่ใช้ควบคุมการเข้าถึงหน่วยความจำ โดย ฟิลด์ R, W และ X ให้สิทธิ์ในการอ่าน (read), เขียน (write) และดำเนินการ (execute) ตามลำดับ ฟิลด์ A ใช้กำหนดโหมดของ PMP และฟิลด์ L ใช้ล็อกการตั้งค่า PMP และรีจิสเตอร์ที่อยู่ที่เกี่ยวข้องไม่ให้ถูกเปลี่ยนแปลง

โค้ดที่ไม่น่าเชื่อถือ (untrusted code) จะต้องถูกจำกัดให้เข้าถึงเฉพาะหน่วยความจำของตนเองเท่านั้น โปรเซสเซอร์ที่รองรับทั้ง M-mode และ U-mode มีคุณสมบัติที่เรียกว่า PMP (*Physical Memory Protection*) ระบบกำหนดสิทธิ์หน่วยความจำระดับฮาร์ดแวร์ ซึ่งช่วยให้ M-mode สามารถกำหนดได้ว่า U-mode สามารถเข้าถึงได้ที่อยู่ใดในหน่วยความจำ PMP ประกอบด้วยรีจิสเตอร์เก็บที่อยู่ (address registers) หลายตัว (โดยทั่วไป 8–16) และรีจิสเตอร์ตั้งค่าที่สอดคล้องกัน โดยรีจิสเตอร์เหล่านี้จะกำหนดสิทธิ์อ่าน (read) เขียน (write) และการดำเนินการ (execute) เมื่อโปรเซสเซอร์ใน U-mode พยายามอ่านคำสั่งหรือดำเนินการคำสั่งโหลด / จัดเก็บ (load/store) ที่อยู่ที่ใช้งาน จะถูกนำไปเปรียบเทียบกับรีจิสเตอร์กำหนดที่อยู่ของ PMP ทั้งหมด ถ้าที่อยู่นั้นมากกว่าหรือเท่ากับที่อยู่ PMP address i แต่น้อยกว่าที่อยู่ PMP $i+1$ รีจิสเตอร์ตั้งค่าที่ PMP $i+1$'s จะตัดสินว่าการเข้าถึงนั้นสามารถดำเนินการต่อได้หรือไม่ หากไม่ผ่านเงื่อนไข จะเกิดข้อยกเว้นการเข้าถึง (access exception)

รูปที่ 11.11 แสดงโครงสร้างของรีจิสเตอร์ที่อยู่และกำหนดค่าของ PMP ทั้งสองเป็นรีจิสเตอร์ควบคุมและสถานะ (CSRs) โดยรีจิสเตอร์ที่อยู่มีชื่อตั้งแต่ `pmpaddr0` ถึง `pmpaddrN` โดยที่ $N+1$ คือจำนวนของ PMP ที่ถูกนำไปออกแบบและมีอยู่จริงในโปรเซสเซอร์ รีจิสเตอร์ที่อยู่ถูกเลื่อนไปทางขวา 2 บิต เนื่องจาก PMP มีความละเอียด (granularity) ที่ระดับ 4 ไบต์ รีจิสเตอร์กำหนดค่าถูกจัดวางอย่างหนาแน่น (densely packed) ใน CSR เพื่อเร่งความเร็วในการสลับบริบท (context switching) ตามที่แสดงรูปที่ 11.12 การกำหนดค่าของ PMP หนึ่งรายการประกอบด้วยบิต R, W และ X ซึ่งเมื่อตั้งค่าแล้วจะอนุญาตให้โหลด (loads), บันทึก (stores), และอ่านค่าคำสั่ง (fetches) ตามลำดับ และมีฟิลด์โหมด A ซึ่งเมื่อเป็น 0 หมายถึงปิดใช้งาน PMP นี้ และเมื่อค่าเป็น 1 เปิดใช้งาน PMP นี้ นอกจากนี้ การกำหนดค่า PMP ยังรองรับโหมดอื่นๆ และสามารถถูกล็อกได้ ซึ่งรายละเอียดเพิ่มเติมอธิบายไว้ในเอกสาร [Waterman and Asanović 2017]



รูปที่ 11.12: โครงสร้างของการกำหนดค่า PMP ภายในรีจิสเตอร์ `pmpcfg` สำหรับ RV32 (ด้านบน) รีจิสเตอร์กำหนดค่าทั้ง 16 ตัว ถูกจัดเก็บแบบ packed อยู่ใน CSR จำนวน 4 ตัว สำหรับ RV64 (ด้านล่าง) รีจิสเตอร์เหล่านี้ถูกจัดเก็บแบบ packed อยู่ใน CSR หมายเลขคู่จำนวน 2 ตัว

11.5 โหมดผู้ดูแลระบบสำหรับระบบปฏิบัติการสมัยใหม่

กลไก PMP ที่อธิบายในส่วนก่อนหน้านี้ เหมาะสมสำหรับระบบฝังตัว เนื่องจากสามารถให้การป้องกันหน่วยความจำได้ในต้นทุนที่ค่อนข้างต่ำ แต่กลไกนี้มีข้อจำกัดหลายประการ ซึ่งทำให้ไม่เหมาะสำหรับการใช้งานในระบบคอมพิวเตอร์ทั่วไป (general-purpose computing) เนื่องจาก PMP รองรับจำนวนขอบเขตหน่วยความจำได้เพียงจำนวนคงที่ (fixed number of memory regions) จึงไม่สามารถขยายตัวเพื่อรองรับแอปพลิเคชันที่มีความซับซ้อนได้ และเนื่องจากขอบเขตเหล่านี้ต้องเป็นช่วงต่อเนื่องกัน (contiguous) ในหน่วยความจำทางกายภาพ ระบบจึงอาจประสบปัญหาการกระจายตัวของหน่วยความจำ (memory fragmentation) นอกจากนี้ PMP ไม่รองรับการแบ่งเพจ (paging) ไปยังหน่วยความจำรอง (secondary storage) ได้อย่างมีประสิทธิภาพ

การกระจายตัวของหน่วยความจำเกิดขึ้นเมื่อหน่วยความจำว่าง แต่ไม่ใช่ส่วนที่ต่อเนื่องกันขนาดใหญ่พอที่จะใช้งานได้

โปรเซสเซอร์ RISC-V ที่ซับซ้อนมากขึ้น จัดการปัญหาเหล่านี้ด้วยวิธีเดียวกับสถาปัตยกรรมประมวลผลทั่วไปเกือบทั้งหมด นั่นคือ การใช้หน่วยความจำเสมือนแบบแบ่งเป็นเพจ (page-based virtual memory) คุณสมบัตินี้เป็นแกนหลักของ โหมดผู้ดูแลระบบ (Supervisor mode หรือ S-mode) ซึ่งเป็นโหมดสิทธิพิเศษแบบทางเลือก (optional privilege mode) ที่ออกแบบมาเพื่อรองรับระบบปฏิบัติการสมัยใหม่ เช่น Linux, FreeBSD และ Windows โดย S-mode มีสิทธิพิเศษมากกว่า U-mode แต่ต่ำกว่า M-mode เช่นเดียวกับ U-mode ซอฟต์แวร์ใน S-mode ไม่สามารถใช้ CSRs หรือคำสั่งของ M-mode ได้ และอยู่ภายใต้ข้อจำกัดของ PMP ในส่วนนี้จะกล่าวถึง อินเทอร์พรีตและข้อยกเว้นใน S-mode และในส่วนถัดไปจะอธิบายรายละเอียดเกี่ยวกับ ระบบหน่วยความจำ

XLEN-1	0
Exception Delegation register [m/s]edeleg	
Interrupt Delegation register [m/s]ideleg	
XLEN	

รูปที่ 11.13: รีจิสเตอร์มอบหมายข้อยกเว้น รีจิสเตอร์มอบหมายข้อยกเว้น และอินเทอร์รับต์ของโหมดเครื่องและโหมดผู้ดูแลระบบ (medeleg, sedeleg, sedeleg, mideleg, sideleg) รีจิสเตอร์เหล่านี้ ช่วยให้สามารถมอบหมายการจัดการไปยังตัวจัดการการแทรก (trap handler) ที่มีสิทธิพิเศษต่ำกว่า โดยตำแหน่งบิตที่เปิดใช้งานจะสอดคล้องกับข้อยกเว้น หรืออินเทอร์รับต์ในรีจิสเตอร์ [m/s]ip

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

รูปที่ 11.14: รีจิสเตอร์ CSR สำหรับ อินเทอร์รับต์ในโหมดผู้ดูแลระบบ รีจิสเตอร์เหล่านี้เป็น CSR แบบอ่าน/เขียน ขนาด XLEN บิต ซึ่งใช้เก็บอินเทอร์รับต์ที่กำลังรอดำเนินการใน (sip) และบิตเปิดใช้งานอินเทอร์รับต์ใน (sie)

ทำไมไม่มอบหมาย (delegate) อินเทอร์รับต์ทั้งหมดไปยัง S-mode โดยไม่มีเงื่อนไข? เหตุผลหนึ่งคือการจำลองเสมือน หาก M-mode ต้องการจำลองอุปกรณ์ (virtualize a device) สำหรับ S-mode อินเทอร์รับต์สำหรับอุปกรณ์นั้นควรถูกส่งไปยัง M-mode ไม่ใช่ S-mode

เสมือนของ S-mode

โดยค่าเริ่มต้น ข้อยกเว้นทั้งหมด ไม่ว่าจะเกิดในโหมดสิทธิพิเศษใด จะถ่ายโอนการควบคุมไปยังตัวจัดการข้อยกเว้นใน M-mode อย่างไรก็ตาม ในระบบ Unix ข้อยกเว้นจำนวนมากควรถูกจัดการโดยระบบปฏิบัติการ ซึ่งทำงานใน S-mode แม้ว่า ตัวจัดการข้อยกเว้นของ M-mode จะสามารถส่งต่อข้อยกเว้นไปยัง S-mode ได้ แต่การมีขั้นตอนเพิ่มนี้จะทำให้การจัดการข้อยกเว้นส่วนใหญ่ช้าลง ดังนั้น RISC-V จึงมีกลไกที่เรียกว่า การมอบหมายข้อยกเว้น (exception delegation) ซึ่งทำให้อินเทอร์รับต์และข้อยกเว้นแบบซิงโครนัส (synchronous exceptions) สามารถมอบหมายไปยัง S-mode ได้แบบเลือกเฉพาะโดยข้าม (bypass) ซอฟต์แวร์ใน M-mode ไปโดยตรง

รีจิสเตอร์ควบคุมและสถานะ mideleg (Machine Interrupt Delegation) ใช้ควบคุมว่าอินเทอร์รับต์ใดถูกมอบหมาย (delegated) ไปยังโหมดผู้ดูแลระบบ (ดังแสดงในรูป 11.13) เช่นเดียวกับ mip และ mie แต่ละบิตใน mideleg จะสอดคล้องกับรหัสข้อยกเว้น หมายเลขเดียวกัน ตามที่แสดงในรูป 11.3 ตัวอย่าง mideleg[5] สอดคล้องกับอินเทอร์รับต์ตัวจับเวลา (timer interrupt) ใน S-mode จะถูกส่งไปยัง อินเทอร์รับต์ตัวจับเวลาใน S-mode แทนที่จะส่งไปยัง M-mode

อินเทอร์รับต์ใดๆ ที่ถูกมอบหมายไปยัง S-mode สามารถถูกปิดกั้น (masked) โดยซอฟต์แวร์ใน S-mode ได้ รีจิสเตอร์ sie และ sip เป็น รีจิสเตอร์ของ S-mode ซึ่งเป็นส่วนย่อยของ mie และ mip (รูปที่ 11.14) รีจิสเตอร์เหล่านี้มีโครงสร้างเหมือนกับรีจิสเตอร์ใน M-mode แต่เฉพาะบิตที่สอดคล้องกับอินเทอร์รับต์ที่ถูกมอบหมายใน mideleg เท่านั้น ที่สามารถอ่านและเขียนได้ผ่าน sie และ sip บิตที่สอดคล้องกับอินเทอร์รับต์ที่

S-mode ไม่สามารถควบคุมอินเทอร์รับต์ตัวจับเวลาและอินเทอร์รับต์ซอฟต์แวร์ได้โดยตรง แต่จะใช้คำสั่ง ecall เพื่อร้องขอให้ M-mode ตั้งค่าตัวจับเวลา หรือส่งอินเทอร์รับต์ระหว่างโปรเซสเซอร์ (interprocessor interrupts) แทน แนวปฏิบัติทางซอฟต์แวร์นี้เป็นส่วนหนึ่งของ SBI Supervisor Binary Interface

XLEN-1		XLEN-2						20		19	18	17		
SD	Reserved						MXR		SUM	Res.				
1	XLEN-21						1		1	1				
16	15	14	13	12	9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]		Res.	SPP	Res.	SPIE	UPIE		Res.	SIE	UIE			
2	2		4	1	2	1	1		2	1	1			

รูปที่ 11.15: sstatus CSR รีจิสเตอร์ sstatus เป็นชุดย่อยของ mstatus (ดูรูปที่ 11.4) จึงมีโครงสร้างที่คล้ายกัน บิต SIE และ SPIE เก็บสถานะการเปิดใช้งานอินเทอร์รัปต์ปัจจุบันและก่อนเกิดข้อยกเว้น (pre-exception interrupt enables) คล้ายกับ MIE และ MPIE ใน mstatus XLEN คือ 32 สำหรับ RV32 หรือ 64 สำหรับ RV64 รูปที่ 4.2 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้ ดูหัวข้อ 4.1 ของเอกสารนั้นสำหรับคำอธิบายของฟิลด์อื่นๆ

ไม่ถูกมอบหมายจะเป็นศูนย์เสมอ

M-mode ยังสามารถมอบหมายข้อยกเว้นแบบซิงโครนัส (synchronous exceptions) ไปยัง S-mode ได้โดยใช้รีจิสเตอร์ medeleg (ดังแสดงในรูป 11.13) กลไกนี้คล้ายกับการมอบหมายอินเทอร์รัปต์ แต่บิตใน medeleg จะสอดคล้องกับรหัสข้อยกเว้นแบบซิงโครนัสตามที่แสดงในรูปที่ 11.3 ตัวอย่าง การตั้งค่า medeleg[15] จะทำให้ store page faults ถูกมอบหมายไปยัง S-mode โดยข้อยกเว้นจะไม่ถูกส่งไปยังโหมดที่มีสิทธิพิเศษต่ำกว่า ไม่ว่าจะเป็นการตั้งค่าการมอบหมายอย่างไรก็ตาม กล่าวคือ ข้อยกเว้นที่เกิดใน M-mode จะถูกจัดการใน M-mode เสมอ ข้อยกเว้นที่เกิดใน S-mode อาจถูกจัดการใน M-mode หรือ S-mode ขึ้นอยู่กับการตั้งค่าการมอบหมาย แต่จะไม่ถูกส่งไปใน U-mode

S-mode มีรีจิสเตอร์ CSRs การจัดการข้อยกเว้นหลายตัว ได้แก่ scause, stvec, sepc, stval, sscratch และ sstatus ซึ่งทำหน้าที่เหมือนกับรีจิสเตอร์ใน M-mode ที่อธิบายไว้ในหัวข้อ 11.2 (ดูรูปที่ 11.7 ถึง 11.8). รูปที่ 11.15 แสดงโครงสร้างของรีจิสเตอร์ sstatus คำสั่ง sret ทำงานเหมือนกับคำสั่ง mret แต่จะทำงานกับรีจิสเตอร์ CSR สำหรับการจัดการข้อยกเว้นของ S-mode แทนที่จะเป็นของ M-mode

กระบวนการเกิดข้อยกเว้น (taking an exception) มีลักษณะคล้ายกับใน M-mode เป็นอย่างมาก หาก hart เกิดข้อยกเว้น และข้อยกเว้นนั้นถูกมอบหมาย (delegate) ไปยัง S-mode ฮาร์ดแวร์จะทำการเปลี่ยนสถานะหลายอย่างแบบอะตอมมิก ในลักษณะที่คล้ายกัน โดยใช้ CSR ของ S-mode แทนที่จะใช้ CSR ของ M-mode เมื่อเกิดข้อยกเว้นใน S-mode จะมีการเปลี่ยนแปลงสถานะดังต่อไปนี้:

- ค่า PC ของคำสั่งที่ทำให้เกิดข้อยกเว้นจะถูกเก็บไว้ใน sepc และ PC จะถูกตั้งค่าเป็น stvec
- scause จะถูกตั้งค่าเป็นรหัสของข้อยกเว้น (ตามที่กำหนดในรูป 11.3) และ stval จะถูกตั้งค่าเป็น ที่อยู่ที่เกิดผิดพลาด หรือข้อมูลเฉพาะของข้อยกเว้นนั้น



31	20	19	10	9	8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V			
12	10	2	1	1	1	1	1	1	1	1			

รูปที่ 11.16: รายการตารางเพจ RV32 Sv32 (PTE)

- อินเทอร์รัปต์ถูกปิดโดยตั้งค่า SIE=0 ในรีจิสเตอร์ sstatus และค่าก่อนหน้าของ SIE ถูกเก็บไว้ใน SPIE
- โหมดก่อนเกิดข้อบกพร่องจะถูกเก็บไว้ในฟิลด์ SPP ของ sstatus จากนั้นโหมดจะถูกเปลี่ยนเป็น S-mode

11.6 หน่วยความจำเสมือนแบบแบ่งเป็นเพจ

S-mode มีระบบหน่วยความจำเสมือนแบบมาตรฐาน ซึ่งแบ่งหน่วยความจำออกเป็นหน่วยขนาดคงที่ที่เรียกว่า *เพจ (pages)* เพื่อใช้ในการแปลงที่อยู่ (address translation) และการป้องกันหน่วยความจำ เมื่อเปิดใช้งานการแบ่งเพจ (paging) แล้ว ที่อยู่ส่วนใหญ่ (รวมถึง effective address ของคำสั่ง load/store และค่า PC) จะเป็น *ที่อยู่เสมือน (virtual addresses)* ซึ่งต้องถูกแปลงเป็น *ที่อยู่จริง (physical addresses)* ก่อนที่จะสามารถเข้าถึงหน่วยความจำจริงได้ การแปลงที่อยู่เสมือนเป็นที่อยู่จริงทำโดยการไล่โครงสร้างต้นไม้แบบหลายทาง (high-radix tree) ที่เรียกว่า *ตารางเพจ (page table)* โหนดปลายทาง (leaf node) ในตารางเพจ จะระบุว่า ที่อยู่เสมือนนั้นมีการแมปไปยังเพจจริงหรือไม่ และถ้าแมป โหมดสิทธิพิเศษใดและประเภทการเข้าถึงใดที่ได้รับอนุญาต ให้เข้าถึงเพจนั้น หากมีการเข้าถึง *หน้าที่ไม่ได้ถูกแมป (unmapped)* หรือมีสิทธิ์ไม่เพียงพอ จะทำให้เกิดข้อบกพร่องเพจฟอลต์ (page fault exception)

รูปแบบการแบ่งเพจของ RISC-V จะถูกตั้งชื่อเป็น SvX โดยที่ X คือขนาดของที่อยู่เสมือนในหน่วยบิต โครงสร้างของ RV32 คือ Sv32 ซึ่งรองรับพื้นที่ที่อยู่เสมือนขนาด 4 GiB โดยพื้นที่นี้ถูกแบ่งออกเป็น 2^{10} *เมกกาเพจ* แต่ละเมกะเพจมีขนาด 4 MiB แต่ละขนาดแบ่งย่อยเป็น 2^{10} *เพจพื้นฐาน (base pages)* —ซึ่งเป็นหน่วยพื้นฐานของการแบ่งเพจ— โดยแต่ละเพจมีขนาด 4 KiB ดังนั้น ตารางเพจของ Sv32 จึงเป็นโครงสร้างแบบต้นไม้สองระดับ (two-level tree) ที่มีฐานเท่ากับ 2^{10} แต่ละรายการในตารางเพจมีขนาด 4 ไบต์ ดังนั้น ตารางเพจ 1 ตารางจึงมีขนาด 4 KiB ซึ่งไม่ใช่เรื่องบังเอิญ ที่ตารางเพจมีขนาดเท่ากับหนึ่งเพจ เพราะการออกแบบนี้ช่วยให้ระบบปฏิบัติการ สามารถจัดสรรหน่วยความจำได้ง่ายขึ้น

รูป 11.16 แสดงโครงสร้างของรายการตารางเพจ Sv32 (PTE) ซึ่งมีฟิลด์ดังต่อไปนี้ อธิบายจากขวาไปซ้าย:

- บิต V ระบุว่ารายการ PTE นี้ถูกต้องหรือไม่ (V=1) ถ้า V=0 การแปลงที่อยู่เสมือนที่ผ่าน PTE นี้จะทำให้เกิดเพจฟอลต์ (page fault)
- บิต R, W และ X ระบุเพจนั้นมีสิทธิ์อ่าน (read), เขียน (write), และดำเนินการ

เพจขนาด 4 KiB ได้รับความนิยมนานแล้วสำหรับคอมพิวเตอร์เริ่มต้นตั้งแต่เครื่อง IBM 360 โมเดล 67 คอมพิวเตอร์ Atlas ซึ่งเป็นเครื่องแรกที่ใช้ระบบ paging มีเพจขนาด 3 KiB (เนื่องจากใช้เวิร์ดขนาด 6 ไบต์) ถือเป็นเรื่องที่น่าสนใจว่า แม้เวลาจะผ่านมากกว่าครึ่งศตวรรษ พร้อมกับการเติบโตแบบทวีคูณของประสิทธิภาพคอมพิวเตอร์ แต่ขนาดของเพจกลับแทบไม่เปลี่ยนแปลงเลย

(execute) ตามลำดับ หากทั้งสามบิตเป็น 0 รายการนี้จะเป็นตัวชี้ไปยังระดับถัดไปของตารางเพจ มิฉะนั้น จะเป็นโหนดสุดท้ายของทรี

- บิต U ระบุว่าเพจนี้เป็นเพจของผู้ใช้หรือไม่ หาก U=0, U-mode ไม่สามารถเข้าถึงเพจนี้ได้ แต่ S-mode สามารถเข้าถึงได้ หาก U=1, U-mode สามารถเข้าถึงได้ แต่ S-mode ไม่สามารถเข้าถึงได้
- บิต G ระบุว่า การแมปนี้มีอยู่ในทุกพื้นที่ของที่อยู่เสมือน (virtual-address spaces) ซึ่งฮาร์ดแวร์ใช้เพื่อเพิ่มประสิทธิภาพการแปลงที่อยู่ โดยทั่วไปใช้สำหรับเพจของระบบปฏิบัติการ
- บิต A ระบุว่าเพจนี้ถูกเข้าถึง (accessed) หรือไม่ ตั้งแต่ครั้งสุดท้ายที่บิต A ถูกเคลียร์
- บิต D ระบุว่าเพจนี้ถูกเขียน (dirty) หรือไม่ ตั้งแต่ครั้งสุดท้ายที่บิต D ถูกเคลียร์
- ฟิลด์ RSW สงวนไว้สำหรับการใช้งานของระบบปฏิบัติการ ฮาร์ดแวร์จะละเว้น
- ฟิลด์ PPN เก็บหมายเลขเพจทางกายภาพ (physical page number) ซึ่งเป็นส่วนหนึ่งของที่อยู่กายภาพ หาก PTE เป็นโหนดใบ PPN จะเป็นส่วนหนึ่งของที่อยู่กายภาพที่แปลได้ มิฉะนั้น PPN จะให้ที่อยู่ของระดับถัดไปของตารางเพจ (รูปที่ 11.16 แบ่ง PPN เป็นสองฟิลด์ย่อยเพื่ออธิบายขั้นตอนการแปลที่อยู่)

ระบบปฏิบัติการใช้บิต A และ D เพื่อตัดสินใจว่าจะสลับเพจใดไปยังที่เก็บข้อมูลสำรองกลางบิต A เป็นระยะ จะช่วยให้ OS ประเมินว่า เพจใดมีการใช้งานน้อยที่สุด เพจ D บิต บ่งชี้ว่าการสลับเพจมีราคาแพงกว่า เพราะต้องเขียนกลับไปในที่เก็บข้อมูลสำรอง

โครงสร้างการแบ่งเพจอื่นๆ สำหรับ RV64 เพียงเพิ่มระดับให้กับตารางเพจ Sv48 คล้ายกับ Sv39 แต่พื้นที่ที่อยู่เสมือนใหญ่กว่า 2^9 เท่า และตารางเพจเพิ่มความลึกอีกหนึ่งระดับ

RV64 รองรับรูปแบบการแบ่งเพจหลายแบบ แต่ในที่นี้จะอธิบายเฉพาะรูปแบบที่ได้รับความนิยมมากที่สุด คือ Sv39 โดย Sv39 ใช้เพจพื้นฐาน (base page) ขนาด 4 KiB เช่นเดียวกับ Sv32 รายการในตารางเพจ (page-table entries) มีขนาดเพิ่มเป็น 8 ไบต์ เพื่อให้สามารถเก็บที่อยู่จริง (physical address) ที่มีขนาดใหญ่ขึ้นได้ เพื่อรักษาหลักการที่ว่าตารางเพจต้องมีขนาดเท่ากับหนึ่งเพจพอดี ค่าฐานของทรีจึงลดลงเป็น 2^9 และโครงสร้างทรีมีความลึก 3 ระดับ พื้นที่ที่อยู่ของ Sv39 มีขนาด 512 GiB และถูกแบ่งออกเป็น 2^9 กิกะเพจ (gigapages) แต่ละกิกะเพจมีขนาด 1 GiB แต่ละกิกะเพจถูกแบ่งย่อยเป็น 2^9 เมกะเพจ (megapages) ซึ่งใน Sv39 มีขนาดเล็กกว่าใน Sv32 เล็กน้อย คือ 2 MiB และแต่ละเมกะเพจจะถูกแบ่งย่อยต่อเป็นขนาด 2^9 เพจพื้นฐาน ขนาด 4 KiB

รูปที่ 11.17 แสดงเค้าโครงสร้างของ PTE ใน Sv39 ซึ่งเหมือนกับ Sv32 ยกเว้นฟิลด์ PPN ที่ขยายเป็น 44 บิต เพื่อรองรับที่อยู่กายภาพ 56 บิต หรือ 2^{26} GiB ของพื้นที่ที่อยู่จริง

63	54 53	28 27	19 18	10 9	8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V	
10	26	9	9	2	1	1	1	1	1	1	1	1	

รูปที่ 11.17: รายการตารางเพจ RV64 Sv39 (PTE)

31	30	22 21	0
MODE	ASID	PPN	RV32
1	9	22	

63	60 59	44 43	0
MODE	ASID	PPN	RV64
4	16	44	

รูปที่ 11.18: รีจิสเตอร์ satp รูปที่ 4.11 และ 4.12 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้

■ รายละเอียดเพิ่มเติม: บิตที่อยู่ที่ไม่ได้ใช้งาน

เนื่องจากที่อยู่เสมือนของ Sv39 มีความกว้างน้อยกว่ารีจิสเตอร์จำนวนเต็มของ RV64 คุณอาจสงสัยว่า บิตอีก 25 บิตที่เหลือจะถูกใช้อย่างไร Sv39 กำหนดว่า บิตของที่อยู่ในช่วง 63–39 จะต้องเป็นสำเนาของบิตที่ 38 ดังนั้น ที่อยู่เสมือนที่ถูกต้องจะอยู่ในช่วง $0000_0000_0000_0000_{hex}-0000_003f_ffff_ffff_{hex}$ และ $ffff_fffc0_0000_0000_{hex}-ffff_ffff_ffff_ffff_{hex}$ ช่องว่างระหว่างสองช่วงนี้มีขนาดใหญ่มากกว่าขนาดของทั้งสองช่วงนั้นถึง 2^{25} เท่า ซึ่งดูเหมือนเป็นการ “เสียพื้นที่” ถึงประมาณ 99.999997% ของค่าที่รีจิสเตอร์ 64 บิตสามารถแทนได้ ทำไมจึงไม่ใช่บิต 25 บิตที่เหลือให้เกิดประโยชน์กว่านี้? คำตอบคือ เมื่อโปรแกรมเริ่มต้องการพื้นที่ที่อยู่เสมือนมากกว่า 512 GiB สถาปนิกต้องการขยายขนาดพื้นที่ที่อยู่โดยไม่ทำลายความเข้ากันได้ย้อนหลัง (backwards compatibility) หากอนุญาตให้โปรแกรมนำบิต 25 บิตส่วนบน ไปเก็บข้อมูลอื่นจะทำให้ไม่สามารถนำบิตเหล่านั้นกลับมาใช้เพื่อรองรับที่อยู่ที่มีขนาดใหญ่ขึ้นในอนาคตได้ การอนุญาตให้เก็บข้อมูลในบิตที่อยู่ที่ไม่ได้ใช้งาน ถือเป็นความผิดพลาดร้ายแรง ซึ่งเคยเกิดขึ้นหลายครั้งในประวัติศาสตร์คอมพิวเตอร์

รีจิสเตอร์ควบคุมและสถานะของ S-mode ชื่อ satp (*Supervisor Address Translation and Protection*) ใช้ควบคุมระบบการแบ่งเพจ (paging system) ดังที่แสดงในรูปที่ 11.18 รีจิสเตอร์ satp ประกอบด้วย 3 ฟิลด์ ฟิลด์ MODE ใช้สำหรับเปิดใช้งานระบบการแบ่งเพจและเลือกความลึกของตารางเพจ การเข้ารหัสดังที่แสดงในรูปที่ 11.19 ฟิลด์ ASID (*Address Space Identifier*) เป็นแบบทางเลือกและสามารถใช้เพื่อลดต้นทุนของการสลับบริบท (context switches) ฟิลด์ PPN (*Physical Page Number*) ใช้เก็บที่อยู่จริงของตารางเพจหลัก (root page table) โดยค่าจะถูกหารด้วยขนาดเพจ 4 KiB โดยทั่วไป ซอฟต์แวร์ในโหมดเครื่อง (M-mode) จะเขียนค่า 0 ลงใน satp ก่อนเข้าสู่ S-mode ครั้งแรกเพื่อปิดใช้งานการแบ่งเพจ จากนั้นซอฟต์แวร์ใน S-mode จะตั้งค่า satp ใหม่อีกครั้ง หลังจากที่ได้สร้างตารางเพจเรียบร้อยแล้ว

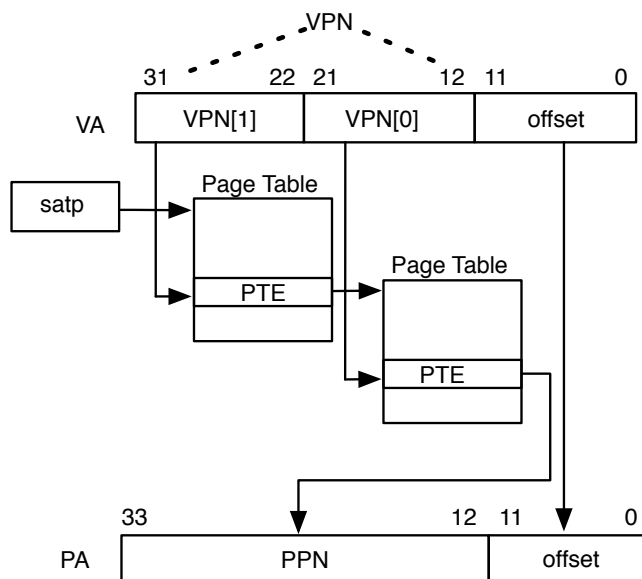
เมื่อเปิดใช้งานการแบ่งหน้า (paging) ในรีจิสเตอร์ satp ที่อยู่เสมือนของโหมด S และโหมด U จะถูกแปลงเป็นที่อยู่จริง โดยการไล่โครงสร้าง ตารางเพจเริ่มจากระดับราก (root) รูป 11.20 แสดงกระบวนการนี้:

1. ฟิลด์ satp.PPN ให้ที่อยู่ฐานของตารางเพจระดับแรก และ VA[31:22] ใช้เป็นดัชนี

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

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

รูปที่ 11.19: การเข้ารหัสของฟิลด์ MODE ในรีจิสเตอร์ควบคุมและสถานะ satp ตาราง 4.3 ของ [Waterman and Asanović 2017] เป็นที่มาของรูปนี้



รูปที่ 11.20: ไดอะแกรมของกระบวนการแปลงที่อยู่ Sv32

ระดับแรก ดังนั้นโปรเซสเซอร์จะอ่านรายการในตารางเพจ (PTE) ตำแหน่งที่อยู่: $(\text{satp.PPN} \times 4096 + \text{VA}[31:22] \times 4)$.

2. PTE ที่ได้จะมีตำแหน่งที่อยู่ฐานของตารางเพจระดับที่สอง และ $\text{VA}[21:12]$ ใช้เป็นดัชนีระดับที่สอง ดังนั้น โปรเซสเซอร์จะอ่านรายการสุดท้ายที่เป็นที่อยู่จริงในตารางเพจ (leaf PTE) ที่ตำแหน่งที่อยู่ $(\text{PTE.PPN} \times 4096 + \text{VA}[21:12] \times 4)$.
3. ฟิลด์ PPN ของรายการสุดท้ายที่เป็นที่อยู่จริงในตารางเพจและบิตที่ระบุตำแหน่งของเพจ *page offset* (12 บิตล่างสุดของที่อยู่เสมือนเดิม) จะให้ผลลัพธ์สุดท้ายเป็นที่อยู่จริง คือ $(\text{LeafPTE.PPN} \times 4096 + \text{VA}[11:0])$

จากนั้นโปรเซสเซอร์จะทำการเข้าถึงหน่วยความจำจริง กระบวนการแปลงที่อยู่ของ Sv39 คล้ายกับ Sv32 แต่ใช้ PTE ขนาดใหญ่กว่าและมีระดับการอ้างอิงเพิ่มขึ้นหนึ่งระดับ (รวมเป็น 3 ระดับ) รูปที่ 11.27 ในตอนท้ายของบทนี้ จะมีการอธิบายอัลกอริทึมการไล่ตารางหน้า (page-table traversal) อย่างครบถ้วน โดยครอบคลุมเงื่อนไขข้อยกเว้นต่างๆ (exceptional conditions) และกรณีพิเศษของการแปลงแบบซูเปอร์เพจ

นี่คือเกือบทั้งหมดของระบบการแบ่งเพจใน RISC-V ยกเว้นเพียงรายละเอียดเล็กน้อยประการหนึ่ง หากทุกคำสั่งการอ่าน (fetches), การโหลด (loads), และการเก็บ (stores) ต้องเข้าถึงตารางเพจหลายครั้ง ระบบการแบ่งเพจจะทำให้ประสิทธิภาพลดลงอย่างมาก! โปรเซสเซอร์สมัยใหม่ทั้งหมดลดภาระนี้ด้วย การใช้แคชสำหรับการแปลงที่อยู่ หรือมักเรียกว่า TLB (Translation Lookaside Buffer) เพื่อให้ต้นทุนของแคชนี้ต่ำลง โปรเซสเซอร์ส่วนใหญ่จะไม่ทำให้ TLB สอดคล้อง (coherent) กับตารางเพจโดยอัตโนมัติ—หากระบบปฏิบัติการแก้ไขตารางเพจ แคชนี้อาจกลายเป็นข้อมูลล้าสมัย (stale) S-mode จึงเพิ่มคำสั่ง `sfence.vma` เพื่อแก้ปัญหานี้ คำสั่ง `sfence.vma` ใช้แจ้งโปรเซสเซอร์ว่า ซอฟต์แวร์อาจมีการแก้ไขตารางเพจ เพื่อให้โปรเซสเซอร์สามารถล้างแคชการแปลง (translation caches) ให้ถูกต้อง คำสั่งนี้มีพารามิเตอร์แบบตัวเลือกสองตัว เพื่อจำกัดขอบเขตของการล้างแคช: `rs1` ระบุที่อยู่เสมือนที่ถูกแก้ไขในตารางเพจ และ `rs2` ระบุ ASID (address-space identifier) ของโปรเซสส์ที่มีการแก้ไขตารางเพจ หากใช้ค่า `x0` ทั้ง `rs1` และ `rs2` จะทำการล้างแคชการแปลงที่อยู่ทั้งหมด

■ รายละเอียดเพิ่มเติม: ความสอดคล้องของแคชการแปลงที่อยู่ในระบบมัลติโปรเซสเซอร์

คำสั่ง `sfence.vma` จะมีผลเฉพาะกับฮาร์ดแวร์การแปลงที่อยู่ของ hart ที่เป็นผู้รันคำสั่งนี้เท่านั้น เมื่อ hart หนึ่งแก้ไขตารางเพจที่ hart อื่นกำลังใช้งานอยู่ hart แรกจะต้องใช้อินเทอร์รัปต์เพื่อแจ้งให้ hart ที่สองทราบว่าควรเรียกใช้คำสั่ง `sfence.vma` ขั้นตอนนี้มักเรียกว่า *TLB shootdown*

11.7 CSR ระบุคุณสมบัติและประสิทธิภาพ

CSR ที่ระบุคุณสมบัติของโปรเซสเซอร์หรือช่วยวัดประสิทธิภาพ CSR มีดังนี้:



ค่าใช้จ่าย

- CSR misa (Machine ISA) ระบุนขนาดของที่อยู่ของโปรเซสเซอร์ (32, 64 หรือ 128 บิต) และระบุว่าส่วนขยายคำสั่งใดถูกรวมอยู่ (รูปที่ 11.21)
- CSR mvendorid (Vendor ID) ระบุนรหัสผู้ผลิต JEDEC ของผู้ให้บริการคอร์ (core) (รูปที่ 11.22)
- CSR marchid (Machine Architecture ID) ระบุนโมโครสถาปัตยกรรมพื้นฐาน การรวม mvendorid และ marchid ระบุนโมโครสถาปัตยกรรมที่พัฒนาขึ้นอย่างไม่ซ้ำกัน (รูปที่ 11.23)
- CSR mimpid (Machine Implementation ID) ระบุนรุ่นของ การพัฒนาของโมโครสถาปัตยกรรมพื้นฐานใน marchid (รูปที่ 11.23)
- CSR mhartid (Hart ID) ระบุนรหัสจำนวนเต็มของ hart ที่กำลังรันอยู่ (รูปที่ 11.23)

CSR การวัดประสิทธิภาพ:

- CSR mtime (Machine Time) เป็นตัวนับแบบเวลาจริง 64 บิต (รูปที่ 11.24)
- CSR mtimecmp (Machine Time Compare) ทำให้เกิดอินเทอร์รัปต์เมื่อ mtime เท่ากับหรือมากกว่าค่าของมัน (รูปที่ 11.24)
- CSR mcounteren และ scounteren รีจิสเตอร์ ควบคุม การ เข้า ถึง ตัว นับ ประสิทธิภาพฮาร์ดแวร์ (hardware performance monitor CSRs) สำหรับระดับ สิทธิพิเศษถัดไปที่ต่ำกว่า (รูปที่ 11.25)
- CSR ตัวนับประสิทธิภาพฮาร์ดแวร์ 31 ตัว (mcycle, minstret, mhpcounter3, ..., mhpcounter31) (clock cycles), คำสั่งที่ดำเนินการเสร็จ (instructions retired), และเหตุการณ์สูงสุด 29 เหตุการณ์ที่เลือกโดยโปรแกรมเมอร์โดยใช้ mhpmevent3,..., mhpmevent31 (รูปที่ 11.26)

11.8 ข้อสังเกตสรุป

การวิจัยแสดงให้เห็นว่านักออกแบบที่ดีที่สุดออกแบบโครงสร้างที่เร็วกว่า เล็กกว่า เรียบง่ายกว่า ชัดเจนกว่า และกระบวนการออกแบบที่ง่ายกว่า ความแตกต่างระหว่างนักออกแบบชั้นยอดและนักออกแบบทั่วไปอาจมาถึงหนึ่งลำดับความสำคัญ

—Fred Brooks, Jr., 1986. ผู้ได้รับรางวัล Turing Award และเป็นสถาปนิกของคอมพิวเตอร์ตระกูล IBM System/360 ซึ่งแสดงให้เห็นถึงความสำคัญของการแยกสถาปัตยกรรมออกจากการพัฒนารุ่นต่อมาของสถาปัตยกรรมปี 1964 นั้นยังคงจำนายอยู่ในปัจจุบัน

ความเป็นโมดูลของสถาปัตยกรรมที่มีสิทธิพิเศษใน RISC-V ตอบสนองความต้องการของระบบหลากหลายประเภท โหมดเครื่องจักร ที่เรียบง่ายรองรับแอปพลิเคชันฝังตัวแบบ bare-metal ด้วยต้นทุนต่ำ โหมดผู้ใช้ (User mode) และการป้องกันหน่วยความจำ



ความเรียบง่าย

odc

การเขียนโปรแกรม

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

รูปที่ 11.21: CSR `misa` รายงาน ISA ที่รองรับ ฟิลด์ MXL (Machine XLEN) เข้ารหัสขนาดของ ISA จำนวนเต็มพื้นฐาน: 1 คือ 32 บิต ,2 คือ 64 บิตและ 3 คือ 128 บิต ฟิลด์ Extensions เข้ารหัสการมีอยู่ของส่วนขยายมาตรฐาน โดยใช้บิตเดียวต่อตัวอักษรของตัวอักษร (บิต 0 ระบุส่วนขยาย “A” บิต 1 ระบุส่วนขยาย “B” จนถึงบิต 25 ซึ่งระบุส่วนขยาย “Z”)

XLEN-1	7	6	0
Vendor ID register (<code>mvendorid</code>)			Offset
XLEN-7			7

รูปที่ 11.22: CSR `mvendorid` ให้รหัสผู้ผลิต JEDEC ของคอร์

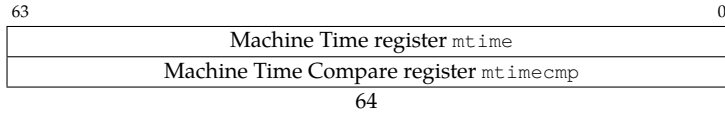
กายภาพ (Physical Memory Protection) ร่วมกันช่วยให้สามารถทำงานหลายงาน (multitasking) ในระบบฝังตัวที่ซับซ้อนมากขึ้น และโหมดผู้ดูแลระบบและหน่วยความจำแบบแบ่งเป็นพจ ให้ความยืดหยุ่นที่จำเป็นในการโฮสต์ระบบปฏิบัติการสมัยใหม่

11.9 เรียนรู้เพิ่มเติม

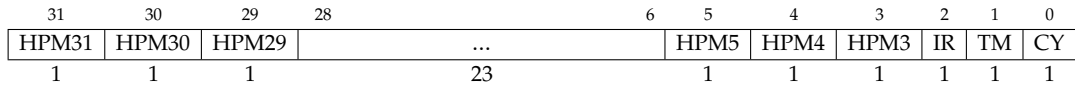
A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017. URL <https://riscv.org/specifications/privileged-isa/>.

