北京大学计算机学院本科生课程

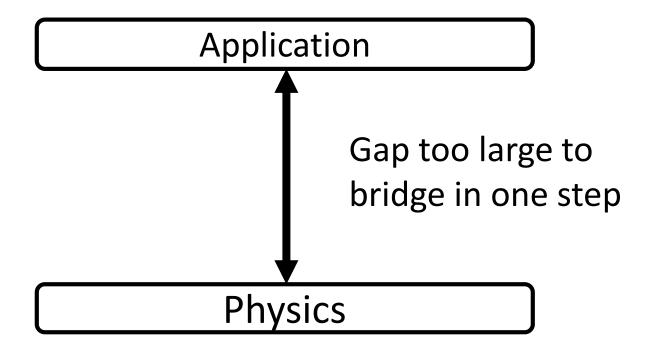
计算机组成与系统结构 实习 RISC-V ISA

北京大学微处理器研发中心

易江芳

2022-10-10

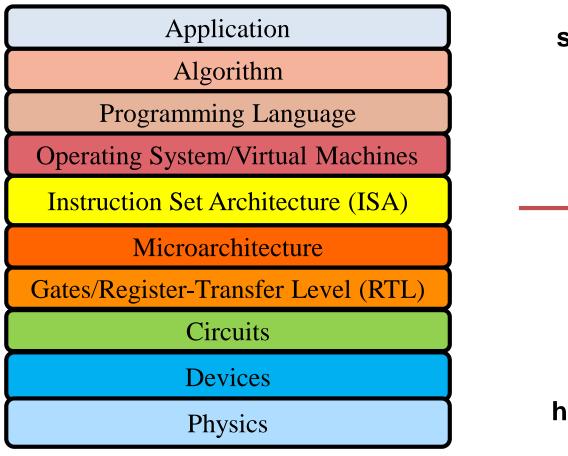
''| Computer Architecture

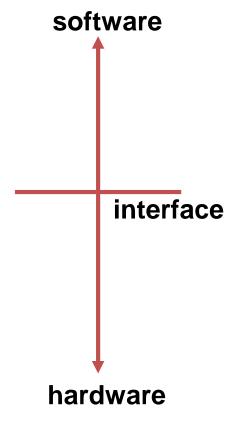


In its broadest definition, computer architecture is the *design of the abstraction layers* that allow us to implement information processing applications efficiently using available manufacturing technologies.

ISA: software/hardware interface







ISA is not only the set of instructions

- ISA不仅仅是指令的集合,还包括了存储管理、异常中断处理、 软硬件协同等多个方面
- 学习和设计ISA,需要建立高级语言与ISA相互映射的系统观
 - 算术语句 vs. 运算指令
 - 循环结构 vs. 分支转移
 - Switch结构 vs. 跳转
 - 静态全局变量 vs. 地址空间的静态数据区
 - 局部变量 vs. 栈
 - 动态分配存储 vs. 堆
 - 进程切换 vs. 存储管理和文件管理
 - 键盘和鼠标 vs. 中断处理

'I RISC ISA

- RISC philosophy
 - IBM 801
 - Cocke IBM, Patterson UCB, Hennessy Stanford, 1980s) Reduced Instruction Set Computing
 - IBM PowerPC、RISC I / RISC II、MIPS etc.
 - Keep the instruction set small and simple, in order to build fast hardware
 - Let software do complicated operations by composing simpler ones

'I RISC-V ISA

- Fifth generation of RISC design from UC Berkeley
- Realistic & complete ISA, but open & simple
- Not over-architected for a certain implementation style
- Both 32-bit and 64-bit address space variants
 - RV32 and RV64
- Easy to subset/extend for education/research
 - RV32IM, RV32IMA, RV32IMAFD, RV32G
 - RV64IM, RV64IMA, RV64IMAFD, RV64G
- Techreport with RISC-V spec available on class website or riscv.org
- We' II be using 64-bit RISC-V this semester in labs. Similar to MIPS you saw in Computer architecture.