XLEN-1	0
Machine Architecture ID register <code>marchid</code>	
Machine Implementation ID register <code>mimpid</code>	
Machine Hart ID register <code>mhartid</code>	
XLEN	

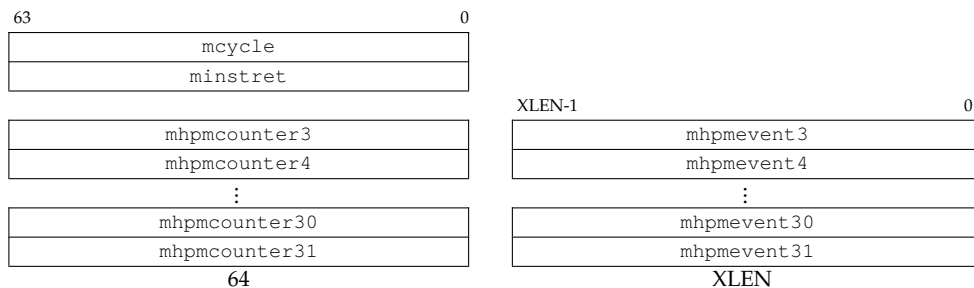
รูปที่ 11.23: CSR ระบุเครื่องจักร (`marchid`, `mimpid`, `mhartid`) ระบุโมโครสถาปัตยกรรมและการพัฒนาของโปรเซสเซอร์ รวมถึงรหัสของ hart ที่กำลังรันอยู่



รูปที่ 11.24: CSR เวลาเครื่องจักร (*mtime* and *mtimecmp*) วัดเวลาและทำให้เกิดอินเทอรัปต์เมื่อ $mtime \geq mtimecmp$



รูปที่ 11.25: รีจิสเตอร์ควบคุมตัวนับ *mcounteren* และ *scounteren* ควบคุมการเข้าถึงตัวนับประสิทธิภาพฮาร์ดแวร์สำหรับโหมดที่มีสิทธิพิเศษต่ำกว่าถัดไป



รูปที่ 11.26: CSR ตัวนับประสิทธิภาพฮาร์ดแวร์ (*mcycle*, *minstret*, *mhpmevent3*, ..., *mhpmevent31*) และเหตุการณ์ที่นับ *mhpmevent3*, ..., *mhpmevent31* สำหรับ RV32 การอ่านค่า CSR ของ *mcycle*, *minstret* และ *mhpmeventn* จะคืนบิตต่ำ 32 บิต, ส่วนการอ่าน รีจิสเตอร์ *mcycleh*, *minstreth* และ *mhpmeventnh* จะคืนบิต 63–32 ของตัวนับที่เกี่ยวข้อง

1. กำหนดให้ a เป็น $\text{satp.ppn} \times \text{PAGESIZE}$, และ ให้ $i = \text{LEVELS} - 1$
2. กำหนดให้ pte เป็นค่าของ PTE ณ ที่อยู่ $a + \text{va.vpn}[i] \times \text{PTESIZE}$
3. ถ้า $pte.v = 0$, หรือ ถ้า $pte.r = 0$ และ $pte.w = 1$, หยุดและเพิ่มข้อยกเว้น page-fault
4. ถ้าไม่ใช่, PTE ไม่ถูกต้อง. ถ้า $pte.r = 1$ หรือ $pte.x = 1$, ไปที่ขั้นตอนที่ 5 มิฉะนั้น PTE นี้จะเป็นตัวชี้ไปยังระดับถัดไปของตารางเพจ กำหนดให้ $i = i - 1$. ถ้า $i < 0$, หยุดและเพิ่มข้อยกเว้น page-fault มิฉะนั้น กำหนดให้ $a = \text{pte.ppn} \times \text{PAGESIZE}$ ไปที่ขั้นตอนที่ 2
5. พบ leaf PTE ตรวจสอบว่าการเข้าถึงหน่วยความจำที่ร้องขอได้รับอนุญาตโดย $pte.r$, $pte.w$, $pte.x$, และ $pte.u$ บิต, กำหนดโหมดสิทธิพิเศษปัจจุบันและค่าของฟิลด์ SUM และ MXR ของรีจิสเตอร์ `mstatus` ถ้าไม่เช่นนั้น ให้หยุดและเพิ่มข้อยกเว้น page-fault
6. ถ้า $i > 0$ และ $\text{pa.ppn}[i - 1 : 0] \neq 0$, นี่คือการเข้าถึงหน่วยความจำที่ไม่ตรงแนว หยุดและเพิ่มข้อยกเว้น page-fault
7. ถ้า $pte.a = 0$, หรือ ถ้า การเข้าถึงหน่วยความจำเป็นที่จัดเก็บ และ $pte.d = 0$, แล้วก็:
 - เพิ่มข้อยกเว้น page-fault หรือ:
 - ตั้งค่าให้ $pte.a$ เป็น 1 และ, หากการเข้าถึงหน่วยความจำเป็นที่เก็บ ให้ตั้งค่าด้วย $pte.d$ เป็น 1
8. การแปลประสบความสำเร็จ ที่อยู่ทางกายภาพที่แปลแล้วจะได้รับดังนี้:
 - $\text{pa.pgoff} = \text{va.pgoff}$
 - ถ้า $i > 0$, นี่คือการแปลเพจขนาดใหญ่และ $\text{pa.ppn}[i - 1 : 0] = \text{va.vpn}[i - 1 : 0]$
 - $\text{pa.ppn}[\text{LEVELS} - 1 : i] = \text{pte.ppn}[\text{LEVELS} - 1 : i]$

รูปที่ 11.27: ขั้นตอนการแปลที่อยู่เสมือนเป็นที่อยู่กายภาพ va เป็นที่อยู่เสมือนที่ป้อนเข้า และ pa เป็นที่อยู่กายภาพที่ส่งออก ค่าคงที่ PAGESIZE คือ 2^{12} สำหรับ Sv32, LEVELS=2 และ PTESIZE=4 ส่วน Sv39, LEVELS=3 และ PTESIZE=8 หัวข้อที่ 4.3.2 ของ [Waterman and Asanović 2017] เป็นที่มาสำหรับรูปนี้

ส่วนขยายเพิ่มเติม RISC-V ในอนาคต

อลัน เพอร์ลิส (1922–1990) เป็นผู้รับรางวัล Turing Award คนแรก (1966) ซึ่งมอบให้สำหรับอิทธิพลของเขาต่อภาษาการเขียนโปรแกรมขั้นสูงและคอมพิวเตอร์ในปี พ.ศ. 2501 เขาได้เข้าร่วมออกแบบ ALGOL ที่มีอิทธิพลต่อภาษาการเขียนโปรแกรมแบบคำสั่งเกือบทุกภาษา รวมถึง C และ Java



คนโง่จะละเลยความซับซ้อน นักปฏิบัติยอมทนกับมัน บางคนหลีกเลี่ยงมันได้ แต่อัจฉริยะจะจัดมันออกไป
—อลัน เพอร์ลิส, 1982

องค์กร RISC-V จะพัฒนาส่วนขยายเสริมเพิ่มเติมอีกหลายตัว

12.1 ส่วนขยายมาตรฐาน “B” สำหรับการจัดการบิต

ในบทที่ 9 เราได้ครอบคลุมชุดย่อยมาตรฐานสี่ชุดของ RV32B แต่ยังมีอีกห้าชุดที่กำลังพิจารณา ตัวอย่างคำสั่งรวมถึง การเก็บดับเบิลเวิร์ดในรีจิสเตอร์เดียว การสับเปลี่ยน การเรียงลำดับ การวางฟิลด์บิต การบีบอัด และการย้ายตามเงื่อนไข

12.2 ส่วนขยายมาตรฐาน “J” สำหรับภาษาที่แปลแบบไดนามิก

ภาษายอดนิยมหลายภาษาจำนวนมากมักถูกนำไปใช้งานผ่านการแปลแบบไดนามิก รวมถึง Java และ Javascript ภาษาเหล่านี้สามารถได้รับประโยชน์จาก ISA ที่สนับสนุนการตรวจสอบแบบไดนามิกและการเก็บรวบรวม (J ย่อมาจาก *Just-In-Time* คอมไพเลอร์))

12.3 ส่วนขยายมาตรฐาน “P” สำหรับคำสั่ง Packed-SIMD

ส่วนขยาย P แบ่งย่อยรีจิสเตอร์สถาปัตยกรรมที่มีอยู่ออก เพื่อให้การคำนวณแบบขนานข้อมูลบนประเภทข้อมูลที่เล็กลง การออกแบบ Packed-SIMD เป็นจุดออกแบบที่สมเหตุสมผลเมื่อใช้ทรัพยากร datapath กว้างที่มีอยู่แล้ว อย่างไรก็ตาม หากต้องการข้อมูลเพิ่มเติมจำนวนมากเกี่ยวกับการดำเนินการแบบขนานข้อมูล บทที่ 8 แสดงให้เห็นว่าการออกแบบสำหรับสถาปัตยกรรมเวกเตอร์เป็นตัวเลือกที่ดีกว่า และสถาปนิกควรใช้ส่วนขยาย RVV

12.4 ส่วนขยายมาตรฐาน “Zk” สำหรับการเข้ารหัสลับแบบสเกลาร์

ส่วนขยายนี้ให้ฟังก์ชันพื้นฐานที่เป็นประโยชน์สำหรับชุดอัลกอริทึมของ NIST, ความหน่วงเวลาการดำเนินการคงที่, และการสร้างบิตสุ่มแบบกำหนดได้

12.5 ข้อสังเกตสรุป

ทำให้ง่าย ไม่ทำให้มันซับซ้อน

—เฮนรี เดวิด ธอโร นักเขียนชื่อดังแห่งศตวรรษที่ 19, 1854

การมีแนวทางคณะกรรมการแบบเปิดและเหมือนมาตรฐานในการขยาย RISC-V คาดว่า จะหมายความว่า การตอบกลับและการถกเถียงจะเกิดขึ้น ก่อนที่คำสั่งจะถูกกำหนดเป็นที่ สิ้นสุด แทนที่จะเป็นหลังจากนั้น เมื่อมันสายเกินไปที่จะเปลี่ยนแปลง ในกรณีที่เหมาะสม สมาชิกบางคนจะนำข้อเสนอไปใช้ก่อนที่มันจะได้รับการรับรอง ซึ่ง FPGA ทำให้ง่ายขึ้นมาก การเสนอส่วนขยายคำสั่งผ่านคณะกรรมการ RISC-V International จะเป็นงานที่ต้องใช้ ความพยายามพอสมควร ซึ่งจะทำให้อัตราการเปลี่ยนแปลงช้าลง ไม่เหมือนกับที่เกิดขึ้นกับ x86-32 (ดูรูปที่ 1.2 หน้าที่ 4 บทที่ 1) อย่าลืมว่าทุกอย่างในบทนี้จะเป็นตัวเลือก ไม่ว่าจะมีการนำส่วนขยายมาใช้ก็ตาม

เราหวังว่า RISC-V จะสามารถพัฒนาไปพร้อมกับความต้องการทางเทคโนโลยี ในขณะที่ รักษาชื่อเสียงในฐานะ ISA ที่เรียบง่ายและมีประสิทธิภาพ หากประสบความสำเร็จ RISC-V จะเป็นการเปลี่ยนแปลงครั้งสำคัญจาก ISA แบบเพิ่มหน่วยของอดีต



ที่ว่างสำหรับส่วนขยาย



สร้างงาน

A

รายการคำสั่ง RISC-V

โคโค ชาแนล (1883-1971)

ผู้ก่อตั้งแบรนด์แฟชั่นชาแนล การแสวงหาความเรียบง่ายที่มีราคาแพงของเธอ ได้กำหนดรูปแบบแฟชั่นแห่งศตวรรษที่ 20



ความเรียบง่ายคือหัวใจสำคัญของความสง่างามที่แท้จริง

—โคโค ชาแนล, 1923

ภาคผนวกนี้แสดงรายการคำสั่งทั้งหมดสำหรับ RV32/64I รวมถึงส่วนขยายทั้งหมดที่ครอบคลุมในหนังสือเล่มนี้ ยกเว้น RVV (RVM, RVA, RVF, RVD และ RVC) และคำสั่งเทียมทั้งหมด แต่ละรายการประกอบด้วยชื่อคำสั่ง, ตัวถูกดำเนินการ, คำจำกัดความในระดับการถ่ายโอนรีจิสเตอร์, ประเภทรูปแบบคำสั่ง คำอธิบายเป็นภาษาไทย คำสั่งบีบอัด (ถ้ามี) และไดอะแกรมที่แสดงโครงสร้างจริงพร้อมรหัสดำเนินการ ซึ่งเป็นสิ่งที่จำเป็นเพื่อทำความเข้าใจคำสั่งทั้งหมดในบทสรุปที่กระชับเหล่านี้ อย่างไรก็ตาม หากคุณต้องการรายละเอียดเพิ่มเติม สามารถอ้างอิงจากข้อกำหนดอย่างเป็นทางการของ RISC-V [Waterman and Asanović 2017]

เพื่อช่วยให้ผู้อ่านพบคำสั่งที่ต้องการในภาคผนวกนี้ ส่วนหัวของหน้าซ้าย (คู่) มีคำสั่งแรกจากด้านบนของหน้านั้น และส่วนหัวของหน้าทางขวา (คี่) จะมีคำสั่งสุดท้ายจากด้านล่างสุดของหน้าหนังสือ รูปแบบคล้ายกับส่วนหัวของพจนานุกรม ซึ่งช่วยให้คุณค้นหาหน้าที่คำสั่งที่คุณต้องการหา ตัวอย่างเช่น ส่วนหัวของหน้า คู่ถัดไปแสดง **AMOADD.D** คำสั่งแรกบนหน้า และส่วนหัวของหน้าคู่ต่อไปแสดง **AMOMAXU.W** ซึ่งเป็นคำสั่งสุดท้ายในหน้านั้น นี่คือสองหน้าที่คุณจะพบคำสั่ง 8 คำสั่งเหล่านี้ `amoadd.d`, `amoadd.w`, `amoand.d`, `amoand.w`, `amomax.d`, `amomax.w`, `amomaxu.d` และ `amomaxu.w`

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

บวก (Add). รูปแบบ-R, RV32I และ RV64I.

บวกค่าในรีจิสเตอร์ $x[rs2]$ กับรีจิสเตอร์ $x[rs1]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.add** rd, rs2; **c.mv** rd, rs2

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

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

บวกค่าทันที (Add Immediate). รูปแบบ-I, RV32I และ RV64I.

บวกค่าทันทีที่ขยายเครื่องหมาย (sign-extended) กับรีจิสเตอร์ $x[rs1]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

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

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

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

บวกเวิร์ดด้วยค่าทันที (Add Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

บวกค่าทันทีที่ขยายเครื่องหมาย (sign-extended) กับรีจิสเตอร์ $x[rs1]$ ตัดทอนผลลัพธ์ให้เหลือ 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมายลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.addiw** rd, imm

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

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

บวกเวิร์ด (Add Word). รูปแบบ-R, RV64I เท่านั้น.

บวกรีจิสเตอร์ $x[rs2]$ กับรีจิสเตอร์ $x[rs1]$ ตัดทอนผลลัพธ์ให้เหลือ 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมาย ลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.addw** rd, rs2

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: บวกดับเบิลเวิร์ด (Add Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น ตั้งค่าดับเบิลเวิร์ดในหน่วยความจำ เป็น $t + x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: บวกเวิร์ด (Add Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น ตั้งค่าเวิร์ดในหน่วยความจำ เป็น $t + x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: AND ดับเบิลเวิร์ด (AND Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต AND ระหว่าง t และค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: AND เวิร์ด (AND word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต AND ระหว่าง t และค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมาย (sign-extended) ของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดดับเบิลเวิร์ด (Maximum Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น เปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองแล้วเขียนค่าที่มากกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดเวิร์ด (Maximum Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้นเปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบส่วนเติมเต็มสองแล้วเขียนค่าที่มากกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมาย (sign-extended) ของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดดับเบิลเวิร์ดแบบไม่มีเครื่องหมาย (Maximum Doubleword, Unsigned). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น เปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย (unsigned) แล้วเขียนค่าที่มากกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าสูงสุดเวิร์ดแบบไม่มีเครื่องหมาย (Maximum Word, Unsigned). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น เปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย แล้วเขียนค่าที่มากกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมาย (sign-extended) ของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดดับเบิลเวิร์ด (Minimum Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น ตั้งค่าดับเบิลเวิร์ดในหน่วยความจำ เป็นค่าที่น้อยกว่าระหว่าง t และ $x[rs2]$ โดยใช้การเปรียบเทียบในรูปแบบส่วนเติมเต็มสองตั้งค่า $x[rd]$ เป็น t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดเวิร์ด (Minimum Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น เปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบส่วนเดิมเต็มสองแล้วเขียนค่าที่น้อยกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดดับเบิลเวิร์ดแบบไม่มีเครื่องหมาย (Minimum Doubleword, Unsigned). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้นเปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย แล้วเขียนค่าที่น้อยกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ โดยใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย ตั้งค่า $x[rd]$ เป็น t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: ค่าต่ำสุดเวิร์ดแบบไม่มีเครื่องหมาย (Minimum Word, Unsigned). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น เปรียบเทียบ t กับค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ โดยใช้การเปรียบเทียบแบบไม่มีเครื่องหมาย (unsigned) แล้วเขียนค่าที่น้อยกว่าลงในตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: OR ดับเบิลเวิร์ด (OR Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต OR ระหว่าง t และค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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) $x[rd] = \text{AMO32}(M[x[rs1]] \mid x[rs2])$

การดำเนินการหน่วยความจำแบบอะตอมิก: OR เวิร์ด (OR Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต OR ระหว่าง t และค่าที่อยู่ในรีจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: สลับดับเบิลเวิร์ด (Swap Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น ตั้งค่าดับเบิลเวิร์ดในหน่วยความจำเป็นค่า $x[rs2]$ ตั้งค่ารีจิสเตอร์ $x[rd]$ เป็น t (ค่าดั้งเดิมจากหน่วยความจำ)

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) $x[rd] = \text{AMO32}(M[x[rs1]] \text{ SWAP } x[rs2])$

การดำเนินการหน่วยความจำแบบอะตอมิก: สลับเวิร์ด (Swap Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น ตั้งค่าเวิร์ดในหน่วยความจำเป็นค่า $x[rs2]$ ตั้งค่า $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t (ค่าดั้งเดิมจากหน่วยความจำ)

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) $x[rd] = \text{AMO64}(M[x[rs1]] \wedge x[rs2])$

การดำเนินการหน่วยความจำแบบอะตอมิก: XOR ดับเบิลเวิร์ด (XOR Doubleword). รูปแบบ-R, RV64A เท่านั้น.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของดับเบิลเวิร์ด (doubleword) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต XOR ระหว่าง t และค่าที่อยู่ในเรจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่าเรจิสเตอร์ $x[rd]$ เป็นค่า t (ค่าดั้งเดิมจากหน่วยความจำ)

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

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

การดำเนินการหน่วยความจำแบบอะตอมิก: XOR เวิร์ด (XOR Word). รูปแบบ-R, RV32A และ RV64A.

ดำเนินการแบบอะตอมิกโดย: กำหนดให้ t เป็นค่าของเวิร์ด (word) ในหน่วยความจำที่ตำแหน่ง $x[rs1]$ จากนั้น คำนวณการดำเนินการแบบบิตต่อบิต XOR ระหว่าง t และค่าที่อยู่ในเรจิสเตอร์ $x[rs2]$ แล้วเขียนผลลัพธ์กลับไปยังตำแหน่งหน่วยความจำเดิม ตั้งค่าเรจิสเตอร์ $x[rd]$ เป็นค่าที่ขยายเครื่องหมายของ t

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

and rd, rs1, rs2 $x[rd] = x[rs1] \& x[rs2]$

AND. รูปแบบ-R, RV32I และ RV64I.

คำนวณผลลัพธ์ของการดำเนินการ AND แบบบิตต่อบิตระหว่างเรจิสเตอร์ $x[rs1]$ และ เรจิสเตอร์ $x[rs2]$ และเขียนผลลัพธ์ลงในเรจิสเตอร์ $x[rd]$.

คำสั่งบีบอัด: **c.and** rd, rs2

31	25	24	20	19	15	14	12	11	7	6	0
0000000	rs2	rs1	111	rd	0110011						

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

AND ค่าทันที (AND Immediate). รูปแบบ-I, RV32I และ RV64I.

คำนวณผลลัพธ์ของการดำเนินการ AND แบบบิตต่อบิตระหว่าง ค่าทันทีที่ขยายเครื่องหมาย (sign-extended) และ เรจิสเตอร์ $x[rs1]$ และเขียนผลลัพธ์ลงในเรจิสเตอร์ $x[rd]$.

คำสั่งบีบอัด: **c.andi** rd, imm

31	20	19	15	14	12	11	7	6	0
immediate[11:0]	rs1	111	rd	0010011					

auipc rd, immediate $x[rd] = pc + sext(immediate[31:12] \ll 12)$

บวกค่าคงที่ส่วนบนกับ PC (Add Upper Immediate to PC). รูปแบบ-U, RV32I และ RV64I.

บวก ค่าคงที่ ขนาด 20 บิตที่ขยายเครื่องหมาย (sign-extended) และเลื่อนซ้าย 12 บิต เข้ากับ (pc) และเขียนผลลัพธ์ลงในรีจิสเตอร์ x[rd]

31	12	11	7	6	0
immediate[31:12]					
rd			0010111		

beq rs1, rs2, offset if (rs1 == rs2) pc += sext(offset)

กระโดดถ้าเท่ากัน (Branch if Equal). รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ x[rs1] เท่ากับค่าใน รีจิสเตอร์ x[rs2] ให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า offset ที่ขยายเครื่องหมาย (sign-extended)

คำสั่งบีบอัด: **c.beq** 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)

กระโดดถ้าเท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

beq rs1, x0, offset

bge rs1, rs2, offset if (rs1 $\geq_s$ rs2) pc += sext(offset)

กระโดดถ้ามากกว่าหรือเท่ากับ. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ x[rs1] มากกว่าหรือเท่ากับค่าในรีจิสเตอร์ x[rs2] โดยพิจารณาค่าเป็นจำนวนเต็มในรูปแบบส่วนเติมเต็มสองให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า offset ที่ขยายเครื่องหมาย (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)

กระโดดถ้ามากกว่าหรือเท่ากับแบบไม่มีเครื่องหมาย. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ x[rs1] ค่ามากกว่าหรือเท่ากับค่าในรีจิสเตอร์ x[rs2] โดยพิจารณาค่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย ให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า offset ที่ขยายเครื่องหมาย (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)
```

กระโดดถ้ามามากกว่าหรือเท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

bge rs1, x0, offset

```
bgt rs1,rs2,offset      if (rs1 >s rs2) pc += sext(offset)
```

กระโดดถ้ามากกว่า. คำสั่งเทียบ, RV32I และ RV64I.

blt rs2, rs1, offset

```
bgtu    rs1,rs2,offset      if (rs1 >u rs2) pc += sext(offset)
```

กระโดดถ้ามามากกว่าแบบไม่มีเครื่องหมาย. คำสั่งเทียม, RV32I และ RV64I.

bltu rs2, rs1, offset

```
bgtz rs2, offset           if (rs2 >_s 0) pc += sext(offset)
```

กระโดดถ้ามามากกว่าศูนย์. คำสั่งเทียบ, RV32I และ RV64I.

blt x0, rs2, offset

```
ble rs1,rs2,offset      if (rs1  $\leq_s$  rs2) pc += sext(offset)
```

กระโดดถ้าน้อยกว่าหรือเท่ากับ. คำสั่งเทียม, RV32I และ RV64I.

bge rs2, rs1, offset

```
bleu rs1,rs2,offset      if (rs1  $\leq_u$  rs2) pc += sext(offset)
```

กระโดดถ้าน้อยกว่าหรือเท่ากับแบบไม่มีเครื่องหมาย. คำสั่งเทียม, RV32I และ RV64I.

bgeu rs2, rs1, offset

```
blez rs2, offset      if (rs2 ≤s 0) pc += sext(offset)
```

กระโดดถ้าน้อยกว่าหรือเท่ากับศูนย์. คำสั่งเพี้ยน, RV32I และ RV64I.

bge x0, rs2, offset

blt $rs1, rs2, offset$ if ($rs1 <_s rs2$) $pc += sext(offset)$

กระโดดถ้าน้อยกว่า. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ $x[rs1]$ น้อยกว่าค่าในรีจิสเตอร์ $x[rs2]$ โดยพิจารณาค่าเป็นจำนวนเต็มในรูปแบบส่วนเติมเต็มสองให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า $offset$ ที่ขยายเครื่องหมาย (sign-extended)

31	25 24	20 19	15 14	12 11	7 6	0
offset[12 10:5]	rs2	rs1	100	offset[4:1 11]	1100011	

bltu $rs1, rs2, offset$ if ($rs1 <_u rs2$) $pc += sext(offset)$

กระโดดถ้าน้อยกว่าแบบไม่มีเครื่องหมาย. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ $x[rs1]$ น้อยกว่าค่าในรีจิสเตอร์ $x[rs2]$ โดยพิจารณาค่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมายให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า $offset$ ที่ขยายเครื่องหมาย (sign-extended)

31	25 24	20 19	15 14	12 11	7 6	0
offset[12 10:5]	rs2	rs1	110	offset[4:1 11]	1100011	

bltz $rs1, offset$ if ($rs1 <_s 0$) $pc += sext(offset)$

กระโดดถ้าน้อยกว่าศูนย์. คำสั่งเทียม, RV32I และ RV64I.

blt $rs1, x0, offset$

bne $rs1, rs2, offset$ if ($rs1 \neq rs2$) $pc += sext(offset)$

กระโดดถ้าไม่เท่ากัน. รูปแบบ-B, RV32I และ RV64I.

ถ้าค่าในรีจิสเตอร์ $x[rs1]$ ไม่เท่ากับค่ารีจิสเตอร์ $x[rs2]$ ให้ตั้งค่า pc เป็น pc ปัจจุบันบวกกับค่า $offset$ ที่ขยายเครื่องหมาย (sign-extended)

คำสั่งบีบอัด: **c.bnez** $rs1, offset$

31	25 24	20 19	15 14	12 11	7 6	0
offset[12 10:5]	rs2	rs1	001	offset[4:1 11]	1100011	

bnez $rs1, offset$ if ($rs1 \neq 0$) $pc += sext(offset)$

กระโดดถ้าไม่เท่ากับศูนย์. คำสั่งเทียม, RV32I และ RV64I.

bne $rs1, x0, offset$

c.add rd, rs2 $x[rd] = x[rd] + x[rs2]$

บวก (Add). RV32IC และ RV64IC.

ขยายเป็น **add** rd, rd, rs2. ไม่ถูกต้องเมื่อ rd=x0 หรือ rs2=x0

15	13	12	11	7	6	2	1	0
100	1		rd		rs2			10

c.addi rd, imm $x[rd] = x[rd] + \text{sext}(imm)$

บวกค่าทันที (Add Immediate). RV32IC และ RV64IC.

ขยายเป็น **addi** rd, rd, imm

15	13	12	11	7	6	2	1	0
000		imm[5]	rd		imm[4:0]			01

c.addi16sp imm $x[2] = x[2] + \text{sext}(imm)$

บวกค่าทันทีที่ปรับขนาดด้วย 16 กับตัวชี้สแต็ก RV32IC และ RV64IC.

ขยายเป็น **addi** x2, x2, imm. ไม่ถูกต้องเมื่อ imm=0

15	13	12	11	7	6	2	1	0
011		imm[9]	00010		imm[4]6[8:7]5			01

c.addi4spn rd', uimm $x[8+rd'] = x[2] + \text{uimm}$

บวกค่าทันทีที่ปรับสเกลด้วย 4 กับตัวชี้สแต็กโดยไม่เปลี่ยนตัวชี้สแต็ก RV32IC และ RV64IC.

ขยายเป็น **addi** rd, x2, uimm, โดยที่ rd=8+rd'. ไม่ถูกต้องเมื่อ uimm=0

15	13	12		5	4	2	1	0
000			uimm[5:4]9:6[2]3		rd'			00

c.addiw rd, imm $x[rd] = \text{sext}((x[rd] + \text{sext}(imm)) [31:0])$

บวกเวิร์ดด้วยค่าทันที (Add Word Immediate). RV64IC เท่านั้น.

ขยายเป็น **addiw** rd, rd, imm. ไม่ถูกต้องเมื่อ rd=x0

15	13	12	11	7	6	2	1	0
001		imm[5]	rd		imm[4:0]			01

c.addw rd', rs2' $x[8+rd'] = \text{sext}((x[8+rd'] + x[8+rs2']) [31:0])$

บวกเวิร์ด (Add Word). RV64IC เท่านั้น.

ขยายเป็น **addw** rd, rd, rs2, โดยที่ rd=8+rd' และ rs2=8+rs2'

15		10	9	7	6	5	4	2	1	0
		100111		rd'		01		rs2'		01

c.and rd', rs2' $x[8+rd'] = x[8+rd'] \& x[8+rs2']$

AND. RV32IC และ RV64IC.

ขยายเป็น **and** rd, rd, rs2, โดยที่ $rd=8+rd'$ และ $rs2=8+rs2'$

15	10	9	7	6	5	4	2	1	0
100011	rd'	11	rs2'	01					

c.andi rd', imm $x[8+rd'] = x[8+rd'] \& sext(imm)$

AND ค่าทันที *AND Immediate*. RV32IC และ RV64IC.

ขยายเป็น **andi** rd, rd, imm, โดยที่ $rd=8+rd'$

15	13	12	11	10	9	7	6	2	1	0
100	imm[5]	10	rd'	imm[4:0]	01					

c.beqz rs1', offset $if (x[8+rs1'] == 0) pc += sext(offset)$

กระโดดถ้าเท่ากับศูนย์ *Branch if Equal to Zero*. RV32IC และ RV64IC.

ขยายเป็น **beq** rs1, x0, offset, โดยที่ $rs1=8+rs1'$

15	13	12	10	9	7	6	2	1	0
110	offset[8:4:3]	rs1'	offset[7:6:2:1:5]	01					

c.bnez rs1', offset $if (x[8+rs1'] \neq 0) pc += sext(offset)$

กระโดดถ้าไม่เท่ากับศูนย์ *Branch if Not Equal to Zero*. RV32IC และ RV64IC.

ขยายเป็น **bne** rs1, x0, offset, โดยที่ $rs1=8+rs1'$

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 $RaiseException(Breakpoint)$

จุดหยุดสภาพแวดล้อม (*Environment Breakpoint*). RV32IC และ RV64IC.

ขยายเป็น **ebreak**.

15	13	12	11	7	6	2	1	0
100	1	00000	00000	10				

c.fld rd', uimm(rs1') $f[8+rd'] = M[x[8+rs1'] + uimm][63:0]$

โหลดดับเบิลเวิร์ดจำนวนลอยตัว (*Floating-point Load Doubleword*). RV32DC และ RV64DC.

ขยายเป็น **fld** rd, uimm(rs1), โดยที่ $rd=8+rd'$ และ $rs1=8+rs1'$

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) $f[rd] = M[x[2] + uimm][63:0]$

โหลดดับเบิลเวิร์ดจำนวนลอยตัวสัมพันธ์กับตัวชี้สแตก (*Floating-point Load Doubleword, Stack-Pointer Relative*). RV32DC และ RV64DC.

ขยายเป็น **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') $f[8+rd'] = M[x[8+rs1'] + uimm][31:0]$

โหลดเวิร์ดจำนวนลอยตัว (*Floating-point Load Word*). RV32FC เท่านั้น.

ขยายเป็น **flw** rd, uimm(rs1), โดยที่ $rd=8+rd'$ และ $rs1=8+rs1'$

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) $f[rd] = M[x[2] + uimm][31:0]$

โหลดเวิร์ดจำนวนลอยตัวสัมพันธ์กับตัวชี้สแตก (*Floating-point Load Word, Stack-Pointer Relative*). RV32FC เท่านั้น.

ขยายเป็น **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') $M[x[8+rs1'] + uimm][63:0] = f[8+rs2']$

บันทึกดับเบิลเวิร์ดจำนวนลอยตัว (*Floating-point Store Doubleword*). RV32DC และ RV64DC.

ขยายเป็น **fsd** rs2, uimm(rs1), where $rs2=8+rs2'$ และ $rs1=8+rs1'$

15	13	12	10	9	7	6	5	4	2	1	0
101	uimm[5:3]			rs1'		uimm[7:6]		rs2'		00	

c.fsdsp rs2, uimm(x2) $M[x[2] + uimm][63:0] = f[rs2]$

บันทึกดับเบิลเวิร์ดจำนวนลอยตัวสัมพันธ์กับตัวชี้สแตก (*Floating-point Store Doubleword, Stack-Pointer Relative*). RV32DC และ RV64DC.

ขยายเป็น **fsd** rs2, uimm(x2)

15	13	12	7	6	2	1	0
101	uimm[5:3 8:6]				rs2		10

c.fsw $rs2', uimm(rs1')$ $M[x[8+rs1'] + uimm][31:0] = f[8+rs2']$

บันทึกเวิร์ดจำนวนลอยตัว (Floating-point Store Word). RV32FC เท่านั้น.

ขยายเป็น **fsw** $rs2, uimm(rs1)$, โดยที่ $rs2=8+rs2'$ และ $rs1=8+rs1'$

15	13	12	10	9	7	6	5	4	2	1	0
111	uimm[5:3]			rs1'		uimm[2:6]		rs2'		00	

c.fswsp $rs2, uimm(x2)$ $M[x[2] + uimm][31:0] = f[rs2]$

บันทึกเวิร์ดจำนวนลอยตัวสัมพันธ์กับตัวชี้สแต็ก (Floating-point Store Word, Stack-Pointer Relative). RV32FC เท่านั้น.

ขยายเป็น **fsw** $rs2, uimm(x2)$

15	13	12	7	6	2	1	0
111	uimm[5:2 7:6]			rs2		10	

c.j $offset$ $pc += sext(offset)$

กระโดด (Jump). RV32IC และ RV64IC.

ขยายเป็น **j** $x0, offset$.

15	13	12	2	1	0
101	offset[11 4 9:8 10 6 7 3:1 5]				
					01

c.jal $offset$ $x[1] = pc+2; pc += sext(offset)$

กระโดดและเชื่อมโยง (Jump and Link). RV32IC เท่านั้น.

ขยายเป็น **jal** $x1, offset$.

15	13	12	2	1	0
001	offset[11 4 9:8 10 6 7 3:1 5]				
					01

c.jalr $rs1$ $t = pc+2; pc = x[rs1]; x[1] = t$

รีจิสเตอร์กระโดดและเชื่อมโยง (Jump and Link Register). RV32IC และ RV64IC.

ขยายเป็น **jalr** $x1, 0(rs1)$. ไม่ถูกต้องเมื่อ $rs1=x0$

15	13	12	11	7	6	2	1	0
100	1	rs1		00000		10		

c.jr $rs1$ $pc = x[rs1]$

รีจิสเตอร์กระโดด (Jump Register). RV32IC และ RV64IC.

ขยายเป็น **jalr** $x0, 0(rs1)$. ไม่ถูกต้องเมื่อ $rs1=x0$

15	13	12	11	7	6	2	1	0
100	0	rs1		00000		10		

c.ld rd', uimm(rs1') $x[8+rd'] = M[x[8+rs1'] + uimm][63:0]$

โหลดดับเบิลเวิร์ด (Load Doubleword). RV64IC เท่านั้น.

ขยายเป็น **ld** rd, uimm(rs1), โดยที่ $rd=8+rd'$ และ $rs1=8+rs1'$

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) $x[rd] = M[x[2] + uimm][63:0]$

โหลดดับเบิลเวิร์ดสัมพันธ์กับตัวชี้สแต็ก (Load Doubleword, Stack-Pointer Relative). RV64IC เท่านั้น.

ขยายเป็น **ld** rd, uimm(x2). ไม่ถูกต้องเมื่อ $rd=x0$

15	13	12	11	7	6	2	1	0
011	uimm[5]		rd		uimm[4:3 8:6]		10	

c.li rd, imm $x[rd] = sext(imm)$

โหลดค่าทันที (Load Immediate). RV32IC และ RV64IC.

ขยายเป็น **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 $x[rd] = sext(imm[17:12] \ll 12)$

โหลดค่าทันทีส่วนบน (Load Upper Immediate). RV32IC และ RV64IC.

ขยายเป็น **lui** rd, imm. ไม่ถูกต้องเมื่อ $rd=x2$ 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') $x[8+rd'] = sext(M[x[8+rs1'] + uimm][31:0])$

โหลดเวิร์ด (Load Word). RV32IC และ RV64IC.

ขยายเป็น **lw** rd, uimm(rs1), โดยที่ $rd=8+rd'$ และ $rs1=8+rs1'$

15	13	12	10	9	7	6	5	4	2	1	0
010	uimm[5:3]			rs1'		uimm[2 6]		rd'		00	

c.lwsp rd, uimm(x2) $x[rd] = sext(M[x[2] + uimm][31:0])$

โหลดเวิร์ดสัมพันธ์กับตัวชี้สแต็ก (Load Word, Stack-Pointer Relative). RV32IC และ RV64IC.

ขยายเป็น **lw** rd, uimm(x2). ไม่ถูกต้องเมื่อ $rd=x0$

15	13	12	11	7	6	2	1	0
010	uimm[5]		rd		uimm[4:2 7:6]		10	

c.mv rd, rs2

$$x[rd] = x[rs2]$$

คัดลอก (*Move*). RV32IC และ RV64IC.ขยายเป็น **add** rd, x0, rs2. ไม่ถูกต้องเมื่อ 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 และ RV64IC.

ขยายเป็น **or** rd, rd, rs2, โดยที่ rd=8+rd' และ rs2=8+rs2'

15	10	9	7	6	5	4	2	1	0
100011	rd'	10	rs2'	01					

c.sd rs2', uimm(rs1')

$$M[x[8+rs1'] + uimm][63:0] = x[8+rs2']$$

บันทึกดับเบิลเวิร์ด (*Store Doubleword*). RV64IC เท่านั้น.ขยายเป็น **sd** rs2, uimm(rs1), โดยที่ rs2=8+rs2' และ rs1=8+rs1'

15	13	12	10	9	7	6	5	4	2	1	0
111	uimm[5:3]	rs1'	uimm[7:6]	rs2'	00						

c.sdsp rs2, uimm(x2)

$$M[x[2] + uimm][63:0] = x[rs2]$$

บันทึกดับเบิลเวิร์ดสัมพันธ์กับตัวชี้สแต็กสัมพันธ์ (*Store Doubleword, Stack-Pointer Relative*). RV64IC เท่านั้น.ขยายเป็น **sd** rs2, uimm(x2)

15	13	12	7	6	2	1	0
111	uimm[5:3 8:6]	rs2	10				

c.slli rd, uimm

$$x[rd] = x[rd] \ll uimm$$

เลื่อนซ้ายแบบลอจิกด้วยค่าทันที (*Shift Left Logical Immediate*). RV32IC และ RV64IC.ขยายเป็น **slli** rd, rd, uimm

15	13	12	11	7	6	2	1	0
000	uimm[5]	rd	uimm[4:0]	10				

c.srai $rd', uimm$ $x[8+rd'] = x[8+rd'] \gg_s uimm$

เลื่อนขวาแบบเลขคณิตด้วยค่าทันที (*Shift Right Arithmetic Immediate*). RV32IC และ RV64IC.

ขยายเป็น **srai** $rd, rd, uimm$, โดยที่ $rd=8+rd'$

15	13	12	11	10	9	7	6	2	1	0
100	uimm[5]			01	rd'		uimm[4:0]		01	

c.srli $rd', uimm$ $x[8+rd'] = x[8+rd'] \gg_u uimm$

เลื่อนขวาแบบลอจิกด้วยค่าทันที (*Shift Right Logical Immediate*). RV32IC และ RV64IC.

ขยายเป็น **srli** $rd, rd, uimm$, โดยที่ $rd=8+rd'$

15	13	12	11	10	9	7	6	2	1	0
100	uimm[5]			00	rd'		uimm[4:0]		01	

c.sub $rd', rs2'$ $x[8+rd'] = x[8+rd'] - x[8+rs2']$

ลบ (*Subtract*). RV32IC และ RV64IC.

ขยายเป็น **sub** $rd, rd, rs2$, โดยที่ $rd=8+rd'$ และ $rs2=8+rs2'$

15	10	9	7	6	5	4	2	1	0
100011			rd'		00		rs2'		01

c.subw $rd', rs2'$ $x[8+rd'] = sext((x[8+rd'] - x[8+rs2'])[31:0])$

ลบเวิร์ด (*Subtract Word*). RV64IC เท่านั้น.

ขยายเป็น **subw** $rd, rd, rs2$, โดยที่ $rd=8+rd'$ และ $rs2=8+rs2'$

15	10	9	7	6	5	4	2	1	0
100111			rd'		00		rs2'		01

C.SW $rs2', uimm(rs1')$ $M[x[8+rs1'] + uimm][31:0] = x[8+rs2']$

บันทึกเวิร์ด (*Store Word*). RV32IC และ RV64IC.

ขยายเป็น **sw** $rs2, uimm(rs1)$, โดยที่ $rs2=8+rs2'$ และ $rs1=8+rs1'$

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) $M[x[2] + \text{uimm}][31:0] = x[\text{rs2}]$

บันทึกเวิร์ดสัมพันธ์กับตัวชี้สแตก (*Store Word, Stack-Pointer Relative*). RV32IC และ RV64IC.

ขยายเป็น **sw** rs2, uimm(x2)

15	13	12	7	6	2	1	0
110	uimm[5:2 7:6]			rs2		10	

c.xor rd', rs2' $x[8+\text{rd}'] = x[8+\text{rd}'] \wedge x[8+\text{rs2}']$ เอกซอร์-XOR (*Exclusive-OR*). RV32IC และ RV64IC.ขยายเป็น **xor** rd, rd, rs2, โดยที่ $\text{rd}=8+\text{rd}'$ และ $\text{rs2}=8+\text{rs2}'$

15	10	9	7	6	5	4	2	1	0
100011			rd'		01	rs2'		01	

call rd, symbol $x[\text{rd}] = \text{pc}+8; \text{pc} = \&\text{symbol}$ เรียกฟังก์ชัน (*Call*). คำสั่งเทียม, RV32I และ RV64I.เขียนตำแหน่งที่อยู่ของคำสั่งถัดไป ($\text{pc}+8$) ลงใน $x[\text{rd}]$ จากนั้นตั้งค่า pc เป็น symbol

ขยายเป็นคำสั่ง **auipc** rd, offsetHi แล้วตามด้วยคำสั่ง **jalc** rd, offsetLo(rd) ถ้าไม่ระบุ rd จะถือว่าใช้ $x1$ โดยปริยาย

csrc csr, rs1 $\text{CSRs}[\text{csr}] \&= \sim x[\text{rs1}]$

ล้างรีจิสเตอร์ควบคุมและสถานะ (*Control and Status Register Clear*). คำสั่งเทียม, RV32I และ RV64I.

สำหรับแต่ละบิตที่ตั้งค่าเป็น 1 ในรีจิสเตอร์ $x[\text{rs1}]$ ให้ล้างบิตที่สอดคล้องกันในรีจิสเตอร์ควบคุมและสถานะ (CSR) ขยายเป็น **csrc** x0, csr, rs1

csrci csr, zimm[4:0] $\text{CSRs}[\text{csr}] \&= \sim \text{zimm}$

ล้างรีจิสเตอร์ควบคุมและสถานะด้วยค่าทันที (*Control and Status Register Clear Immediate*). คำสั่งเทียม, RV32I และ RV64I.

สำหรับแต่ละบิตที่ถูกตั้งค่าเป็น 1 ในค่าทันทีขนาด 5 บิตที่ขยายศูนย์ (zero-extended) ให้ล้างบิตที่สอดคล้องกันในรีจิสเตอร์ควบคุมและสถานะ (CSR) ขยายเป็น **csrci** x0, csr, zimm

CSRR rd, csr $x[rd] = \text{CSRs}[csr]$

อ่านรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read). คำสั่งเทียม, RV32I และ RV64I.

คัดลอกรีจิสเตอร์ควบคุมและสถานะ (csr) ไปที่ $x[rd]$ ขยายเป็น **csrrs** rd, csr, x0

CSRRC rd, csr, rs1 $t = \text{CSRs}[csr]; \text{CSRs}[csr] = t \& \sim x[rs1]; x[rd] = t$

อ่านค่าและล้างบิตรีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Clear). รูปแบบ-I, RV32I และ RV64I.

ให้ t เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) เขียนผลลัพธ์ของการดำเนินการ AND แบบบิตต่อบิตระหว่าง t และส่วนเติมเต็มหนึ่งของรีจิสเตอร์ $x[rs1]$ ไปยัง csr แล้วเขียนค่า t ลงใน $x[rd]$

31	20	19	15	14	12	11	7	6	0	
csr			rs1		011		rd		1110011	

csrrci rd, csr, zimm[4:0] $t = \text{CSRs}[csr]; \text{CSRs}[csr] = t \& \sim \text{zimm};$

$x[rd] = t$

อ่านและล้างค่ารีจิสเตอร์ควบคุมและสถานะทันที (Control and Status Register Read and Clear Immediate). รูปแบบ-I, RV32I และ RV64I.

ให้ t เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) เขียนผลลัพธ์ของการดำเนินการ AND แบบบิตต่อบิตระหว่าง t และส่วนเติมเต็มหนึ่งของค่า $zimm$ (ค่าทันทีที่ขยายศูนย์ขนาด 5 บิต) ลงใน csr จากนั้นเขียนค่า t ลงในรีจิสเตอร์ $x[rd]$ (บิตที่ 5 ขึ้นไปใน csr จะไม่ถูกแก้ไข)

31	20	19	15	14	12	11	7	6	0	
csr			zimm[4:0]		111	rd		1110011		

csrrs rd, csr, rs1 $t = \text{CSRs}[csr]; \text{CSRs}[csr] = t \mid x[rs1]; x[rd] = t$

อ่านและตั้งค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Set). รูปแบบ-I, RV32I และ RV64I.

ให้ t เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) เขียนผลลัพธ์ของการดำเนินการ OR แบบบิตต่อบิตระหว่าง t และ $x[rs1]$ ลงใน csr จากนั้นเขียนค่า t ลงใน $x[rd]$

31	20	19	15	14	12	11	7	6	0	
csr				rs1		010	rd		1110011	

csrrsi $rd, csr, zimm[4:0]$ $t = CSRs[csr]; CSRs[csr] = t \mid zimm; x[rd] = t$
 อ่านและตั้งค่ารีจิสเตอร์ควบคุมและสถานะทันที (Control and Status Register Read and Set Immediate). รูปแบบ-I, RV32I และ RV64I.

ให้ t เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) เขียนผลลัพธ์ของการดำเนินการ OR แบบบิตต่อบิตระหว่าง t และค่า $zimm$ (ค่าทันทีแบบขยายศูนย์ขนาด 5 บิต) ลงใน csr จากนั้นเขียนค่า t ลงในรีจิสเตอร์ $x[rd]$ (บิตที่ 5 ขึ้นไปใน csr จะไม่ถูกแก้ไข)

31	20	19	15	14	12	11	7	6	0
csr					zimm[4:0]		110	rd	1110011

csrrw $rd, csr, rs1$ $t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t$
 อ่านและเขียนค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Read and Write). รูปแบบ-I, RV32I และ RV64I.

ให้ t เป็นค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) คัดลอก $x[rs1]$ ไปยัง csr จากนั้นเขียนค่า t ลงใน $x[rd]$.

31	20	19	15	14	12	11	7	6	0
csr					rs1		001	rd	1110011

csrrwi $rd, csr, zimm[4:0]$ $x[rd] = CSRs[csr]; CSRs[csr] = zimm$
 อ่านและเขียนรีจิสเตอร์ควบคุมและสถานะทันที (Control and Status Register Read and Write Immediate). รูปแบบ-I, RV32I และ RV64I.

คัดลอกค่าของรีจิสเตอร์ควบคุมและสถานะ (CSR) ไปยัง $x[rd]$ จากนั้นเขียนค่าทันที 5 บิตที่ขยายศูนย์ $zimm$ ไปยัง csr

31	20	19	15	14	12	11	7	6	0
csr					zimm[4:0]		101	rd	1110011

csrs $csr, rs1$ $CSRs[csr] \mid= x[rs1]$
 ตั้งค่ารีจิสเตอร์ควบคุมและสถานะ (Control and Status Register Set). คำสั่งเทียม, RV32I และ RV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในรีจิสเตอร์ $x[rs1]$ ให้ตั้งค่าบิตที่สอดคล้องในรีจิสเตอร์ควบคุมและสถานะ (CSR) เป็น 1 ขยายเป็น **csrrs** $x0, csr, rs1$

csrsi csr, zimm[4:0] CSRs[csr] |= zimm

ตั้งค่ารีจิสเตอร์ควบคุมและสถานะด้วยค่าทันที (*Control and Status Register Set Immediate*). คำสั่งเทียม, RV32I และRV64I.

สำหรับแต่ละบิตที่ตั้งค่า (set) ในค่าทันที 5 บิตที่ขยายศูนย์ ตั้งค่าบิตที่สอดคล้องในรีจิสเตอร์ตัวควบคุมและสถานะ (CSR) เป็น 1 ขยายเป็น **csrsi** x0, csr, zimm

csrw csr, rs1 CSRs[csr] = x[rs1]

เขียนรีจิสเตอร์ควบคุมและสถานะ (*Control and Status Register Write*). คำสั่งเทียม, RV32I และRV64I.

คัดลอก x[rs1] ไปยังรีจิสเตอร์ควบคุมและสถานะ (CSR) ขยายเป็น **csrw** x0, csr, rs1

csrwi csr, zimm[4:0] CSRs[csr] = zimm

เขียนค่ารีจิสเตอร์ควบคุมและสถานะด้วยค่าทันที (*Control and Status Register Write Immediate*). คำสั่งเทียม, RV32I และRV64I.

คัดลอกค่าทันทีที่ขยายศูนย์ขนาด 5 บิต ไปยังรีจิสเตอร์ควบคุมและสถานะ (csr) ขยายเป็น **csrwi** x0, csr, zimm

div rd, rs1, rs2 $x[rd] = x[rs1] \div_s x[rs2]$

การหาร (*Divide*). รูปแบบ-R, RV32M และ RV64M.

หารรีจิสเตอร์ x[rs1] ด้วยรีจิสเตอร์ x[rs2], ปิดผลลัพธ์ไปที่ศูนย์ โดยถือว่าค่าเป็น

จำนวนเต็มแบบส่วนเติมเต็มสองและเขียนผลหารลงในรีจิสเตอร์ x[rd]

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	100	rd	0110011	

divu rd, rs1, rs2 $x[rd] = x[rs1] \div_u x[rs2]$

หารแบบไม่มีเครื่องหมาย (*Divide, Unsigned*). รูปแบบ-R, RV32M และ RV64M.

หารค่ารีจิสเตอร์ x[rs1] ด้วยค่าในรีจิสเตอร์ x[rs2] โดยปิดผลลัพธ์ไปที่ศูนย์ โดยถือว่าค่า

เป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนผลหารลงในรีจิสเตอร์ x[rd]

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	101	rd	0110011	

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

หารเวิร์ดแบบไม่มีเครื่องหมาย (*Divide Word, Unsigned*). รูปแบบ-R, RV64M เท่านั้น.

หาร 32 บิตล่างของรีจิสเตอร์ $x[rs1]$ ด้วย 32 บิตล่างของรีจิสเตอร์ $x[rs2]$ โดยปัดผลลัพธ์ไปที่ศูนย์ โดยถือว่าค่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนผลหาร 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
00000001	rs2	rs1	101	rd	0111011	

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

หารเวิร์ด (*Divide Word*). รูปแบบ-R, RV64M เท่านั้น.

หาร 32 บิตล่างของรีจิสเตอร์ $x[rs1]$ ด้วย 32 บิตล่างของรีจิสเตอร์ ของรีจิสเตอร์ $x[rs2]$ โดยปัดผลลัพธ์ไปที่ศูนย์ โดยถือว่าค่าเป็นจำนวนเต็มแบบส่วนเติมเต็มสองและเขียนผลหาร 32 บิต ที่ขยายเครื่องหมาย (sign-extended) ลงใน $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
00000001	rs2	rs1	100	rd	0111011	

ebreak

RaiseException(Breakpoint)

จุดหยุดสภาพแวดล้อม (*Environment Breakpoint*). รูปแบบ-I, RV32I และ RV64I.

สร้างคำร้องขอไปยังดีบั๊กเกอร์ โดยการเรียกข้อยกเว้นหยุดสภาพแวดล้อม

31	20 19	15 14	12 11	7 6	0
00000000000001	00000	000	00000	1110011	

ecall

RaiseException(EnvironmentCall)

การเรียกสภาพแวดล้อม (*Environment Call*). รูปแบบ-I, RV32I และ RV64I.

สร้างคำร้องขอไปยังสภาพแวดล้อมการประมวลผล โดยการเรียกข้อยกเว้นการเรียกสภาพแวดล้อม

31	20 19	15 14	12 11	7 6	0
00000000000000	00000	000	00000	1110011	

fabs.d rd,rs1

$f[rd] = |f[rs1]|$

ค่าสัมบูรณ์จำนวนลอยตัว (*Floating-point Absolute Value*). คำสั่งเติม, RV32D และ RV64D.

เขียนค่าสัมบูรณ์ของจำนวนลอยตัวความแม่นยำคู่ในรีจิสเตอร์ $f[rs1]$ ลงในรีจิสเตอร์ $f[rd]$ ขยายเป็น **fsgnjx.d** rd, rs1, rs1

fabs.s rd, rs1 $f[rd] = |f[rs1]|$

ค่าสัมบูรณ์จำนวนลอยตัว (Floating-point Absolute Value). คำสั่งเทียม, RV32F และ RV64F.

เขียนค่าสัมบูรณ์ของจำนวนลอยตัวความแม่นยำเดียว ในรีจิสเตอร์ $f[rs1]$ ลงในรีจิสเตอร์ $f[rd]$ ขยายเป็น **fsgnjx.s** rd, rs1, rs1

fadd.d rd, rs1, rs2 $f[rd] = f[rs1] + f[rs2]$

บวกจำนวนลอยตัวความแม่นยำคู่ (Floating-point Add, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

บวกจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ และเขียนผลรวมแบบความแม่นยำคู่ที่ปัดผลลัพธ์แล้ว ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	rm	rd	1010011	

fadd.s rd, rs1, rs2 $f[rd] = f[rs1] + f[rs2]$

บวกจำนวนลอยตัวความแม่นยำเดียว (Floating-point Add, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

บวกจำนวนลอยตัวความแม่นยำเดียว ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ และเขียนผลรวมแบบความแม่นยำเดียวที่ปัดผลลัพธ์แล้ว ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	rm	rd	1010011	

fclass.d rd, rs1 $x[rd] = \text{classify}_d(f[rs1])$

จำแนกประเภทจำนวนลอยตัวความแม่นยำคู่ (Floating-point Classify, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

เขียนมาส์กลงในรีจิสเตอร์ $x[rd]$ เพื่อระบุประเภทของจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ โดยดูคำอธิบายของ **fclass.s** สำหรับการตีความมาส์กที่เขียนลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1110001	00000	rs1	001	rd	1010011	

fclass.s $rd, rs1$ $x[rd] = \text{classify}_s(f[rs1])$

จำแนกประเภทจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Classify, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียนมาสก์ลงในรีจิสเตอร์ $x[rd]$ เพื่อระบุประเภทของจำนวนลอยตัวแบบความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ โดยตั้งค่าเพียงบิตเดียวในรีจิสเตอร์ $x[rd]$ เป็น 1 ตามตารางต่อไปนี้:

$[rd]$ bit	ความหมาย
0	$f[rs1]$ ค่าลบ อินฟินิตี้ ($-\infty$)
1	$f[rs1]$ ค่าลบ จำนวนปกติ
2	$f[rs1]$ ค่าลบ จำนวนย่อยปกติ
3	$f[rs1]$ ค่าลบ ศูนย์ (-0)
4	$f[rs1]$ ค่าบวก ศูนย์ ($+0$)
5	$f[rs1]$ ค่าบวก จำนวนย่อยปกติ
6	$f[rs1]$ ค่าบวก จำนวนปกติ
7	$f[rs1]$ ค่าบวก อินฟินิตี้ ($+\infty$)
8	$f[rs1]$ NaN แบบส่งสัญญาณ
9	$f[rs1]$ NaN แบบเงียบ

31	25 24	20 19	15 14	12 11	7 6	0
1110000	00000	rs1	001	rd	1010011	

fcvt.d.l $rd, rs1$ $f[rd] = f64_{s64}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำคู่จากจำนวนเต็มยาว (to Double from Long). รูปแบบ-R, RV64D เท่านั้น.

แปลงจำนวนเต็ม 64 บิต ในรูปแบบส่วนเติมเต็มสองจากรีจิสเตอร์ $x[rs1]$ เป็นตัวจำนวนลอยตัวความแม่นยำคู่ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101001	00010	rs1	rm	rd	1010011	

fcvt.d.lu $rd, rs1$ $f[rd] = f64_{u64}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำคู่จากจำนวนเต็มยาวแบบไม่มีเครื่องหมาย

(Floating-point Convert to Double from Unsigned Long). รูปแบบ-R, RV64D เท่านั้น.

แปลงจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย จากรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวแบบความแม่นยำคู่ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101001	00011	rs1	rm	rd	1010011	

fcvt.d.s rd,rs1 $f[rd] = f64_{f32}(f[rs1])$

แปลงจำนวนลอยตัวความแม่นยำคู่จากความแม่นยำเดี่ยว (Floating-point Convert to Double from Single). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนลอยตัวแบบความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนลอยตัวแบบความแม่นยำคู่ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0100001	00000	rs1	rm	rd	1010011	

fcvt.d.w rd,rs1 $f[rd] = f64_{s32}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำคู่จากจำนวนเต็มเวิร์ด (Floating-point Convert to Double from Word). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนเต็ม 32 บิต ในรูปแบบส่วนเติมเต็มสองในรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวแบบความแม่นยำคู่ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101001	00000	rs1	rm	rd	1010011	

fcvt.d.wu rd,rs1 $f[rd] = f64_{u32}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำคู่จากจำนวนเต็มเวิร์ดแบบไม่มีเครื่องหมาย

(Floating-point Convert to Double from Unsigned Word). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนเต็ม 32 บิต แบบไม่มีเครื่องหมาย จากรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวความแม่นยำคู่ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101001	00001	rs1	rm	rd	1010011	

fcvt.l.d rd,rs1 $x[rd] = s64_{f64}(f[rs1])$

แปลงจำนวนเต็มยาวจากจำนวนลอยตัวความแม่นยำคู่ (Floating-point Convert to Long from Double). รูปแบบ-R, RV64D เท่านั้น.

แปลงตัวจำนวนลอยตัวความแม่นยำคู่ จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 64 บิตในรูปแบบส่วนเติมเต็มสอง และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100001	00010	rs1	rm	rd	1010011	

fcvt.l.s rd, rs1 $x[rd] = s64_{f32}(f[rs1])$

แปลงจำนวนเต็มยาวจากจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Convert to Long from Single). รูปแบบ-R, RV64F เท่านั้น.

แปลงจำนวนลอยตัวแบบความแม่นยำเดี่ยว จากระจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 64 บิตในรูปแบบส่วนเติมเต็มสอง และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100000	00010	rs1	rm	rd		1010011

fcvt.lu.d rd, rs1 $x[rd] = u64_{f64}(f[rs1])$

แปลงจำนวนเต็มยาวแบบไม่มีเครื่องหมายจากจำนวนลอยตัวความแม่นยำคู่ (Floating-point Convert to Unsigned Long from Double). รูปแบบ-R, RV64D เท่านั้น.

แปลงจำนวนลอยตัวความแม่นยำคู่ จากระจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100001	00011	rs1	rm	rd		1010011

fcvt.lu.s rd, rs1 $x[rd] = u64_{f32}(f[rs1])$

แปลงจำนวนเต็มยาวแบบไม่มีเครื่องหมายจากจำนวนลอยตัวความแม่นยำเดี่ยว

(Floating-point Convert to Unsigned Long from Single). รูปแบบ-R, RV64F เท่านั้น.

แปลงจำนวนลอยตัวความแม่นยำเดี่ยว จากระจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100000	00011	rs1	rm	rd		1010011

fcvt.s.d rd, rs1 $f[rd] = f32_{f64}(f[rs1])$

แปลงจำนวนลอยตัวความแม่นยำเดี่ยวจากจำนวนลอยตัวความแม่นยำคู่ (Floating-point Convert to Single from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนลอยตัวความแม่นยำคู่ จากระจิสเตอร์ $f[rs1]$ เป็นจำนวนลอยตัวความแม่นยำเดี่ยว และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0100000	00001	rs1	rm	rd		1010011

fcvt.s.l rd, rs1 $f[rd] = f32_{s64}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำเดี่ยวจากจำนวนเต็มยาว (Floating-point Convert to Single from Long). รูปแบบ-R, RV64F เท่านั้น.

แปลงจำนวนเต็ม 64 บิตในรูปแบบ ส่วนเต็มเต็มสองจากรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวความแม่นยำเดี่ยว และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00010	rs1	rm	rd	1010011	

fcvt.s.lu rd, rs1 $f[rd] = f32_{u64}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำเดี่ยวจากจำนวนเต็มยาวแบบไม่มีเครื่องหมาย

(Floating-point Convert to Single from Unsigned Long). รูปแบบ-R, RV64F เท่านั้น.

แปลงจำนวนเต็ม 64 บิตแบบไม่มีเครื่องหมาย จากรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวความแม่นยำเดี่ยว และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00011	rs1	rm	rd	1010011	

fcvt.s.w rd, rs1 $f[rd] = f32_{s32}(x[rs1])$

แปลงจำนวนลอยตัวเป็นความแม่นยำเดี่ยวจากเวิร์ด (Floating-point Convert to Single from Word). รูปแบบ-R, RV32F และ RV64F.

แปลงจำนวนเต็ม 32 บิตในรูปแบบส่วนเต็มเต็มสองจากรีจิสเตอร์ $x[rs1]$ เป็นจำนวนลอยตัวความแม่นยำเดี่ยว และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00000	rs1	rm	rd	1010011	

fcvt.s.wu rd, rs1 $f[rd] = f32_{u32}(x[rs1])$

แปลงจำนวนลอยตัวความแม่นยำเดี่ยวจากเวิร์ดแบบไม่มีเครื่องหมาย (Floating-point Convert to Single from Unsigned Word). รูปแบบ-R, RV32F และ RV64F.

แปลงจำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย ใน $x[rs1]$ เป็นจำนวนลอยตัวความแม่นยำเดี่ยว และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1101000	00001	rs1	rm	rd	1010011	

fcvt.w.d rd, rs1 $x[rd] = \text{sext}(s32_{f64}(f[rs1]))$

แปลงจำนวนเต็มเวิร์ดจากจำนวนลอยตัวความแม่นยำคู่ (Floating-point Convert to Word from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนลอยตัวแบบความแม่นยำคู่ จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 32 บิต ในรูปแบบส่วนเติมเต็มสองและเขียนผลลัพธ์ที่ขยายเครื่องหมายลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100001	00000	rs1	rm	rd		1010011

fcvt.wu.d rd, rs1 $x[rd] = \text{sext}(u32_{f64}(f[rs1]))$

แปลงจำนวนเต็มเวิร์ดแบบไม่มีเครื่องหมายจากจำนวนลอยตัวความแม่นยำคู่ (Floating-point Convert to Unsigned Word from Double). รูปแบบ-R, RV32D และ RV64D.

แปลงจำนวนลอยตัวแบบความแม่นยำคู่ จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ที่ขยายเครื่องหมายลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100001	00001	rs1	rm	rd		1010011

fcvt.w.s rd, rs1 $x[rd] = \text{sext}(s32_{f32}(f[rs1]))$

แปลงจำนวนเต็มเวิร์ดจากจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Convert to Word from Single). รูปแบบ-R, RV32F และ RV64F.

แปลงจำนวนลอยตัวความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 32 บิตในรูปแบบส่วนเติมเต็มสองและเขียนผลลัพธ์ที่ขยายเครื่องหมายลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100000	00000	rs1	rm	rd		1010011

fcvt.wu.s rd, rs1 $x[rd] = \text{sext}(u32_{f32}(f[rs1]))$

แปลงจำนวนเต็มเวิร์ดเวิร์ดแบบไม่มีเครื่องหมายจากจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Convert to Unsigned Word from Single). รูปแบบ-R, RV32F และ RV64F.

แปลงจำนวนลอยตัวแบบความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs1]$ เป็นจำนวนเต็ม 32 บิตแบบไม่มีเครื่องหมาย และเขียนผลลัพธ์ที่ขยายเครื่องหมาย ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1100000	00001	rs1	rm	rd		1010011

fdiv.d rd, rs1, rs2 $f[rd] = f[rs1] \div f[rs2]$

หารจำนวนลอยตัวความแม่นยำคู่ (Floating-point Divide, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

หารจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ ด้วย $f[rs2]$ และเขียนผลหารแบบความแม่นยำคู่ที่ปัดผลลัพธ์แล้วลงใน รีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0001101	rs2	rs1	rm	rd	1010011	

fdiv.s rd, rs1, rs2 $f[rd] = f[rs1] \div f[rs2]$

หารจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Divide, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

หารจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ ด้วย $f[rs2]$ และเขียนผลหารแบบความแม่นยำเดี่ยวที่ปัดผลลัพธ์แล้วลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0001100	rs2	rs1	rm	rd	1010011	

fence pred, succ Fence(pred, succ)

กำหนดลำดับการทำงานและการเข้าถึงหน่วยความจำและ I/O (Fence Memory and I/O). รูปแบบ-I, RV32I และ RV64I.

ทำให้การเข้าถึงหน่วยความจำและ I/O ก่อนหน้านั้นในชุด predecessor สามารถมองเห็นได้โดยเรดอื่นหรืออุปกรณ์อื่น ก่อนที่การเข้าถึงในชุด successor จะมองเห็นได้ บิต 3, 2, 1 และ 0 ในชุดเหล่านี้สอดคล้องกับ: การรับข้อมูลจากอุปกรณ์ (input), การส่งข้อมูลไปยังอุปกรณ์ (output), การอ่านหน่วยความจำ (reads) และการเขียนหน่วยความจำ (writes) ตามลำดับ ตัวอย่างเช่น คำสั่ง **fence r, rw** จัดลำดับการอ่านที่เก่ากว่า กับการอ่านและการเขียนที่ใหม่กว่า โดยเข้ารหัสด้วย: $pred=0010$ (หมายถึงควบคุมการอ่าน) และ $succ=0011$ (หมายถึงควบคุมการอ่านและการเขียน) หากไม่ระบุอาร์กิวเมนต์ จะถือว่าเป็น **fence iorw, iorw** แบบเต็มรูปแบบ

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)

ควบคุมลำดับของคำสั่งที่ถูกดึงจากหน่วยความจำ (Fence Instruction Stream). รูปแบบ-I, RV32I และ RV64I.

ทำให้การเขียนข้อมูลลงหน่วยความจำคำสั่ง (instruction memory) สามารถมองเห็นได้ (observable) สำหรับการดึงคำสั่ง (instruction fetches) ที่ตามมา

31	20	19	15	14	12	11	7	6	0		
000000000000					00000		001	00000		0001111	

feq.d rd, rs1, rs2 $x[rd] = f[rs1] == f[rs2]$

เปรียบเทียบการเท่ากันของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Equals, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนลอยตัวความแม่นยำคู่ในรีจิสเตอร์ $f[rs1]$ เท่ากับจำนวนลอยตัวในรีจิสเตอร์ $f[rs2]$ และเขียนค่า 0 ถ้าไม่เท่ากัน

31	25	24	20	19	15	14	12	11	7	6	0	
1010001			rs2		rs1		010		rd		1010011	

feq.s rd, rs1, rs2 $x[rd] = f[rs1] == f[rs2]$

เปรียบเทียบการเท่ากันของจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Equals, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนจำนวนลอยตัวความแม่นยำเดี่ยวในรีจิสเตอร์ $f[rs1]$ เท่ากับจำนวนลอยตัว ในรีจิสเตอร์ $f[rs2]$ และเขียน 0 ถ้าไม่เท่ากัน

31	25	24	20	19	15	14	12	11	7	6	0	
1010000			rs2		rs1		010		rd		1010011	

fld rd, offset(rs1) $f[rd] = M[x[rs1] + sext(offset)][63:0]$

โหลดจำนวนลอยตัวดับเบิลเวิร์ด (Floating-point Load Doubleword). รูปแบบ-I, RV32D และ RV64D.

โหลดจำนวนลอยตัวแบบความแม่นยำคู่ จากตำแหน่งที่อยู่หน่วยความจำ $x[rs1] + (offset)$ ที่ขยายเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$.

คำสั่งบีบอัด: **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 $x[rd] = f[rs1] \leq f[rs2]$

เปรียบเทียบการน้อยกว่าหรือเท่ากับของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Less Than or Equal, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ มีค่าน้อยกว่าหรือเท่ากับจำนวนลอยตัวในรีจิสเตอร์ $f[rs2]$ และเขียนค่า 0 ถ้าไม่ใช่

31	25 24	20 19	15 14	12 11	7 6	0
1010001	rs2	rs1	000	rd	1010011	

fle.s rd, rs1, rs2 $x[rd] = f[rs1] \leq f[rs2]$

เปรียบเทียบการน้อยกว่าหรือเท่ากับของจำนวนลอยตัวความแม่นยำเดี่ยว

(Floating-point Less Than or Equal, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ มีค่าน้อยกว่าหรือเท่ากับจำนวนลอยตัวในรีจิสเตอร์ $f[rs2]$ และเขียนค่า 0 ถ้าไม่ใช่

31	25 24	20 19	15 14	12 11	7 6	0
1010000	rs2	rs1	000	rd	1010011	

flt.d rd, rs1, rs2 $x[rd] = f[rs1] < f[rs2]$

เปรียบเทียบการน้อยกว่าของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Less Than, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ มีค่าน้อยกว่าจำนวนลอยตัวในรีจิสเตอร์ $f[rs2]$ และเขียนค่า 0 ถ้าไม่ใช่

31	25 24	20 19	15 14	12 11	7 6	0
1010001	rs2	rs1	001	rd	1010011	

flt.s rd, rs1, rs2 $x[rd] = f[rs1] < f[rs2]$

เปรียบเทียบการน้อยกว่าของจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Less Than, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ มีค่าน้อยกว่าจำนวนลอยตัวในรีจิสเตอร์ $f[rs2]$ และเขียนค่า 0 ถ้าไม่ใช่

31	25 24	20 19	15 14	12 11	7 6	0
1010000	rs2	rs1	001	rd	1010011	

flw rd, offset(rs1) $f[rd] = M[x[rs1] + sext(offset)][31:0]$

โหลดเวิร์ดจำนวนลอยตัว (Floating-point Load Word). รูปแบบ-I, RV32F และ RV64F.

โหลดจำนวนลอยตัวความแม่นยำเดี่ยว จากตำแหน่งที่อยู่หน่วยความจำ $x[rs1] + (offset)$ ที่ขยายเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$.

คำสั่งบีบอัด: **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 $f[rd] = f[rs1] \times f[rs2] + f[rs3]$

คูณและบวกจำนวนลอยตัวความแม่นยำคู่แบบผสม (Floating-point Fused Multiply-Add, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.

คูณจำนวนลอยตัวความแม่นยำคู่ใน $f[rs1]$ กับ $f[rs2]$ บวกผลคูณที่ยังไม่ปัดเข้ากับจำนวนลอยตัวความแม่นยำคู่ใน $f[rs3]$ และเขียนผลลัพธ์แบบความแม่นยำคู่ที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

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 $f[rd] = f[rs1] \times f[rs2] + f[rs3]$

คูณและบวกจำนวนลอยตัวความแม่นยำเดี่ยวแบบผสม (Floating-point Fused Multiply-Add, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.

คูณจำนวนลอยตัวความแม่นยำเดี่ยวใน $f[rs1]$ กับ $f[rs2]$ บวกผลคูณที่ยังไม่ปัดเข้ากับจำนวนลอยตัวความแม่นยำเดี่ยวใน $f[rs3]$ และเขียนผลลัพธ์แบบความแม่นยำเดี่ยวที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

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 $f[rd] = \max(f[rs1], f[rs2])$

ค่าสูงสุดของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Maximum, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

คัดลอกจำนวนลอยตัวความแม่นยำคู่ ที่มีค่ามากกว่าระหว่างรีจิสเตอร์ $f[rs1]$ และ $f[rs2]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25	24	20	19	15	14	12	11	7	6	0	
0010101			rs2		rs1		001		rd		1010011	

fmax.s rd, rs1, rs2 $f[rd] = \max(f[rs1], f[rs2])$

ค่าสูงสุดของจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Maximum, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนลอยตัวความแม่นยำเดี่ยวที่มีค่ามากกว่า ระหว่างรีจิสเตอร์ $f[rs1]$ และ $f[rs2]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25	24	20	19	15	14	12	11	7	6	0
0010100	rs2			rs1			001	rd		1010011	

fmin.d rd, rs1, rs2 $f[rd] = \min(f[rs1], f[rs2])$

ค่าต่ำสุดของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Minimum, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

คัดลอกจำนวนลอยตัวความแม่นยำคู่ที่มีค่าน้อยกว่า ระหว่างรีจิสเตอร์ $f[rs1]$ และ $f[rs2]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25	24	20	19	15	14	12	11	7	6	0
0010101	rs2			rs1			000	rd		1010011	

fmin.s rd, rs1, rs2 $f[rd] = \min(f[rs1], f[rs2])$

ค่าต่ำสุดของจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Minimum, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนลอยตัวความแม่นยำเดี่ยวที่มีค่าน้อยกว่า ระหว่างรีจิสเตอร์ $f[rs1]$ และ $f[rs2]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25	24	20	19	15	14	12	11	7	6	0
0010100	rs2			rs1			000	rd		1010011	

fmsub.d rd, rs1, rs2, rs3 $f[rd] = f[rs1] \times f[rs2] - f[rs3]$

คูณและลบจำนวนลอยตัวความแม่นยำคู่แบบผสม (Floating-point Fused Multiply-Subtract, Double-Precision). รูปแบบ-R4, RV32D และ RV64D.

คูณจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ ลบจำนวนลอยตัวแบบความแม่นยำคู่ใน $f[rs3]$ ออกจากผลคูณที่ยังไม่ปัด และเขียนผลลัพธ์ที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

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 $f[rd] = f[rs1] \times f[rs2] - f[rs3]$

คูณและลบจำนวนลอยตัวความแม่นยำเดี่ยวแบบผสม (Floating-point Fused Multiply-Subtract, Single-Precision). รูปแบบ-R4, RV32F และ RV64F.

คูณจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ ลบจำนวนลอยตัวความแม่นยำเดี่ยวใน $f[rs3]$ ออกจากผลคูณที่ยังไม่ปัด และเขียนผลลัพธ์แบบความแม่นยำเดี่ยวที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

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 $f[rd] = f[rs1] \times f[rs2]$

คูณจำนวนลอยตัวความแม่นยำคู่ (Floating-point Multiply, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

คูณจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ และเขียนผลคูณที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0001001	rs2	rs1	rm	rd	1010011	

fmul.s rd, rs1, rs2 $f[rd] = f[rs1] \times f[rs2]$

คูณจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Multiply, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

คูณจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ และเขียนผลคูณที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0001000	rs2	rs1	rm	rd	1010011	

fmv.d rd, rs1 $f[rd] = f[rs1]$

ย้ายจำนวนลอยตัว (Floating-point Move). คำสั่งเทียม, RV32D และ RV64D.

คัดลอกจำนวนลอยตัวความแม่นยำคู่ จากรีจิสเตอร์ $f[rs1]$ ไปยังรีจิสเตอร์ $f[rd]$ ขยายเป็น

fsgnj.d rd, rs1, rs1

fmv.d.x rd, rs1 $f[rd] = x[rs1][63:0]$

ย้ายจำนวนลอยตัวความแม่นยำคู่ (Floating-point Move Doubleword from Integer). รูปแบบ-R, RV64D เท่านั้น.

คัดลอกจำนวนลอยตัวความแม่นยำคู่ จากรีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1111001	00000	rs1	000	rd	1010011	

fmv.s rd, rs1 $f[rd] = f[rs1]$

ย้ายจำนวนลอยตัว (Floating-point Move). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกจำนวนลอยตัวความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs1]$ ไปยังรีจิสเตอร์ $f[rd]$ ขยายเป็น **fsgnj.d** rd, rs1, rs1

fmv.w.x rd, rs1 $f[rd] = x[rs1][31:0]$

ย้ายเวิร์ดจำนวนลอยตัวจากจำนวนเต็ม (Floating-point Move Word from Integer).

รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนลอยตัวความแม่นยำเดี่ยว จากรีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1111000	00000	rs1	000	rd	1010011	

fmv.x.d rd, rs1 $x[rd] = f[rs1][63:0]$

ย้ายดับเบิลเวิร์ดจำนวนลอยตัวไปยังจำนวนเต็ม (Floating-point Move Doubleword to Integer). รูปแบบ-R, RV64D เท่านั้น.

คัดลอกจำนวนลอยตัวแบบความแม่นยำคู่จากรีจิสเตอร์ $f[rs1]$ ไปยังรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
1110001	00000	rs1	000	rd	1010011	

fmv.x.w rd, rs1 $x[rd] = \text{sext}(f[rs1][31:0])$

ย้ายเวิร์ดจำนวนลอยตัวไปยังจำนวนเต็ม (Floating-point Move Word to Integer). รูปแบบ-R, RV32F และ RV64F.

คัดลอกจำนวนลอยตัวแบบความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs1]$ ไปยัง $x[rd]$ โดยใน RV64F จะขยายเครื่องหมายของผลลัพธ์ให้เป็น 64 บิต

31	25 24	20 19	15 14	12 11	7 6	0
1110000	00000	rs1	000	rd	1010011	

fneg.d rd, rs1 $f[rd] = -f[rs1]$

กลับเครื่องหมายจำนวนลอยตัว (Floating-point Negate). คำสั่งเทียม, RV32D และ RV64D.

เขียนค่าจำนวนลอยตัวความแม่นยำคู่ ที่มีเครื่องหมายตรงข้ามจากรีจิสเตอร์ $f[rs1]$ ลงในรีจิสเตอร์ $f[rd]$ ขยายเป็น **fsgnjn.d** rd, rs1, rs1

$$f[rd] = -f[rs1]$$

เขียนค่าจำนวนลอยตัวแบบความแม่นยำเดียว ที่มีเครื่องหมายตรงข้ามจากรีจิสเตอร์ `f[rs1]`
ลงในรีจิสเตอร์ `f[rd]` ขยายเป็น **fsnjin.s** rd, rs1, rs1

$$f[rd] = -f[rs1] \times f[rs2] - f[rs3]$$

คุณเชิงลบและบวกจำนวนลอยตัวความแม่นยำคู่แบบผสม (Floating-point Fused Negative Multiply-Add, Double-Precision). รูปแบบ-R4, RV32D และ RV64D. คุณจำนวนลอยตัวแบบความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ กลับเครื่องหมายของ ผลคูณ ลบจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs3]$ ออกจากผลคูณที่ยังไม่ปิด และ เขียนผลลัพธ์แบบความแม่นยำคู่ที่ปิดแล้วลงใน $f[rd]$

rs3		01	rs2		rs1		rm		rd		1001111	
-----	--	----	-----	--	-----	--	----	--	----	--	---------	--

$$f[rd] = -f[rs1] \times f[rs2] - f[rs3]$$

คุณเชิงลบและบวกจำนวนลอยตัวความแม่นยำเดี่ยวแบบผสม (Floating-point Fused Negative Multiply-Add, Single-Precision). รูปแบบ-R4, RV32F และ RV64F. คุณจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ กลับเครื่องหมายของผลคูณ ลบจำนวนลอยตัวความแม่นยำเดี่ยวในรีจิสเตอร์ $f[rs3]$ ออกจากผลคูณที่ยังไม่ถูกปิด และเขียนผลลัพธ์แบบความแม่นยำเดี่ยวที่ปิดแล้วลงใน $f[rd]$

31	27	26	25	24	20	19	15	14	12	11	7	6	0
rs3				00	rs2			rs1		rm	rd		1001111