'II RISC-V simple green card

-		IECN	n.	0	ARITHMETIC CORE	IN!	STRUCTION SET		2	
2/1	マ	DC V	Reference	Data	RV64M Multiply Extens	ion	NAME		-	
WELL BASE	INTE	GER INSTRUCTIONS, in al	phabetical order	Numer	mul, mulw		NAME MULtiply (Word)	DESCRIPTION (in Verilog)	NOTE	
WHEMONIC			DESCRIPTION (in Verilog) R[rd] = R[rs1] + R[rs2]	NOTE 1)	mulh	R	MULtiply upper Half	R[cd] = (R[cs1] * R[cs2]x(63:0) R[cd] = (R[cs1] * R[cs2]x(127:64)	1)	
House and W		ADD (Word) ADD Immediate (Word)	R[rd] = R[rs1] + R[rs2] $R[rd] = R[rs1] + imm$	1)	mulhsu	R	MULtiply upper Half Sign/Un	R[nl] = (R[nl] * R[n2])(127.64)		
di, addiw	I R	ADD Immediate (Word)	R[rd] = R[rs1] & R[rs2]	",	mulhu	R	MULtiply upper Half	R[nf] = (R[n1] * R[n2])(127.64)	6)	
d	1	AND Immediate	R[rd] = R[rs1] & imm		div, divw		DIVide (Word)	R[nl] = (R[nl] / R[n2])		
ipc	U	Add Upper Immediate to PC	R[rd] = PC + (imm, 12'b0)		divu		DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	1)	
id id	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}		remu, remuw	R	REMainder (Word)	Rindl = (Right) % Pro-20	1)	
	co	Branch Greater than or Equal	if(Rfrs17>=Rfrs2)		RV64F and RV64D Floa	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)	
ie.			PC=PC+{imm,100}		fld, flw	ting-	Point Extensions Load (Word)	F[rd] = M[R[rs1]+inm)		
eu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)	fod, faw	S	Store (Ward)	M[R[rs1]+imm] = Finf1	1)	
	60	Branch Less Than	if(R[rs1] <r[rs2) pc="PC+{imm,1b'0}</td"><td></td><td>fadd.s,fadd.d</td><td></td><td>ADD</td><td>F[rd] = F[rs1] + F[rs2]</td><td>7)</td><td></td></r[rs2)>		fadd.s,fadd.d		ADD	F[rd] = F[rs1] + F[rs2]	7)	
	SB	Branch Less Than Unsigned	if(R[rs1] <r[rs2) pc="PC+{imm,1b'0}</td"><td>2)</td><td>fsub.s,fsub.d fmul.s,fmul.d</td><td></td><td>SUBtract</td><td>$\mathbb{F}[\mathrm{rd}] = \mathbb{F}[\mathrm{rs1}] - \mathbb{F}[\mathrm{rs2}]$</td><td>7)</td><td></td></r[rs2)>	2)	fsub.s,fsub.d fmul.s,fmul.d		SUBtract	$\mathbb{F}[\mathrm{rd}] = \mathbb{F}[\mathrm{rs1}] - \mathbb{F}[\mathrm{rs2}]$	7)	
e u	cn	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+(imm,1b'0)		fdiv.s.fdiv.d		MULtiply DIVide	F[rd] = F[rd1] * F[rd2]	7)	
atc.	1	Cont/Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & -R[rs1]		faqrt.s,faqrt.d		SQuare RooT	F[rd] = F[rs1] / F[rs2] F[rd] = sqrt(F[rs1])	7)	
rrol	1	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~imm		fmadd.s, fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)	
	-	Imm Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR [R[rs1]		fmsub.s,fmsub.d	R	Multiply-SUBtract	F[m] = F[m1] * F[m2] - F[m3]	7)	