```
f[rd] = -f[rs1]*f[rs2]+f[rs3]
```

คุณเชิงลบและลบจำนวนลอยตัวความแม่นยำคู่แบบผสม (Floating-point Fused Negative Multiply-Subtract, Double-Precision). รูปแบบ-R4, RV32D และ RV64D. คุณจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ กลับเครื่องหมายของผล คุณ บวกผลคุณที่ยังไม่ปิดเข้ากับจำนวนลอยตัวความแม่นยำคู่ใน $f[rs3]$ และเขียนผลลัพธ์ แบบความแม่นยำคู่ที่ปิดแล้วลงใน $f[rd]$

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 $f[rd] = -f[rs1] \times f[rs2] + f[rs3]$

คูณเชิงลบและลบจำนวนลอยตัวความแม่นยำเดี่ยวแบบผสม (Floating-point Fused Negative Multiply-Subtract, Single-Precision). รูปแบบ-R4, RV32F และ RV64F. คูณจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ กับ $f[rs2]$ กลับเครื่องหมายของ ผลคูณ บวกผลคูณที่ยังไม่ปิดเข้ากับจำนวนลอยตัวความแม่นยำเดี่ยวใน $f[rs3]$ และเขียน ผลลัพธ์แบบความแม่นยำเดี่ยวที่ปิดแล้วลงใน $f[rd]$

31	27	26	25	24	20	19	15	14	12	11	7	6	0
rs3	00				rs2		rs1		rm		rd		1001011

frcsr rd $x[rd] = \text{CSRs}[fcsr]$

อ่านรีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว (Floating-point Read Control and Status Register). คำสั่งเพิ่มเติม, RV32F และ RV64F.

คัดลอกค่ารีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว ไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น

csrrs rd, fcsr, x0

frflags rd $x[rd] = \text{CSRs}[fflags]$

อ่านแฟล็กข้อยกเว้นจำนวนลอยตัว (Floating-point Read Exception Flags). คำสั่งเพิ่มเติม, RV32F และ RV64F.

คัดลอกแฟล็กข้อยกเว้นจำนวนลอยตัวไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrs** rd, fflags, x0

frrm rd $x[rd] = \text{CSRs}[frm]$

อ่านโหมดการปัดจำนวนลอยตัว (Floating-point Read Rounding Mode). คำสั่งเพิ่มเติม, RV32F และ RV64F.

คัดลอกโหมดการปัดจำนวนลอยตัวไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrs** rd, frm, x0

fscsr rd, rs1 $t = \text{CSRs}[fcsr]; \text{CSRs}[fcsr] = x[rs1]; x[rd] = t$

สลับรีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว (Floating-point Swap Control and Status Register). คำสั่งเพิ่มเติม, RV32F และ RV64F.

คัดลอกรีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว จากนั้นคัดลอกค่าก่อนหน้าของรีจิสเตอร์ควบคุมและสถานะจำนวนลอยตัว ไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrw** rd, fcsr, rs1 ถ้าไม่ระบุ rd จะถือว่าใช้ x0

fsd rs2, offset(rs1) $M[x[rs1] + sext(offset)] = f[rs2][63:0]$

บันทึกดับเบิลเวิร์ดจำนวนลอยตัว (Floating-point Store Doubleword). รูปแบบ-S, RV32D และ RV64D.

บันทึกจำนวนลอยตัวความแม่นยำคู่ จากรีจิสเตอร์ $f[rs2]$ ลงในหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (offset)$ ที่ขยายเครื่องหมาย

คำสั่งบีบอัด: **c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)

31	25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100111	

fsflags rd, rs1 $t = CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd] = t$

สลับแฟล็กข้อยกเว้นจำนวนลอยตัว (Floating-point Swap Exception Flags). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกค่ารีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์แฟล็กข้อยกเว้นจำนวนลอยตัว จากนั้นคัดลอกค่าแฟล็กข้อยกเว้นจำนวนลอยตัวก่อนหน้าไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrw** rd, fflags, rs1 ถ้าไม่ระบุ rd ถูกละไว้ จะถือว่าใช้ x0 โดยปริยาย

fsgnj.d rd, rs1, rs2 $f[rd] = \{f[rs2][63], f[rs1][62:0]\}$

แทรกเครื่องหมายจำนวนลอยตัวความแม่นยำคู่ (Floating-point Sign Inject, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

สร้างจำนวนลอยตัวแบบความแม่นยำคู่ใหม่ โดยใช้เลขชี้กำลังและและส่วนจำนวนลอยตัวจากรีจิสเตอร์ $f[rs1]$ และใช้เครื่องหมาย (sign) จากรีจิสเตอร์ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0010001	rs2	rs1	000	rd	1010011	

fsgnj.s rd, rs1, rs2 $f[rd] = \{f[rs2][31], f[rs1][30:0]\}$

แทรกเครื่องหมายจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Sign Inject, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

สร้างจำนวนลอยตัวแบบความแม่นยำเดี่ยวใหม่ จากเลขชี้กำลังและและส่วนจำนวนลอยตัว (significand) จากรีจิสเตอร์ $f[rs1]$ และใช้เครื่องหมาย (sign) จากรีจิสเตอร์ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0010000	rs2	rs1	000	rd	1010011	

fsgnjn.d rd,rs1,rs2 $f[rd] = \{\sim f[rs2][63], f[rs1][62:0]\}$

แทรกเครื่องหมายเชิงลบจำนวนลอยตัวความแม่นยำคู่ (Floating-point Sign Inject-Negate, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

สร้างจำนวนลอยตัวความแม่นยำคู่ใหม่ จากเลขชี้กำลังและส่วนจำนวนลอยตัว (significand) จากรีจิสเตอร์ $f[rs1]$ โดยใช้เครื่องหมายที่ตรงข้ามกับเครื่องหมายของรีจิสเตอร์ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25	24	20	19	15	14	12	11	7	6	0
0010001			rs2		rs1		001	rd		1010011	

fsgnjn.s rd,rs1,rs2 $f[rd] = \{\sim f[rs2][31], f[rs1][30:0]\}$

แทรกเครื่องหมายเชิงลบจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Sign Inject-Negate, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

สร้างจำนวนลอยตัวแบบความแม่นยำเดี่ยวใหม่ จากเลขชี้กำลังและส่วนจำนวนลอยตัว (significand) จากรีจิสเตอร์ $f[rs1]$ โดยใช้เครื่องหมาย (sign) ที่ตรงข้ามกับเครื่องหมายของรีจิสเตอร์ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$.

31	25	24	20	19	15	14	12	11	7	6	0
0010000			rs2		rs1		001	rd		1010011	

fsgnjx.d rd,rs1,rs2 $f[rd] = \{f[rs1][63] \wedge f[rs2][63], f[rs1][62:0]\}$

แทรกเครื่องหมายด้วย XOR จำนวนลอยตัวความแม่นยำคู่ (Floating-point Sign Inject-XOR, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

สร้างจำนวนลอยตัวความแม่นยำคู่ใหม่ จากเลขชี้กำลังและส่วนจำนวนลอยตัว (significand) จากรีจิสเตอร์ $f[rs1]$ และใช้เครื่องหมาย (sign) ที่ได้จากการดำเนินการ XOR ระหว่างเครื่องหมายของ $f[rs1]$ และ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0010001	rs2	rs1	010	rd	1010011	

fsgnjx.s rd, rs1, rs2 $f[rd] = \{f[rs1][31] \wedge f[rs2][31], f[rs1][30:0]\}$

แทรกเครื่องหมายด้วย XOR จำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Sign Inject-XOR, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

สร้างจำนวนลอยตัวความแม่นยำเดี่ยวใหม่ จากเลขชี้กำลังและส่วนจำนวนลอยตัว (significand) จากรีจิสเตอร์ $f[rs1]$ และใช้เครื่องหมาย (sign) ที่ได้จากการดำเนินการ XOR ระหว่างเครื่องหมายของ $f[rs1]$ และ $f[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0010000	rs2	rs1	010	rd	1010011	

fsqrt.d rd, rs1 $f[rd] = \sqrt{f[rs1]}$

รากที่สองของจำนวนลอยตัวความแม่นยำคู่ (Floating-point Square Root, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

คำนวณรากที่สองของจำนวนลอยตัวแบบความแม่นยำคู่ ในรีจิสเตอร์ $f[rs1]$ และเขียนผลลัพธ์ความแม่นยำคู่ที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0101101	00000	rs1	rm	rd	1010011	

fsqrt.s rd, rs1 $f[rd] = \sqrt{f[rs1]}$

รากที่สองจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Square Root, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

คำนวณรากที่สองของจำนวนลอยตัวแบบความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs1]$ และเขียนผลลัพธ์ความแม่นยำเดี่ยวที่ปัดแล้วลงใน $f[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0101100	00000	rs1	rm	rd	1010011	

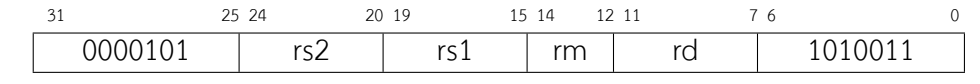
fsrm rd, rs1 $t = \text{CSR}[frm]; \text{CSR}[frm] = x[rs1]; x[rd] = t$
สลับโหมดการปัดจำนวนลอยตัว (Floating-point Swap Rounding Mode). คำสั่งเทียม, RV32F และ RV64F.

คัดลอกค่าจากรีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์โหมดการปัดจำนวนลอยตัว จากนั้นคัดลอกค่าโหมดการปัดจำนวนลอยตัวก่อนหน้าไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrw** rd, frm, rs1 ถ้าไม่ระบุ rd จะถือว่าใช้ x0 โดยปริยาย

fsub.d rd, rs1, rs2 $f[rd] = f[rs1] - f[rs2]$

ลบจำนวนลอยตัวความแม่นยำคู่ (Floating-point Subtract, Double-Precision). รูปแบบ-R, RV32D และ RV64D.

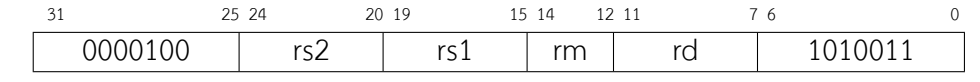
ลบจำนวนลอยตัวความแม่นยำคู่ ในรีจิสเตอร์ $f[rs2]$ ออกจาก $f[rs1]$ และเขียนผลลัพธ์ของผลต่างความแม่นยำคู่ที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$



fsub.s rd, rs1, rs2 $f[rd] = f[rs1] - f[rs2]$

ลบจำนวนลอยตัวความแม่นยำเดี่ยว (Floating-point Subtract, Single-Precision). รูปแบบ-R, RV32F และ RV64F.

ลบจำนวนลอยตัวความแม่นยำเดี่ยว ในรีจิสเตอร์ $f[rs2]$ ออกจาก $f[rs1]$ และเขียนผลลัพธ์ของผลต่างความแม่นยำเดี่ยวที่ปัดแล้วลงในรีจิสเตอร์ $f[rd]$

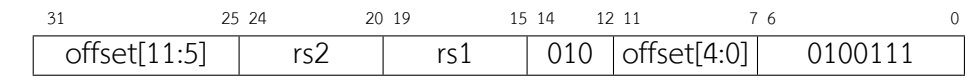


fsw rs2, offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = f[rs2][31:0]$

บันทึกเวิร์ดจำนวนลอยตัว (Floating-point Store Word). รูปแบบ-S, RV32F และ RV64F.

บันทึกจำนวนลอยตัวความแม่นยำเดี่ยว จากรีจิสเตอร์ $f[rs2]$ ลงในหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย

คำสั่งบีบอัด: **c.fswsp** rs2, offset; **c.fsw** rs2, offset(rs1)



j offset $pc += \text{sext}(\text{offset})$

กระโดด (Jump). คำสั่งเทียม, RV32I และ RV64I.

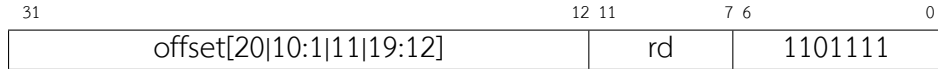
ตั้งค่า pc (ตัวนับโปรแกรม) เป็น pc ปัจจุบัน บวกกับ $offset$ ที่ขยายเครื่องหมาย (sign-extended) ขยายเป็น **j** **jal** x0, offset

jal rd, offset $x[rd] = pc+4; pc += sext(offset)$

การกระโดดและเชื่อมโยง (Jump and Link). รูปแบบ-J, RV32I และ RV64I.

เขียนที่อยู่ของคำสั่งถัดไป ($pc+4$) ลงในรีจิสเตอร์ $x[rd]$ จากนั้นตั้งค่า pc (ตัวนับโปรแกรม) เป็น pc ปัจจุบันบวกกับ $offset$ ที่ขยายเครื่องหมาย (sign-extended) ถ้าไม่ระบุ rd จะถือว่าใช้ $x1$ โดยปริยาย

คำสั่งบีบอัด: **c.j** offset; **c.jal** offset

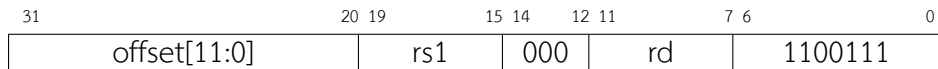


jalr rd, offset(rs1) $t=pc+4; pc=(x[rs1]+sext(offset))\&\sim 1; x[rd]=t$

กระโดดและเชื่อมโยงผ่านรีจิสเตอร์ (Jump and Link Register). รูปแบบ-I, RV32I และ RV64I.

ตั้งค่า pc (ตัวนับโปรแกรม) เป็นผลรวมของ $x[rs1]+(offset)$ ที่ขยายเครื่องหมาย (sign-extended) โดยตัดบิตที่มีนัยสำคัญน้อยที่สุด (least-significant bit) ของตำแหน่งที่อยู่ที่คำนวณได้ออก จากนั้นเขียนค่า $pc+4$ ก่อนหน้า ลงในรีจิสเตอร์ $x[rd]$ ถ้าไม่ระบุ rd จะถือว่าใช้ $x1$ โดยปริยาย

คำสั่งบีบอัด: **c.jr** rs1; **c.jalr** rs1



jr rs1 $pc = x[rs1]$

กระโดดผ่านเรจิสเตอร์ (Jump Register). คำสั่งเทียม, RV32I และ RV64I.

ตั้งค่า pc เป็น $x[rs1]$ ขยายเป็น **jalr** $x0, 0(rs1)$

la rd, symbol $x[rd] = \&symbol$

โหลดที่อยู่ (Load Address). คำสั่งเทียม, RV32I และ RV64I.

โหลดที่อยู่ของ ($symbol$) ลงในรีจิสเตอร์ $x[rd]$ เมื่อประกอบโค้ดแบบไม่ขึ้นกับตำแหน่ง (position-independent code) คำสั่งนี้จะขยายเป็นการโหลดจาก GOT (Global Offset Table):

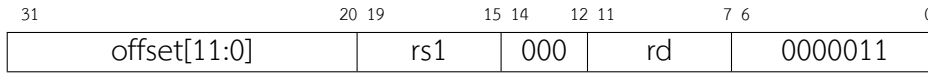
สำหรับ RV32I, **auipc** rd, offsetHi แล้วตามด้วย **lw** rd, offsetLo(rd);

สำหรับ RV64I, **auipc** rd, offsetHi แล้วตามด้วย **ld** rd, offsetLo(rd) ในกรณีอื่น คำสั่งจะถูกขยายเป็น **auipc** rd, offsetHi จากนั้น **addi** rd, rd, offsetLo

lb rd, offset(rs1) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][7:0])$

โหลดไบต์ (Load Byte). รูปแบบ-I, RV32I และ RV64I.

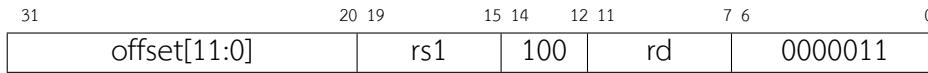
โหลดไบต์จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extended) และเขียนลงในรีจิสเตอร์ $x[rd]$ โดยขยายเครื่องหมายค่าที่โหลดได้



lbu rd, offset(rs1) $x[rd] = M[x[rs1] + \text{sext}(\text{offset})][7:0]$

โหลดไบต์แบบไม่มีเครื่องหมาย (Load Byte, Unsigned). รูปแบบ-I, RV32I และ RV64I.

โหลดไบต์จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extended) และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ โดยขยายศูนย์ (zero-extended) ค่าที่โหลดได้

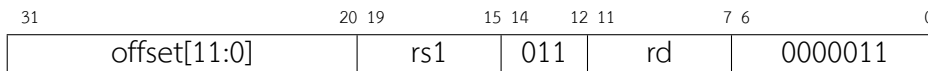


ld rd, offset(rs1) $x[rd] = M[x[rs1] + \text{sext}(\text{offset})][63:0]$

โหลดดับเบิลเวิร์ด (Load Doubleword). รูปแบบ-I, RV64I เท่านั้น.

โหลด 8 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extended) และเขียนผลลัพธ์ลงใน $x[rd]$.

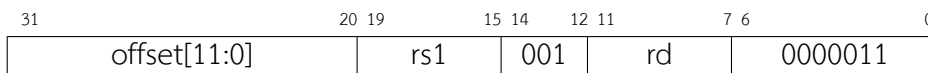
คำสั่งบีบอัด: **c.ldsp** rd, offset; **c.ld** rd, offset(rs1)



lh rd, offset(rs1) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][15:0])$

โหลดฮาล์ฟเวิร์ด (Load Halfword). รูปแบบ-I, RV32I และ RV64I.

โหลด 2 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extended) และเขียนลงใน $x[rd]$ โดยขยายเครื่องหมายค่าที่โหลดได้



lhu rd, offset(rs1) $x[rd] = M[x[rs1] + \text{sext}(\text{offset})][15:0]$

โหลดฮาล์ฟเวิร์ดแบบไม่มีเครื่องหมาย (Load Halfword, Unsigned). รูปแบบ-I, RV32I และ RV64I.

โหลด 2 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extend) และเขียนผลลัพธ์ลงใน $x[rd]$ โดยขยายศูนย์ (zero-extending) ค่าที่โหลดได้

31	20	19	15	14	12	11	7	6	0
offset[11:0]					rs1		101		rd
									0000011

li rd, immediate $x[rd] = \text{immediate}$

โหลดค่าคงที่ (Load Immediate). คำสั่งเทียม, RV32I และ RV64I.

โหลดค่าคงที่ลงใน $x[rd]$ โดยใช้คำสั่งน้อยที่สุดเท่าที่เป็นไปได้ สำหรับ RV32I จะขยายเป็น

lui และ/หรือ **addi**; สำหรับ RV64I ยาวถึง **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**

lla rd, symbol $x[rd] = \&\text{symbol}$

โหลดที่อยู่ภายใน (Load Local Address). คำสั่งเทียม, RV32I และ RV64I.

โหลดที่อยู่ของ *symbol* ลงใน $x[rd]$ โดยขยายเป็นเป็นคำสั่ง **auipc** rd, offsetHi แล้วตามด้วย **insnaddird**, rd, offsetLo

lr.d rd, (rs1) $x[rd] = \text{LoadReserved64}(M[x[rs1]])$

การดำเนินการหน่วยความจำแบบอะตอมิก: โหลดแบบจองดับเบิลเวิร์ด

(Load-Reserved Doubleword). รูปแบบ-R, RV64A เท่านั้น.

โหลด 8 ไบต์ จากหน่วยความจำที่ตำแหน่งที่อยู่ $x[rs1]$ เขียนลงในรีจิสเตอร์ $x[rd]$ และบันทึกการจอง (reservation) บนดับเบิลเวิร์ดในหน่วยความจำนั้น

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) $x[rd] = \text{LoadReserved32}(M[x[rs1]])$

การดำเนินการหน่วยความจำแบบอะตอมิก: โหลดแบบจองเวิร์ด (Load-Reserved

Word). รูปแบบ-R, RV32A และ RV64A.

โหลด 4 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1]$ เขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

โดยขยายเครื่องหมาย (sign-extended) ค่าที่โหลดได้ และบันทึกการจอง (reservation) บนเวิร์ดในหน่วยความจำนั้น

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00010				aq	rl	00000				rs1		010	
												rd	
												0101111	

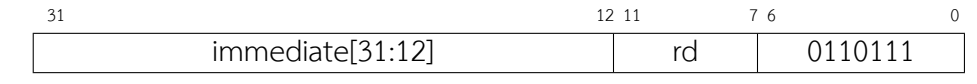
lui rd, immediate $x[rd] = \text{sext}(\text{immediate}[31:12] \ll 12)$

โหลดค่าคงที่ส่วนบน (Load Upper Immediate). รูปแบบ-U, RV32I และ RV64I.

เขียนค่าคงที่ ขนาด 20 บิตที่ขยายเครื่องหมาย (sign-extended) และเลื่อนซ้าย 12 บิต

ลงในรีจิสเตอร์ $x[rd]$ โดยตั้งค่าบิตต่ำสุด 12 บิตเป็นศูนย์

คำสั่งบีบอัด: **c.lui** rd, imm



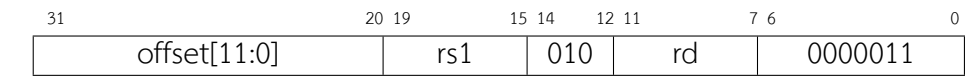
lw rd, offset(rs1) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][31:0])$

โหลดเวิร์ด (Load Word). รูปแบบ-I, RV32I และ RV64I.

โหลด 4 ไบต์จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย และ

เขียนลงในรีจิสเตอร์ $x[rd]$ สำหรับ RV64I ผลลัพธ์จะขยายเครื่องหมาย (sign-extended)

คำสั่งบีบอัด: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)



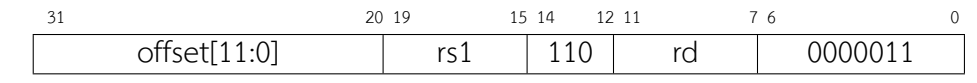
lwu rd, offset(rs1) $x[rd] = M[x[rs1] + \text{sext}(\text{offset})][31:0]$

โหลดเวิร์ดแบบไม่มีเครื่องหมาย (Load Word, Unsigned). รูปแบบ-I, RV64I เท่านั้น.

โหลด 4 ไบต์ จากหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย

(sign-extended) และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ โดยขยายศูนย์

(zero-extended) ค่าที่โหลดได้



mret ExceptionReturn (Machine)

คืนการจัดการข้อยกเว้นโหมดเครื่องจักร (Machine-mode Exception Return). รูป

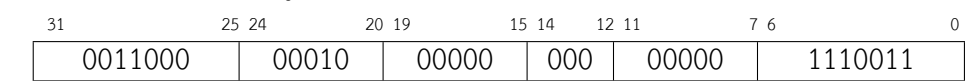
แบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.

คืนการจัดการข้อยกเว้นในโหมดเครื่องจักร โดยตั้งค่า pc เป็นค่าในเรจิสเตอร์

CSRs[mepc], ตั้งโหมดสิทธิพิเศษเป็นค่าใน CSRs[mstatus].MPP, ตั้ง

CSRs[mstatus].MIE เป็นค่าใน CSRs[mstatus].MPIE และตั้งค่า CSRs[mstatus].MPIE

เป็น 1; หากรองรับโหมดผู้ใช้ จะตั้ง CSRs[mstatus].MPP เป็น 0



mul rd, rs1, rs2 $x[rd] = x[rs1] \times x[rs2]$

คูณ (Multiply). รูปแบบ-R, RV32M และ RV64M.

คูณค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ และเขียนผลคูณลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	000	rd	0110011	

mulh rd, rs1, rs2 $x[rd] = (x[rs1] \times_s x[rs2]) \gg_s XLEN$

คูณส่วนบน (Multiply High). รูปแบบ-R, RV32M และ RV64M.

คูณค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ โดยพิจารณาค่าเป็นจำนวนเต็มในรูปแบบส่วนเต็มเต็มสองและเขียนส่วนบนของผลคูณ (upper half) ลงใน $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	001	rd	0110011	

mulhsu rd, rs1, rs2 $x[rd] = (x[rs1] \times_{su} x[rs2]) \gg_s XLEN$

คูณส่วนบนแบบมีเครื่องหมาย-ไม่มีเครื่องหมาย (Multiply High Signed-Unsigned). รูปแบบ-R, RV32M และ RV64M.

คูณค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ โดยพิจารณาค่าใน $x[rs1]$ เป็นจำนวนเต็มในรูปแบบส่วนเต็มเต็มสองแบบมีเครื่องหมาย และค่าใน $x[rs2]$ เป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนส่วนบนของผลคูณ (upper half) ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	010	rd	0110011	

mulhu rd, rs1, rs2 $x[rd] = (x[rs1] \times_u x[rs2]) \gg_u XLEN$

คูณส่วนบนแบบไม่มีเครื่องหมาย (Multiply High Unsigned). รูปแบบ-R, RV32M และ RV64M.

คูณค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ โดยพิจารณาค่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนส่วนบนของผลคูณ (upper half) ลงใน $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	011	rd	0110011	

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

คูณเวิร์ด (Multiply Word). รูปแบบ-R, RV64M เท่านั้น.

คูณค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ ตัดทอนผลคูณให้เหลือ 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	000	rd	0111011	

mv rd, rs1 $x[rd] = x[rs1]$

ย้าย (Move). คำสั่งเทียม, RV32I และ RV64I.

คัดลอกค่าจากรีจิสเตอร์ $x[rs1]$ ไปยังรีจิสเตอร์ $x[rd]$ ขยายเป็น **addi** rd, rs1, 0

neg rd, rs2 $x[rd] = -x[rs2]$

กลับเครื่องหมาย (Negate). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่าตรงข้ามในรูปแบบส่วนเติมเต็มสองของรีจิสเตอร์ $x[rs2]$ ลงในรีจิสเตอร์ $x[rd]$

ขยายเป็น **sub** rd, x0, rs2

negw rd, rs2 $x[rd] = \text{sext}((-x[rs2])[31:0])$

กลับเครื่องหมายเวิร์ด (Negate Word). คำสั่งเทียม, RV64I เท่านั้น.

คำนวณค่าส่วนเติมเต็มสองของรีจิสเตอร์ $x[rs2]$ ตัดทอนผลลัพธ์ให้เหลือ 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ ขยายเป็น **subw** rd, x0, rs2

nop Nothing

ไม่ดำเนินการ (No operation). คำสั่งเทียม, RV32I และ RV64I.

เลื่อน pc ไปยังคำสั่งถัดไปโดยไม่ดำเนินการใดๆ ขยายเป็น **addi** x0, x0, 0

not rd, rs1 $x[rd] = \sim x[rs1]$

ส่วนเติมเต็มหนึ่ง (NOT). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่าในรูปแบบส่วนเติมเต็มหนึ่งของ $x[rs1]$ ลงในรีจิสเตอร์ $x[rd]$ ขยายเป็น **xori** rd, rs1,

or rd, rs1, rs2 $x[rd] = x[rs1] \mid x[rs2]$

ดำเนินการระดับบิต OR (OR). รูปแบบ-R, RV32I และ RV64I.

คำนวณการดำเนินการแบบบิตต่อบิต Inclusive-OR ระหว่างรีจิสเตอร์ $x[rs1]$ และ $x[rs2]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

คำสั่งบีบอัด: **c.or** rd, rs2

31	25 24	20 19	15 14	12 11	7 6	0
00000000	rs2	rs1	110	rd	0110011	

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

ดำเนินการระดับบิต OR ด้วยค่าทันที (OR Immediate). รูปแบบ-I, RV32I และ RV64I.

คำนวณการดำเนินการแบบบิตต่อบิต Inclusive-OR ระหว่างค่าทันทีที่ขยายเครื่องหมาย (sign-extended) และรีจิสเตอร์ $x[rs1]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	110	rd	0010011	

rdcycle rd $x[rd] = \text{CSRs}[\text{cycle}]$

อ่านตัวนับรอบ (Read Cycle Counter). คำสั่งเทียม, RV32I และ RV64I.

เขียนจำนวนรอบ (cycles) ที่ผ่านไปลงในรีจิสเตอร์ $x[rd]$ โดยขยายเป็น **csrrs** rd, cycle, x0

rdcycleh rd $x[rd] = \text{CSRs}[\text{cycleh}]$

อ่านตัวนับรอบส่วนบน (Read Cycle Counter High). คำสั่งเทียม, RV32I เท่านั้น.

เขียนจำนวนรอบ (cycles) ที่ผ่านไป โดยเลื่อนขวา 32 บิต ลงในรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrs** rd, cycleh, x0

rdinstret rd $x[rd] = \text{CSRs}[\text{instret}]$

อ่านตัวนับคำสั่งที่เสร็จสิ้น (Read Instructions-Retired Counter). คำสั่งเทียม, RV32I และ RV64I.

เขียนจำนวนคำสั่งที่เสร็จสิ้นการทำงาน (retired instructions) ลงในรีจิสเตอร์ $[rd]$ โดยขยายเป็น **csrrs** rd, instret, x0

rdinstreth rd $x[rd] = \text{CSRs}[\text{instreth}]$

อ่านตัวนับคำสั่งที่เสร็จสิ้นส่วนบน (*Read Instructions-Retired Counter High*). คำสั่งเทียม, RV32I เท่านั้น.

เขียนจำนวนคำสั่งที่เสร็จสิ้นการทำงาน (retired instructions) โดยเลื่อนขวา 32 บิต ลงในรีจิสเตอร์ $x[rd]$ ขยายเป็น **csrrs** rd, instreth, x0

rdtime rd $x[rd] = \text{CSRs}[\text{time}]$

อ่านเวลา (*Read Time*). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่าเวลาปัจจุบันลงในรีจิสเตอร์ $x[rd]$ โดยความถี่ของตัวจับเวลา (timer frequency) ขึ้นอยู่กับแพลตฟอร์ม โดยขยายเป็น **csrrs** rd, time, x0

rdtimeh rd $x[rd] = \text{CSRs}[\text{timeh}]$

อ่านเวลาส่วนบน (*Read Time High*). คำสั่งเทียม, RV32I เท่านั้น.

เขียนค่าเวลาปัจจุบัน โดยเลื่อนขวา 32 บิต ลงในรีจิสเตอร์ $x[rd]$ โดยความถี่ของตัวจับเวลา (timer frequency) ขึ้นอยู่กับแพลตฟอร์ม โดยขยายเป็น **csrrs** rd, timeh, x0

rem rd, rs1, rs2 $x[rd] = x[rs1] \% x[rs2]$

หารเอาเศษ (*Remainder*). รูปแบบ-R, RV32M และ RV64M.

หารค่าในรีจิสเตอร์ $x[rs1]$ ด้วย $x[rs2]$ โดยผลหารถูกคำนวณโดยใช้การปัดทศทางไปที่ยุ่ (rounding towards zero) พิจารณาว่าเป็นจำนวนเต็มในรูปแบบส่วนเติมเต็มสอง) และเขียนค่าเศษลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	110	rd	0110011	

remu rd, rs1, rs2 $x[rd] = x[rs1] \%_{\text{u}} x[rs2]$

หารเอาเศษแบบไม่มีเครื่องหมาย (*Remainder, Unsigned*). รูปแบบ-R, RV32M และ RV64M.

หารค่าในรีจิสเตอร์ $x[rs1]$ ด้วย $x[rs2]$ โดยผลหารถูกคำนวณโดยใช้การปัดทศทางไปที่ยุ่ (rounding towards zero) พิจารณาว่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนค่าเศษลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	111	rd	0110011	

remuw rd,rs1,rs2 $x[rd] = \text{sext}(x[rs1][31:0] \%_{\text{u}} x[rs2][31:0])$

หารเอาเศษเวิร์ดแบบไม่มีเครื่องหมาย(Remainder Word, Unsigned). รูปแบบ-R, RV64M เท่านั้น.

หาร 32 บิตที่มีค่าน้อยที่สุด (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ด้วย 32 บิตที่มีค่าน้อยที่สุดของ $x[rs2]$ โดยปัดทิศทางไปศูนย์ (rounding towards zero) พิจารณาค่าเป็นจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนค่าเศษ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	111	rd	0111011	

remw rd,rs1,rs2 $x[rd] = \text{sext}(x[rs1][31:0] \%_{\text{s}} x[rs2][31:0])$

หารเอาเศษเวิร์ด (Remainder Word). รูปแบบ-R, RV64M เท่านั้น.

หาร 32 บิตที่มีค่าน้อยที่สุด (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ด้วย 32 บิตที่มีค่าน้อยที่สุดของ $x[rs2]$ โดยปัดทิศทางไปศูนย์ (rounding towards zero) พิจารณาค่าเป็นจำนวนเต็มในรูปแบบส่วนเติมเต็มสองและเขียนค่าเศษ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$

31	25 24	20 19	15 14	12 11	7 6	0
0000001	rs2	rs1	110	rd	0111011	

ret

$pc = x[1]$

คืนจากฟังก์ชัน (Return). คำสั่งเทียม, RV32I และ RV64I.

คืนจากการเรียกรูทिनย่อย โดยตั้งค่า pc เป็นค่าที่อยู่ในรีจิสเตอร์ $x1$ (ลิงก์รีจิสเตอร์) ซึ่งโดยทั่วไปเก็บที่อยู่สำหรับการคืนจากรูทินย่อย ขยายเป็น **jair** $x0, 0(x1)$

sb rs2,offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][7:0]$

บันทึกไบต์ (Store Byte). รูปแบบ-S, RV32I และ RV64I.

บันทึกไบต์ที่มีนัยสำคัญน้อยที่สุด จากรีจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำที่ตำแหน่ง $x[rs1] + (\text{offset} \text{ ที่ขยายเครื่องหมาย})$

31	25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011	

sc.d rd, rs2, (rs1) $x[rd] = \text{StoreConditional64}(M[x[rs1]], x[rs2])$

การดำเนินการหน่วยความจำแบบอะตอมิก: บันทึกรูปแบบมีเงื่อนไขดับเบิลเวิร์ด (Store-Conditional Doubleword). รูปแบบ-R, RV64A เท่านั้น.

บันทึก 8 ไบต์ จากระจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำที่ตำแหน่ง $x[rs1]$ โดยมีเงื่อนไขว่า ต้องมีการจองการจองโหลด (load reservation) บนที่อยู่นั้นอยู่ก่อนแล้ว เขียนค่า 0 ลงใน $x[rd]$ หากการบันทึกสำเร็จ หรือเขียนรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ หากการบันทึกไม่สำเร็จ

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00011	aq	rl	rs2	rs1	011	rd	0101111						

sc.w rd, rs2, (rs1) $x[rd] = \text{StoreConditional32}(M[x[rs1]], x[rs2])$

การดำเนินการหน่วยความจำแบบอะตอมิก: บันทึกรูปแบบมีเงื่อนไขเวิร์ด (Store-Conditional Word). รูปแบบ-R, RV32A และ RV64A.

บันทึก 4 ไบต์ จากระจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำที่ตำแหน่ง $x[rs1]$ โดยมีเงื่อนไขว่า ต้องมีการจองการจองโหลด (load reservation) บนที่อยู่นั้นอยู่ก่อนแล้ว เขียนค่า 0 ลงในรีจิสเตอร์ $x[rd]$ หากการบันทึกสำเร็จ หรือเขียนรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ หากการบันทึกไม่สำเร็จ

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00011	aq	rl	rs2	rs1	010	rd	0101111						

sd rs2, offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][63:0]$

บันทึกดับเบิลเวิร์ด (Store Doubleword). รูปแบบ-S, RV64I เท่านั้น.

บันทึก 8 ไบต์ จากระจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำที่ตำแหน่ง $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย

คำสั่งบีบอัด: **c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)

31	25	24	20	19	15	14	12	11	7	6	0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100011						

seqz rd, rs1 $x[rd] = (x[rs1] == 0)$

ตั้งค่าเมื่อเท่ากับศูนย์ (Set if Equal to Zero). คำสั่งเติม, RV32I and RV64I.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าค่าใน $x[rs1]$ เท่ากับ 0 หรือเขียนค่า 0 ถ้าไม่เท่ากับ 0 โดยขยายเป็น **sltiu** rd, rs1, 1

sext.w rd, rs1 $x[rd] = \text{sext}(x[rs1][31:0])$

ขยายเครื่องหมายเวิร์ด (Sign-extend Word). คำสั่งเทียม, RV64I only.

อ่าน 32 บิตที่มีค่าน้อยที่สุด (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ขยายเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ โดยขยายเป็น **addiw** rd, rs1, 0

sfence.vma rs1, rs2 Fence(Store, AddressTranslation)

กำหนดลำดับการเขียนในหน่วยความจำเสมือน (Fence Virtual Memory). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.

กำหนดลำดับการเขียนข้อมูล (stores) ที่เกิดขึ้นก่อนหน้าลงในตารางหน้า (page tables) กับการแปลงที่อยู่เสมือน (virtual-address translations) ที่ตามมา เมื่อ $rs2 = 0$ มีผลต่อการแปลงที่อยู่สำหรับทุกพื้นที่ที่อยู่ (address spaces) หากไม่ใช่ 0 จะจัดลำดับเฉพาะการแปลงสำหรับพื้นที่ที่อยู่ที่ระบุโดย $x[rs2]$ เมื่อ $rs1 = 0$ จะจัดลำดับการแปลงสำหรับทุกที่อยู่เสมือนในพื้นที่ที่อยู่ที่เลือก หากไม่ใช่ 0 จะจัดลำดับเฉพาะการแปลงสำหรับหน้า (page) ที่มีที่อยู่เสมือน $x[rs1]$ ในพื้นที่ที่อยู่ที่เลือก

31	25 24	20 19	15 14	12 11	7 6	0
0001001	rs2	rs1	000	00000	1110011	

sgtz rd, rs2 $x[rd] = (x[rs2] >_s 0)$

ตั้งค่าเมื่อมากกว่าศูนย์ (Set if Greater Than to Zero). คำสั่งเทียม, RV32I และ RV64I. เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้าค่าใน $x[rs2]$ มากกว่า 0 หรือเขียนค่า 0 ถ้าน้อยกว่าหรือเท่ากับ 0 ขยายเป็น **slt** rd, x0, rs2

sh rs2, offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][15:0]$

บันทึกฮาล์ฟเวิร์ด (Store Halfword). รูปแบบ-S, RV32I และ RV64I.

บันทึกสองไบต์ที่มีนัยสำคัญน้อยที่สุด จากรีจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset})$ ที่ขยายเครื่องหมาย (sign-extended)

31	25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011	

sll rd, rs1, rs2 $x[rd] = x[rs1] \ll x[rs2]$

เลื่อนบิตซ้าย (Shift Left Logical). รูปแบบ-R, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปทางซ้ายตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ลงใน $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีค่าน้อยที่สุดของ $x[rs2]$ (หรือใช้ 6 บิตสำหรับ RV64I) โดยบิตส่วนบนจะถูกละเว้น

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	001	rd	0110011	

slli rd, rs1, shamt $x[rd] = x[rs1] \ll \text{shamt}$

เลื่อนบิตขวาด้วยค่าทันที (Shift Left Logical Immediate). รูปแบบ-I, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปซ้ายตามจำนวนบิตที่ระบุใน shamt โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ สำหรับ RV32I คำสั่งนี้ถูกต้องตามกฎหมาย (เฉพาะเมื่อ $\text{shamt}[5]=0$ คำสั่งบีบอัด **c.slli** rd, shamt

31	26 25	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	001	rd	0010011	

slliw rd, rs1, shamt $x[rd] = \text{sext}((x[rs1] \ll \text{shamt})[31:0])$

เลื่อนบิตซ้ายด้วยค่าทันทีแบบเวิร์ด (Shift Left Logical Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

เลื่อนรีจิสเตอร์ $x[rs1]$ ไปซ้ายตามตำแหน่งบิต shamt บิตที่ว่างจะถูกเติมด้วยศูนย์ ผลลัพธ์จะถูกตัดให้เหลือ 32 บิต และผลลัพธ์ 32 บิตที่ขยายเครื่องหมายจะถูกเขียนไปยัง $x[rd]$ คำสั่งจะถูกต้องก็ต่อเมื่อ $\text{shamt}[5]=0$

31	26 25	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	001	rd	0011011	

sllw rd, rs1, rs2 $x[rd] = \text{sext}((x[rs1] \ll x[rs2][4:0])[31:0])$

เลื่อนบิตซ้ายเวิร์ด (Shift Left Logical Word). รูปแบบ-R, RV64I เท่านั้น.

เลื่อน 32 บิตที่มีนัยสำคัญน้อยที่สุด (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ไปซ้ายตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีนัยสำคัญน้อยที่สุดของ $x[rs2]$ โดยบิตส่วนบนจะถูกละเว้น

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	001	rd	0111011	

slt rd, rs1, rs2 $x[rd] = x[rs1] <_s x[rs2]$

ตั้งค่าน้อยกว่า (Set if Less Than). รูปแบบ-R, RV32I และ RV64I.

เปรียบเทียบค่าในรีจิสเตอร์ $x[rs1]$ และ $x[rs2]$ ในรูปแบบจำนวนเต็มส่วนเติมเต็มสองและเขียน 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้า $x[rs1]$ น้อยกว่า $x[rs2]$ หรือเขียน 0 ถ้ามากกว่าหรือเท่ากับ

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	010	rd	0110011	

slti rd, rs1, immediate $x[rd] = x[rs1] <_s \text{sext}(\text{immediate})$

ตั้งค่าน้อยกว่าค่าทันที (Set if Less Than Immediate). รูปแบบ-I, RV32I และ RV64I.

เปรียบเทียบค่าในรีจิสเตอร์ $x[rs1]$ กับค่า *ค่าทันที* ที่ขยายเครื่องหมาย (sign-extended) ในรูปแบบจำนวนเต็มส่วนเติมเต็มสองและเขียน 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้า $x[rs1]$ น้อยกว่า *ค่าทันที* หรือเขียน 0 ถ้ามากกว่าหรือเท่ากับ

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	010	rd	0010011	

sltiu rd, rs1, immediate $x[rd] = x[rs1] <_u \text{sext}(\text{immediate})$

ตั้งค่าน้อยกว่าค่าทันทีแบบไม่มีเครื่องหมาย (Set if Less Than Immediate, Unsigned). รูปแบบ-I, RV32I และ RV64I.

เปรียบเทียบค่าในรีจิสเตอร์ $x[rs1]$ กับ *ค่าทันที* ที่ขยายเครื่องหมาย (sign-extended) ในรูปแบบจำนวนเต็มแบบไม่มีเครื่องหมาย (unsigned) และเขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้า $x[rs1]$ น้อยกว่า *ค่าทันที* หรือเขียน 0 ถ้ามากกว่าหรือเท่ากับ

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	011	rd	0010011	

sltu rd, rs1, rs2 $x[rd] = x[rs1] <_u x[rs2]$

ตั้งค่าน้อยกว่าแบบไม่มีเครื่องหมาย (Set if Less Than, Unsigned). รูปแบบ-R, RV32I และ RV64I.

เปรียบเทียบค่าในรีจิสเตอร์ $x[rs1]$ กับ $x[rs2]$ เป็นตัวเลขแบบไม่มีเครื่องหมาย และเขียน 1 ลงใน $x[rd]$ ถ้า $x[rs1]$ น้อยกว่า $x[rs2]$ หรือเขียน 0 ถ้ามากกว่าหรือเท่ากับ

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	011	rd	0110011	

sltz rd, rs1 $x[rd] = (x[rs1] <_s 0)$

ตั้งค่าน้อยกว่าศูนย์ (Set if Less Than to Zero). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่า 1 ลงในรีจิสเตอร์ $x[rd]$ ถ้า $x[rs1]$ น้อยกว่าศูนย์ หรือเขียน 0 ถ้ามากกว่าหรือเท่ากับ 0 ขยายเป็น **slti** rd, rs1, x0

snez rd, rs2 $x[rd] = (x[rs2] \neq 0)$

ตั้งค่าเมื่อไม่เท่ากับศูนย์ (Set if Not Equal to Zero). คำสั่งเทียม, RV32I และ RV64I.

เขียนค่า 0 ลงในรีจิสเตอร์ $x[rd]$ ถ้า $x[rs2]$ เท่ากับ 0 หรือเขียนค่า 1 ถ้าไม่เท่ากับ 0 ขยายเป็น **sltu** rd, x0, rs2

sra rd, rs1, rs2 $x[rd] = x[rs1] >>_s x[rs2]$

เลื่อนขวาแบบเลขคณิต (Shift Right Arithmetic). รูปแบบ-R, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปขวาตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยสำเนาของบิตที่มีนัยสำคัญสูงสุดของ $x[rs1]$ เพื่อรักษาเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีนัยสำคัญน้อยที่สุดของ $x[rs2]$ (ใน RV32I) หรือ 6 บิต (ใน RV64I) โดยบิตส่วนบนจะถูกกลบ

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	101	rd	0110011	

srai rd, rs1, shamt $x[rd] = x[rs1] >>_s shamt$

เลื่อนขวาแบบเลขคณิตด้วยค่าทันที (Shift Right Arithmetic Immediate). รูปแบบ-I, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $shamt$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยสำเนาของบิตที่มีนัยสำคัญสูงสุดของ $x[rs1]$ เพื่อรักษาเครื่องหมาย และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ สำหรับ RV32I คำสั่งนี้จะถูกต้องตามกฎหมายเฉพาะเมื่อ $shamt[5]=0$.

คำสั่งบีบอัด: **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 $x[rd] = \text{sext}(x[rs1][31:0] \gg_{\text{shamt}})$

เลื่อนขวาแบบเลขคณิตแบบเวิร์ดด้วยค่าทันที (Shift Right Arithmetic Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

เลื่อน 32 บิตล่าง (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน *shamt* โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยสำเนาของบิตที่ 31 ของ $x[rs1][31]$ เพื่อรักษาเครื่องหมาย และเขียนผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ คำสั่งนี้จะถูกต้องตามกฎหมายเฉพาะเมื่อ *shamt*[5]=0

31	26 25	20 19	15 14	12 11	7 6	0
010000	shamt	rs1	101	rd	0011011	

sraw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \gg_{x[rs2][4:0]})$

เลื่อนขวาแบบเลขคณิตแบบเวิร์ด (Shift Right Arithmetic Word). รูปแบบ-R, RV64I เท่านั้น.

เลื่อน 32 บิตล่าง (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยสำเนาของบิตที่ 31 ของ $x[rs1][31]$ เพื่อรักษาเครื่องหมาย และเขียนผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีนัยสำคัญน้อยที่สุดของ $x[rs2]$ โดยบิตส่วนบนจะถูกละเว้น

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	101	rd	0111011	

sret

ExceptionReturn (Supervisor)

คืนจากการจัดการข้อยกเว้นโหมดซูเปอร์ไวเซอร์ (Supervisor-mode Exception Return). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิ์พิเศษ.

คืนจากการจัดการข้อยกเว้นในโหมดซูเปอร์ไวเซอร์ โดยตั้งค่า *pc* เป็นค่าในรีจิสเตอร์ $CSRs[sepc]$ ตั้งโหมดสิทธิ์พิเศษเป็นค่าใน $CSRs[sstatus].SPP$, ตั้ง $CSRs[sstatus].SIE$ เป็นค่าใน $CSRs[sstatus].SPIE$, ตั้ง $CSRs[sstatus].SPIE$ เป็น 1 และตั้ง $CSRs[sstatus].SPP$ เป็น 0

31	25 24	20 19	15 14	12 11	7 6	0
0001000	00010	00000	000	00000	1110011	

srl rd, rs1, rs2 $x[rd] = x[rs1] \gg_u x[rs2]$

เลื่อนขวาแบบลอจิกัล (Shift Right Logical). รูปแบบ-R, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ เขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีนัยสำคัญน้อยที่สุดของ $x[rs2]$ (หรือ 6 บิตใน RV64I) โดยบิตส่วนบนจะถูกละเว้น

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	101	rd	0110011	

srl rd, rs1, shamt $x[rd] = x[rs1] \gg_u shamt$

เลื่อนขวาแบบลอจิกัลด้วยค่าทันที (Shift Right Logical Immediate). รูปแบบ-I, RV32I และ RV64I.

เลื่อนค่าในรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $shamt$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ สำหรับ RV32I คำสั่งนี้จะถูกต้องเฉพาะเมื่อ $shamt[5]=0$.

คำสั่งบีบอัด: **c.srli** rd, shamt

31	26 25	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	101	rd	0010011	

srliw rd, rs1, shamt $x[rd] = sext(x[rs1][31:0] \gg_u shamt)$

เลื่อนขวาแบบลอจิกัลเวิร์ดด้วยค่าทันที (Shift Right Logical Word Immediate). รูปแบบ-I, RV64I เท่านั้น.

เลื่อน 32 บิตล่าง (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $shamt$ บิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ คำสั่งจะถูกต้องตามกฎหมาย (legal) เฉพาะเมื่อ $shamt[5]=0$

31	26 25	20 19	15 14	12 11	7 6	0
0000000	shamt	rs1	101	rd	0011011	

srlw rd, rs1, rs2 $x[rd] = \text{sext}(x[rs1][31:0] \gg_u x[rs2][4:0])$

เลื่อนขวาแบบลอจิคัลเวิร์ด (Shift Right Logical Word). รูปแบบ-R, RV64I เท่านั้น.

เลื่อน 32 บิตล่าง (lower 32 bits) ของรีจิสเตอร์ $x[rs1]$ ไปทางขวาตามจำนวนบิตที่ระบุใน $x[rs2]$ โดยบิตที่ว่างลง (vacated bits) จะถูกเติมด้วยศูนย์ และเขียนผลลัพธ์ 32 บิตที่ขยายเครื่องหมาย (sign-extended) ลงในรีจิสเตอร์ $x[rd]$ จำนวนบิตที่เลื่อน (shift amount) ใช้ 5 บิตที่มีนัยสำคัญน้อยที่สุดของ $x[rs2]$ โดยบิตส่วนบนจะถูกละเว้น

31	25 24	20 19	15 14	12 11	7 6	0
0000000	rs2	rs1	101	rd	0111011	

sub rd, rs1, rs2 $x[rd] = x[rs1] - x[rs2]$

ลบ (Subtract). รูปแบบ-R, RV32I และ RV64I.

ลบค่าในรีจิสเตอร์ $x[rs2]$ ออกจากรีจิสเตอร์ $x[rs1]$ และเขียนผลลัพธ์ลงในรีจิสเตอร์ $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **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 $x[rd] = \text{sext}((x[rs1] - x[rs2])[31:0])$

ลบเวิร์ด (Subtract Word). รูปแบบ-R, RV64I เท่านั้น.

ลบค่าในรีจิสเตอร์ $x[rs2]$ ออกจากรีจิสเตอร์ $x[rs1]$ ตัดทอนผลลัพธ์เหลือ 32 บิต และเขียนผลลัพธ์ที่ขยายเครื่องหมายลงใน $x[rd]$ ไม่ตรวจสอบการล้น (overflow) ของการคำนวณ

คำสั่งบีบอัด: **c.subw** rd, rs2

31	25 24	20 19	15 14	12 11	7 6	0
0100000	rs2	rs1	000	rd	0111011	

sw rs2, offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][31:0]$

บันทึกเวิร์ด (Store Word). รูปแบบ-S, RV32I และ RV64I.

บันทึกสี่ไบต์ที่มีนัยสำคัญน้อยที่สุด (least-significant bytes) จากรีจิสเตอร์ $x[rs2]$ ลงในหน่วยความจำตำแหน่งที่อยู่ $x[rs1] + (\text{offset ที่ขยายเครื่องหมาย})$.

คำสั่งบีบอัด: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)

31	25 24	20 19	15 14	12 11	7 6	0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011	

tail symbol $pc = \&symbol; ; \text{clobber } x[6]$

เรียกไปยังที่อยู่ของฟังก์ชัน (*Tail call*). คำสั่งเติม, RV32I และ RV64I.

ตั้งค่า pc เป็นที่อยู่ของ $symbol$ โดยมีการเขียนทับรีจิสเตอร์ $x[6]$ ในกระบวนการ โดยขยายเป็นคำสั่ง **auipc** $x6, offsetHi$ จากนั้นตามด้วยคำสั่ง **jair** $x0, offsetLo(x6)$

wfi $\text{while } (\text{noInterruptsPending}) \text{ idle}$

รอการขัดจังหวะ (*Wait for Interrupt*). รูปแบบ-R, RV32I และ RV64I สถาปัตยกรรมสิทธิพิเศษ.

ทำให้โปรเซสเซอร์อยู่ในสถานะว่าง (*idle*) เพื่อประหยัดพลังงาน หากไม่มีการขัดจังหวะ (*interrupts*) ที่เปิดใช้งานอยู่ในขณะนั้น

31	25 24	20 19	15 14	12 11	7 6	0
0001000	00101	00000	000	00000	1110011	

xor rd, rs1, rs2 $x[rd] = x[rs1] \wedge x[rs2]$

เอกซ์คลูซีฟ-OR (*Exclusive-OR*). รูปแบบ-R, RV32I และ RV64I.

คำนวณค่าระดับบิต X-OR ของรีจิสเตอร์ $x[rs1]$ และ $x[rs2]$ และเขียนผลลัพธ์ลงใน $x[rd]$

คำสั่งบีบอัด: **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 $x[rd] = x[rs1] \wedge \text{sext}(\text{immediate})$

เอกซ์คลูซีฟ-OR ค่าทันที (*Exclusive-OR Immediate*). รูปแบบ-I, RV32I และ RV64I.

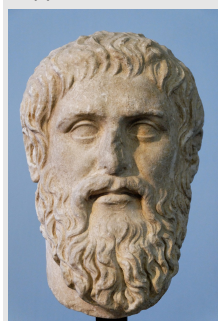
คำนวณค่าระดับบิต X-OR ของค่าทันทีที่ขยายเครื่องหมาย และ $x[rs1]$ และเขียนผลลัพธ์ลงใน $x[rd]$

31	20 19	15 14	12 11	7 6	0
immediate[11:0]	rs1	100	rd	0010011	

B

การแปลงคำสั่ง RISC-V

เพลโต (พ.ศ.971–891 ก่อนคริสต์ศักราช) เป็นนักปรัชญากรีกโบราณผู้วางรากฐานให้กับคณิตศาสตร์ปรัชญา และวิทยาศาสตร์ตะวันตก



ความงามของรูปแบบ ความกลมกลืน ความสง่างาม และจังหวะที่ดีขึ้นอยู่กับความเรียบง่าย

—เพลโต, สาธารณรัฐ

B.1 บทนำ

ภาคผนวกนี้ประกอบด้วยตารางที่แปลงคำสั่งทั่วไปและรูปแบบการเขียนคำสั่งเฉพาะที่เป็นแบบแผนใน RV32I ให้อยู่ในรูปแบบโค้ดที่เทียบเท่ากันใน ARM-32 และ x86-32 จุดประสงค์ของการเขียนภาคผนวกนี้คือ เพื่อช่วยโปรแกรมเมอร์ที่ไม่คุ้นเคยกับ RISC-V แต่มีความคุ้นเคยกับ ARM-32 หรือ x86-32 ให้สามารถเรียนรู้ RISC-V ได้ง่ายขึ้น และช่วยให้สามารถแปลง ISA รุ่นเก่าเหล่านั้นให้เป็นโค้ด RISC-V พื้นฐานได้ ภาคผนวกนี้ปิดท้ายด้วยโค้ดภาษา C สำหรับการสำรวจต้นไม้ทวิภาค (binary tree traversal) พร้อมโค้ดแอสเซมบลีที่มีคำอธิบายกำกับสำหรับ ISA ทั้งสามแบบ โดยจัดลำดับคำสั่งของการทำงานทั้งสามรูปแบบให้มีความใกล้เคียงกันมากที่สุด เพื่อให้เห็นความสอดคล้องกันของคำสั่งเหล่านั้นได้อย่างชัดเจน



ความเรียบง่าย

คำสั่งถ่ายโอนข้อมูลในรูปที่ B.1 แสดงให้เห็นถึงความคล้ายคลึงกันระหว่างคำสั่งโหลดและคำสั่งเก็บข้อมูลของ RV32I และ ARM-32 สำหรับรูปแบบการกำหนดแอดเดรส (addressing mode) ที่ได้รับความนิยมมากที่สุด เนื่องจาก ISA ของ x86 ใช้แนวทางที่ให้หน่วยความจำและรีจิสเตอร์สามารถเป็นโอเปอเรนด์ของคำสั่งได้ (memory-register orientation) แทนที่จะใช้แนวทางแบบโหลด-เก็บ (load-store orientation) เช่นเดียวกับ ISA

Description	RV32I	ARM-32	x86-32
Load word	lw t0, 4(t1)	ldr r0, [r1, #4]	mov eax, [edi+4]
Load halfword	lh t0, 4(t1)	ldrsh r0, [r1, #4]	movsx eax, WORD PTR[edi+4]
Load halfword unsigned	lhu t0, 4(t1)	ldrh r0, [r1, #4]	movzx eax, WORD PTR[edi+4]
Load byte	lb t0, 4(t1)	ldrb 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

รูปที่ B.1: คำสั่งการเข้าถึงหน่วยความจำ RV32I แปลงเป็น ARM-32 และ x86-32

Description	RV32I	ARM-32	x86-32
Zero register	li t0, 0	mov r0, #0	xor eax, eax
Move register	mv t0, t1	mov r0, r1	mov eax, edi
Complement register	not t0, t1	mvn r0, r1	not eax, edi
Negate register	neg t0, t1	rsb r0, r1, #0	mov eax, edi neg eax
Load large constant	lui t0, 0xABCDE addi t0, t0, 0x123	movw r0, #0xE123 movt r0, #0xABCD	mov eax, 0xABCDE123
Move PC to register	auipc t0, 0	ldr r0, [pc, #-8]	call lf 1: pop eax
Add	add t0, t1, t2	add r0, r1, r2	lea eax, [edi+esi]
Add (imm.)	addi t0, t0, 1	add r0, r0, #1	add eax, 1
Subtract	sub t0, t0, t1	sub r0, r0, r1	sub eax, edi
Set reg. to (reg=0)	sltiu t0, t1, 1	rsbs r0, r1, #1 movcc r0, #0	xor eax, eax test edx, edx sete al
Set reg. to (reg≠0)	sltu t0, x0, t1	adds r0, r1, #0 movne r0, #1	xor eax, eax test edx, edx setne al
Bitwise-OR	or t0, t0, t1	orr r0, r0, r1	or eax, edi
Bitwise-AND	and t0, t0, t1	and r0, r0, r1	and eax, edi
Bitwise-XOR	xor t0, t0, t1	eor r0, r0, r1	xor eax, edi
Bitwise-OR (imm.)	ori t0, t0, 1	orr r0, r0, #1	or eax, 1
Bitwise-AND (imm.)	andi t0, t0, 1	and r0, r0, #1	and eax, 1
Bitwise-XOR (imm.)	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 logical	srl t0, t0, t1	lsr r0, r0, r1	shr eax, cl
Shift right arith.	sra t0, t0, t1	asr r0, r0, r1	sar eax, cl
Shift left (imm.)	slli t0, t0, 1	lsl r0, r0, #1	sal eax, 1
Shift right logical (imm.)	srlr t0, t0, 1	lsr r0, r0, #1	shr eax, 1
Shift right arith. (imm.)	srair t0, t0, 1	asr r0, r0, #1	sar eax, 1

รูปที่ B.2: คำสั่งทางคณิตศาสตร์ของ RV32I ที่แปลงเป็น ARM-32 และ x86-32 รูปแบบคำสั่งแบบสองตัวถูกดำเนินการ (two-operand) ของ x86-32 มักต้องใช้คำสั่งมากกว่าเมื่อเทียบกับรูปแบบคำสั่งแบบสามตัวถูกดำเนินการ (three-operand) ของ ARM-32 และ RV32I

ของ RV32I และ ARM-32 ดังนั้น x86 จึงใช้ คำสั่งย้ายข้อมูลในการถ่ายโอนข้อมูลแทน นอกเหนือจากคำสั่งเลขคณิต คำสั่งตรรกะ และคำสั่งเลื่อนบิตมาตรฐานแล้ว รูปที่ B.2 แสดงวิธีการดำเนินการทั่วไปในแต่ละ ISA ตัวอย่างเช่น การทำให้เรจิสเตอร์เป็นศูนย์ใช้ คำสั่งเทียบ li ใน RV32I คำสั่งคัดลอกค่าคงที่ (move immediate) ใน ARM-32 และการดำเนินการ exclusive-OR รีจิสเตอร์กับตัวเองใน x86-32 ข้อจำกัดสองตัวถูกดำเนินการของคำสั่ง x86-32 ทำให้ต้องใช้คำสั่งมากขึ้นในบางกรณี แม้ว่ารูปแบบคำสั่งความยาวแปรผัน (variable length instruction format) จะช่วยให้โหลดค่าคงที่ขนาดใหญ่ได้ในคำสั่งเดียว คำสั่งบวก ลบ ตรรกะ และเลื่อนตำแหน่งแบบทั่วไป—ซึ่งเป็นคำสั่งที่ใช้มากที่สุด—มีการแมปแบบหนึ่งต่อหนึ่งระหว่าง ISA

รูปที่ B.3 รายการคำสั่งกระโดดแบบมีเงื่อนไขและไม่มีเงื่อนไข รวมถึงคำสั่งเรียก (call instructions) แสดงให้เห็นว่าแนวทางรหัสเงื่อนไขต้องใช้สองคำสั่งสำหรับ ARM-32 และ

Description	RV32I	ARM-32	x86-32
Branch if =	beq t0, t1, foo	cmp r0, r1 beq foo	cmp eax, esi je foo
Branch if $\neq$	bne t0, t1, foo	cmp r0, r1 bne foo	cmp eax, esi jne foo
Branch if <	blt t0, t1, foo	cmp r0, r1 blt foo	cmp eax, esi jl foo
Branch if $\geq_s$	bge t0, t1, foo	cmp r0, r1 bge foo	cmp eax, esi jge foo
Branch if $<_u$	bltu t0, t1, foo	cmp r0, r1 bcc foo	cmp eax, esi jb foo
Branch if $\geq_u$	bgeu t0, t1, foo	cmp r0, r1 bcs foo	cmp eax, esi jnb foo
Branch if =0	beqz t0, foo	cmp r0, #0 beq foo	test eax, eax je foo
Branch if $\neq 0$	bnez t0, foo	cmp r0, #0 bne foo	test eax, eax jne foo
Direct jump or tail call	jal x0, foo	b foo	jmp foo
Subroutine call	jal ra, foo	bl foo	call foo
Subroutine return	jalr x0, 0(ra)	bx lr	ret
Indirect call	jalr ra, 0(t0)	blx r0	call eax
Indirect jump or tail call	jalr x0, 0(t0)	bx r0	jmp eax

รูปที่ B.3: คำสั่งการควบคุม RV32I แปลงเป็น ARM-32 และ x86-32 คำสั่งเปรียบเทียบและกระโดด (compare-and-branch) ของ RV32I ใช้จำนวนคำสั่งเพียงครั้งหนึ่งเมื่อเทียบกับการกระโดดตามรหัสเงื่อนไข (condition code based branches) ของ ARM-32 และ x86-32

x86-32 ในขณะที่ RV32I ใช้เพียงคำสั่งเดียว ดังที่บทที่ 2 แสดงในรูปที่ 2.5 ถึง 2.11 แม้ว่า RISC-V จะออกแบบ ISA แบบเรียบง่าย แต่การกระโดดแบบเปรียบเทียบและดำเนินการ (compare-and-execute branches) ลดจำนวนคำสั่งใน Insertion Sort ได้มากเท่ากับ โหมดการกำหนดที่อยู่ที่ซับซ้อนและคำสั่ง push และ pop ของ ARM-32 และ x86-32



ประสิทธิ์ภาพ

B.2 การเปรียบเทียบ RV32I, ARM-32 และ x86-32 โดยใช้ Tree Sum

โค้ดภาษา C ในรูปที่ B.4 เป็นตัวอย่างที่เราใช้เปรียบเทียบทั้งสาม ISA แบบเคียงข้างกันในรูปที่ B.5 ถึง B.7 โดยคำนวณผลรวมของค่าในไบนารีทรี โดยใช้การสำรวจทรีแบบตามลำดับ (in-order) ทรีเป็นโครงสร้างข้อมูลพื้นฐาน และแม้ว่าการดำเนินการทรีนี้อาจดูเรียบง่ายเกินไป เราเลือกเพราะมันแสดงทั้งการเรียกซ้ำ (recursion) และการวนซ้ำ (iteration) ในคำสั่งแอสเซมบลีเพียงไม่กี่คำสั่ง โปรแกรมนี้ใช้การเรียกซ้ำเพื่อคำนวณผลรวมของทรีย่อยด้านซ้าย (left subtree) แต่ใช้การวนซ้ำเพื่อคำนวณผลรวมของทรีย่อยด้านขวา (right subtree) ซึ่งลดขนาดหน่วยความจำและจำนวนคำสั่ง คอมไพเลอร์ที่ปรับแต่งแล้วสามารถแปลงโค้ดที่เรียกซ้ำทั้งหมดเป็นรุ่นนี้ได้ เราแสดงการวนซ้ำอย่างชัดเจนเพื่อความชัดเจน

ความแตกต่างที่ใหญ่ที่สุดระหว่างขนาดของโปรแกรมภาษาแอสเซมบลีทั้งสามคือใน ส่วนการเข้าและออกจากฟังก์ชัน (function entry and exit) RISC-V ใช้สี่คำสั่งเพื่อบันทึกและกู้คืนสามรีจิสเตอร์ในสแต็กและปรับตัวชี้สแต็ก (stack pointer) x86-32 บันทึกและกู้คืนเพียงสองรีจิสเตอร์ในสแต็ก เพราะสามารถดำเนินการเลขคณิตกับตัวถูกดำเนินการใน หน่วยความจำได้โดยไม่ต้องโหลดทั้งหมดลงในรีจิสเตอร์ และยังใช้คำสั่ง push และ pop ซึ่งปรับตัวชี้สแต็กโดยนัย (implicitly) แทนการทำอย่างชัดเจนอย่างใน RISC-V ARM-32 สามารถบันทึกสามรีจิสเตอร์รวมถึงลิงก์รีจิสเตอร์ (link register) ที่มีที่อยู่คืนกลับ (return address) ลงในสแต็กด้วยคำสั่ง push เดียว และกู้คืนด้วยคำสั่ง pop เดียว RISC-V ดำเนิน การลูปหลัก (main loop) ด้วยเจ็ดคำสั่ง แทนแปดคำสั่งใน ISA อื่นๆ เพราะตามที่แสดงในรูปที่ B.3 RISC-V สามารถเปรียบเทียบและกระโดดในคำสั่งเดียว (bnez) ในขณะที่ ARM-32 และ x86-32 ต้องใช้สองคำสั่ง คำสั่งที่เหลือในลูปมีการแมปแบบหนึ่งต่อหนึ่งระหว่าง RV32I และ ARM-32 ดังที่แสดงในรูปที่ B.1 และ B.2 ความแตกต่างคือ คำสั่ง call และ ret ของ x86-32 จะทำการ push และ pop ที่อยู่คืนกลับและจากสแต็กโดยนัย ในขณะที่ ISA อื่นๆ ทำการบันทึกและกู้คืนอย่างชัดเจนในส่วน prologue และ epilogue (โดย บันทึกและกู้คืน ra สำหรับ RV32I หรือ push lr และ pop ไปยัง pc สำหรับ ARM-32) นอกจากนี้ เนื่องจากข้อกำหนดการเรียกฟังก์ชัน (calling convention) ของ x86-32 ส่ง พารามิเตอร์ผ่านสแต็ก โค้ด x86-32 จึงมีคำสั่ง push และ pop ในลูปที่ ISA อื่นๆ หลีก เลี่ยงได้ การถ่ายโอนข้อมูลเพิ่มเติมนี้ลดประสิทธิภาพ



ประสิทธิ์ภาพ

B.3 บทสรุป

แม้ว่าปรัชญาของ ISA จะแตกต่างกันอย่างมาก แต่โปรแกรมที่ได้นั้นค่อนข้างคล้ายกัน

```

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;
}

```

รูปที่ B.4: ฟังก์ชัน C ที่รวมค่าต่างๆ ในไบนารีทรี โดยใช้การสำรวจแบบตามลำดับ (in-order traversal)

ทำให้การแปลงจากโปรแกรมของสถาปัตยกรรมเก่าไปยัง RISC-V ค่อนข้างตรงไปตรงมา การมี 32 รีจิสเตอร์ใน RISC-V เทียบกับ 16 ใน ARM-32 และ 8 ใน x86-32 ทำให้การถอดความไปยัง RISC-V ง่ายขึ้น ซึ่งจะยากกว่ามากในทิศทางตรงกันข้าม ขั้นตอนการถอดความคือ: ขั้นแรก ปรับส่วน prologue และ epilogue ของฟังก์ชัน จากนั้นเปลี่ยนการกระโดดแบบมีเงื่อนไขจากแบบรหัสเงื่อนไข (condition code oriented) เป็นคำสั่งเปรียบเทียบและกระโดด (compare-and-branch) และสุดท้าย แทนที่ชื่อเรจิสเตอร์และคำสั่งทั้งหมดด้วยเทียบเท่าใน RISC-V อาจมีการปรับแต่งเพิ่มเติม เช่น การจัดการค่าคงที่ขนาดใหญ่และที่อยู่ใน ISA x86-32 ที่มีความยาวแปรผัน หรือการเพิ่มคำสั่ง RISC-V เพื่อรองรับโหมดการกำหนดที่อยู่อันซับซ้อนหากใช้ในการถ่ายโอนข้อมูล แต่การทำตามสามขั้นตอนนี้จะทำให้เข้าใจผลลัพธ์แล้ว

```

addi sp, sp, -16 # จัดสรร stack frame
sw    s1, 4(sp)  # บันทึก s1
sw    s0, 8(sp)  # บันทึก s0
sw    ra, 12(sp) # บันทึก ra
li    s1, 0      # sum = 0
beqz  a0, .L1    # Skip loop if node == 0
mv    s0, a0     # s0 = node
.L3:
lw    a0, 0(s0)  # a0 = node->left
jal   tree_sum   # เรียกซ้ำ; result in 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    # Loop if node != 0
.L1:
mv    a0, s1     # Return sum in a0
lw    s1, 4(sp)  # คืน s1
lw    s0, 8(sp)  # คืน s0
lw    ra, 12(sp) # คืน ra
addi  sp, sp, 16 # กลับคืน stack frame
ret   # คืนค่า

```

รูปที่ B.5: โค้ด RV32I สำหรับการสำรวจทรีแบบตามลำดับ (in-order) ลูปหลักสั้นกว่าใน ISA อื่นๆ เนื่องจากคำสั่งเปรียบเทียบและกระโดด bnez

```

push {r4, r5, r6, lr} # บันทึก รีจิสเตอร์
mov  r5, #0           # sum = 0
subs r4, r0, #0       # r4 = node; node == 0?
beq  .L1              # Skip loop if so
.L3:
ldr  r0, [r4]         # r0 = node->left
bl   tree_sum         # เรียกซ้ำ; result in 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           # node == 0?
bne  .L3              # Loop if not
.L1:
mov  r0, r5           # Return sum in r0
pop  {r4, r5, r6, pc} # คืนค่ารีจิสเตอร์และส่งคืน

```

รูปที่ B.6: โค้ด ARM-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) คำสั่ง push และ pop แบบมัลติเวิร์ดช่วยลดขนาดโค้ดสำหรับ ARM-32 เมื่อเทียบกับ ISA อื่นๆ

```

push esi          # บันทึก esi
push ebx          # บันทึก ebx
xor esi, esi      # sum = 0
mov ebx, [esp+12] # ebx = node
test ebx, ebx     # node == 0?
je .L1           # Skip if so
.L3:
push [ebx]        # Load node->left; push to stack
call tree_sum     # เรียกซ้ำ; result in eax
pop edx           # Pop old arg and discard
add esi, [ebx+8]  # sum += node->value
mov ebx, [ebx+4]  # node = node->right
add esi, eax      # sum += eax
test ebx, ebx     # node == 0?
jne .L3          # Loop if not
.L1:
mov eax, esi      # Return sum in eax
pop ebx          # กู้คืน ebx
pop esi          # กู้คืน esi
ret              # คืนค่า