rrsi	1	Cont/Stat.RegRead&Set	R[rd] = CSR; CSR = CSR imm		fmnsub.s, fmnsub.d		Negative Multiply-SUBtract	F[rd] = -(F[rs1] * F[rs2] - F[rs3])	7)	
1100		Imm			fmnadd.s,fmnadd.d fsgni.s,fsgni.d		Negative Multiply-ADD SiGN source	F[rd] = -(F[rs1] * F[rs2] + F[rs3])	7)	
ITW	1	Cont/Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]		fsgnjn.s.fsgnjn.d		Negative SiGN source	F[nf] = (F[n2]=63>,F[n1]=62:0>) F[nf] = (c-F[n2]=63>)	7)	
LLM3	1	Cont/Stat.Reg Read&Write	R[rd] = CSR; CSR = imm					F[nd] = { (-F[n2]=63>), F[n1]=62:0>)	7)	
reak	1	Environment BREAK	Transfer control to debugger		fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = (F[rs2]<63>*F[rs1]<63>, F[rs1]<62:0>)	7)	
nll	1	Environment CALL	Transfer control to operating system		fmin.s, fmin.d	R	Minimum	F[m] = (F[m1] < F[m2]) 7 F[m1] : F[m2]	7)	
nce		Synch thread	Synchronizes threads		fmax.s, fmax.d	R	MAXimum	F[n2] F[n] = (F[n1] > F[n2]) ? F[n1] : F[n2]	7)	
nce.i	1	Synch Instr & Data	Synchronizes writes to instruction stream		feq.s, feq.d	D	Compare Float EQual	F[es2] R[ed] = (F[es1]== F[es2]) † 1 : 0		
1	111	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}		flt.s,flt.d		Compare Float Less Than	R[rd] = (F[rs1] = F[rs2]) 7 1 : 0 R[rd] = (F[rs1] < F[rs2]) 7 1 : 0	7)	
	1	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)	fle.s.fle.d	R	Compare Float Loss than or -	R[rd] = (F[rs1] == F[rs2]) 7:1:0	7)	
		Load Byte	R[rd] =	4)	fclass.s,fclass.d		Classify Type	R[rd] = class(F[rs1])	7,8)	
		Load Diggs Hardwood	{56'bM[](7),M[R[rs1]+imm](7:0)} P[rt] = (56'b0 M[R[rs1]+imm](7:0)}		fav.s.x,fav.d.x		Move from Imager	F[rd] = R[rs1]	7)	
1		Load Byte Unsigned Load Doubleword	R[rd] = {56'b0,M[R[rs1]+imm](7:0)} R[rd] = M[R[rs1]+imm](63:0)		fmv.x.s,fmv.x.d fcvt.s.d		Move to Integer Convert from DP to SP	R[rd] = F[rs1] F[rd] = single(F[rs1])	7)	
		Load Doubleword Load Halfword	Rfrd1 =	4)	fevt.m.d fevt.d.s		Convert from SP to SP Convert from SP to DP	F[rd] = single(F[rs1]) F[rd] = double(F[rs1])		
			{48'bM[](15),M[R[rs1]+imm](15:0)}		fort.s.w.fort.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:9))	7)	
d		Load Halfword Unsigned	$R[rd] = \{48'b0,M[R[rs1]+imm](15:0)\}$		fovt.s.l,fovt.d.l	R	Convert from 64b Integer	F[rd] - float(R[rs1](63:9))	7)	
	U	Load Upper Immediate	R[rd] = (32b'imm<31>, imm, 12'b0)	-	fort.s.wu, fort.d.wu	R	Convert from 32b Int Linsigne		2,7)	
	E	Load Word	$R[rd] = $ ${32^{b}M[](31),M[R[rs1]+imm](31:0)}$	4)	fort.s.lu, fort.d.lu	R	Convert from 64b Int Unsigne		2,7)	
1	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}		fcvt.w.s.fcvt.w.d fcvt.l.s.fcvt.l.d		Convert to 32b Integer Convert to 64b Integer	R[rd](31:0) = integer(F[rs1]) R[rd](63:0) = integer(F[rs1])	7)	