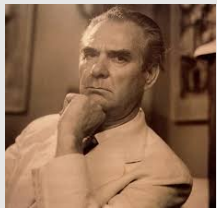
```

รูปที่ B.7: โค้ด x86-32 สำหรับการสำรวจทรีแบบตามลำดับ (in-order) ลูปหลักมีคำสั่ง push และ pop ที่ไม่มีในโปรแกรมของ ISA อื่นๆ ซึ่งสร้างการถ่ายโอนข้อมูลเพิ่มเติม

C

การแปลภาษาไทย

ศิลป์ พีระศรี (15 กันยายน พ.ศ. 2435–14 พฤษภาคม พ.ศ. 2505) ชื่อเดิม คอร์ราโด เฟโรซี เป็นชาวอิตาลี สัญชาติไทย ท่านเป็นปูชนียบุคคลคนหนึ่งของไทย โดยได้สร้างคุณูปการในทางศิลปะจนเป็นที่รู้จักกว้างขวาง ทั้งยังเป็นผู้ก่อตั้งมหาวิทยาลัยศิลปากร ได้รับการยกย่องเป็นปูชนียบุคคลของมหาวิทยาลัยแห่งนี้ มีผลงานที่โดดเด่นหลายอย่างในประเทศไทย



“ศิลปะยืนยาว ชีวิตสั้น” ที่มาจากภาษาละตินว่า “ars longa vita brevis” คำนี้หมายถึง ศิลปะหรือความรู้มีอายุยืนยาวเกินกว่าชีวิตของมนุษย์ ซึ่งสั้นนัก

—ศาสตราจารย์ ศิลป์ พีระศรี

C.1 บทนำ

ภาค ผนวกนี้ เป็นเวอร์ชันภาษาไทยเพื่อให้เป็นแหล่งอ้างอิง สำหรับการแปลหนังสือสถาปัตยกรรมชุดคำสั่ง RISC-V เล่มนี้ ผู้แปลคำนึงถึงความถูกต้องและแม่นยำทางเทคนิคที่สอดคล้องกับนิยามใน RISC-V และ IEEE 754-2019 จึงได้รวบรวมคำศัพท์เฉพาะทาง (technical terms) และคำศัพท์ด้านการเรียนรู้ (Learning Terms) ในลักษณะที่เข้าใจง่าย และรักษาความหมายทางเทคนิค คำศัพท์ใหม่ เช่น pipeline hazards หรือ strip-mining อาจไม่มีคำแปลที่ตายตัวในภาษาไทย ในทางปฏิบัติ วงการเทคโนโลยีและคอมพิวเตอร์มักใช้คำศัพท์ภาษาอังกฤษ เพื่อให้การสื่อสารชัดเจนและเป็นสากล เนื่องจากเป็นศัพท์เฉพาะที่ผู้อ่านเข้าใจตรงกัน แต่ภาษาไทยอาจต้องใช้คำขยายหรือวลีที่ยาวกว่าเพื่อให้สมบูรณ์ ภาคผนวกนี้ เน้นการอธิบายเหตุผลและความสำคัญของ RISC-V ในฐานะสถาปัตยกรรมชุดคำสั่งแบบเปิด ด้วยความหวังว่าผู้อ่าน จะสามารถเข้าใจถึงความสวยงาม และความเรียบง่ายขั้นพื้นฐาน ตามปณิธานที่หนังสือเล่มนี้เขียนไว้แต่แรก

C.2 RISC-V in a Nutshell

คำว่า “in a nutshell” เป็นสำนวนในภาษาอังกฤษที่แปลว่า “โดยย่อ” “อย่างกระชับ” หรือ “ใจความสำคัญ” หมายถึงการสรุปหรืออธิบายบางสิ่ง ที่ซับซ้อนให้เข้าใจง่ายและสั้นที่สุดเท่าที่จะเป็นไปได้ คำว่า “Nutshell” มาจากภาพของการพยายามใส่ข้อมูลจำนวนมากลงในเปลือกถั่ว (nutshell) ซึ่งมีขนาดเล็กมาก “คู่มือสรุปชุดคำสั่ง RISC-V : RISC-V in a Nutshell” สื่อถึงคู่มือที่ให้ภาพรวมของ RISC-V ISA ครอบคลุมใจความสำคัญ เข้าใจง่ายและกระชับ โดยสรุปแนวคิดที่ซับซ้อนให้อยู่ในรูปแบบที่ย่อและชัดเจน เพื่อให้ผู้อ่านเข้าใจพื้นฐานอย่างรวดเร็ว การสรุปสาระสำคัญของ สถาปัตยกรรมชุดคำสั่ง RISC-V รวมถึงส่วนขยาย เช่น Zfh, RV32F, RV32D และการทำงานของจุดลอยตัว (floating-point



ความเรียบง่าย



ประสิทธิภาพ



การเขียนโปรแกรม

operations) โดยไม่ต้องอ่านเอกสารเต็ม เช่น The RISC-V Instruction Set Manual เหมาะสำหรับผู้อ่านที่ต้องการเข้าใจภาพรวมของ RISC-V โดยไม่ต้องเจาะลึกในรายละเอียดทุกด้าน หนังสือเล่มเล็กนี้จึงเป็นคู่มือหรือเอกสารอ้างอิงสำหรับผู้เริ่มต้น หรือผู้ที่ต้องการทบทวนแนวคิดหลักอย่างรวดเร็ว โดยไม่ลงลึกในรายละเอียดทางเทคนิคที่ซับซ้อนเกินไป

C.3 RISC-V ง่ายเหมือน Pi

“RISC Five As Easy As Pi” เป็นคำกล่าวของ Paul Sherman นักพัฒนาซอฟต์แวร์และผู้เชี่ยวชาญด้านฮาร์ดแวร์ เขาใช้คำกล่าวนี้เป็นบทนำสำหรับผู้เริ่มต้นในการพัฒนา RISC-V เป็นการเปรียบเทียบที่ชาญฉลาดระหว่าง RISC-V และ Raspberry Pi โดยเน้นความเรียบง่าย การเข้าถึงได้ และความเหมาะสม สำหรับการพัฒนาและการเรียนรู้ในสภาพแวดล้อมที่ไม่ซับซ้อน โดย “RISC Five” เป็นการเล่นคำกับชื่อสถาปัตยกรรม RISC-V ที่ออกเสียงคล้ายกันระหว่าง “RISC Five” และ “Pi” เพื่อสื่อถึงลักษณะเด่นของ RISC-V ที่คล้ายคลึงกับ Raspberry Pi

คำว่า “Pi” ไม่เพียงแต่พ้องเสียงกับ “Five” แต่ยังดึงภาพลักษณ์ของ Raspberry Pi ซึ่งเป็นสัญลักษณ์ของแพลตฟอร์มที่ใช้งานง่ายและเป็นที่ยอมรับ บริบทที่ระบุถึงกระบวนการพัฒนา (compiling, assembling, linking, loading) และการใช้ JTAG บน Raspberry Pi แสดงให้เห็นว่า RISC-V สามารถพัฒนาและทดสอบได้ในสภาพแวดล้อมที่คุ้นเคยและไม่ซับซ้อน วลี “new thing on the block” เน้นย้ำว่า RISC-V เป็นนวัตกรรมใหม่ที่มีศักยภาพในการเปลี่ยนแปลงวงการ เช่นเดียวกับที่ Raspberry Pi ทำได้ในวงการคอมพิวเตอร์ขนาดเล็ก สื่อถึงความเป็นมิตรต่อผู้ใช้และชุมชน

Paul Sherman เขียนบทความนี้ (ในรูปแบบของ repository บน GitHub ชื่อ riscv-easy-as-pi) เพื่อให้ความรู้พื้นฐานที่เรียบง่ายและเข้าถึงได้ สำหรับผู้เริ่มต้นหรือผู้ที่สนใจฮาร์ดแวร์ โดยมุ่งเน้นที่กระบวนการ compiling, assembling, linking และ loading โปรแกรม RISC-V โดยใช้ Raspberry Pi เป็นเครื่องมือหลักในการพัฒนา คำกล่าวนี้เปรียบเทียบว่าการเริ่มต้นใช้งาน RISC-V ง่ายตายเหมือนกับการใช้ Raspberry Pi (ที่เป็นบอร์ดราคาถูกและคุ้นเคย) โดยไม่ต้องพึ่งพาเครื่องมือแพงหรือซับซ้อน เนื้อหาเป็นแนวปฏิบัติที่สนุกสนานและถ่ายทอดได้ (transferable) โดยใช้เครื่องมือพื้นฐานที่หาได้ง่าย เช่น Raspberry Pi ที่ “กำลังเก็บฝุ่นอยู่” ในบ้าน เพื่อช่วยเหลือผู้ที่ไม่มีประสบการณ์มากนัก โดยครอบคลุมตัวอย่างง่าย ๆ เช่น การกะพริบไฟ LED หรือแสดงข้อความ “Hello, World!” และอธิบายปัญหาที่พบบ่อย เช่น การตั้งค่า JTAG

บทความนี้ถูกนำเสนอครั้งแรกที่งาน Flash Memory Summit ในเดือนสิงหาคม 2022 ซึ่ง Paul Sherman ใช้โอกาสนี้เพื่อแบ่งปันความรู้ให้กับชุมชน โดยเน้นความเรียบง่ายของ RISC-V ที่เหมาะสำหรับทั้งมือใหม่และผู้เชี่ยวชาญด้านข้อมูล บทความนี้ได้เปรียบเทียบในหลายด้านของ RISC-V ที่คล้ายคลึงกับ Raspberry Pi ซึ่งเป็นจุดเด่นของทั้งสองแพลตฟอร์มดังนี้:



ความง่ายและการเข้าถึงได้ (Ease of Use and Accessibility)

- Raspberry Pi เป็นแพลตฟอร์มที่ออกแบบมาให้ใช้งานง่าย ราคาถูก และเข้าถึงได้สำหรับทุกคน ไม่ว่าจะเป็นนักเรียน นักพัฒนา หรือผู้ที่สนใจงาน DIY คำกล่าว “from the comfort of a leftover Raspberry Pi gathering dust” เน้นว่า Raspberry Pi เป็นอุปกรณ์ที่หาได้ง่าย แม้แต่เครื่องเก่าที่ไม่ได้ใช้งานก็สามารถนำมาใช้ได้
- RISC-V ถูกออกแบบให้เป็นสถาปัตยกรรมชุดคำสั่ง (ISA) ที่เรียบง่ายตามปรัชญา Reduced Instruction Set Computing (RISC) ซึ่งทำให้ง่ายต่อการเรียนรู้และพัฒนา คำกล่าว “As Easy As Pi” สื่อว่า RISC-V มีความเรียบง่ายในระดับที่ทุกคนสามารถเริ่มต้นใช้งานได้ เช่นเดียวกับ Raspberry Pi
- การเปรียบเทียบ: ผู้กล่าวเปรียบเทียบ RISC-V กับ Raspberry Pi เพื่อเน้นว่า RISC-V เป็น ISA ที่เข้าถึงง่ายและเหมาะสำหรับการพัฒนาในสภาพแวดล้อมที่ไม่ซับซ้อน เช่น การใช้ Raspberry Pi เก่าๆ เพื่อรันกระบวนการพัฒนา (compile, assemble, link, load) หรือทดสอบ JTAG



ความเรียบง่าย

การเป็นแพลตฟอร์มสำหรับการพัฒนาและการเรียนรู้ (Simplified Development Platform)

- Raspberry Pi เป็นแพลตฟอร์ม ยอด นิยม สำหรับการ เรียน รู้ กระบวนการพัฒนาซอฟต์แวร์ และฮาร์ดแวร์ เช่น การเขียนโปรแกรม การคอมไพล์ และการดีบั๊กผ่านเครื่องมืออย่าง JTAG คำกล่าวระบุถึงการใช้ Raspberry Pi ในการ “compiling, assembling, linking, and loading” ซึ่งเป็นขั้นตอนพื้นฐานในการพัฒนาโปรแกรม
- RISC-V ถูกออกแบบให้เป็น ISA ที่เหมาะสำหรับการพัฒนาและการเรียนรู้ โดยมีชุดคำสั่งที่เรียบง่ายและชัดเจน คำกล่าว “It deserves a more simplified development platform” บ่งชี้ว่า RISC-V ควรใช้แพลตฟอร์มที่เข้าถึงง่าย เช่น Raspberry Pi เพื่อให้การพัฒนาและการทดสอบทำได้สะดวก
- การเปรียบเทียบ: ผู้กล่าวใช้ Raspberry Pi เป็นตัวอย่างของแพลตฟอร์มที่เรียบง่ายและคุ้นเคย เพื่อแสดงว่า RISC-V สามารถพัฒนาและทดสอบได้บนแพลตฟอร์มที่ทุกคนเข้าถึงได้ โดยไม่ต้องใช้เครื่องมือหรืออุปกรณ์ราคาแพง

ความเป็นนวัตกรรมใหม่ (New Thing on the Block)

- Raspberry Pi ถือเป็นนวัตกรรมที่ปฏิวัติวงการคอมพิวเตอร์ขนาดเล็กเมื่อเปิดตัวในปี 2012 โดยนำเสนอแพลตฟอร์มที่ราคาถูกและทรงพลังสำหรับการเรียนรู้และพัฒนา
- คำว่า “new thing on the block” หมายถึง RISC-V ในฐานะ ISA ที่เป็นนวัตกรรมใหม่ ท้าทาย ISA แบบกรรมสิทธิ์ เช่น ARM และ x86 ด้วยความเป็น open-source และความยืดหยุ่น



ค่าใช้จ่าย

- การเปรียบเทียบ: ผู้กล่าวเปรียบเทียบ RISC-V กับ Raspberry Pi เพื่อเน้นว่า RISC-V เป็นสิ่งใหม่ที่มีศักยภาพในการเปลี่ยนแปลงวงการ ISA เช่นเดียวกับที่ Raspberry Pi เปลี่ยนวงการคอมพิวเตอร์ขนาดเล็ก

ความเป็น Open-Source และชุมชน (Openness and Community)

- Raspberry Pi สนับสนุนชุมชน open-source โดยมีซอฟต์แวร์ เอกสาร และเครื่องมือที่เปิดเผย ทำให้ชุมชนสามารถพัฒนาและปรับแต่งได้อย่างอิสระ
- RISC-V เป็น ISA แบบเปิดที่ไม่มีค่าลิขสิทธิ์ อนุญาตให้ทุกคนนำไปพัฒนาและปรับแต่งได้ คำกล่าวเน้นความง่ายในการใช้ RISC-V บนแพลตฟอร์มอย่าง Raspberry Pi ซึ่งเป็นสัญลักษณ์ของ open-source
- การเปรียบเทียบ: ผู้กล่าวเน้นว่า RISC-V ความเป็น open-source คล้ายกับ Raspberry Pi ทำให้เหมาะสำหรับชุมชนนักพัฒนาที่ต้องการทดลองและสร้างสรรค์ โดยไม่มีข้อจำกัดด้านลิขสิทธิ์



สง่างาม

การทดสอบและดีบั๊กด้วยเครื่องมือพื้นฐาน (JTAG and Simplified Debugging)

- คำกล่าวที่ว่า “RISC-V is the new thing on the block. We go through the basic process of compiling, assembling, linking, and loading” ระบุถึงการใช้ Raspberry Pi เพื่อทดสอบ JTAG ซึ่งเป็นวิธีการดีบั๊กและทดสอบฮาร์ดแวร์ที่ใช้งานง่ายและเข้าถึงได้ Raspberry Pi สามารถทำหน้าที่เป็นแพลตฟอร์มสำหรับ JTAG เพื่อเชื่อมต่อและทดสอบชิป RISC-V
- RISC-V ออกแบบให้เรียบง่าย ซึ่งเหมาะกับการดีบั๊กด้วยเครื่องมือพื้นฐานอย่าง JTAG คำกล่าวที่ว่า “describe the basics how JTAG works (and how it fails)” ชี้ให้เห็นว่า RISC-V สามารถทดสอบได้ในสภาพแวดล้อมที่ไม่ซับซ้อน
- การเปรียบเทียบ: ผู้กล่าวใช้ Raspberry Pi เป็นตัวอย่างของแพลตฟอร์มที่สามารถใช้ทดสอบและดีบั๊ก RISC-V ได้ง่ายๆ เช่นเดียวกับการใช้ JTAG เพื่อตรวจสอบการทำงานของชิป โดยไม่ต้องใช้เครื่องมือราคาแพง



การเขียนโปรแกรม



ประสิทธิภาพ

การลดความซับซ้อนในการพัฒนา (Simplified Development)

- Raspberry Pi เป็นแพลตฟอร์มที่ลดความซับซ้อนในการพัฒนา โดยให้เครื่องมือและซอฟต์แวร์ที่พร้อมใช้งาน เช่น GCC สำหรับคอมไพล์และลิงก์
- คำกล่าวที่ว่า “Reduced Instruction Set Computing is a simple thing. It deserves a more simplified development platform” เน้นว่า RISC-V เป็น ISA ที่เรียบง่ายและควรพัฒนาบนแพลตฟอร์มที่ไม่ซับซ้อน เช่น Raspberry Pi
- การเปรียบเทียบ: ผู้กล่าวเปรียบเทียบเพื่อแสดงว่า RISC-V สามารถพัฒนาได้บนแพลตฟอร์มที่เรียบง่ายและเป็นที่ยอมรับอย่าง Raspberry Pi ทำให้การพัฒนาและทดสอบ ISA นี้เป็นเรื่องง่ายสำหรับทุกคน

C.4 ทำไมต้องใช้ รหัสดำเนินการ (Opcode)?

รหัสดำเนินการ (Opcode) คือส่วนของคำสั่งเครื่อง (machine instruction) ในรูปแบบไบนารีที่ระบุการดำเนินการหลักของคำสั่งนั้น เช่น การบวก ลบ หรือการโหลดข้อมูล ซึ่งช่วยให้หน่วยประมวลผลกลาง (CPU) สามารถถอดรหัส (decode) และดำเนินการได้อย่างถูกต้อง Opcode มีความสำคัญอย่างยิ่งต่อ hardware efficiency ซึ่งหมายถึงประสิทธิภาพของฮาร์ดแวร์ในด้านการใช้พลังงานต่ำ การลดพื้นที่ชิป (die area) ความเร็วในการประมวลผลสูง และการลดความซับซ้อนของ logic circuits โดย opcode ช่วยเพิ่ม efficiency ดังนี้:

- **การถอดรหัสคำสั่งที่รวดเร็วและมีประสิทธิภาพ (Efficient Decoding):** Opcode มักถูกวางไว้ที่ตำแหน่งแรกหรือตำแหน่งสุดท้ายของคำสั่ง (เช่น bits 6:0 ใน RISC-V) ทำให้ CPU สามารถระบุประเภทคำสั่งได้ทันทีโดยไม่ต้องสแกนทั้งคำสั่ง ซึ่งลดจำนวน cycles ใน pipeline ลด latency และเพิ่ม throughput (จำนวนคำสั่งต่อหน่วยเวลา). ใน ISA ที่ opcode สั้นและสม่ำเสมอ (เช่น RISC architectures) ฮาร์ดแวร์ใช้ logic gates น้อยลงในการ decode เทียบกับ ISA ที่ opcode ยาวหรือ variable-length (เช่น x86) ซึ่งอาจต้องการ decoder ซับซ้อนและใช้พลังงานมากกว่า
- **การลดการใช้ทรัพยากรฮาร์ดแวร์ (Resource Optimization):** Opcode ที่ออกแบบดีช่วยจองพื้นที่ (opcode space) สำหรับคำสั่งและส่วนขยาย (extensions) อย่างมีประสิทธิภาพ ทำให้ไม่ต้องเพิ่มขนาดคำสั่งหรือจำนวน instructions ซึ่งลดพื้นที่ชิปและความซับซ้อนของ control unit (เช่น arithmetic logic unit: ALU) ที่ต้องจัดการ microoperations. ตัวอย่างเช่น ใน RISC-V opcode 7 บิต ช่วยให้ decoding ใช้ hardware น้อย ส่งผลให้ชิปประหยัดพลังงานและเหมาะสำหรับ embedded systems
- **การเพิ่มประสิทธิภาพโดยรวมของระบบ (System-Level Efficiency):** Opcode ช่วยให้ instruction format สม่ำเสมอ ซึ่งทำให้ cache และ memory access มีประสิทธิภาพสูงขึ้น (ลด cache misses) และสนับสนุน optimizations เช่น fused operations หรือ compressed instructions (เช่น RV32C ใน RISC-V) ซึ่งลดขนาดโค้ดและเพิ่ม instruction throughput. นอกจากนี้ opcode ที่ชัดเจนยังช่วยลดความเสี่ยง hazards ใน pipeline ลดการใช้พลังงานจาก stalls หรือ mispredictions

opcode เป็น "หัวใจ" ของการถอดรหัสคำสั่งที่รวดเร็วและทำให้ฮาร์ดแวร์ทำงานได้อย่างมีประสิทธิภาพ โดยลดความซับซ้อน ลดพลังงาน และเพิ่มความเร็ว หาก opcode ออกแบบไม่ดี (เช่น space ไม่พอหรือไม่สม่ำเสมอ) อาจนำไปสู่ฮาร์ดแวร์ที่ inefficient เช่น ใน ISA รุ่นเก่าบางตัวที่ต้องใช้ decoder ขนาดใหญ่.

รูปที่ C.1 Assembly Anatomy หมายถึงการ dissect หรือวิเคราะห์โครงสร้าง (anatomy) ของคำสั่งแอสเซมบลี ซึ่งแสดงให้เห็นส่วนประกอบต่างๆ ของคำสั่ง เพื่อให้

Address	Opcode	Mnemonic	Operands	Comments
0:	00450693	addi	a3,a0,4	# a3 ตัวชี้ไปที่ a[i]
4:	4705	c.li	a4,1	# (ขยายเป็น addi a4,x0,1) i = 1
Outer Loop:				
6:	00b76363	bltu	a4,a1,c	# ถ้า i < n กระโดดไปที่ Continue Outer loop
a:	8082	c.ret		# (ขยายเป็น jalr x0,ra,0) ออกจากฟังก์ชัน
Continue Outer Loop:				
c:	0006a803	lw	a6,0(a3)	# x = a[i]
10:	8636	c.mv	a2,a3	# (ขยายเป็น add a2,x0,a3) a2 ตัวชี้ไปที่ a[j]
12:	87ba	c.mv	a5,a4	# (ขยายเป็น add a5,x0,a4) j = i
InnerLoop:				
14:	ffc62883	lw	a7,-4(a2)	# a7 = a[j-1]
18:	01185763	ble	a7,a6,26	# ถ้า a[j-1] <= a[i] กระโดดไปที่ Exit InnerLoop
1c:	01162023	sw	a7,0(a2)	# a[j] = a[j-1]
20:	17fd	c.addi	a5,-1	# (ขยายเป็น addi a5,a5,-1) j--
22:	1671	c.addi	a2,-4	# (ขยายเป็น addi a2,a2,-4) ลด a2 ชี้ไปที่ a[j]
24:	fbe5	c.bnez	a5,14	# (ขยายเป็น bne a5,x0,14) ถ้า j != 0 กระโดดไปที่ InnerLoop
Exit InnerLoop:				
26:	078a	c.slli	a5,0x2	# (ขยายเป็น slli a5,a5,0x2) คูณ a5 by 4
28:	97aa	c.add	a5,a0	# (ขยายเป็น add a5,a5,a0) a5 = byte address of a[j]
2a:	0107a023	sw	a6,0(a5)	# a[j] = x

รูปที่ C.1: โครงสร้างโค้ดโปรแกรมแอสเซมบลี แสดงเลเบล (labels), คำอธิบายโค้ด (comments), ที่อยู่หน่วยความจำ (memory address), รหัสดำเนินการ (opcode), รหัสคำสั่ง (mnemonic), และตัวถูกดำเนินการ (operand) หมายถึง: รูปแบบการจัดเก็บข้อมูลแบบลำดับไบนารี (endianness) ของระบบที่ใช้จะกำหนดว่า ที่อยู่หน่วยความจำที่แสดงนั้นชี้ไปยังไบต์ที่อยู่ด้านขวาสุด (little-endian) หรือด้านซ้ายสุด (big-endian) ของรหัสคำสั่ง (opcode)

เข้าใจว่าคำสั่งเครื่องจักร (machine code) ถูกสร้างขึ้นอย่างไร โดยมักแสดงในรูปแบบ diagram หรือ bit layout (เช่น ในเอกสารหรือ cheat sheet) ในบริบท RISC-V Assembly Anatomy แสดง opcode เป็นส่วนแรกและสำคัญที่สุดที่ระบุ “operation” หรือประเภทการดำเนินการหลัก ดังนี้:

- Diagram Bit Layout: มักแสดง opcode เป็นช่องซ้ายสุด (ต่ำสุด) ที่ระบุความหมาย เช่น opcode 0110011 หมายถึง “OP” (arithmetic operations) จากนั้น funct fields ขยายความหมาย (e.g., funct3=000 + funct7=0000000 = ADD) ซึ่งช่วยให้เห็นว่า opcode เป็นตัวกำหนด “ประเภทหลัก” ก่อนที่ fields อื่นจะระบุรายละเอียด
- RISC-V Cheat Sheet หรือ Manual: เช่น จาก Project F หรือ Cornell CS3410, Anatomy แสดง opcode เป็นส่วนที่ทำให้ instruction เข้าใจง่าย โดย mnemonic ตรงกับ opcode + operands (e.g., “addi rd, rs1, imm” → opcode 0010011) เพื่ออธิบายกระบวนการ assembly-to-machine code.
- ความหมายที่แสดงถึง “operation semantics” เช่น การคำนวณ การเข้าถึงหน่วยความจำ หรือ control flow โดย Anatomy เน้นว่า opcode ช่วยป้องกัน ambiguity (ความคลุมเครือ) ในคำสั่ง และทำให้ debugging ง่ายเพราะสามารถ disassemble (แปลง machine code กลับเป็น assembly) ได้ตรง

โครงสร้างของรายการโปรแกรมแอสเซมบลี ทำให้เห็นวาระห์ดำเนินการ เป็นตัวกำหนดการทำงานของ mnemonic เช่น 0110011 อาจแปลเป็น add, sub, หรือคำสั่งอื่นขึ้นกับฟิลด์ funct3 และ funct7 โดย Opcode ระบุว่า CPU ต้องทำอะไร ในบริบท RISC-V รหัสดำเนินการจึงเป็นหัวใจของคำสั่งเครื่อง ที่กำหนดพฤติกรรมของคำสั่ง

C.5 ส่วนเติมเต็มสอง

ส่วนเติมเต็มสอง Two's Complement ในบริบทของสถาปัตยกรรม RISC-V เป็นวิธีการมาตรฐานที่ใช้ในการแทนเลขจำนวนเต็มที่มีเครื่องหมาย (signed integers) ในหน่วยความจำและเรจิสเตอร์ เพื่อให้สามารถจัดการทั้งเลขบวกและเลขลบได้อย่างมีประสิทธิภาพ Two's Complement เป็นวิธีการเข้ารหัสเลขจำนวนเต็มที่มีเครื่องหมายในระบบเลขฐานสอง โดยมีคุณสมบัติที่ทำให้การคำนวณทางคณิตศาสตร์ (เช่น การบวก การลบ) ง่ายและไม่ต้องใช้ฮาร์ดแวร์พิเศษสำหรับการจัดการเลขลบ หลักการสำคัญคือ:

- บิตเครื่องหมาย (sign bit): บิตสูงสุด (most significant bit) ระบุว่าเลขเป็นบวกหรือลบ
 - ถ้าเป็น 0 (บวกหรือศูนย์) บิตที่เพิ่มใหม่จะเติมด้วย 0
 - ถ้าเป็น 1 (ลบ) บิตที่เพิ่มใหม่จะเติมด้วย 1

- บิตเครื่องหมาย (sign bit): บิตสูงสุด (most significant bit) ระบุว่าเลขเป็นบวกหรือลบ

ใน RISC-V ซึ่งเป็นสถาปัตยกรรมที่ออกแบบมาให้เรียบง่ายและมีประสิทธิภาพ Two's Complement ถูกใช้ในทุกการดำเนินการเกี่ยวกับจำนวนเต็มที่มีเครื่องหมาย (signed integers) เช่น ในคำสั่ง **addi** rd, rs1, imm, **addiw** rd, rs1, imm, **sub** rd, rs1, rs2, และอื่นๆ ดังนี้:

- การแทนจำนวนในเรจิสเตอร์:
 - ใน RISC-V รูปแบบ RV32 (32 บิต) และ RV64 (64 บิต) ใช้ Two's Complement เพื่อเก็บเลขจำนวนเต็มที่มีเครื่องหมายในเรจิสเตอร์ (x0 ถึง x31)
 - ตัวอย่าง: ใน RV32 ค่าในเรจิสเตอร์ x5 ที่มีค่า 0xFFFFFFF B แทนเลข -5 ใน Two's Complement
- การจัดการค่า Immediate:
 - ในคำสั่งเช่น **addi** rd, rs1, imm ค่า immediate (มักเป็น 12 บิตหรือขนาดอื่น) จะถูกขยายเครื่องหมาย (sign-extended) เป็นขนาดของเรจิสเตอร์ (32 หรือ 64 บิต) โดยใช้ Two's Complement
 - ตัวอย่าง: immediate = 0xFFB (12 บิต, -5) ขยายเป็น 32 บิต = 0xFFFFFFF B (-5)

ความสำคัญของ Signed and Zero Extend ใน RISC-V Signed และ Zero Extend มีความสำคัญในเชิงเทคนิคและประสิทธิภาพดังนี้:

- **ความสม่ำเสมอ:** Two's Complement ใช้ในทุกคำสั่งที่เกี่ยวข้องกับจำนวนเต็มที่มีเครื่องหมาย ทำให้การออกแบบและการเขียนโปรแกรมง่ายขึ้น
- **การจัดการหน่วยความจำเสมือน:** การคำนวณที่อยู่ (address calculation) ในคำสั่งเช่น **lw** rd, offset(rs1) (load word) หรือ **sw** rs2, offset(rs1) (store word) ใช้ Two's Complement สำหรับค่า offset
- **ความยืดหยุ่นใน RV32 และ RV64:** Two's Complement รองรับการขยายขนาดเรจิสเตอร์และการคำนวณที่สอดคล้องกันในทุกสองโหมด

ใน RISC-V Two's Complement เป็นวิธีการหลักในการแทนเลขจำนวนเต็มที่มีเครื่องหมาย ช่วยให้การคำนวณเลขบวกและลบทำได้ง่ายโดยไม่ต้องใช้ฮาร์ดแวร์พิเศษ การขยายเครื่องหมายของค่า immediate การบวก/ลบ และการเปรียบเทียบล้วนอาศัย Two's Complement เพื่อรักษาความถูกต้องและประสิทธิภาพ การเข้าใจ Two's Complement จึงสำคัญสำหรับการเขียนโปรแกรมและการออกแบบระบบใน RISC-V โดยเฉพาะในคำสั่งที่จัดการจำนวนเต็ม เช่น **addi** rd, rs1, imm, **addiw** rd, rs1, imm, และ **slti** rd, rs1, rs2

C.6 การขยายบิตเครื่องหมาย

การขยายเครื่องหมาย (Sign-Extension) เป็นกระบวนการในระบบคอมพิวเตอร์ที่ใช้เพื่อรักษาความหมายของเครื่องหมาย (sign) ของตัวเลขของเลขในระบบ two's complement ใน RISC-V การขยายเครื่องหมายใช้ในคำสั่งที่มีค่า immediate เมื่อแปลงจากขนาดบิตที่เล็กกว่าไปเป็นขนาดบิตที่ใหญ่กว่า เช่น จาก 16 บิตไปเป็น 32 บิต หรือจาก 32 บิตไปเป็น 64 บิต โดยเฉพาะในบริบทของการคำนวณในสถาปัตยกรรม RISC-V เช่น คำสั่ง `addiw` หรือ `addi` ที่ใช้ค่า immediate ที่ขยายเครื่องหมาย

ขั้นตอนการขยายเครื่องหมาย:

- **ตรวจสอบบิตเครื่องหมาย** ที่ตำแหน่งบิตสูงสุด (most significant bit) ของตัวเลขขนาดเล็ก (immediate) ใน two's complement มีบิตเครื่องหมาย (sign bit)
 - ถ้าเป็น 0 (บวกหรือศูนย์) บิตที่เพิ่มใหม่จะเติมด้วย 0
 - ถ้าเป็น 1 (ลบ) บิตที่เพิ่มใหม่จะเติมด้วย 1
- **เติมบิตใหม่** เมื่อต้องการขยายขนาดของตัวเลข (เช่น จาก 12 บิตใน immediate ไปเป็น 32 บิตในเรจิสเตอร์) ขั้นตอนการขยายเครื่องหมาย :
 - คัดลอกบิตเครื่องหมายไปยังทุกบิตที่เพิ่มขึ้นในขนาดใหม่ เพื่อรักษาความหมายของตัวเลข (เลขบวกหรือลบ)
 - ส่วนบิตที่เหลือ (บิตของตัวเลขเดิม) จะยังคงเหมือนเดิม

ความสำคัญของ Signed and Zero Extend ใน RISC-V Signed และ Zero Extend มีความสำคัญในเชิงเทคนิคและประสิทธิภาพดังนี้:

- **รักษาความถูกต้องของข้อมูล (Data Integrity):** เมื่อโหลดข้อมูลขนาดเล็กจากหน่วยความจำ (ซึ่งมักจัดเก็บเป็น byte หรือ halfword เพื่อประหยัดพื้นที่) ต้องขยายให้เข้ากับขนาดรีจิสเตอร์ (XLEN) โดยไม่เปลี่ยนความหมาย เช่น signed extend ป้องกันไม่ให้จำนวน negative กลายเป็น positive เมื่อขยาย ซึ่งจำเป็นสำหรับ arithmetic operations (เช่น การบวก signed integers) หรือ comparisons
- **รองรับการทำงานกับข้อมูลผสมขนาด (Mixed-Width Operations):** RISC-V รองรับ XLEN ที่หลากหลาย (32/64/128 บิต) และ extensions (เช่น RV64 สำหรับ 64 บิต) เทคนิคนี้ช่วยให้โค้ดเข้ากันได้ (compatibility) ระหว่าง 32 บิตและ 64 บิต เช่น ใน RV64 การโหลด byte ต้อง extend ให้เต็ม 64 บิตเพื่อหลีกเลี่ยง undefined behavior
- **เพิ่มประสิทธิภาพฮาร์ดแวร์และซอฟต์แวร์ (Efficiency):** ลดจำนวนคำสั่งที่ต้องใช้ เช่น แทนที่จะใช้ลูปหรือ mask เพื่อ extend ด้วยมือ คำสั่ง built-in (เช่น `sxt.b`)

ทำได้ใน cycle เดียว ลด code size และพลังงาน เหมาะสำหรับ embedded systems ที่ทรัพยากรจำกัด นอกจากนี้ ยังช่วย optimize compiler (เช่น GCC สำหรับ RISC-V) ให้สร้างโค้ดที่มีประสิทธิภาพสูงขึ้น

- **เพิ่มประสิทธิภาพฮาร์ดแวร์และซอฟต์แวร์ (Efficiency):** ลดจำนวนคำสั่งที่ต้องใช้ เช่น แทนที่จะใช้ลูปหรือ mask เพื่อ extend ด้วยมือ คำสั่ง built-in (เช่น sext.b) ทำได้ใน cycle เดียว ลด code size และพลังงาน เหมาะสำหรับ embedded systems ที่ทรัพยากรจำกัด นอกจากนี้ ยังช่วย optimize compiler (เช่น GCC สำหรับ RISC-V) ให้สร้างโค้ดที่มีประสิทธิภาพสูงขึ้น
- **ป้องกันข้อผิดพลาดและความปลอดภัย (Error Prevention and Security):** ใน signed operations การ extend ผิดอาจนำไปสู่ overflow หรือ buffer overflows ซึ่งอันตรายในระบบจริง (e.g., security vulnerabilities) Extend ที่ถูกต้องช่วยรักษา semantics ของภาษาโปรแกรมสูงระดับ (เช่น C/C++) เมื่อ compile ลง RISC-V
- **ความยืดหยุ่นใน ISA แบบ Modular:** ใน RISC-V ซึ่งเป็น open ISA: เทคนิคนี้ถูกนำมาใช้ใน extensions เช่น B (Bit Manipulation) เพื่อขยายฐาน ISA โดยไม่เพิ่มความซับซ้อน ช่วยให้ RISC-V รองรับ applications หลากหลาย เช่น cryptography (ที่ต้องจัดการบิตแม่นยำ) หรือ data processing

ใน RISC-V การขยายเครื่องหมาย ถูกจัดการโดยฮาร์ดแวร์ในขั้นตอน Decode หรือ Execute ของไปป์ไลน์ ทำให้ไม่ต้องเพิ่มคำสั่งพิเศษ การออกแบบที่เรียบง่ายของ RV32I เทคนิคเหล่านี้ถูกนำมาใช้ในคำสั่ง load (เช่น lb สำหรับ signed extend byte, lbu สำหรับ zero extend byte) ทำให้ Sign Extension เป็นส่วนหนึ่งของคำสั่งมาตรฐาน ลดจำนวนคำสั่งพิเศษและเพิ่มความสม่ำเสมอ ทำให้การขยายเครื่องหมายรวดเร็วและไม่เพิ่มความซับซ้อนมากนัก

C.7 การจัดเรียงลำดับไบนารี

การจัดเรียงลำดับไบนารี (Endianness) คือวิธีการกำหนดลำดับไบนารีของข้อมูลในหน่วยความจำ โดย Big-Endian เริ่มจาก MSB ที่ที่อยู่ต่ำสุด และ Little-Endian เริ่มจาก LSB ที่ที่อยู่ต่ำสุด ซึ่งส่งผลต่อการแสดงที่อยู่ในหน่วยความจำและการเข้าถึงข้อมูล การทำความเข้าใจและจัดการกับ Endianness ให้เหมาะสมจึงเป็นสิ่งสำคัญในการพัฒนาซอฟต์แวร์และการสื่อสารข้อมูลในระบบคอมพิวเตอร์ ในระบบ Big-Endian การเข้าถึงข้อมูลจะเริ่มจากไบนารีที่มีน้ำหนักมากที่สุด (MSB) ก่อน ซึ่งสอดคล้องกับลำดับการเขียนตัวเลขตามปกติ (จากซ้ายไปขวา) หรืออ่านแบบคนไทยอ่านหนังสือในระบบ Little-Endian การเข้าถึงข้อมูลจะเริ่มจากไบนารีที่มีน้ำหนักน้อยที่สุด (LSB) ก่อน ซึ่งอาจดูไม่เป็นธรรมชาติ แต่มีข้อดีในด้านการคำนวณทางคณิตศาสตร์บางประเภท เช่น การบวกเลขที่มีความยาวต่างกัน

ภาษาคอมพิวเตอร์จะมีโครงสร้างคล้ายกัน ขึ้นอยู่กับอุปกรณ์ฮาร์ดแวร์ที่ใช้ จำกัดภาษาระดับต่ำไว้เพียงคำสั่งพื้นฐานเพื่อให้สร้างได้ง่ายขึ้น ลดค่าใช้จ่าย และประหยัดทรัพยากรมากขึ้น แต่ยังคงทำงานอย่างมีประสิทธิภาพสูงสุด ภาษาที่คอมพิวเตอร์เข้าใจคือลำดับของเลข 0 และ 1 เราสามารถเรียงลำดับของตัวเลขจาก ซ้ายไปขวาหรือจากขวาไปซ้ายได้ ซึ่งการอ่านในแต่ละทิศทางสามารถแปลความหมายได้ต่างกัน ดังนั้นเราจึงมีการกำหนดการเข้าถึงตำแหน่งของไบต์ (byte addressing) เพื่อกำหนดลำดับของไบต์ (byte order) ในทางสถาปัตยกรรมคอมพิวเตอร์ คำว่า “เอนเดียน” (endian) หมายถึง วิธีที่คอมพิวเตอร์จัดเก็บไบต์ (“byte order” เป็นแนวคิดในระบบคอมพิวเตอร์ที่กำหนดลำดับการจัดเก็บไบต์ของข้อมูลที่มีขนาดหลายไบต์ (เช่น ค่าตัวเลข 16 บิต หรือ 32 บิต) ในหน่วยความจำ โดยในระบบดิจิทัล ข้อมูลที่มีขนาดใหญ่กว่าหนึ่งไบต์จะถูกแบ่งออกเป็นไบต์ย่อย ๆ และลำดับที่ไบต์เหล่านี้ถูกจัดเก็บจะขึ้นอยู่กับรูปแบบของ Endianness Endianness ส่งผลโดยตรงต่อความเข้ากันได้ (compatibility) ประสิทธิภาพ (performance) ความซับซ้อนของฮาร์ดแวร์ (hardware complexity) และ ecosystem ของซอฟต์แวร์ Endianness ไม่ใช่แค่เรื่องทางเทคนิคเล็กน้อย แต่เป็นการตัดสินใจพื้นฐานที่อาจกระทบต่อการนำ ISA ไปใช้งานในอุตสาหกรรมต่าง ๆ เช่น embedded systems, servers, หรือ devices ที่ต้องทำงานร่วมกับระบบอื่น ๆ

การเลือก Endianness (ลำดับการจัดเก็บไบต์ในหน่วยความจำสำหรับข้อมูลหลายไบต์ เช่น big-endian ที่เก็บไบต์สูงสุดก่อน หรือ little-endian ที่เก็บไบต์ต่ำสุดก่อน) มีผลกระทบอย่างมีนัยสำคัญต่อการออกแบบ Instruction Set Architecture (ISA) โดยสถาปนิกต้องพิจารณาเพื่อให้ ISA มีความยั่งยืนและใช้งานได้จริง ในบริบท RISC-V ซึ่งเป็น ISA แบบเปิดและ modular สถาปนิกเลือก little-endian เป็น default สำหรับทั้ง instructions และ data เพื่อความเข้ากันได้ แต่รองรับ big-endian สำหรับ data ใน non-standard variants หรือ runtime switch ใน privileged mode เพื่อความยืดหยุ่น ผลกระทบหลักต่อ ISA สามารถสรุปได้ดังนี้:

- **ด้านความเข้ากันได้ (Compatibility)** Endianness กระทบต่อการทำงานร่วมกับซอฟต์แวร์ ฮาร์ดแวร์ และ protocols ที่มีอยู่ เช่น ถ้า ISA เป็น big-endian แต่ ecosystem ส่วนใหญ่ (เช่น Linux, x86, ARM) เป็น little-endian จะทำให้ต้องแปลงข้อมูล (byte swapping) ซึ่งเพิ่ม overhead และเสี่ยงต่อ bugs ใน network (e.g., TCP/IP ที่ big-endian) หรือ file formats (e.g., JPEG, PNG). ใน RISC-V: เลือก little-endian เพื่อเข้ากันกับ ARM และ x86 ทำให้ porting ซอฟต์แวร์ง่ายขึ้น แต่สำหรับ legacy systems (เช่น จาก MIPS ซึ่ง big-endian) รองรับ big-endian สำหรับ data เพื่อไม่ให้ผู้ใช้ต้องปรับ code มาก.
- **ด้านประสิทธิภาพ (Performance)** Endianness ส่งผลต่อการเข้าถึงหน่วยความจำและ cache efficiency เช่น little-endian อาจเร็วขึ้นใน arithmetic operations (เพราะ align กับ addressing ใน CPU สมัยใหม่) แต่ big-endian อาจดี

กว่าใน network processing ที่ข้อมูลมาจาก big-endian sources ถ้าไม่ match ต้องมี swapping ซึ่งเพิ่ม cycles และใช้พลังงานมากขึ้น ใน RISC-V: Instructions เป็น little-endian เสมอเพื่อลด latency ใน fetch/decode แต่ data สามารถ switch ได้ runtime ใน privileged spec เพื่อ optimize สำหรับ workload เฉพาะ (e.g., big-endian สำหรับ compatibility กับ old code) ซึ่งช่วยลด performance penalty ใน embedded systems.

- **ด้านความซับซ้อนของฮาร์ดแวร์ (Hardware Complexity)** ถ้า ISA รองรับ bi-endian (ทั้งสองแบบ) ต้องเพิ่ม logic ใน CPU สำหรับ swapping และ control registers (e.g., CSRs ใน RISC-V) ซึ่งเพิ่มพื้นที่ชิป (die area) และพลังงาน แต่ถ้า fixed-endian (เช่น little-endian เท่านั้น) จะลด complexity และต้นทุนผลิต ใน RISC-V: ไม่รองรับ bi-endian สำหรับ instructions เพื่อหลีกเลี่ยง complexity ใน decoder แต่สำหรับ data รองรับ switch ผ่าน privileged ISA เพื่อ balance ระหว่าง simplicity และ flexibility ซึ่งช่วยลดพื้นที่ชิปประมาณ 5-10
- **ด้านการพัฒนาซอฟต์แวร์และ Ecosystem (Software Development and Ecosystem)** Endianness กระทบต่อ compiler, assembler และ libraries เช่น compiler ต้อง generate code สำหรับ conversion ถ้า endian ไม่ match ซึ่งเพิ่ม code size และ complexity ใน development นอกจากนี้ ยังกระทบต่อ portability ของ binary code ระหว่าง platforms ใน RISC-V: การเลือก little-endian ช่วยให้ ecosystem (เช่น GCC, LLVM, Linux kernel) เติบโตเร็วเพราะ port จาก x86/ARM ง่าย แต่สำหรับ big-endian ยังมี support ใน tools (e.g., cross-compilation) เพื่อ specific uses เช่นใน automotive หรือ legacy ports ซึ่งช่วยขยาย adoption ในอุตสาหกรรม.
- **ด้านความปลอดภัยและความยั่งยืน (Security and Sustainability)** Endian mismatch อาจนำไปสู่ vulnerabilities เช่น side-channel attacks ถ้า swapping ไม่ secure หรือ bugs ใน data parsing แต่ถ้าออกแบบดี จะช่วยให้ ISA ยั่งยืน (e.g., switchable endian สำหรับ future tech) ใน RISC-V: Little-endian ช่วยลด risks ใน security-sensitive apps (เช่น crypto) เพราะ consistent กับ standards แต่ big-endian support ช่วยให้ ISA ยั่งยืนสำหรับ markets ที่ยังใช้ big-endian (e.g., some embedded).

โดยรวม การเลือก Endianness เป็นการ trade-off ที่สถาปนิกต้องชั่งน้ำหนักตามเป้าหมายของ ISA เช่น ใน RISC-V ที่เน้น openness และ compatibility เลือก little-endian เพื่อแข่งขันกับ ARM/x86 แต่ยังยืดหยุ่นพอสำหรับ non-standard uses ซึ่งช่วยให้ RISC-V เติบโตใน ecosystem ทั่วโลก

C.8 รายการคำศัพท์ย่อ

ABI Application Binary Interface: อินเทอร์เฟซไบนารีสำหรับแอปพลิเคชันซึ่งกำหนดวิธีที่ซอฟต์แวร์โต้ตอบกับสถาปัตยกรรม RISC-V เช่น มาตรฐานการเรียกฟังก์ชันและบทบาทของรีจิสเตอร์

ALU Arithmetic Logic Unit: หน่วยคำนวณทางคณิตศาสตร์และตรรกะ

AMO Atomic Memory Operations: การดำเนินการหน่วยความจำอะตอมิก เป็นประเภทคำสั่งที่ดำเนินการอ่าน-แก้ไข-เขียนแบบอะตอมิกบนหน่วยความจำ โดยไม่มีการขัดจังหวะระหว่างการอ่านและเขียน

API Application Program Interface: เป็นวิธีสำหรับโปรแกรมคอมพิวเตอร์หรือส่วนประกอบตั้งแต่สองโปรแกรมขึ้นไปในการสื่อสารระหว่างกัน

ASICs Application-Specific Integrated Circuit: ชิปวงจรรวม (IC) เฉพาะการใช้งาน ซึ่งออกแบบมาเพื่อแอปพลิเคชันเฉพาะเจาะจง

BCD Binary Coded Decimal: เป็นคลาสของการเข้ารหัสไบนารีของเลขทศนิยม โดยแต่ละหลักจะแสดงด้วยจำนวนบิตคงที่ โดยปกติจะเป็นสี่หรือแปด

CISC Complex Instruction Set Computer: คอมพิวเตอร์ชุดคำสั่งซับซ้อน

CPI Cycles Per Instruction: จำนวนรอบสัญญาณนาฬิกาเฉลี่ยต่อคำสั่งสำหรับโปรแกรม

CSR Control and Status Registers: รีจิสเตอร์พิเศษสำหรับควบคุมระบบและสถานะ ใช้จัดการ traps, exceptions และสถานะระบบ

DAXPY Double-precision AX Plus Y: อัลกอริทึมสำหรับการคำนวณเวกเตอร์

DP Double Precision: รูปแบบ floating-point 64 บิต

SP Single Precision: รูปแบบ floating-point 32 บิต

ELF Executable and Linkable Format: รูปแบบไฟล์ปฏิบัติการและลิงก์ได้ ซึ่งเป็นมาตรฐานสำหรับไฟล์ออบเจกต์ และไฟล์ปฏิบัติการที่ใช้โดยลิงเกอร์ในระบบ Unix เช่น ไฟล์ .o เป็นอินพุตและ a.out เป็นเอาต์พุต

FPD field programmable devices: วงจรรวมที่มีตรรกะเชิงผสมและอุปกรณ์หน่วยความจำที่กำหนดได้โดยผู้ใช้ปลายทาง

FCSR Floating-Point Control and Status Registers: รีจิสเตอร์ควบคุมและสถานะจุดลอยตัว ซึ่งเก็บโหมดการปัดเศษแบบไดนามิก (FRM) และธงข้อผิดพลาดที่สะสม (FFLAGS) สอดคล้องกับมาตรฐาน IEEE 754-2008 และเข้าถึงด้วยคำสั่งเช่น FRCSR, FSCSR

FFLAGS Floating-Point Accrued Exceptions: ช่องใน FCSR ที่ติดตามข้อผิดพลาดสะสม เช่น NV (ไม่ถูกต้อง), DZ (หารด้วยศูนย์), OF (ล้นเกิน), UF (ต่ำเกิน), NX (ไม่แม่นยำ) เข้าถึงด้วย FRFLAGS, FSFLAGS

FRM Floating-Point Rounding Mode: ช่องใน FCSR ที่กำหนดพฤติกรรมการปัดเศษ เช่น RTE (ปัดไปหาเลขคู่ใกล้ที่สุด), RTZ (ปัดไปหาศูนย์), RDN (ปัดลง), RUP (ปัดขึ้น), RMM (ปัดไปหาเลขที่มีขนาดใหญ่มากใกล้ที่สุด) ตั้งค่าด้วย FSRM, FRRM

FPGAs Field-Programmable Gate Arrays: วงจรรวมที่กำหนดค่าได้ ซึ่งเป็นวงจรรวมที่สามารถปรับโปรแกรมได้หลังผลิต

FPUs Floating Point Units: เป็นส่วนหนึ่งของระบบคอมพิวเตอร์ที่ออกแบบมาเป็นพิเศษเพื่อดำเนินการกับจำนวนจุดทศนิยม

LR/SC Load Reserved/Store Conditional: คู่ของคำสั่งที่โหลดหน่วยความจำที่จองไว้ และบันทึกการจอง และเก็บหน่วยความจำแบบมีเงื่อนไขหากมีการจองอยู่ โดยคืนค่าเป็นศูนย์เมื่อสำเร็จหรือรหัสข้อผิดพลาดที่ไม่ใช่ศูนย์ในกรณีอื่น

LMUL Length Multiplier: ตัวคูณความยาวเวกเตอร์ ใน RV32V

GDB GNU Debugger: เครื่องมือดีบั๊ก

GP Global Pointer: ตัวชี้ส่วนกลาง

PC Program Counter: รีจิสเตอร์ที่เก็บที่อยู่ของคำสั่งถัดไปที่จะรัน

Hart Hardware thread: หน่วยประมวลผลแบบขนานใน CPU สำหรับจัดการ interrupts และ contexts

MFLOPS Mega floating-point operations per second: เป็นหน่วยวัดประสิทธิภาพของคอมพิวเตอร์ในด้านการคำนวณ ซึ่งมีประโยชน์ในด้านการคำนวณทางวิทยาศาสตร์ที่ต้องใช้การคำนวณจุดลอยตัว

NaN Not a Number: ค่าพิเศษจุดลอยตัวสำหรับผลลัพธ์ที่ไม่กำหนด (เช่น 0/0) แบ่งเป็น signaling NaN หรือ quiet NaN ตรวจสอบด้วยคำสั่ง FCLASS.S/D

IMM Immediate: ข้อมูลค่าคงที่

IPI Inter-Processor Interrupt: ประสานงานระหว่าง hart

Inf Infinity: ค่าพิเศษจุดลอยตัวสำหรับล้นเกินหรือหารด้วยศูนย์ (บวก +Inf หรือลบ -Inf) ตรวจสอบด้วยคำสั่ง FCLASS.S/D

ISA Instruction Set Architecture: สถาปัตยกรรมชุดคำสั่ง กำหนดกฎสำหรับคำสั่งและฮาร์ดแวร์ที่รองรับ

MARCHID Machine Architecture ID: รหัสสถาปัตยกรรมเครื่อง

MHARTID Machine Hardware Thread ID: ตัวระบุคอร์ที่กำลังดำเนินงาน

MMU Memory Management Unit: หน่วยจัดการหน่วยความจำ สำหรับ virtual memory

MIE Machine mode Interrupt Enable: เปิดใช้งานการขัดจังหวะ

MIP Machine mode Interrupt Pending: การขอขัดจังหวะรอดำเนินการ

NOP No Operation: คำสั่งที่ไม่ทำอะไร

PC Program Counter: รีจิสเตอร์ที่เก็บที่อยู่ของคำสั่งถัดไปที่จะรัน ขนาด 32 บิตใน RV32I และ 64 บิตใน RV64I

PIC Position Independent Code: โค้ดที่ไม่ขึ้นกับตำแหน่ง ซึ่งสามารถรันได้โดยไม่ต้องรู้ที่อยู่หน่วยความจำแน่นอน โดยใช้การกระโดดแบบสัมพันธ์กับ PC

PMP Physical Memory Protection: การป้องกันหน่วยความจำกายภาพ ซึ่งกำหนดสิทธิ์การเข้าถึงหน่วยความจำ ผ่านรีจิสเตอร์ pmpaddr และ pmpcfg เพื่อความปลอดภัยในโหมด M

PTE Page Table Entry: รายการตารางหน้า ซึ่งเก็บข้อมูลการแปลที่อยู่เสมือนเป็นกายภาพ รวมถึงสิทธิ์การเข้าถึง เช่น read/write/execute

RISC Reduced Instruction Set Computer: คอมพิวเตอร์ชุดคำสั่งลดลง

RISC-I Reduced Instruction Set Computer 1st major design: เป็นแนวทางการออกแบบหลักที่คิดค้นโดย เดวิด แพตเตอร์สัน ในปี 1980 ที่มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ มีชื่อเรียกว่า GOLD ซึ่งมีแนวโน้มว่าอาจได้รับแรงบันดาลใจจากแนวคิดของ จอห์น ค็อก และ IBM 801 ที่พัฒนาขึ้นก่อนหน้านั้นไม่กี่ปี

RISC-II Reduced Instruction Set Computer 2nd major design: โครงการที่พัฒนาโดยมหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ มีชื่อเรียกว่า BLUE

RISC-III Reduced Instruction Set Computer 3rd major design: โครงการที่พัฒนาโดยมหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ มีชื่อเรียกว่า SOAR (Smalltalk on a RISC) เพื่อปรับปรุงประสิทธิภาพการทำงานของ Smalltalk

RISC-IV Reduced Instruction Set Computer 4th major design: โครงการที่พัฒนาโดยมหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ มีชื่อเรียกว่า SPUR (Symbolic Processing Using RISCs) เพื่อสร้างเวิร์กสเตชันแบบหลายตัวประมวลผล ที่รันโปรแกรมทั่วไปและภาษา Lisp

RISC-V Reduced Instruction Set Computer 5th major design: เป็น ISA แบบโมดูลาร์ที่ออกแบบเพื่อความยืดหยุ่นและความเรียบง่าย ที่ใช้งานได้กับโปรเซสเซอร์ทุกขนาด พัฒนาขึ้นในปี 2010 โดย มหาวิทยาลัยแคลิฟอร์เนีย เบิร์กลีย์ เพื่อให้สามารถปรับแต่งและใช้งานได้ฟรี โดยไม่ต้องเสียค่าลิขสิทธิ์

ISA Instruction Set Architecture: สถาปัตยกรรมชุดคำสั่ง

ชุด

คำสั่ง ที่ กำหนด วิธี การ สื่อสาร ระหว่าง ซอฟต์แวร์ และ ฮาร์ดแวร์ ซึ่ง กำหนด คำ สั่ง ที่ โปรเซสเซอร์สามารถรันได้

RPM Revolution Per Minute: รอบการทำงานของคำสั่งต่อนาที

RVC RISC-V Compressed: ส่วนขยายคำสั่งย่อ 16 บิต ที่ map ไปยังคำสั่ง 32 บิตเดิม เพื่อลดขนาดโค้ด

RVGCV RISC-V General Compressed Vector: ชุดรวมฐาน + extensions หลัก (I, M, A, F, D, C, V)

SATP Supervisor Address Translation and Protection: การแปลและป้องกันที่อยู่ของ supervisor ซึ่งเป็น CSR ที่เก็บ root page table และโหมดการแปล (เช่น Sv39 สำหรับ 39-bit virtual address)

SBI Supervisor Binary Interface: ส่วนต่อประสานมาตรฐานในสถาปัตยกรรม RISC-V

SP Single Precision: รูปแบบ floating-point 32 บิต

SIMD Single Instruction Multiple Data: คำสั่งเดียวข้อมูลหลาย ซึ่งเป็นเทคนิคสำหรับการประมวลผลขนาน

SISD Single Instruction stream, Single Data stream: คำสั่งเดียวสตรีมข้อมูลเดียว

SPLD simple programmable logic device: อุปกรณ์ลอจิกแบบตั้งโปรแกรมอย่างง่าย

SP Single Precision: รูปแบบ floating-point 32 บิต

SEW Standard Element Width: ขนาดของแต่ละ element ใน vector (หน่วยบิต เช่น 8, 16, 32, 64) กำหนดรูปแบบข้อมูลใน VTYPE

TLB Translation Lookaside Buffer: แคชสำหรับการแปลที่อยู่เสมือนเป็นกายภาพอย่างรวดเร็ว เพื่อจัดการกับ page faults ในหน่วยความจำเสมือน

VLSI very-large-scale integrated: อุปกรณ์ที่มีทรานซิสเตอร์หลายแสนตัวถึงหลายล้านตัว คำสั่งที่ยาวมาก (VLIW) รูปแบบของ สถาปัตยกรรมชุดคำสั่ง ที่เรียกใช้การดำเนินการจำนวนมาก

LMUL Length Multiplier: ตัวคูณความยาวรีจิสเตอร์เวกเตอร์ใน RV32V

SEW Standard Element Width: ความกว้างสมาชิกเวกเตอร์ใน RV32V

vL Vector Length: จำนวนข้อมูลย่อยที่จะประมวลผลใน vector operation หนึ่งครั้ง กำหนดโดย CSR และสามารถเปลี่ยน dynamically

VMA Vector Mask: Mask register สำหรับ conditional execution (ใช้ v0 เป็น default) แต่ละ bit ควบคุมว่าประมวลผล element นั้นหรือไม่ (1=process, 0=don't)

LMUL (Vector) Length Multiplier: ตัวคูณสำหรับขนาด vector register (1, 2, 4, 8) เพื่อขยาย VLEN เป็น AVLEN; ใช้ vreggroup (vector register group) จากหลาย physical registers

VLEN Vector Length: ขนาด คงที่ ของ vector register แต่ละ ตัว ใน หน่วย บิต (implementation-defined, e.g., 128-65536 bits) ไม่รวม tail elements

VTYPE Vector Type: Control and Status Register (CSR) ที่เกี่ยวกับการกำหนดค่า vector เช่น SEW, LMUL, และ tail/mask policy ใช้คำสั่ง vsetvli เพื่อตั้งค่า

AVLEN Actual Vector Length: ขนาดจริงของ vector register หลังคูณด้วย LMUL (AVLEN = VLEN * LMUL) ใช้สำหรับคำนวณ VLMAX

XLEN eXecution LENgth: ความกว้างของรีจิสเตอร์จำนวนเต็มในหน่วยบิต (32 สำหรับ RV32I ,64 สำหรับ RV64I)

Zba Bit Manipulation Address Generation: การสร้างที่อยู่สำหรับการจัดการบิต ซึ่งรวมคำสั่งช่วยในการคำนวณที่อยู่หน่วยความจำ เช่น sh1add, sh2add, sh3add

Zbb Bit Manipulation Basic: การจัดการบิตพื้นฐาน ซึ่งรวมคำสั่งพื้นฐานเช่น andn, orn, xnor, clz, ctz, cpop, sext, zext

Zbc Bit Manipulation Carry-less Multiplication: การคูณแบบไม่มีแครี่สำหรับการจัดการบิต ซึ่งรวมคำสั่ง clmul, clmulh, clmulr สำหรับการคำนวณใน cryptography

Zbs Bit Manipulation Single-bit Instructions: คำสั่งจัดการบิตเดี่ยว ซึ่งรวมคำสั่งเช่น bclr, bclri, bext, bexti, binv, binvi, bset, bseti สำหรับการจัดตั้งค่า ลบ หรือสกัตบิตเดี่ยว

Zbkb Bit Manipulation for Cryptography (Basic): การจัดการบิตสำหรับ cryptography พื้นฐาน ซึ่งรวมคำสั่ง rotate และ bitwise operations เพิ่มเติมสำหรับความปลอดภัย

Zbkc Bit Manipulation for Cryptography (Carry-less): การจัดการบิตสำหรับ cryptography แบบไม่มีแครี่ ซึ่งขยาย Zbc สำหรับการคำนวณปลอดภัย

Zbks Bit Manipulation for Cryptography (Single-bit): การจัดการบิตสำหรับ cryptography แบบบิตเดี่ยว ซึ่งขยาย Zbs สำหรับการใช้งานใน cryptography

C.9 ความสอดคล้องของการแปล

Technical Terms:

- Modular ISA สถาปัตยกรรมชุดคำสั่งแบบโมดูลาร์
- Incremental ISA สถาปัตยกรรมชุดคำสั่งแบบเพิ่มคำสั่งทีละน้อย
- Backward Compatibility ความเข้ากันได้ย้อนหลัง
- Complex Instruction Set Computer (CISC) คอมพิวเตอร์ชุดคำสั่งซับซ้อน
- Embedded Systems ระบบฝังตัว
- RISC-V Foundation มูลนิธิ RISC-V
- Virtual Memory หน่วยความจำเสมือน
- Cost ต้นทุน
- Simplicity ความเรียบง่าย
- Performance ประสิทธิภาพ
- Isolation การแยกสถาปัตยกรรมนำไปใช้งาน
- growth ที่ว่างสำหรับส่วนขยาย
- size ขนาดโปรแกรม
- easy ง่ายต่อการเรียนรู้
- Extensibility ความสามารถในการขยาย
- Software Compatibility ความเข้ากันได้ของซอฟต์แวร์
- Binary Compatibility ความเข้ากันได้ไบนารี
- General-Purpose Registers เรจิสเตอร์อเนกประสงค์
- Hardwired to Zero ติดตั้งเป็นศูนย์
- Byte-Addressable Address Space พื้นที่ที่อยู่แบบไบต์
- Displacement Addressing การกำหนดที่อยู่แบบ displacement
- Sign-Extended Immediate ค่าทันทีแบบขยายเครื่องหมาย
- Arithmetic Instructions คำสั่งเลขคณิต

- Logical Instructions คำสั่งตรรกะ
- Shift Instructions คำสั่งเลื่อนบิต
- To Learn More เรียนรู้เพิ่มเติม
- Comparison Instructions
- Sign Extension การขยายสัญญาณ
- Zero Extension การขยายศูนย์
- Conditional Branches กระโดดแบบมีเงื่อนไข
- Unconditional Jumps กระโดดแบบไม่มีเงื่อนไข
- PC-Relative Addressingการกำหนดที่อยู่แบบ PC-relative
- Control and Status Registers (CSR)เรจิสเตอร์ควบคุมและสถานะ
- Performance Counters ตัวนับประสิทธิภาพ
- Misaligned Accesses การเข้าถึงที่ไม่ aligned
- Memory-Mapped I/O I/O แบบ memory-mapped
- Pseudoinstruction คำสั่งเทียม
- Calling Conventions มาตรฐานการเรียกฟังก์ชัน
- Application Binary Interface (ABI) อินเทอร์เฟซไบนารีแอปพลิเคชัน
- Position-Independent Code (PIC) โค้ดที่ไม่ขึ้นกับตำแหน่ง
- Function Callการเรียกฟังก์ชัน
- Calling Conventionมาตรฐานการเรียกฟังก์ชัน
- ABIABI (อินเทอร์เฟซไบนารีแอปพลิเคชัน)
- Register Usage การใช้งานเรจิสเตอร์
- Temporary Registersเรจิสเตอร์ชั่วคราว
- Saved Registersเรจิสเตอร์บันทึก
- Assembler Directives คำสั่ง assembler
- Static Linking การลิงก์แบบ static
- Dynamic Linking การลิงก์แบบ dynamic

- Shared Libraries ไลบรารีแชร์
- Position-Independent Code โค้ดที่ไม่ขึ้นกับตำแหน่ง
- Symbol Table ตาราง symbol
- Stack Pointer ตัวชี้สแต็ก
- Return Address ที่อยู่ส่งกลับ
- Single-Precision ความแม่นยำเดี่ยว
- Double-Precision ความแม่นยำคู่
- Floating-Point Registers รีจิสเตอร์จุดลอยตัว
- Rounding Modes โหมดการปัดเศษ
- Fused Multiply-Add การคูณ-บวกแบบผสม
- R4 Format รูปแบบ R4
- Comparison Operations การดำเนินการเปรียบเทียบ
- Conversion Instructions คำสั่งแปลง
- Sign-Injection การฉีดเครื่องหมาย
- Classify การจำแนกประเภท
- Dynamic Rounding Mode โหมดปัดเศษแบบไดนามิก
- Static Rounding Mode โหมดปัดเศษแบบคงที่
- Invalid Operation การดำเนินการไม่ถูกต้อง
- Divide by Zero หารด้วยศูนย์
- Underflow ต่ำกว่าขอบเขต
- Inexact ไม่แม่นยำ
- Atomic Memory Operations (AMOs) การดำเนินการหน่วยความจำอะตอมมิก
- Critical Section ส่วนวิกฤต
- Sequentially Consistent การสอดคล้องแบบลำดับ
- Memory Consistency Model แบบจำลองความสอดคล้องหน่วยความจำ
- Concluding Remarks ข้อสังเกตสรุป

- Dynamic Typing การพิมพ์แบบไดนามิก
- Vector Length (vl) ความยาวเวกเตอร์
- Vector Arithmetic เลขคณิตเวกเตอร์
- Masked Operations การดำเนินการแบบมาสก์
- Element Width ความกว้างองค์ประกอบ
- Vector Configuration การกำหนดค่าเวกเตอร์
- Vector Reduction การลดรูปเวกเตอร์
- Concluding Remarks ข้อสังเกตสรุป
- Stripmining การ stripmining
- Vector Memory Operations การดำเนินการหน่วยความจำเวกเตอร์
- Cray-Style Vectorization การทำเวกเตอร์แบบ Cray
- Scalable Vector Architecture สถาปัตยกรรมเวกเตอร์ที่ปรับขนาดได้
- Compiler Support การสนับสนุนคอมไพเลอร์
- Insertion Sort การเรียงแบบแทรก
- Privileged Architecture สถาปัตยกรรมสิทธิพิเศษ
- Machine Mode (M-mode) โหมดเครื่อง (M-mode)
- Supervisor Mode (S-mode) โหมดหัวหน้า (S-mode)
- User Mode (U-mode) โหมดผู้ใช้ (U-mode)
- Exception Handling การจัดการยกเว้น
- Virtual Memory หน่วยความจำเสมือน
- Page Table ตารางเพจ
- Address Translation การแปลที่อยู่
- External Interrupt อินเทอร์รัปต์ภายนอก
- XLEN: เพื่อแสดงถึงความกว้างของรีจิสเตอร์จำนวนเต็ม (integer register width) ซึ่งขึ้นอยู่กับรุ่นของสถาปัตยกรรม

- Open Instruction Set Architecture สถาปัตยกรรมชุดคำสั่งแบบเปิด
- RISC Philosophy ปรัชญา RISC
- CISC Comparison การเปรียบเทียบกับ CISC
- Instruction Simplicity ความเรียบง่ายของคำสั่ง
- Modularity ความเป็นโมดูล
- Extensibility ความสามารถในการขยาย
- Open Source Hardware ฮาร์ดแวร์แบบเปิด
- Cost Reduction การลดต้นทุน
- Innovation Enablement การส่งเสริมนวัตกรรม
- Community Collaboration การร่วมมือของชุมชน
- Teaching Tool เครื่องมือการสอน
- Historical Context บริบททางประวัติศาสตร์
- Design Trade-offs การแลกเปลี่ยนในการออกแบบ
- Performance Optimization การปรับปรุงประสิทธิภาพ
- Code Density ความหนาแน่นของโค้ด
- Backward Compatibility ความเข้ากันได้ย้อนหลัง
- Industry Adoption การยอมรับในอุตสาหกรรม
- Research Platform แพลตฟอร์มการวิจัย
- Simplified Execution Model แบบจำลองการประมวลผลที่เรียบง่าย
- Instruction Alignment การจัดเรียงคำสั่ง
- Scalability ความสามารถในการขยายขนาด
- Customization การปรับแต่ง
- Licensing Freedom อิสระภาพด้านไลเซนส์
- Ecosystem Development การพัฒนาระบบนิเวศ
- Academic Contribution การมีส่วนร่วมจากวิชาการ
- Hardware Accessibility การเข้าถึงฮาร์ดแวร์

- Simplified Design Process กระบวนการออกแบบที่เรียบง่าย
- Cross-Platform Compatibility ความเข้ากันได้ข้ามแพลตฟอร์ม
- Learning Curve เส้นโค้งการเรียนรู้ ความง่ายในการเรียนรู้ RISC-V
- Future-Proofing การเตรียมพร้อมสำหรับอนาคต
- Instruction Simplicity ความเรียบง่ายของคำสั่ง
- Modularity ความเป็นโมดูล
- Open Architecture สถาปัตยกรรมแบบเปิด
- Design Philosophy ปรัชญาการออกแบบ
- Sign-Extended Immediate ค่าทันทีแบบขยายเครื่องหมาย
- Pipelining Concept แนวคิดการทำงานแบบท่อ
- Instruction Encoding การเข้ารหัสคำสั่ง
- Memory Access การเข้าถึงหน่วยความจำ
- Control Flow การควบคุมการไหล
- Base ISA ชุดคำสั่งพื้นฐาน
- Sign Extension การขยายสัญญาณ
- Zero Extension การขยายศูนย์
- Extensibility ความสามารถในการขยาย
- Instruction Format รูปแบบคำสั่ง
- Orthogonality ความเป็นอิสระต่อกัน
- Performance Optimization การปรับปรุงประสิทธิภาพ
- Performance Counters ตัวนับประสิทธิภาพ
- Misaligned Accesses การเข้าถึงที่ไม่ aligned
- Pseudoinstruction คำสั่งเทียม
- Code Density ความหนาแน่นของโค้ด
- Instruction Alignment การจัดเรียงคำสั่ง
- Backward Compatibility ความเข้ากันได้ย้อนหลัง

- Instruction Decoding การถอดรหัสคำสั่ง
- Teaching Tool เครื่องมือการสอน
- Simplified Execution Model แบบจำลองการประมวลผลที่เรียบง่าย
- Calling Convention ข้อตกลงการเรียกใช้
- Pseudoinstruction คำสั่งเทียม
- Linker ลิงเกอร์
- Loader โหลดเดอร์
- Static Linking การลิงก์แบบคงที่
- Dynamic Linking การลิงก์แบบไดนามิก
- Caller-Saved Registers รีจิสเตอร์ที่ผู้เรียกบันทึก
- Callee-Saved Registers รีจิสเตอร์ที่ถูกเรียกบันทึก
- Function Prologue โพรล็อกฟังก์ชัน
- Function Epilogue เอพิล็อกฟังก์ชัน
- Position-Independent Code โค้ดที่ไม่ขึ้นกับตำแหน่ง
- Symbol Resolution การแก้ไขสัญลักษณ์
- Relocation การย้ายที่อยู่
- Assembler Directive คำสั่งนำทางแอสเซมบลอร์
- Machine Code โค้ดเครื่อง
- Object File ไฟล์ออบเจกต์
- Executable ไฟล์ที่รันได้
- Leaf Function ฟังก์ชันใบไม้
- Register Allocation การจัดสรรเรจิสเตอร์
- Stack Frame เฟรมสแต็ก
- Global Pointer พอยน์เตอร์โกลบอล
- Thread Pointer พอยน์เตอร์เธรด
- KISS Principle หลักการ KISS
- Application Binary Interface (ABI) อินเทอร์เฟซไบนารีแอปพลิเคชัน
- Linker Relaxation การผ่อนคลายลิงเกอร์
- Stub Function ฟังก์ชันสตับ
- Dynamic Linker ลิงเกอร์ไดนามิก

- Modular Extension ส่วนขยายแบบโมดูล
- Arithmetic Enhancement การปรับปรุงการคำนวณเลขคณิต
- Signed vs Unsigned Operations การดำเนินการแบบมีเครื่องหมายเทียบกับไม่มี
- Constant Division Optimization การปรับปรุงการหารด้วยค่าคงที่
- Reciprocal Approximation การประมาณค่าผกผัน
- ISA Comparison การเปรียบเทียบ ISA
- Architectural State Management การจัดการสถานะสถาปัตยกรรม
- Task Switching Efficiency ประสิทธิภาพการสลับงาน
- Instruction Opcode Mapping การแมปไค้ดโอเปอเรชันคำสั่ง
- Graphical Representation การแทนด้วยกราฟิก
- Code Example Analysis การวิเคราะห์ตัวอย่างไค้ด
- Numerical Analysis Reference การอ้างอิงวิเคราะห์เชิงตัวเลข
- Extension Utility ประโยชน์ของส่วนขยาย
- Hardware Flexibility ความยืดหยุ่นของฮาร์ดแวร์
- Cost-Effectiveness ความคุ้มค่าด้านต้นทุน
- Simplicity Principle หลักการความเรียบง่าย
- Embedded Application Suitability ความเหมาะสมกับแอปพลิเคชันฝังตัว
- Performance Trade-offs การแลกเปลี่ยนด้านประสิทธิภาพ
- Historical Evolution วิวัฒนาการทางประวัติศาสตร์
- Further Study Suggestions ข้อเสนอแนะการศึกษาต่อ
- IEEE 754 Compliance การสอดคล้องกับ IEEE 754
- Single Precision ความแม่นยำเดี่ยว
- Double Precision ความแม่นยำคู่
- Floating-Point Registers รีจิสเตอร์จุดลอยตัว
- Instruction Format Preservation การรักษาารูปแบบคำสั่ง
- Fused Multiply-Add การคูณ-บวกแบบผสม
- Rounding Modes โหมดการปัดเศษ
- Exception Handling การจัดการข้อยกเว้น
- Sign Injection การฉีดเครื่องหมาย
- Floating-Point Classification การจำแนกประเภทจุดลอยตัว

- Conversion Operations การดำเนินการแปลงค่า
- Benchmark Comparison การเปรียบเทียบ benchmark
- DAXPY Loop ลูป DAXPY
- Scalar vs Vector Operations การดำเนินการสเกลาร์เทียบเวกเตอร์
- Dynamic Instruction Count จำนวนคำสั่งแบบไดนามิก
- Code Size Efficiency ประสิทธิภาพขนาดโค้ด
- Architectural Simplicity ความเรียบง่ายทางสถาปัตยกรรม
- Register Bandwidth แบนด์วิดท์เรจิสเตอร์
- Precision Conversion การแปลงความแม่นยำ
- NaN Handling การจัดการ NaN
- Infinity Representation การแทนค่าอนันต์
- Subnormal Number รจำนวนย่อยปกติ
- Quiet NaN vs Signaling NaN NaN แบบเงียบเทียบสัญญาณ
- Control Status Register รีจิสเตอร์ควบคุมสถานะ
- Dynamic Rounding การปัดเศษแบบไดนามิก
- Static Rounding การปัดเศษแบบคงที่
- Performance Optimization การปรับปรุงประสิทธิภาพ
- ISA Comparison การเปรียบเทียบ ISA
- Historical Context บริบททางประวัติศาสตร์
- Operation การดำเนินการอะตอมมิก
- Synchronization การประสานงาน
- Concurrency การทำงานพร้อมกัน
- Race Condition สภาวะแข่งกัน
- Mutual Exclusion การยกเว้นซึ่งกันและกัน
- Load-Reserved โหลดแบบสงวน
- Store-Conditional สโตร์แบบเงื่อนไข
- Atomic Memory Operation การดำเนินการหน่วยความจำอะตอมมิก
- Memory Consistency ความสอดคล้องของหน่วยความจำ
- Relaxed Memory Model โมเดลหน่วยความจำแบบผ่อนคลาย
- Acquire Semantics ความหมายแบบรับมา

- Release Semantics ความหมายแบบปล่อยออก
- Compare-and-Swap เปรียบเทียบและสลับ
- Lock-Free Programming การเขียนโปรแกรมแบบไร้ล็อก
- Thread Safety ความปลอดภัยของเธรด
- Critical Section ส่วนวิกฤติ
- Shared Memory หน่วยความจำร่วม
- Multi-Threading การทำงานหลายเธรด
- Multi-Processing การทำงานหลายโปรเซสเซอร์
- Data Race การแข่งข้อมูล
- Indivisibility ความไม่สามารถแบ่งแยก
- Synchronization Primitive ตัวประสานงานพื้นฐาน
- Memory Ordering การเรียงลำดับหน่วยความจำ
- Reduction Operation การดำเนินการลดค่า
- I/O Device Communication การสื่อสารกับอุปกรณ์ I/O
- Reservation Validity ความถูกต้องของการสงวน
- Error Code Return การคืนค่ารหัสข้อผิดพลาด
- Universal Synchronization การประสานงานสากล
- Scalability ความสามารถในการขยายขนาด
- Performance Overhead ค่าใช้จ่ายด้านประสิทธิภาพ
- Software Synchronization การประสานงานด้วยซอฟต์แวร์
- Hardware Thread เธรดฮาร์ดแวร์
- Data Consistency ความสอดคล้องของข้อมูล
- Parallel Programming การเขียนโปรแกรมแบบขนาน
- Deadlock Avoidance การหลีกเลี่ยงเดดล็อก
- Atomic Counter เคาน์เตอร์อะตอมมิก
- Memory Barrier รั้วหน่วยความจำ
- Thread Synchronization การประสานเธรด
- Sub-Extension ส่วนขยายย่อย
- Design Philosophy ปรัชญาการออกแบบ
- Performance Improvement การปรับปรุงประสิทธิภาพ

- Code Density ความหนาแน่นของโค้ด
- Power Efficiency ประสิทธิภาพด้านพลังงาน
- Modularity ความเป็นโมดูล
- Bitwise Operations การดำเนินการแบบบิต
- Shift and Rotate การเลื่อนและหมุนบิต
- Bit Counting การนับบิต
- Bit Extraction การสกัดบิต
- Bit Insertion การแทรกบิต
- Bit Swapping การสลับบิต
- Carry-Less Multiplication การคูณแบบไร้แครรี
- Address Generation การสร้างที่อยู่
- Base Bit Manipulation การจัดการบิตพื้นฐาน
- Shifter Funnel ช่องทางเลื่อนบิต
- Single-Bit Operations การดำเนินการบิตเดี่ยว
- ISA Comparison การเปรียบเทียบ ISA
- Cryptographic Applications แอปพลิเคชันเข้ารหัส
- Embedded Systems Suitability ความเหมาะสมกับระบบฝังตัว
- Historical Evolution วิวัฒนาการทางประวัติศาสตร์
- Optimization Techniques เทคนิคการปรับปรุง
- Bit Field Operations การดำเนินการฟิลด์บิต
- Endian Conversion การแปลง endian
- Checksum Calculation การคำนวณ checksum
- Parallel Bit Operations การดำเนินการบิตแบบขนาน
- Register Utilization การใช้รีจิสเตอร์
- Extension Ratification การรับรองส่วนขยาย
- Compatibility ความเข้ากันได้
- Benchmarking การทดสอบ benchmark
- Code Size Reduction การลดขนาดโค้ด
- Energy Consumption การใช้พลังงาน
- General-Purpose Computing การคำนวณทั่วไป

- Specialized Instructions คำสั่งเฉพาะทาง
- Open Standard มาตรฐานแบบเปิด
- Community Contribution การมีส่วนร่วมจากชุมชน
- Educational Value คุณค่าทางการศึกษา
- Vector Predicate Registers รีจิสเตอร์ตัวกำหนดเวกเตอร์
- Vector Length Register รีจิสเตอร์ความยาวเวกเตอร์
- Maximum Vector Length ความยาวเวกเตอร์สูงสุด
- Dynamic Typing การพิมพ์แบบไดนามิก
- Data-Level Parallelism ความสมมาตรในระดับข้อมูล
- SIMD Comparison การเปรียบเทียบ SIMD
- Vector Computation การคำนวณเวกเตอร์
- Vector Loads การโหลดเวกเตอร์
- Vector Stores การเก็บเวกเตอร์
- Strided Loads การโหลดแบบก้าว
- Indexed Loads การโหลดแบบดัชนี
- Strided Stores การเก็บแบบก้าว
- Indexed Stores การเก็บแบบดัชนี
- Conditional Execution การดำเนินการแบบมีเงื่อนไข
- Parallelism ความสมมาตร
- Vector Arithmetic การคำนวณเลขคณิตเวกเตอร์
- Vector Logical Operations การดำเนินการตรรกะเวกเตอร์
- Predicate Operations การดำเนินการตัวกำหนด
- Set Vector Length การกำหนดความยาวเวกเตอร์
- Vector Configuration การกำหนดค่าเวกเตอร์
- Vector Type Register รีจิสเตอร์ประเภทเวกเตอร์
- Vector Start Index ดัชนีเริ่มต้นเวกเตอร์
- Chaining การเชื่อมต่อ
- Reduction Operations การดำเนินการลดค่า
- Masking การปกปิด
- Polymorphism ความหลากหลายรูปแบบ

- Scalability ความสามารถในการขยายขนาด
- Portability ความสามารถในการพกพา
- Vector Memory Access การเข้าถึงหน่วยความจำเวกเตอร์
- Unit-Stride ก้าวหน่วย
- Constant-Stride ก้าวคงที่
- Indexed Access การเข้าถึงแบบดัชนี
- Vector Permutation การสลับตำแหน่งเวกเตอร์
- Slide Operations การดำเนินการเลื่อน
- Merge Operations การดำเนินการรวม
- Compress Operations การดำเนินการบีบอัด
- Expand Operations การดำเนินการขยาย
- Vector Floating-Point จุดลอยตัวเวกเตอร์
- 64-bit Address Space พื้นที่ที่อยู่ 64 บิต
- RV64 ISA Extension ส่วนขยาย ISA RV64
- Register Width Expansion การขยายความกว้างรีจิสเตอร์
- Default Data Size ขนาดข้อมูลเริ่มต้น
- Word-Level Operations การดำเนินการระดับคำ
- Doubleword Operations การดำเนินการดับเบิลเวิร์ด
- Sign Extension การขยายเครื่องหมาย
- Zero Extension การขยายศูนย์
- Truncation การตัดทอน
- Backward Compatibility ความเข้ากันได้ย้อนหลัง
- Modularity in ISA ความเป็นโมดูลใน ISA
- Performance in Large Data ประสิทธิภาพกับข้อมูลขนาดใหญ่
- Address Width Concern ความกังวลเรื่องความกว้างที่อยู่
- Instruction Encoding Consistency ความสอดคล้องในการเข้ารหัสคำสั่ง
- Mixed Data Size Handling การจัดการขนาดข้อมูลผสม
- Atomic 64-bit Operations การดำเนินการอะตอมมิก 64 บิต
- Design Simplicity ความเรียบง่ายในการออกแบบ
- ISA Comparison การเปรียบเทียบ ISA

- Large Memory Support การรองรับหน่วยความจำขนาดใหญ่
- Data Type Flexibility ความยืดหยุ่นประเภทข้อมูล
- Rounding Towards Zero การปัดเศษไปยังศูนย์
- Remainder Computation การคำนวณเศษ
- Shift Operations Extension การขยายการดำเนินการเลื่อน
- Code Portability ความสามารถในการพกพาโค้ด
- Hardware Efficiency ประสิทธิภาพฮาร์ดแวร์
- Future-Proofing การเตรียมพร้อมสำหรับอนาคต
- Instruction Set Modularity ความเป็นโมดูลของชุดคำสั่ง
- Compatibility Layer r^u ชั้นความเข้ากันได้
- Address Translation การแปลที่อยู่
- Data Alignment การจัดเรียงข้อมูล
- Overflow Handling การจัดการโอเวอร์โฟลว์
- Extension Philosophy ปรัชญาการขยาย
- Benchmark Comparison การเปรียบเทียบ benchmark
- Code Size Impact ผลกระทบต่อขนาดโค้ด
- Power Consumption Reduction การลดการใช้พลังงาน
- Server Application Suitability ความเหมาะสมกับแอปพลิเคชันเซิร์ฟเวอร์
- Historical Address Limitations ข้อจำกัดที่อยู่ทางประวัติศาสตร์
- Incremental Design การออกแบบแบบเพิ่มทีละส่วน
- Open Architecture Benefits ประโยชน์ของสถาปัตยกรรมแบบเปิด
- Educational ISA Extension การขยาย ISA ทางการศึกษา
- Privileged Architecture สถาปัตยกรรมสิทธิพิเศษ
- Privilege Levels ระดับสิทธิพิเศษ
- Machine Mode (M-mode) โหมดเครื่อง (M-mode)
- Supervisor Mode (S-mode) โหมดผู้ดูแล (S-mode)
- User Mode (U-mode) โหมดผู้ใช้ (U-mode)
- Hypervisor Mode (HS-mode) โหมดไฮเปอร์ไวเซอร์ (HS-mode)
- Control and Status Registers (CSRs) รีจิสเตอร์ควบคุมและสถานะ (CSRs)
- Trap กับดัก

- Exception ข้อยกเว้น
- Interrupt การขัดจังหวะ
- Trap Vector เวกเตอร์กับดัก
- Trap Handler ตัวจัดการกับดัก
- Mode Transition การเปลี่ยนโหมด
- Return from Trap การคืนจากกับดัก
- Memory Protection การป้องกันหน่วยความจำ
- Virtual Memory หน่วยความจำเสมือน
- Page Table ตารางหน้า
- Supervisor Address Translation and Protection (SATP) การแปลและป้องกันที่อยู่ผู้ดูแล (SATP)
- Machine Status Register (mstatus) รีจิสเตอร์สถานะเครื่อง (mstatus)
- Supervisor Status Register (sstatus) รีจิสเตอร์สถานะผู้ดูแล (sstatus)
- Machine Trap Vector (mtvec) เวกเตอร์กับดักเครื่อง (mtvec)
- Supervisor Trap Vector (stvec) เวกเตอร์กับดักผู้ดูแล (stvec)
- Machine Exception Program Counter (mepc) เคาน์เตอร์โปรแกรมข้อยกเว้นเครื่อง (mepc)
- Supervisor Exception Program Counter (sepc) เคาน์เตอร์โปรแกรมข้อยกเว้นผู้ดูแล
- Machine Cause Register (mcause) รีจิสเตอร์สาเหตุเครื่อง (mcause)
- Supervisor Cause Register (scause) รีจิสเตอร์สาเหตุผู้ดูแล (scause)
- Machine Scratch Register (mscratch) รีจิสเตอร์ชั่วคราวเครื่อง (mscratch)
- Supervisor Scratch Register (sscratch) รีจิสเตอร์ชั่วคราวผู้ดูแล (sscratch)
- Interrupt Enable การเปิดใช้งานขัดจังหวะ
- Previous Privilege Mode โหมดสิทธิพิเศษก่อนหน้า
- CSR Access Instructions คำสั่งเข้าถึง CSR
- Atomic CSR Operations การดำเนินการ CSR แบบอะตอมมิก
- Security Isolation การแยกความปลอดภัย
- Virtualization Support การรองรับ virtualization
- Timer Interrupt การขัดจังหวะตัวจับเวลา
- External Interrupt การขัดจังหวะภายนอก

- Software Interrupt การขัดจังหวะซอฟต์แวร์
- Environment Call (ecall) การเรียกสภาพแวดล้อม (ecall)
- Breakpoint (ebreak) จุดหยุด (ebreak)
- Fence Instruction คำสั่งรั้ว
- Physical Memory Protection (PMP) การป้องกันหน่วยความจำกายภาพ (PMP)
- Supervisor Fence Virtual Memory (sfence.vma) รั้วผู้ดูแลหน่วยความจำเสมือน (sfence.vma)
- Machine Interrupt Enable (mie) การเปิดใช้งานขัดจังหวะเครื่อง
- Supervisor Interrupt Enable (sie) การเปิดใช้งานขัดจังหวะผู้ดูแล
- Delegation การมอบหมาย
- absolute address ที่อยู่จริงของตัวแปรหรือรูทีนในหน่วยความจำ
- access bit บิตการเข้าถึง เรียกอีกอย่างว่าบิตใช้ หรือบิตอ้างอิงฟิลด์ ที่ตั้งค่าทุกครั้งที่มีการเข้าถึงเพจ และใช้เพื่อนำ LRU ไปใช้ งานหรือโครงสร้างการแทนที่อื่นๆ
- active matrix display การแปลที่อยู่ เรียกอีกอย่างว่าการทำแผนที่ที่อยู่ กระบวนการที่แมปที่อยู่เหมือนกับที่อยู่ที่ใช้ในการเข้าถึงหน่วยความจำในหนังสือ The RISC-V Reader : An Open Architecture Atlas แสดงคำศัพท์ Technical Terms ของบทที่ 10 : RV32/64 Privileged Architecture ที่เหมาะสมเป็นคำศัพท์ Translator's Concordance จำนวน 40 คำ
- address mode โหมดการกำหนดตำแหน่งที่อยู่
- Amdahl's Law กฎของ Amdahl กฎที่ระบุว่า การเพิ่มประสิทธิภาพที่เป็นไปได้ ด้วยการปรับปรุงที่กำหนดนั้น ถูกจำกัดด้วยจำนวนที่ใช้คุณลักษณะที่ปรับปรุงแล้ว เป็นเวอร์ชันเชิงปริมาณของกฎว่าด้วยผลตอบแทนที่ลดลง
- architectural registers รีจิสเตอร์ที่มองเห็นได้ของโปรเซสเซอร์
- assembler โปรแกรมที่แปลเวอร์ชันสัญลักษณ์ของคำสั่งเป็นเวอร์ชันไบนารี
- assembler directive การดำเนินการที่บอกแอสเซมเบลอร์ถึงวิธีการแปลโปรแกรม แต่ไม่สร้างคำสั่ง เครื่องเริ่มต้นด้วยจุดเสมอ
- benchmark โปรแกรมที่เลือกใช้ในการเปรียบเทียบประสิทธิภาพของคอมพิวเตอร์
- binary digit เรียกอีกอย่างว่าบิตไบนารี หนึ่งในสองตัวเลขในฐาน 2, 0 หรือ 1 ซึ่งเป็นส่วนประกอบของข้อมูล
- branch address table ตารางตำแหน่งที่อยู่ของลำดับคำสั่งทางเลือก
- branch prediction วิธีการแก้ไขอันตรายของการบรานช์ ที่ถือว่าผลลัพธ์ที่กำหนดสำหรับบรานช์แบบมีเงื่อนไขและดำเนินการจากสมมติฐานนั้น แทนที่จะรอเพื่อยืนยันผลลัพธ์จริง

- bus ชุดของสายข้อมูล ที่ถือว่าเป็นสัญญาณลอจิกเดียว
- callee โพรซีเดอร์ที่รับชุดคำสั่งที่เก็บไว้ตามพารามิเตอร์ที่ผู้เรียกให้มา
- callee-saved register รีจิสเตอร์ที่บันทึกโดยรูทีนในการเรียกโพรซีเดอร์
- caller โปรแกรมที่กระตุ้น โพรซีเดอร์และจัดเตรียมค่าพารามิเตอร์ที่จำเป็น
- caller-saved register รีจิสเตอร์ที่บันทึกผู้โทร รีจิสเตอร์ที่บันทึกโดยรูทีนที่ถูกเรียก
- clock cycle เวลาสำหรับ ช่วงสัญญาณ นาฬิกา ในช่วงโดยปกติของ นาฬิกาของ โปรเซสเซอร์
- clock cycles per instruction (CPI) รอบสัญญาณนาฬิกาต่อคำสั่ง (CPI) จำนวนรอบสัญญาณนาฬิกาเฉลี่ยต่อคำสั่งสำหรับโปรแกรมหรือส่วนย่อยของโปรแกรม
- clock period ความยาวของรอบนาฬิกาแต่ละรอบ
- die ส่วนสี่เหลี่ยมแต่ละส่วน ซึ่งถูกตัดจาก แผ่นเวเฟอร์ หรือที่รู้จักกันอย่างไม่เป็นทางการว่าชิป
- double precision ความแม่นยำสองเท่าของค่าจุดทศนิยมที่แสดงในดับเบิลเวิร์ด 64 บิต
- doubleword หน่วยของการเข้าถึงข้อมูลในคอมพิวเตอร์ปกติแล้วจะเป็นกลุ่ม 64 บิต
- dynamic branch prediction การทำนายบรานซ์ที่รันใหม่โดยใช้ข้อมูลรันใหม่
- dynamically linked libraries (DLLs) ไลบรารีที่เชื่อมโยงแบบไดนามิก (DLLs) รูทีนของไลบรารีที่เชื่อมโยงกับโปรแกรมระหว่างการดำเนินการ
-
- data-level parallelism ความขนานระดับข้อมูลความขนานที่ทำได้โดยการ ดำเนินการเดียวกันกับข้อมูลอิสระ
- datapath ส่วนประกอบของโปรเซสเซอร์ที่ทำการคำนวณทางคณิตศาสตร์
- datapath element องค์ประกอบ datapath หน่วยที่ใช้ในการดำเนินการหรือเก็บข้อมูลภายในตัวประมวลผล
- decoder ตัวถอดรหัสบิตลอจิกที่มีอินพุตแบบ n-bit และเอาต์พุต 2n โดยจะมีการยืนยันเอาต์พุตเพียงรายการเดียวสำหรับชุดอินพุตและชุดข้อบกพร่อง
- delayed branch ประเภทของบรานซ์ที่ดำเนินการคำสั่งทันที โดยไม่ขึ้นกับว่าเงื่อนไขของสาขาเป็นจริงหรือเท็จ
- embedded computer คอมพิวเตอร์ระบบฝังตัวภายในอุปกรณ์อื่น ที่ใช้สำหรับเรียกใช้แอปพลิเคชัน หรือชุดซอฟต์แวร์ที่กำหนดไว้ล่วงหน้า

- exception การขัดจังหวะเหตุการณ์ที่ไม่ได้กำหนดไว้ ซึ่งขัดขวางการทำงานของโปรแกรม ใช้ในการตรวจจับโอเวอร์โฟลล์
- exception enable เรียกอีกอย่างว่าเปิดใช้งานการขัดจังหวะสัญญาณ หรือการกระทำที่ควบคุม ว่ากระบวนการตอบสนองต่อข้อบกพร่อง หรือไม่จำเป็นสำหรับการป้องกันการเกิดข้อบกพร่องในช่วงเวลาก่อนที่จะ โปรเซสเซอร์จะบันทึกสถานะที่จำเป็นในการรีเซ็ตอย่างปลอดภัย
- global miss rate เศษส่วนของการอ้างอิงที่ขาดหายไปในทุกๆระดับ ของแคชหลายระดับ
- global pointer รีจิสเตอร์ที่สงวนไว้เพื่อชี้ไปยังพื้นที่สแตค
- guard ป้องกันบิตพิเศษสองบิตแรก เก็บไว้ทางด้านขวา ระหว่างการคำนวณตัวเลขทศนิยม เพื่อปรับปรุงความแม่นยำ ในการปัดเศษ
- handler ตัวจัดการชื่อของ รูทีนซอฟต์แวร์ที่เรียกใช้เพื่อ"จัดการ" ข้อบกพร่องหรือขัดจังหวะ
- hardware multithreading เพิ่มการใช้โปรเซสเซอร์โดยสลับไปยังเรดอื่น เมื่อเรดหนึ่งหยุดทำงาน
- ecall (Environment Call): ใช้เรียกบริการจากสภาพแวดล้อม เช่น การเรียกระบบในระบบปฏิบัติการ (system calls) เพื่อขอทรัพยากรหรือบริการ
- ebreak (Environment Break): ใช้ในการดีบั๊ก โดยหยุดการทำงานของโปรแกรม และถ่ายโอนการควบคุมไปยังดีบั๊กเกอร์
- fence (Fence): ควบคุมลำดับการเข้าถึงหน่วยความจำและ I/O เพื่อให้แน่ใจว่าเรดอื่นหรืออุปกรณ์ภายนอกเห็นการเปลี่ยนแปลงในลำดับที่ถูกต้อง
- hit rate เศษของการเข้าถึงหน่วยความจำ ที่พบในระดับของลำดับชั้นหน่วยความจำ
- hit time เวลาที่ต้องใช้ในการเข้าถึงระดับของลำดับชั้นหน่วยความจำ รวมถึงเวลาที่จำเป็นในการพิจารณาว่าการเข้าถึงนั้นเป็น hit หรือ miss
- hold time เวลาต่ำสุดระหว่างที่อินพุต ต้องถูกต้องหลังจากขอบนาฬิกาการ
- imprecise interrupt ขัดจังหวะไม่ แน่ชัด เรียกอีกอย่างว่า ข้อบกพร่องที่ไม่แนชัด การขัดจังหวะหรือข้อบกพร่องในคอมพิวเตอร์แบบไปป์ไลน์ที่ไม่เกี่ยวข้อง กับคำสั่งที่แน่นอนซึ่งเป็นสาเหตุของการขัดจังหวะหรือข้อบกพร่อง
- implementation การนำฮาร์ดแวร์ไปใช้ที่เป็นไปตามนามธรรมของสถาปัตยกรรม
- input device อุปกรณ์อินพุตที่คอมพิวเตอร์รับข้อมูล
- instruction คำสั่งคำสั่งที่ฮาร์ดแวร์คอมพิวเตอร์เข้าใจและปฏิบัติตาม
- instruction count จำนวนคำสั่งจำนวนคำสั่งที่ดำเนินการโดยโปรแกรม

- instruction format รูปแบบคำสั่งรูปแบบการแสดงคำสั่งที่ประกอบด้วยช่อง เลขฐานสอง
- instruction-level parallelism ความขนานระดับคำสั่ง ความขนานระหว่างคำสั่ง คำสั่งผสม การวัดความถี่ไดนามิกของคำสั่ง ในโปรแกรมเดียวหรือหลายโปรแกรม
- instruction set architecture สถาปัตยกรรมชุด คำสั่ง เรียก อีก อย่าง ว่า สถาปัตยกรรมอินเทอร์เฟซแบบนามธรรม ระหว่างฮาร์ดแวร์และ ซอฟต์แวร์ระดับต่ำสุด ที่รวมข้อมูลทั้งหมดที่จำเป็นในการเขียนโปรแกรมภาษาเครื่อง ที่จะทำงานอย่างถูกต้อง รวมถึงคำสั่งรีจิสเตอร์ การเข้าถึงหน่วยความจำ I/O และอื่นๆ
- Interrupts: ขัดจังหวะข้อยกเว้นที่เกิดจากภายนอกโปรเซสเซอร์ เช่น อุปกรณ์ I/O ส่งสัญญาณ
- interrupt handler ตัวจัดการการขัดจังหวะส่วนของรหัส ที่รันอันเป็นผลมาจากข้อยกเว้น หรือการขัดจังหวะ
- loader โปรแกรมระบบที่วางโปรแกรม อ็อบเจกต์ในหน่วยความจำหลักเพื่อให้พร้อมที่จะดำเนินการ
- latch องค์กรประกอบหน่วยความจำ ที่เอาต์พุตเท่ากับค่าของสถานะที่เก็บไว้ภายในองค์กรประกอบ และสถานะจะเปลี่ยนแปลงทุกครั้งที่อินพุตที่เหมาะสมเปลี่ยนและนาฬิกาถูกยืนยัน
- latency เวลาแฝง (ไปป์ไลน์) จำนวนขั้นตอนในไปป์ไลน์หรือจำนวนขั้นตอนระหว่างสองคำสั่งระหว่างการดำเนินการ
- least significant bit บิตที่มีนัยสำคัญน้อยที่สุดบิตขวาสุดใน doubleword RISC-V
- linker เรียกอีกอย่างว่าตัวแก้ไขลิงค์โปรแกรมระบบ ที่รวมโปรแกรมภาษาเครื่อง ที่ประกอบกันอย่างอิสระและแก้ไขเลเบล ที่ไม่ได้กำหนดทั้งหมดให้เป็นไฟล์เรียกทำงาน
- loader โปรแกรมระบบที่วางโปรแกรมอ็อบเจกต์ในหน่วยความจำหลัก เพื่อให้พร้อมที่จะดำเนินการ
- local label เลเบลที่อ้างถึงอ็อบเจกต์ที่สามารถใช้ได้เฉพาะภายในไฟล์ที่กำหนดไว้เท่านั้น
- lock อุปกรณ์ซิงโครไนซ์ที่อนุญาตให้เข้าถึงข้อมูลไปยังโปรเซสเซอร์ได้ ครั้งละหนึ่งตัวเท่านั้น
- mcause: รีจิสเตอร์ใน RISC-V ที่ใช้เก็บข้อมูลเกี่ยวกับสาเหตุของข้อยกเว้นหรือการขัดจังหวะ
- MIMD หรือสตรีมคำสั่งหลายรายการ สตรีมข้อมูลหลายรายการ มัลติโปรเซสเซอร์
- most significant bit บิตที่สำคัญที่สุดบิตซ้ายสุดใน doubleword RISC-V

- nop คำสั่งที่ไม่มีการดำเนินการเพื่อเปลี่ยนสถานะ
- out-of-order execution การดำเนินการที่ไม่อยู่ในลำดับสถานการณ์ ในการดำเนินการไปฟิไลน์ เมื่อคำสั่งที่ถูกบล็อกไม่ให้ดำเนินการไม่ทำให้คำสั่งต่อไปนี้รอ
- opcode ฟิรต์ที่แสดงถึงการดำเนินการและรูปแบบของคำสั่ง
- parallel processing program โปรแกรมประมวลผลแบบขนาน โปรแกรมเดียวที่ทำงานบนโปรเซสเซอร์หลายตัวพร้อมกัน
- PC-relative addressing การกำหนดแอดเดรสแบบสัมพันธ์กับพีซี ระบบการกำหนดแอดเดรสที่ตำแหน่งที่อยู่เป็นผลรวมของตัวนับโปรแกรม (PC) และค่าคงที่ในคำสั่ง
- physical address ที่อยู่ทางกายภาพ
- pipelining เทคนิคการนำไปใช้โดยที่คำสั่งหลายคำสั่งซ้อนทับกัน ในการดำเนินการเหมือนกับสายการประกอบ
- pop ลบองค์ประกอบออกจากสแต็ก
- procedure รูทีนย่อยที่เก็บไว้ซึ่งทำงานเฉพาะตามพารามิเตอร์ ที่มีให้เฟรมการเรียกโปรซีเจอร์
- Precise exceptions: คุณสมบัติที่รับประกันว่าสถานะของโปรแกรมจะคงที่เมื่อเกิดข้อบกพร่อง ทำให้สามารถจัดการและกู้คืนได้อย่างแม่นยำ
- program counter (PC) รีจิสเตอร์ที่มีตำแหน่งที่อยู่ของคำสั่งในโปรแกรมที่กำลังดำเนินการ
- pseudo instruction รูปแบบทั่วไปของคำสั่ง ภาษาแอสเซมบลีมักจะถูกปฏิบัติราวกับว่ามันเป็นคำสั่งในตัวของมันเอง
- Push เพิ่มองค์ประกอบเพื่อสแต็ค
- parallel processing program โปรแกรมประมวลผลแบบขนาน โปรแกรมเดียวที่ทำงานบนโปรเซสเซอร์หลายตัวพร้อมกัน
- quotient ผลหารคือจำนวนที่เมื่อคูณด้วยตัวหารและบวกกับส่วนที่เหลือ ทำให้ได้ตัวตั้งหาร (dividend)
- Race Condition: สถานการณ์ที่หลายเธรดเข้าถึงหน่วยความจำร่วมกันและทำให้ผลลัพธ์ไม่แน่นอน
- round ปัดเศษวิธีการทำผลทศนิยมกลางให้พอดีกับรูปแบบจุดลอยตัว เป้าหมายโดยทั่วไปคือการหาตัวเลขที่ใกล้ที่สุด
- scientific notation สัญกรณ์วิทยาศาสตร์ สัญกรณ์ที่แสดงตัวเลขที่มีหลักเดียวทางด้านซ้ายของจุดทศนิยม

- sign-extend การ ขยาย เครื่องหมาย ของ ค่า แบบ พันที โดยใช้ บิต เครื่องหมาย ที่ ตำแหน่งคงที่ช่วยให้การประมวลผลเร็วขึ้น โดยไม่ต้องรอรอตรหัสคำสั่ง
- SIMD หรือสตรีมคำสั่งเดียวสตรีมข้อมูล หลายรายการ คำสั่งเดียวกันนี้ใช้กับสตรีมข้อมูลจำนวนมาก
- single precision ค่าทศนิยมที่แสดงในเวิร์ดแบบ 32 บิต
- stack โครงสร้างข้อมูลสำหรับการลงทะเบียนการรั่วไหลที่จัดเป็นคิวสุดท้ายก่อนออก
- stack pointer ตัวชี้สแต็กค่าที่แสดง ถึงที่อยู่จัดสรรล่าสุดในสแต็กที่แสดงตำแหน่งที่รีจิสเตอร์ควรรั่วไหล หรือตำแหน่งที่สามารถหาค่ารีจิสเตอร์เก่าได้ใน RISC-V คือ register x2 หรือที่เรียกว่า sp
- stack segment ส่วนของหน่วย ความจำที่โปรแกรมใช้เพื่อเก็บโปรซีเดอร์ call frames องค์ประกอบสถานะ
- supercomputer คลาสของคอมพิวเตอร์ที่มีการปรับขนาดที่แข็งแกร่ง การเพิ่มความเร็วทำได้บนมัลติโพรเซสเซอร์
- superscalar เทคนิคการ pipe lline ขั้นสูงที่ช่วยให้โพรเซสเซอร์สามารถดำเนินการได้มากกว่าหนึ่งคำสั่งต่อรอบสัญญาณนาฬิกา
- Synchronization:: การประสานงานระหว่างเธรดเพื่อให้แน่ใจว่าการเข้าถึงหน่วย ความจำเกิดขึ้นอย่างถูกต้อง
- Illegal instructions: รูปแบบบิตที่เป็นศูนย์ทั้งหมดหรือหนึ่งทั้งหมดถูกกำหนดให้เป็นคำสั่งที่ไม่ถูกต้องใน RV32I เพื่อช่วยตรวจจับข้อผิดพลาด
- scientific notation สัญกรณ์วิทยาศาสตร์สัญกรณ์ที่แสดงตัวเลขที่มี หลักเดียวทางด้านซ้ายของจุดทศนิยม
- truth table ตารางความจริงจากตรรกะ การแสดงการดำเนินการทางตรรกะ โดย แสดงรายการค่าทั้งหมดของอินพุตจากนั้น ในแต่ละกรณีจะแสดงว่าผลลัพธ์ ที่ได้ควรเป็นอย่างไร
- throughput ปริมาณงาน เรียกอีกอย่างว่าแบนด์วิดท์ การวัดประสิทธิภาพอีกอย่าง คือ จำนวนงานที่เสร็จสิ้นต่อหน่วยเวลา
- underflow (floating-point) อันเดอร์โฟลว์(จุดทศนิยม) สถานการณ์ที่เลขชี้กำลังลบมีขนาดใหญ่เกินกว่าจะใส่ลงในช่องเลขชี้กำลัง
- valid bit บิตที่ถูกต้องฟิลต์ในตารางของลำดับชั้นหน่วยความจำ ที่ระบุว่าบล็อกที่เกี่ยวข้องในลำดับชั้น ประกอบด้วยข้อมูลที่ถูกต้อง
- virtual addressx ที่อยู่เสมือนที่สอดคล้องกับตำแหน่งในพื้นที่เสมือนและแปล โดยการจับคู่ตำแหน่งที่อยู่เป็นที่อยู่จริงเมื่อเข้าถึงหน่วยความจำ
- word หน่วยปกติของการเข้าถึงในคอมพิวเตอร์โดย ปกติจะเป็นกลุ่ม 32 บิต

- wafer แผ่นเวเฟอร์แผ่นจากแท่งซิลิกอนหนาไม่เกิน 0.1 นิ้วใช้ทำเป็นแผ่น
- Yield ผลผลิตคิดเป็นเปอร์เซ็นต์ของแม่พิมพ์ที่ดี จากจำนวนทั้งหมดของแม่พิมพ์ บนแผ่นเวเฟอร์

C.10 สรุปท้ายบท

RISC-V เป็น ISA แบบเปิดที่ปฏิวัติวงการด้วยความเรียบง่าย ออกแบบให้เป็นโมดูลาร์ มีชุดคำสั่งพื้นฐาน (เช่น RV32I) และส่วนขยาย (เช่น RV32M, RV32FD, RV32A, RV32V) ผู้ใช้สามารถเลือกส่วนขยายตามความต้องการ RV32I มีคำสั่งพื้นฐาน 47 คำสั่ง สำหรับจำนวนเต็ม 32 บิต ให้ง่ายต่อการเรียนรู้ และ RV64 ขยายเป็น 64 บิต รองรับคำสั่งเดียวกัน แต่จัดการข้อมูลขนาดใหญ่กว่า ส่วนขยาย RV32M เพิ่มคำสั่งคูณและหาร RV32A รองรับการทำงานแบบอะตอมิก RV32V รองรับงานประมวลผลสูง เช่น การคำนวณเวกเตอร์ใน DAXPY ความยืดหยุ่น และการไม่มีค่าใบอนุญาต จึงเหมาะสำหรับงานหลากหลาย ตั้งแต่ IoT (เช่น ESP32-C3) ไปจนถึงการคำนวณประสิทธิภาพสูง ชิพ SiFive Performance P550 ใช้ในเซิร์ฟเวอร์ ชิพ Kendryte K210 (RISC-V) รองรับการประมวลผลภาพและ ML มีคำสั่งและส่วนขยายที่สามารถกำหนดเองตอบโจทย์งานเฉพาะ RV32E เป็นส่วนขยายของ RISC-V ที่ออกแบบสำหรับระบบฝังตัว (embedded systems) โดยลดจำนวนรีจิสเตอร์จาก 32 ตัวใน RV32I เหลือ 16 ตัว (x0 ถึง x15) เพื่อลดต้นทุนและขนาดของฮาร์ดแวร์ RISC-V มีส่วนขยายที่กำหนดเอง ผู้ผลิตสามารถเพิ่มคำสั่งเฉพาะ เช่น ในชิพ AI หรือ IoT การพัฒนา RISC-V สามารถทำได้บนแพลตฟอร์มอย่าง Raspberry Pi โดยใช้ JTAG และเครื่องมือเช่น OpenOCD

นอกจากนี้ RISC-V ยังมีชุมชนที่กำลังเติบโต โดยได้รับการสนับสนุนจาก RISC-V International และบริษัทชั้นนำ RISC-V มีระบบปฏิบัติการที่รองรับ RISC-V ในปี 2025 มีหลากหลาย ตั้งแต่ระบบปฏิบัติการทั่วไป เช่น Linux ไปจนถึง RTOS สำหรับระบบฝังตัว การสนับสนุนที่กว้างขวางนี้แสดงถึงความยืดหยุ่นของ RISC-V ในระบบฝังตัว, IoT, และการประมวลผลทั่วไป ทำให้ RISC-V เหมาะสำหรับการเรียนรู้ ISA และภาษาแอสเซมบลี

ดัชนี

- ABI, ดูเพิ่มเติม application binary interface
- Add, 24, 167
- immediate, 24, 167
 - immediate word, 126, 167
 - upper immediate to PC, 174
 - word, 126, 167
- add, 24, 95, ดูเพิ่มเติม c.add, 167
- Add upper immediate to PC, 25
- add.uw, 130
- addi, ดูเพิ่มเติม Add immediate, ดูเพิ่มเติม c.addi16sp, ดูเพิ่มเติม c.addi4spn, ดูเพิ่มเติม c.addi, ดูเพิ่มเติม c.li, 167
- addiw, ดูเพิ่มเติม Add immediate word, ดูเพิ่มเติม c.addiw, 167
- addw, ดูเพิ่มเติม Add word, ดูเพิ่มเติม c.addw, 167
- ALGOL, 164
- Allen, Fran, 18
- AMD64, 134
- amoad.d, ดูเพิ่มเติม Atomic Memory Operation Add Doubleword, 168
- amoad.w, ดูเพิ่มเติม Atomic Memory Operation Add Word, 168
- amoand.d, ดูเพิ่มเติม Atomic Memory Operation And Doubleword, 168
- amoand.w, ดูเพิ่มเติม Atomic Memory Operation And Word, 169
- amomax.d, ดูเพิ่มเติม Atomic Memory Operation Maximum Doubleword, 169
- amomax.w, ดูเพิ่มเติม Atomic Memory Operation Maximum Word, 169
- amomaxu.d, ดูเพิ่มเติม Atomic Memory Operation Maximum Unsigned Doubleword, 170
- amomaxu.w, ดูเพิ่มเติม Atomic Memory Operation Maximum Unsigned Word, 170
- amomin.d, ดูเพิ่มเติม Atomic Memory Operation Minimum Doubleword, 170
- amomin.w, ดูเพิ่มเติม Atomic Memory Operation Minimum Word, 171
- amominu.d, ดูเพิ่มเติม Atomic Memory Operation Minimum Unsigned Doubleword, 171
- amominu.w, ดูเพิ่มเติม Atomic Memory Operation Minimum Unsigned Word, 171
- amoor.d, ดูเพิ่มเติม Atomic Memory Operation Or Doubleword, 172
- amoor.w, ดูเพิ่มเติม Atomic Memory Operation Or Word, 172
- amoswap.d, ดูเพิ่มเติม Atomic Memory Operation Swap Doubleword, 172
- amoswap.w, ดูเพิ่มเติม Atomic Memory Operation Swap Word, 172
- amoxor.d, ดูเพิ่มเติม Atomic Memory Operation Exclusive Or Doubleword, 173
- amoxor.w, ดูเพิ่มเติม Atomic Memory Operation Exclusive Or Word, 173
- And, 24, 173
- immediate, 24, 173
- and, ดูเพิ่มเติม c.and, 173
- andi, 24, ดูเพิ่มเติม And immediate, ดูเพิ่มเติม c.andi, 173
- andh, 123
- application binary interface, 22, 33, 39, 41, 57
- Application Specific Integrated Circuits, 2
- architecture, 9
- ARM
- code size, 10, 134
 - Cortex-A5, 8
 - Cortex-A9, 9
 - DAXPY, 62
 - Insertion Sort, 30
 - instruction reference manual
 - number of pages, 15
 - Load Multiple, 8, 10
 - number of registers, 12
 - Thumb, 10
 - Thumb-2, 10
 - Tree Sum, 229
- ARM SVE, 111
- ARM64 SVE Vector, 109
- ARMv8, 134
- Art is long, life is short, 234
- ASIC, ดูเพิ่มเติม Application Specific Integrated Circuits
- assembler directives, 46
- Atomic Memory Operation
- Add
 - Doubleword, 126, 168
 - Word, 70, 168
 - And
 - Doubleword, 126, 168
 - Word, 70, 169
 - Exclusive Or
 - Doubleword, 126, 173
 - Word, 70, 173
 - Maximum
 - Doubleword, 126, 169
 - Word, 70, 169
 - Maximum Unsigned
 - Doubleword, 126, 170
 - Word, 70, 170
 - Minimum
 - Doubleword, 126, 170

- Word, 70, 171
- Minimum Unsigned
 - Doubleword, 126, 171
 - Word, 70, 171
- Or
 - Doubleword, 126, 172
 - Word, 70, 172
- Swap
 - Doubleword, 126, 172
 - Word, 70, 172
- auipc, *ดูเพิ่มเติม* Add upper immediate to PC, 174
- backwards
 - binary-compatibility, 4
- bclr, **124**
- Bell, C. Gordon, 54, 126
- beq, *ดูเพิ่มเติม* Branch if equal, *ดูเพิ่มเติม* c.beqz, 174
- beqz, 41, 174
- bext, **124**
- bge, *ดูเพิ่มเติม* Branch if greater or equal, 174
- bgeu, *ดูเพิ่มเติม* Branch if greater or equal unsigned, 174
- bgez, 41, 175
- bgt, 41, 175
- bgtu, 41, 175
- bgtz, 41, 175
- Bhirasri, Silpa, 234
- binv, **124**
- ble, 41, 175
- bleu, 41, 175
- blez, 41, 175
- blt, *ดูเพิ่มเติม* Branch if less than, 176
- bltu, *ดูเพิ่มเติม* Branch if less than unsigned, 176
- bltz, 41, 176
- bne, *ดูเพิ่มเติม* Branch if not equal, *ดูเพิ่มเติม* c.bnez, 176
- bnez, 41, 176
- Boole, George, 122
- Branch
 - if equal, 27, 174
 - if greater or equal, 27, 174
 - if greater or equal unsigned, 27, 174
 - if less than, 27, 176
 - if less than unsigned, 27, 176
 - if not equal, 27, 176
- branch prediction, 9, 24
- Brooks, Fred, 161
- Browning, Robert, 63
- bset, **124**
- C programs
 - DAXPY, **62**
 - Insertion Sort, **30**
 - Tree Sum, **229**
 - c.add, *ดูเพิ่มเติม* add, 177
 - c.addi, *ดูเพิ่มเติม* addi, 177
 - c.addi16sp, *ดูเพิ่มเติม* addi, 177
 - c.addi4spn, *ดูเพิ่มเติม* addi, 177
 - c.addiw, 126, *ดูเพิ่มเติม* addiw, 177
 - c.addw, 126, *ดูเพิ่มเติม* addw, 177
 - c.and, *ดูเพิ่มเติม* and, 178
 - c.andi, *ดูเพิ่มเติม* andi, 178
 - c.beqz, *ดูเพิ่มเติม* beq, 178
 - c.bnez, *ดูเพิ่มเติม* bne, 178
 - c.ebreak, *ดูเพิ่มเติม* ebreak, 178
 - c.fld, *ดูเพิ่มเติม* fld, 178
 - c.fldsp, *ดูเพิ่มเติม* fld, 179
 - c.flw, *ดูเพิ่มเติม* flw, 179
 - c.flwsp, *ดูเพิ่มเติม* flw, 179
 - c.fsd, *ดูเพิ่มเติม* fsd, 179
 - c.fsdsp, *ดูเพิ่มเติม* fsd, 179
 - c.fsw, *ดูเพิ่มเติม* fsw, 180
 - c.fswsp, *ดูเพิ่มเติม* fsw, 180
 - c.j, *ดูเพิ่มเติม* jal, 180
 - c.jal, *ดูเพิ่มเติม* jal, 180
 - c.jalr, *ดูเพิ่มเติม* jalr, 180
 - c.jr, *ดูเพิ่มเติม* jalr, 180
 - c.ld, 126, *ดูเพิ่มเติม* ld, 181
 - c.ldsp, 126, *ดูเพิ่มเติม* ld, 181
 - c.li, *ดูเพิ่มเติม* addi, 181
 - c.lui, *ดูเพิ่มเติม* lui, 181
 - c.lw, *ดูเพิ่มเติม* lw, 181
 - c.lwsp, *ดูเพิ่มเติม* lw, 181
 - c.mv, *ดูเพิ่มเติม* add, 182
 - c.or, *ดูเพิ่มเติม* or, 182
 - c.sd, 126, *ดูเพิ่มเติม* sd, 182
 - c.sdsp, 126, *ดูเพิ่มเติม* sd, 182
 - c.slli, *ดูเพิ่มเติม* slli, 182
 - c.srai, *ดูเพิ่มเติม* srai, 183
 - c.srli, *ดูเพิ่มเติม* srli, 183
 - c.sub, *ดูเพิ่มเติม* sub, 183
 - c.subw, 126, *ดูเพิ่มเติม* subw, 183
 - c.sw, *ดูเพิ่มเติม* sw, 183
 - c.swsp, *ดูเพิ่มเติม* sw, 184
 - c.xor, *ดูเพิ่มเติม* xor, 184
 - call, 41, 184
 - Callee saved registers, 39
 - Caller saved registers, 39
 - Calling conventions, **38**
 - Carryless multiply, **124**
 - Chanel, Coco, 166
 - chip, *ดูเพิ่มเติม* die, 6
 - clmul, **124**
 - clmulh, **124**
 - clmulr, **124**
 - clz, **123**
 - clzw, 130
 - Compilers
 - Turing Award, 164
 - Control and Status Register
 - marchid, 160, 161
 - mcause, 146, 148, 149, 151
 - mcounteren, 160, 161
 - mcycle, 160, 161
 - medeleg, 154, 154, 155
 - mepc, 146, 148, 149, 151
 - mhartid, 160, 161
 - mhpmmcounter*i*, 160, 161
 - mhpmevent*i*, 160, 161
 - mideleg, 154
 - mie, 146, 148, 151
 - mimpid, 160, 161
 - minstret, 160, 161
 - mip, 146, 148, 151
 - misa, 160, 161
 - mscratch, 146, 148, 149
 - mstatus, 146, 148, 149, 151
 - mtimecmp, 160, 161
 - mtime, 160, 161
 - mtval, 146, 148, 149, 151
 - mtvec, 146, 148, 149
 - mvendorid, 160, 161
 - pmpcfg, 152
 - satp, 158, 158
 - scause, 146, 155
 - scounteren, 160, 161
 - sedeleg, 154
 - sepc, 146, 155
 - sideleg, 154
 - sie, 154, 155
 - sip, 154, 155
 - sscratch, 146, 155
 - sstatus, 155, 156
 - stval, 146, 155
 - stvec, 146, 155
 - read and clear, 30, 185
 - read and clear immediate, 30, 185
 - read and set, 30, 185
 - read and set immediate, 30, 186
 - read and write, 30, 186
 - read and write immediate, 30, 186
 - CoreMark benchmark, 9
 - cost, *ดูเพิ่มเติม* instruction set architecture, principles of design, cost
 - cpopw, 130
 - Cray, Seymour, 82, 135
 - CRC32, **124**

- CSR, *ดู* Control and Status Register
- csrc, 41, 184
- csrci, 41, 184
- csrr, 41, 185
- csrrc, *ดูเพิ่มเติม* Control and Status Register read and clear, 185
- csrrci, *ดูเพิ่มเติม* Control and Status Register read and clear immediate, 185
- csrrs, *ดูเพิ่มเติม* Control and Status Register read and set, 185
- csrrsi, *ดูเพิ่มเติม* Control and Status Register read and set immediate, 186
- csrrw, *ดูเพิ่มเติม* Control and Status Register read and write, 186
- csrrwi, *ดูเพิ่มเติม* Control and Status Register read and write immediate, 186
- csrs, 41, 186
- csrsi, 41, 187
- csrw, 41, 187
- csrwi, 41, 187
- ctzw, 130

- da Vinci, Leonardo, 2
- data-level parallelism, 82
- DAXPY, 62, 93, 108, 111, 114
- de Saint-Exupéry, Antoine, 56
- delay slot, 9
- delayed branch, 9
- die, *ดูเพิ่มเติม* chip, 6
 - yield, 6
- div, 187
- Divide, 53, 187
 - unsigned, 53, 187
 - unsigned word, 126, 188
 - using shift right, 53
 - word, 126, 188
- divu, *ดูเพิ่มเติม* Divide unsigned, 187
- divuw, *ดูเพิ่มเติม* Divide unsigned word, 188
- divvw, *ดูเพิ่มเติม* Divide word, 188
- dynamic linking, 49
- dynamic typing, 87, 132

- ease of programming, compiling, and linking, *ดูเพิ่มเติม* instruction set architecture, principles of design, ease of programming, compiling, and linking
- ebreak, 188
- ecall, 188
- EEW, 88, 96
- Effective Element Width, 88
- Effective register length multipliler, 88
- Einstein, Albert, 70
- Element length, 88
- ELEN, 88
- ELF, *ดูเพิ่มเติม* executable and linkable format
- EMUL, 88
- endianness, 27
- epilogue, *ดูเพิ่มเติม* function epilogue
- Exception, 145
- Exception Return Machine, 149, 211
 - Supervisor, 155, 222
- Exclusive Or, 24, 225
 - immediate, 24, 225
- executable and linkable format, 46

- fabs.d, 41, 188
- fabs.s, 41, 189
- fadd.d, *ดูเพิ่มเติม* Floating-point Add double-precision, 189
- fadd.s, *ดูเพิ่มเติม* Floating-point Add single-precision, 189
- fclass.d, *ดูเพิ่มเติม* Floating-point Classify double-precision, 189
- fclass.s, *ดูเพิ่มเติม* Floating-point Classify single-precision, 190
- fcvt.d.l, *ดูเพิ่มเติม* Floating-point Convert double from long, 190
- fcvt.d.lu, *ดูเพิ่มเติม* Floating-point Convert double from long unsigned, 190
- fcvt.d.s, *ดูเพิ่มเติม* Floating-point Convert double from single, 191
- fcvt.d.w, *ดูเพิ่มเติม* Floating-point Convert double from word, 191
- fcvt.d.wu, *ดูเพิ่มเติม* Floating-point Convert double from word unsigned, 191
- fcvt.l.d, *ดูเพิ่มเติม* Floating-point Convert long from double, 191
- fcvt.l.s, *ดูเพิ่มเติม* Floating-point Convert long from single, 192
- fcvt.lu.d, *ดูเพิ่มเติม* Floating-point Convert long unsigned from double, 192
- fcvt.lu.s, *ดูเพิ่มเติม* Floating-point Convert long unsigned from single, 192
- fcvt.s.d, *ดูเพิ่มเติม* Floating-point Convert single from double, 192
- fcvt.s.l, *ดูเพิ่มเติม* Floating-point Convert single from long, 193
- fcvt.s.lu, *ดูเพิ่มเติม* Floating-point Convert single from long unsigned, 193
- fcvt.s.w, 193
- fcvt.s.wu, *ดูเพิ่มเติม* Floating-point Convert single from word unsigned, 193
- fcvt.w.d, *ดูเพิ่มเติม* Floating-point Convert word from double, 194
- fcvt.w.s, *ดูเพิ่มเติม* Floating-point Convert word from single, 194
- fcvt.wu.d, *ดูเพิ่มเติม* Floating-point Convert word unsigned from double, 194
- fcvt.wu.s, *ดูเพิ่มเติม* Floating-point Convert word unsigned from single, 194
- fdiv.d, *ดูเพิ่มเติม* Floating-point Divide double-precision, 195
- fdiv.s, *ดูเพิ่มเติม* Floating-point Divide single-precision, 195
- Fence
 - Instruction Stream, 196
 - Memory and I/O, 195
 - Virtual Memory, 160, 218
- fence, 41, *ดูเพิ่มเติม* Fence Memory and I/O, 195
- fence.i, *ดูเพิ่มเติม* Fence Instruction Stream, 196
- feq.d, *ดูเพิ่มเติม* Floating-point Equals double-precision, 196
- feq.s, *ดูเพิ่มเติม* Floating-point Equals single-precision, 196
- Field-Programmable Gate Array, 2
- find-first-set mask bit, 102
- fld, *ดูเพิ่มเติม* c.fldsp, *ดูเพิ่มเติม* c.fld, *ดูเพิ่มเติม* Floating-point load doubleword, 196
- fle.d, *ดูเพิ่มเติม* Floating-point Less or Equals double-precision, *ดูเพิ่มเติม* Floating-point Less Than double-precision, 197
- fle.s, *ดูเพิ่มเติม* Floating-point Less or Equals

single-precision, ดูเพิ่มเติม

Floating-point Less Than

single-precision, 197

Floating-Point

dynamic rounding mode, 58

fused multiply-add, 58

IEEE 754-2019 floating-point

standard, 56

sign-injection, 61

static rounding mode, 58

Floating-point, 56

Add

double-precision, 56, 189

half-precision, 56

single-precision, 56, 189

binary128, 64

binary16, 64

binary32, 64

binary64, 64

Classify

double-precision, 56, 189

half-precision, 56

single-precision, 56, 190

Convert

double from half, 56

double from long, 126,

190

double from long

unsigned, 126, 190

double from single, 56,

191

double from word, 56,

191

double from word

unsigned, 56, 191

half from double, 56

half from long, 126

half from long unsigned,

126

half from single, 56

half from word, 56

half from word unsigned,

56

long from double, 126,

191

long from half, 126

long from single, 126, 192

long unsigned from

double, 126, 192

long unsigned from half,

126

long unsigned from

single, 126, 192

single from double, 56,

192

single from half, 56

single from long, 126, 193

single from long

unsigned, 126, 193

single from word, 56

single from word

unsigned, 56, 193

word from double, 56,

194

word from half, 56

word from single, 56, 194

word unsigned from

double, 56, 194

word unsigned from half,

56

word unsigned from

single, 56, 194

decimal128, 64

decimal32, 64

decimal64, 64

Divide

double-precision, 56, 195

half-precision, 56

single-precision, 56, 195

Equals

double-precision, 56, 196

half-precision, 56

single-precision, 56, 196

Fused multiply-add

double-precision, 56, 198

single-precision, 56, 198

Fused multiply-half

single-precision, 56

Fused multiply-subtract

double-precision, 56, 199

half-precision, 56

single-precision, 56, 200

Fused negative multiply-add

double-precision, 56, 202

half-precision, 56

single-precision, 56, 202

Fused negative

multiply-subtract

double-precision, 56, 202

half-precision, 56

single-precision, 56, 203

half precision, 64

Less or Equals

double-precision, 56, 197

half-precision, 56

single-precision, 56, 197

Less Than

double-precision, 56, 197

half-precision, 56

single-precision, 56, 197

Load

doubleword, 56, 196

half, 56

word, 56, 198

Maximum

double-precision, 56, 198

half-precision, 56

single-precision, 56, 199

Minimum

double-precision, 56, 199

half-precision, 56

single-precision, 56, 199

Move

doubleword from integer,

200

doubleword to integer,

201

half from integer, 56

half to integer, 56

single from integer, 56

single to integer, 56

word from integer, 201

word to integer, 201

Multiply

double-precision, 56, 200

half-precision, 56

single-precision, 56, 200

octuple precision, 64

quadruple precision, 64

Sign-inject

double-precision, 56, 204

half-precision, 56

single-precision, 56, 204

Sign-inject negative

double-precision, 56, 205

half-precision, 56

single-precision, 56, 205

Sign-inject XOR

double-precision, 56, 205

half-precision, 56

single-precision, 56, 206

Square root

double-precision, 56, 206

half-precision, 56

single-precision, 56, 206

Store

doubleword, 56, 204

half, 56

word, 56, 207

Subtract

double-precision, 56, 207

half-precision, 56

single-precision, 56, 207

floating-point

control and status register,

57

flt.d, 197

flt.s, 197

flw, ดูเพิ่มเติม c.flwsp, ดูเพิ่มเติม

c.flw, ดูเพิ่มเติม Floating-point

load word, 198

- fmadd.d, *ดูเพิ่มเติม*
Floating-point fused multiply-add double-precision, 198
- fmadd.s, *ดูเพิ่มเติม*
Floating-point fused multiply-add single-precision, 198
- fmax.d, *ดูเพิ่มเติม* Floating-point maximum double-precision, 198, *ดูเพิ่มเติม* Floating-point maximum single-precision
- fmax.s, 199
- fmin.d, *ดูเพิ่มเติม* Floating-point minimum double-precision, 199
- fmin.s, *ดูเพิ่มเติม* Floating-point minimum single-precision, 199
- fmsub.d, *ดูเพิ่มเติม*
Floating-point fused multiply-subtract double-precision, 199
- fmsub.s, *ดูเพิ่มเติม*
Floating-point fused multiply-subtract single-precision, 200
- fmul.d, *ดูเพิ่มเติม* Floating-point Multiply double-precision, 200
- fmul.s, *ดูเพิ่มเติม* Floating-point Multiply single-precision, 200, *ดูเพิ่มเติม* Floating-point Subtract single-precision
- fmv.d, 41, 200
- fmv.d.x, *ดูเพิ่มเติม* Floating-point move doubleword from integer, 200
- fmv.s, 41, 201
- fmv.w.x, *ดูเพิ่มเติม*
Floating-point move word from integer, 201
- fmv.x.d, *ดูเพิ่มเติม* Floating-point move doubleword to integer, 201
- fmv.x.w, *ดูเพิ่มเติม*
Floating-point move word to integer, 201
- fneg.d, 41, 201
- fneg.s, 41, 202
- fnmadd.d, *ดูเพิ่มเติม*
Floating-point fused negative multiply-add double-precision, *ดูเพิ่มเติม* Floating-point fused negative multiply-add single-precision, 202
- fnmadd.s, 202
- fnmsub.d, *ดูเพิ่มเติม*
Floating-point fused negative multiply-subtract double-precision, 202
- fnmsub.s, *ดูเพิ่มเติม*
Floating-point fused negative multiply-subtract single-precision, 203
- FPGA, *ดูเพิ่มเติม*
Field-Programmable Gate Array, 2
- frcsr, 41, 203
- frflags, 41, 203
- frmm, 41, 203
- fscsr, 41, 203
- fsd, *ดูเพิ่มเติม* c.fsdsp, *ดูเพิ่มเติม* c.fsd, *ดูเพิ่มเติม* Floating-point store doubleword, 204
- fsflags, 41, 204
- fsgnj.d, *ดูเพิ่มเติม* Floating-point Sign-inject double-precision, 204
- fsgnj.s, *ดูเพิ่มเติม* Floating-point Sign-inject single-precision, 204
- fsgnjn.d, *ดูเพิ่มเติม*
Floating-point Sign-inject negative double-precision, 205
- fsgnjn.s, *ดูเพิ่มเติม* Floating-point Sign-inject negative single-precision, 205
- fsgnjx.d, *ดูเพิ่มเติม*
Floating-point Sign-inject XOR double-precision, 205
- fsgnjx.s, *ดูเพิ่มเติม* Floating-point Sign-inject XOR single-precision, 206
- fsqrt.d, *ดูเพิ่มเติม* Floating-point Square Root double-precision, 206
- fsqrt.s, *ดูเพิ่มเติม* Floating-point Square Root single-precision, 206
- fsm, 41, 206
- fsub.d, *ดูเพิ่มเติม* Floating-point Subtract double-precision, 207
- fsub.s, 207
- fsw, *ดูเพิ่มเติม* c.fswsp, *ดูเพิ่มเติม* c.fsw, *ดูเพิ่มเติม* Floating-point store word, 207
- function epilogue, 41
- function prologue, 41
- Fused multiply-add, 58
- gather, 90
- Hart, 143
- IEEE 754-2008, 64
- IEEE 754-2019, 64
- IEEE 754-2019 floating-point standard, 56
- Illiac IV, 114
- implementation, 9
- Insertion Sort, 30
- Instruction diagram
Privileged instructions, 143
- RV32A, 70
- RV32B, 123
- RV32C, 74
- RV32D, 56
- RV32F, 56
- RV32I, 18
- RV32M, 52
- RV32V, 83
- RV64A, 126
- RV64B, 130
- RV64C, 126
- RV64D, 126
- RV64F, 126
- RV64I, 126
- RV64M, 126
- Zfh, 56
- instruction set architecture, 3
backwards
binary-compatibility, 4
elegance, 16, 31, 50, 77, 115, 135, 165
incremental, 4
metrics of design, 6
cost, 6, 18, 21, 25, 27, 31, 54, 76, 124, 135, 159
ease of programming, compiling, and linking, 12, 21, 22, 24, 27, 31, 61, 82, 87, 90, 91, 115, 132, 135, 151, 161
isolation of architecture from implementation, 9, 31, 82, 115, 143
performance, 8, 18, 22, 31, 53, 56, 58, 62, 72, 82, 91, 92, 95, 104, 113, 132, 135
program size, 10, 31, 74, 77, 130, 134, 135
room for growth, 10, 21, 31, 135
simplicity, 8, 14, 15, 24–26, 28, 30, 31, 62, 72, 74, 83, 112, 149, 155, 161
mistakes of the past, 31
modularity, 5
open, 3
principles of design

- cost, 50
- ease of programming,
- compiling, and linking, 48, 50
- performance, 39, 50, 164
- room for growth, 165
- simplicity, 41, 46
- Interrupt, 146
- ISA, *ดูเพิ่มเติม* instruction set architecture
- isolation of architecture from implementation, *ดูเพิ่มเติม*
- instruction set architecture,
- principles of design, isolation of architecture from implementation
- Itanium, **133**
- j, 41, 207
- jal, 41, *ดูเพิ่มเติม* c.jal, *ดูเพิ่มเติม*
- c.j, *ดูเพิ่มเติม* Jump and link, 208
- jahr, 41, *ดูเพิ่มเติม* c.jahr, *ดูเพิ่มเติม*
- c.jr, *ดูเพิ่มเติม* Jump and link register, 208
- Johnson, Kelly, 50
- jr, 41, 208
- Jump and link, 29, 208
- register, 29, 208
- Kuck, David, 114
- la, 208
- lb, *ดูเพิ่มเติม* Load byte, 209
- lbu, *ดูเพิ่มเติม* Load byte unsigned, 209
- ld, *ดูเพิ่มเติม* c.ldsp, *ดูเพิ่มเติม*
- c.ld, *ดูเพิ่มเติม* Load doubleword, 209
- leaf function, 39
- lh, *ดูเพิ่มเติม* Load halfword, 209
- lhu, *ดูเพิ่มเติม* Load halfword unsigned, 210
- li, 41, 210
- Lindy effect, 33
- linker relaxation, **49**
- little-endian, 27
- lla, 210
- LMUL, 88, 88, 94, 97, 113
- Load
 - byte, 26, 209
 - byte unsigned, 26, 209
 - doubleword, 126, 209
 - halfword, 26, 209
 - halfword unsigned, 26, 210
 - reserved
 - doubleword, 126, 210
 - word, 70, 210
 - upper immediate, 25, 211
 - word, 26, 211
 - word unsigned, 26, 211
- Load upper immediate, 25
- lr.d, *ดูเพิ่มเติม* Load reserved doubleword, 210
- lr.w, *ดูเพิ่มเติม* Load reserved word, 210
- lui, *ดูเพิ่มเติม* c.lui, *ดูเพิ่มเติม*
- Load upper immediate, 211
- lw, *ดูเพิ่มเติม* c.lwsp, *ดูเพิ่มเติม*
- c.lw, *ดูเพิ่มเติม* Load word, 211
- lwu, *ดูเพิ่มเติม* Load word unsigned, 211
- Machine mode, 143
- macrofusion, 6, 8, 76
- marchid, *ดูเพิ่มเติม* Control and Status Register
- max, **123**
- maximum vector length, 87
- maxu, **123**
- mcause, *ดูเพิ่มเติม* Control and Status Register
- mcounteren, *ดูเพิ่มเติม* Control and Status Register
- mcycle, *ดูเพิ่มเติม* Control and Status Register
- mepc, *ดูเพิ่มเติม* Control and Status Register
- metrics of ISA design, *ดูเพิ่มเติม* instruction set architecture, metrics of design
- mhartid, *ดูเพิ่มเติม* Control and Status Register
- mhpmmcounteri, *ดูเพิ่มเติม* Control and Status Register
- mhpmeventi, *ดูเพิ่มเติม* Control and Status Register
- microMIPS, 62
- mie, *ดูเพิ่มเติม* Control and Status Register
- mimpid, *ดูเพิ่มเติม* Control and Status Register
- min, **123**
- minstret, *ดูเพิ่มเติม* Control and Status Register
- minu, **123**
- mip, *ดูเพิ่มเติม* Control and Status Register
- MIPS
 - assembler, 25
 - DAXPY, **62**
 - delayed branch, 9, 28, 33, 64, 138
 - delayed load, 27, 33, 138
 - Insertion Sort, **30**
 - MIPS-IV, **134**
 - misa, *ดูเพิ่มเติม* Control and Status Register
 - Moore's Law, 2
 - mret, *ดูเพิ่มเติม* Exception Return Machine, 211
 - mscratch, *ดูเพิ่มเติม* Control and Status Register
 - mstatus, *ดูเพิ่มเติม* Control and Status Register
 - mtime, *ดูเพิ่มเติม* Control and Status Register
 - mtimecmp, *ดูเพิ่มเติม* Control and Status Register
 - mtval, *ดูเพิ่มเติม* Control and Status Register
 - mtvec, *ดูเพิ่มเติม* Control and Status Register
 - mul, *ดูเพิ่มเติม* Multiply, 212
 - mulh, *ดูเพิ่มเติม* Multiply high, 212
 - mulhsu, *ดูเพิ่มเติม* Multiply high signed-unsigned, 212
 - mulhu, *ดูเพิ่มเติม* Multiply high unsigned, 212
 - Multiply, 53, 212
 - high, 53, 212
 - high signed-unsigned, 53, 212
 - high unsigned, 53, 212
 - multi-word, 54
 - using shift left, 53
 - word, 126, 213
 - mulw, *ดูเพิ่มเติม* Multiply word, 213
 - mv, 41, 213
 - mvendorid, *ดูเพิ่มเติม* Control and Status Register
 - Narrowing instructions, 95, 96
 - neg, 41, 213
 - negw, 41, 213
 - nop, 41, 213
 - not, 41, 213
 - Occam, William of, 52
 - Or, 24, 214
 - immediate, 24, 214
 - or, *ดูเพิ่มเติม* c.or, 214
 - ori, 24, *ดูเพิ่มเติม* Or immediate, 214
 - orn, **123**
 - out-of-order processors, 22
 - Page, 156
 - Page fault, 156
 - Page table, 156
 - Pascal, Blaise, 77

- performance, ดูเพิ่มเติม
 - instruction set architecture,
 - principles of design,
 - performance
 - CoreMark benchmark, 9
 - equation, 8
- Perlis, Alan, 164, 164
- PIC, ดูเพิ่มเติม position
 - independent code
- pipelined processor, 24
- Plato, 226
- popc, **123**
- position independent code,
 - 12, 27, 48, 133
- Privilege mode, 142
 - Machine mode, 143
 - User mode, 151
- program size, ดูเพิ่มเติม
 - instruction set architecture,
 - principles of design, program size
- Programming languages
 - Turing Award, 164
- prologue, ดูเพิ่มเติม function
- prologue
- Pseudoinstruction, **44**
 - beqz, 41, 174
 - bgez, 41, 175
 - bgt, 41, 175
 - bgtu, 41, 175
 - bgtz, 41, 175
 - ble, 41, 175
 - bleu, 41, 175
 - blez, 41, 175
 - bltz, 41, 176
 - bnez, 41, 176
 - call, 41, 184
 - csrc, 41, 184
 - csrci, 41, 184
 - csrr, 41, 185
 - csrs, 41, 186
 - csrsi, 41, 187
 - csrw, 41, 187
 - csrwi, 41, 187
 - fabs.d, 41, 188
 - fabs.s, 41, 189
 - fence, 41
 - fmv.d, 41, 200
 - fmv.s, 41, 201
 - fneg.d, 41, 201
 - fneg.s, 41, 202
 - frcsr, 41, 203
 - frflags, 41, 203
 - frmm, 41, 203
 - fscsr, 41, 203
 - fsflags, 41, 204
 - fsm, 41, 206
 - j, 41, 207
 - jr, 41, 208
 - la, 208
 - li, 41, 210
 - lla, 210
 - mv, 41, 213
 - neg, 41, 213
 - negw, 41, 213
 - nop, 41, 213
 - not, 41, 213
 - rdcycle, 41, 214
 - rdcycleh, 41, 214
 - rdinstret, 41, 214
 - rdinstreth, 41, 215
 - rdtime, 41, 215
 - rdtimeh, 41, 215
 - ret, 41, 216
 - seqz, 41, 217
 - sxt.w, 41, 218
 - sgtz, 41, 218
 - sltz, 41, 221
 - snez, 41, 221
 - tail, 41, 225
- Ralph Brazelton Peck, 124
- rdcycle, 41, 214
- rdcycleh, 41, 214
- rdinstret, 41, 214
- rdinstreth, 41, 215
- rdtime, 41, 215
- rdtimeh, 41, 215
- reductions, 101
- registers
 - number of, 12
- rem, ดูเพิ่มเติม Remainder, 215
- Remainder, 53, 215
 - unsigned, 53, 215
 - unsigned word, 126, 216
 - word, 126, 216
- remu, ดูเพิ่มเติม Remainder
 - unsigned, 215
- remuw, ดูเพิ่มเติม Remainder
 - unsigned word, 216
- remw, ดูเพิ่มเติม Remainder
 - word, 216
- ret, 41, 216
- rev8, **124**
- RISC-V
 - application binary interface, 22, 33, 39, 41, 57
 - assembler directives, 46
 - BOOM, 9
 - Calling conventions, **41**
 - code size, 10, 134
 - DAXPY, **62**
 - function epilogue, 41
 - function prologue, 41
 - heap region, 47
 - Insertion Sort, **30**
 - instruction reference manual
 - number of pages, 15
 - instruction set naming scheme, **5**
 - International, **3**
 - lessons learned, **31**
 - Linker, **47**
 - Loader, **50**
 - long, 130
 - macrofusion, 6
 - memory allocation, **47**
 - modularity, 5
 - number of registers, 12
 - pseudoinstruction, **44**
 - pseudoinstructions, 13
 - Rocket, 8
 - RV128, 136
 - RV32A, **70**
 - RV32B, **122**
 - RV32C, 10, 13, **74**
 - RV32D, **56**
 - RV32F, **56**
 - RV32G, 10, 13
 - RV32I, **18**
 - RV32M, **52**
 - RV32V, 14, **82**
 - RV64A, **126**
 - RV64C, **126, 134**
 - RV64D, **126**
 - RV64F, **126**
 - RV64G, 14
 - RV64I, **126**
 - RV64M, **126**
 - saved registers, 39
 - stack region, **47**
 - static region, **47**
 - temporary registers, 39
 - text region, **47**
 - Tree Sum, **229**
 - Zfh, **56**
- RISC-V ABI, ดู RISC-V
 - Application Binary Interface, 48
- RISC-V Application Binary Interface
 - ilp32, 48
 - ilp32d, 48
 - ilp32f, 48
 - lp64, 130
 - lp64d, 130
 - lp64f, 130
- RISC-V International, ii, 3, **3**, **5**, 14, 33, 164, 165
- rol, **123**

- rolw, 130
- room for growth, ดูเพิ่มเติม
 - instruction set architecture,
 - principles of design, room for growth
- ror, **123**
- roriw, 130
- rorw, 130
- RV128, 136
- RV32B, 164
- RV32C, 62
- RV32J, **164**
- RV32P, **164**
- RV32Q, **62**
- RV64V, 111

- Santayana, George, 31
- sb, ดูเพิ่มเติม Store byte, 216
- sc.d, ดูเพิ่มเติม Store conditional doubleword, 217
- sc.w, ดูเพิ่มเติม Store conditional word, 217
- scatter, 90
- scause, ดู Control and Status Register
- Schumacher, E. F., 74
- scounteren, ดู Control and Status Register
- sd, ดูเพิ่มเติม c.sdsp, ดูเพิ่มเติม c.sd, ดูเพิ่มเติม Store doubleword, 217
- selected element width, 87
- sepc, ดู Control and Status Register
- seqz, 41, 217
- Set less than, 24, 220
 - immediate, 24, 220
 - immediate unsigned, 24, 220
 - unsigned, 24, 220
- set-before-first mask bit, 102
- set-including-first mask bit, 102
- set-only-first mask bit, 102
- seven metrics of ISA design, ดูเพิ่มเติม instruction set architecture, metrics of design
- SEW, 87, 88, 94, 96
- sext, **123**
- sext.w, 41, 218
- sfence.vma, ดูเพิ่มเติม Fence Virtual Memory, 218
- sgtz, 41, 218
- sh, ดูเพิ่มเติม Store halfword, 218
- sh1add, **123**
- sh1add.uw, 130
- sh2add, **123**
- sh2add.uw, 130
- sh3add, **123**
- sh3add.uw, 130
- Shift
 - left logical, 24, 219
 - left logical immediate, 24, 219
 - left logical immediate word, 126, 219
 - left logical word, 126, 219
 - right arithmetic, 24, 221
 - right arithmetic immediate, 24, 221
 - right arithmetic immediate word, 126, 222
 - right arithmetic word, 126, 222
 - right logical, 24, 223
 - right logical immediate, 24, 223
 - right logical immediate word, 126, 223
 - right logical word, 126, 224
- SIMD, ดูเพิ่มเติม Single Instruction Multiple Data
- simplicity, ดูเพิ่มเติม instruction set architecture, principles of design, simplicity
- single bit manipulation, **124**
- Single Instruction Multiple Data, 3, 14
- Single Instruction, Multiple Data, 82
- sll, ดูเพิ่มเติม Shift left logical, 219
- slli, ดูเพิ่มเติม c.slli, ดูเพิ่มเติม Shift left logical immediate, 219
- slli.uw, 130
- slliw, ดูเพิ่มเติม Shift left logical immediate word, 219
- sllw, ดูเพิ่มเติม Shift left logical word, 219
- slt, ดูเพิ่มเติม Set less than, 220
- slti, ดูเพิ่มเติม Set less than immediate, 220
- sltiu, ดูเพิ่มเติม Set less than immediate unsigned, 220
- sltu, ดูเพิ่มเติม Set less than unsigned, 220
- sltz, 41, 221
- Small is Beautiful, 74
- Smith, Jim, 114
- snez, 41, 221
- sra, ดูเพิ่มเติม Shift right arithmetic, 221
- srai, ดูเพิ่มเติม c.srai, ดูเพิ่มเติม Shift right arithmetic immediate, 221
- sraiw, ดูเพิ่มเติม Shift right arithmetic immediate word, 222
- sraw, ดูเพิ่มเติม Shift right arithmetic word, 222
- sret, ดูเพิ่มเติม Exception Return Supervisor, 222
- srl, ดูเพิ่มเติม Shift right logical, 223
- srli, ดูเพิ่มเติม c.srli, ดูเพิ่มเติม Shift right logical immediate, 223
- srlw, ดูเพิ่มเติม Shift right logical immediate word, 223
- srlw, ดูเพิ่มเติม Shift right logical word, 224
- sscratch, ดู Control and Status Register
- static linking, **49**
- Store
 - byte, 26, 216
 - conditional doubleword, 126, 217
 - word, 70, 217
 - doubleword, 126, 217
 - halfword, 26, 218
 - word, 26, 224
- strip-mining, 93
- stripmining, 109
- stval, ดู Control and Status Register
- stvec, ดู Control and Status Register
- sub, 24, ดูเพิ่มเติม Subtract, 95, ดูเพิ่มเติม c.sub, ดูเพิ่มเติม Subtract, 224
- Subtract, 24, 224
 - word, 126, 224
- subw, ดูเพิ่มเติม c.subw, ดูเพิ่มเติม Subtract word, 224
- superscalar, 3, 9, 76
- Sutherland, Ivan, 38
- sw, ดูเพิ่มเติม c.swsp, ดูเพิ่มเติม c.sw, ดูเพิ่มเติม Store word, 224

- tail, 41, 225
- Thoreau, Henry David, 165
- Thumb-2, 62
- TLB, 160
- TLB shutdown, 160
- Translation
 - Mnemonic, **240**
 - Opcode, **240**
- Translation Lookaside Buffer, 160

- Tree Sum, **229**
- Turing Award
 - Allen, Fran, *18*
 - Brooks, Fred, *161*
 - Dijkstra, Edsger W., *142*
 - Patterson, David, *ii*
 - Perlis, Alan, *164*
 - Sutherland, Ivan, *38*
 - Wirth, Niklaus, *77*
- User mode, *151*
- vadc, *105*
- vcompress, *105*
- vcpop, *102*
- Vector
 - gather, *90*
 - indexed load, *90*
 - indexed ordered load, *90*
 - indexed ordered store, *90*
 - indexed store, *90*
 - indexed unordered load, *90*
 - indexed unordered store, *90*
 - scatter, *90*
 - strided load, *89*
 - strided store, *89*
 - strip-mining, *109*
 - unit-stride store, *89*
 - unit-stride load, *89*
 - vectorizable, *92, 112*
- vector add with carry, *105*
- Vector architecture, **83**
 - dynamic typing, *87, 132*
 - type encoding, *88*
- vector element index, *102*
- vector length, *87*
- vector length multiplier, *88*
- vector length register, *87*
- vector mask population count, *102*
- Vector register gather, *103*
- vector segment load, *107*
- vector segment store, *107*
- vector slide-down, *104*
- vector unit-stride
 - fault-only-first load, *107*
- vector whole-register load, *106*
- vector whole-register store, *106*
- vectorizable, *92, 112*
- vfcvt, *99*
- vfirst, *102*
- vfmacc, *95*
- vfmv, *98*
- vfnvct, *99*
- vfrec7, *103*
- vfredosum, *101*
- vfredusum, *101*
- vfrsqrt7, *103*
- vfslide1down, *104*
- vfslide1up, *104*
- vfwcvt, *99*
- vid, *102*
- viota, *102*
- Virtual address, *156*
- Virtual memory, *156*
- vl, *87, 88*
- vl1re32, *106*
- vl2re32, *106*
- vl4re32, *106*
- vl8re32, *106*
- vldx, *90*
- vle16.v, *89*
- vle16ff, *107*
- vle32.v, *89*
- vle32ff, *107*
- vle64, *95*
- vle64.v, *89*
- vle64ff, *107*
- vle8.v, *89*
- vle8ff, *107*
- VLEN, **87, 88**
- VLMAX, *87, 95, 103*
- vlmul, *88*
- vloxe16.v, *90*
- vloxe32.v, *90*
- vloxe64.v, *90*
- vloxe8.v, *90*
- vlse16.v, *89*
- vlse32.v, *89*
- vlse64.v, *89*
- vlse8.v, *89*
- vlseg2e32, *107*
- vlseg3e8, *107*
- vlseg3e8, *107*
- vluxe16.v, *90*
- vluxe32.v, *90*
- vluxe64.v, *90*
- vluxe8.v, *90*
- vluxseg3ei32, *107*
- vma, *88*
- vmacd, *105*
- vmand, *92, 101*
- vmandn, *92, 101*
- vmerge, *103*
- vmnand, *92, 101*
- vmnor, *92, 101*
- vmor, *92, 101*
- vmorn, *92, 101*
- vmsbf, *102*
- vmsof, *102*
- vmv, *98*
- vmv1r, *98*
- vmv2r, *98*
- vmv4r, *98*
- vmv8r, *98*
- vmxnor, *92, 101*
- vmxor, *92, 101*
- vnclip, *103*
- vnclipu, *103*
- vnsl, *97*
- von Neumann architecture, *15*
- von Neumann, John, *15*
- vredmax, *101*
- vredmaxu, *101*
- vredmin, *101*
- vredminu, *101*
- vredsum, *101*
- vrgather, *103*
- vse16.v, *89*
- vse32.v, *89*
- vse64, *95*
- vse64.v, *89*
- vse8.v, *89*
- vsetivli, *93*
- vsetvl, *93*
- vsetvli, *93, 94*
- vsew, *88*
- vslide1down, *104*
- vslide1up, *104*
- vslidedown, *104*
- vslideup, *104*
- vsoxe16.v, *90*
- vsoxe32.v, *90*
- vsoxe64.v, *90*
- vsoxe8.v, *90*
- vsse16.v, *89*
- vsse32.v, *89*
- vsse64.v, *89*
- vsse8.v, *89*
- vstx, *90*
- vsuxe16.v, *90*
- vsuxe32.v, *90*
- vsuxe64.v, *90*
- vsuxe8.v, *90*
- vta, *88*
- vtype, *88, 88, 93*
- vwmul, *97*
- vwredsum, *101*
- vwredsumu, *101*
- Wait for Interrupt, *151, 225*
- wait, ดูเพิ่มเติม Wait for Interrupt, *225*
- Whole vector register move, *98*
- Widening instructions, *95, 96*
- William of Occam, *52*
- Wirth, Niklaus, *77*
- x86

- aaa instruction, 4
- aad instruction, 4
- aam instruction, 4
- aas instruction, 4
- code size, 10, 134
- DAXPY, **62**
- enter instruction, 8
- Insertion Sort, **30**
- instruction reference
- manual
 - number of pages, 15
 - ISA growth, 3
 - number of registers, 12
 - position independent code, 12
 - Tree Sum, **229**
- x86-64
 - AMD64, **133**
- x86-64 AVX-512, 111
- x86-64 AVX-512 SIMD, 109
- x86-64 AVX2, 111
- x86-64 AVX2 SIMD, 108, 109
- XLEN, 143
- xnor, **123**
- xor, 24, ดูเพิ่มเติม/ c.xor, ดูเพิ่มเติม/
 - Exclusive Or, 225
- xor properties, 26
- xor register exchange, 26
- xori, 24, ดูเพิ่มเติม/ Exclusive Or
 - immediate, 225
- yield, 6
- Zba, 122, **123**
- Zbb, 122, **123**
- Zbc, 122, **124**
- Zbs, 122, **124**
- zext, **123**
- Zk, **164**

คอลัฟอน Colophon

การออกแบบตัวอักษร / Typography

ภาษาไทยในหนังสือเล่มนี้เรียงพิมพ์ด้วยฟอนต์ TH Sarabun New (ทีเอช สารบรรณ นิว) ออกแบบโดย สุพจน์ เจริมลาก แห่งบริษัท กด รัตน์ ซึ่งเป็นบริษัทในเครือ Cadson Demak สำนักพิมพ์ตัวอักษรอิสระในกรุงเทพมหานคร

Suppakit Chalermarp, born and raised in Chonburi, designed TH Sarabun New for the 2006 national font competition organised by SIPA — Thailand’s Software Industry Promotion Agency — under a government mandate to replace proprietary commercial fonts, including Microsoft’s Angsana New, with freely licensed alternatives. His entry won. The font became the official typeface of Thai government documents and the Royal Gazette of Thailand. สารบรรณ หมายถึง กิจการเอกสาร – *documentary affairs*. It seemed appropriate for a book about computer architecture.

TH Sarabun New was chosen over Angsana New for reasons that matter in long technical text: its looped consonant heads follow classical Thai calligraphic proportion rather than the compressed strokes of older digital Thai fonts; its descenders clear properly beneath vowel marks without collision; and its slightly condensed geometry gives each line a natural reading rhythm even when Thai and English terms alternate within a sentence. When ผู้ใช้ (user) brings a below-vowel, an above-vowel, and a tone mark together against a ใ leading vowel, all four elements occupy separate vertical zones without fighting for space — a detail invisible when correct, painful when wrong. Angsana New gets this wrong. TH Sarabun New does not. Suppakit is also recognised internationally as an expert on South-east Asian minority script support, having extended that same care to Lao and others. We are glad his work is free. ขอขอบคุณคุณสุพจน์ที่ทำให้ผลงานนี้เข้าถึงได้อย่างเสรี

TH Sarabun New is scaled to match Latin uppercase height rather than lowercase, accommodating Thai’s tall vowel stacks that extend above the Latin cap line. English and mathematics are set in TeX Gyre Pagella, the free Palatino companion. Monospace code is set in TeX Gyre Cursor, the free Courier companion, scaled to 85% to optically match Thai body size, requiring custom escape mechanisms and explicit font restoration after every Thai term to prevent the verbatim environment from abandoning its post mid-listing.

textbfการเรียงพิมพ์หลายภาษา / Multilingual Typesetting

This book uses XeLaTeX with the Polyglossia multilingual framework rather than pdfLaTeX with Babel. The decision was not taken lightly — it was taken the moment we discovered that pdfLaTeX cannot render Thai at all.

Thai language comments inside assembly code listings are rendered via a custom escape mechanism in the listings package, with explicit font restoration after every term. Thai text inside BVerbatim environments required a separate approach. Both solutions are documented at riscvthai.org. Getting Thai text into verbatim code blocks is a feat the L^AT_EX documentation does not mention, because no one expected anyone to attempt it. We ported both techniques to the Spanish, Japanese, Greek, Chinese, and other language branches of this book.

Thai line breaking uses XeTeX’s ICU dictionary engine rather than TeX’s traditional hyphenation patterns. Thai has no spaces between words and no hyphens to break on. Getting this right without producing rivers of white space between borrowed transliterated words such as รีจิสเตอร์ (register) and โปรเซสเซอร์ (processor) is an ongoing negotiation between linguists, typographers, and the ICU consortium. นี่คือการพิมพ์หลายภาษาควรจะเป็น

Non-Gregorian dates use a purpose-built Buddhist Era calendar macro (thaitoday.tex) converting Gregorian dates to the Thai solar calendar with full Thai numeral output. This edition is dated พ.ศ. ๒๕๖๘.

สถิติอัตโนมัติ / Automatic Statistics

The document statistics appearing in the Preface — figures, sidebars, margin icons, elaboration boxes, citations, quotations, and page counts comparing this edition to the previous — are computed automatically each build by a two-pass counter system (stats.tex) that accumulates values across the build and persists them through a .sum file. David Patterson and Andrew Waterman counted these figures by hand from 2018. They will not need to do so again.

ระบบสร้างเอกสาร / Build System

The build system is a pair of shell scripts (`makepdf.sh` / `makepdf.cmd` for Linux, Mac, and Windows) that replaced an inherited `Makefile` of uncertain provenance and occasionally uncertain behaviour. The scripts handle four $\text{\LaTeX}$ passes, thirteen BibTeX compilation units, custom `makeindex` preamble injection for the index anchor, and intermediate file archiving with version-stamped filenames. They ask before destroying anything. The `Makefile` is survived by its comments.

The $\text{\LaTeX}$ class underlying this book descends from work by Prof. Armando Fox of UC Berkeley, co-creator with David Patterson of *Engineering Software as a Service* and Berkeley’s first MOOC — a book and course that themselves embody the open educational philosophy we try to honour here. Armando’s dream was a technical textbook that lived freely on the web, updated continuously, available to anyone. We inherited his `textbook.cls`, argued with it at length, kept it alive, and added Thai. We hope he would approve.

ภาพปก / Cover Art

Cover art was produced in GIMP from original artwork, with typography matched by hand to the interior fonts. The artwork preceded this book in its present form: it gave birth to the RISC-V Thai Calendar, twelve months of ISA chapter illustrations and quotations, copies of which hang on the walls of the authors in California. The calendar preceded this colophon. This colophon is, in some sense, the calendar’s revenge.

การเผยแพร่ / Availability

หนังสือเล่มนี้มีให้อ่านฟรีที่ riscvthai.org and at the official RISC-V Reader site riscvbook.com. Technical notes on Thai $\text{\LaTeX}$ typesetting developed during this project are documented at riscvthai.org. If you paid for this book, we would like to know who sold it to you.

คู่มือ RISC-V ฉบับภาษาไทย (Risc Five Reader, Thai language edition)
open architecture Atlas

October 22, 2018 1st edition 1st printing issued

November 12, 2018 2nd printing issued

Author David Patterson

Andrew Waterman

Translator Paul Sherman & Naruemon Ratanakunakorn

Issuer freely distributed

Distributor riscvthai.org • riscvbook.com

Selling no charge

Book design & production Paul Sherman & Naruemon Ratanakunakorn

Printing / Bookbinding JP Graphics, San Jose CA (jp-graphics.com)

Unauthorized copying of this document is prohibited except as permitted by copyright law. Electronic data and e-books by third parties other than the purchaser are not permitted, including for private use.

For inquiries regarding this book, please contact:

<http://riscvthai.org>

© 2017 David Patterson and Andrew Waterman

Thai translation rights arranged with David Patterson and Andrew Waterman through Paul Sherman.

Printed in Thailand

ISBN 978-4-8222-9281-2