	R	OR	$R[rd] = R[rs1] \mid R[rs2]$		fevt.wu.s,fevt.wu.d			R[rd](31:0) = integer(F[rs1])	2,7)	
		OR Immediate	$R[rd] = R[rs1] \mid imm$		fort.lu.s, fort.lu.d	R			2,7)	
		Store Byte Store Doubleword	M[R[rs1]+imm](7:0) = R[rs2](7:0) M[R[rs1]+imm](63:0) = R[rs2](63:0)		RV64A Atomic Extensi	on			9)	
		Store Doubleword Store Halfword	M[R[rs1]+imm](63:0) = R[rs2](63:0) M[R[rs1]+imm](15:0) = R[rs2](15:0)		amoadd.w, amoadd.d		ADD	M[R[rs1]] = M[R[rs1]] + R[rs2]		
,allw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)	amound.w, amound.d		AND	K[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)	
i,slliw	1	Shift Left Immediate (Word)	$R[rd] = R[rs1] \ll imm$	1)	amomax.w,amomax.d		MAXimum	R[d] = M[R[n1]], M[R[n1]] = M[R[n1]] + R[n2] R[d] = M[R[n1]], M[R[n1]] = M[R[n1]] & R[n2] R[d] = M[R[n1]], I(R[n2] > M[R[n1]]) M[R[n1]] = R[n2]	9)	
		Set Less Than	$R[rd] = (R[rs1] \le R[rs2]) ? 1 : 0$		anomasu.w,anomasu.d	R	MAXimum Unsigned		2,9)	
i iu		Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1:0	(2)	amomin.w,amomin.d		MINimum	$\vec{\pi}(R[n2] > M[R[n1]]) M[R[n1]] = R[n2]$ R[n] = M[R[n1]], $\vec{\pi}(R[n2] < M[R[n1]]) M[R[n1]] = R[n2]$	9)	
0		Set < Immediate Unsigned Set Less Than Unsigned	$R[rd] = (R[rs1] \le imm) ? 1 : 0$ $R[rd] = (R[rs1] \le R[rs2]) ? 1 : 0$	2)	amominu.w,amominu.d		MINimum Unsigned	f(R[n2] < M[R[n1]) M[R[n1]] = R[n2] R[n] = M[R[n1]].	2.9)	
, SIAN		Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)			OR		9)	
i, sraiw	1	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> R[rs2] $R[rd] = R[rs1] >> imm$	1,5)	amoor.w,amoor.d			M[R[cs1]] - M[R[cs1]] R[cs2]	9)	
nelw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)	anoswap.w, amoswap.d		SWAP XOR	R[ni] = M[R[ni]], M[R[ni]] = R[n2] R[ni] = M[R[ni]].	9) 9)	
, scliw	1	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)	amoxor.w,amoxor.d			R[nl] = M[R[nl]], $M[R[nl]] = M[R[nl]] \cap R[n2],$ R[nl] = M[R[nl]],	*)	
subv	R	SUBtract (Word)	R[rd] = R[rs1] - R[rs2]	1)	ir.w,ir.d		Load Reserved	R[n1] = M[R[n1]], reservation on M[R[rs1]] if reserved, M[R[rs1]] = R[rs2],		
			M[R[rs1]+imm](31:0) = R[rs2](31:0)		sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rs1] = 0; else R[rs1] = 1		
	1	XOR Immediate	$R[rd] = R[rs1] \wedge R[rs2]$ $R[rd] = R[rs1] \wedge imm$							
a: I) The F	Ford v	ersion only operates on the ri	R[rd] = R[rs1] ^ imm ghtmost 32 bits of a 64-bit registers		CORE INSTRUCTIO			15 14 12	6	
					31 27	26.	25 24 20 19	15 14 12 11 7	6 0	
-y. 2190 0	COAT AT	gnificant bit of the branch ado	fress in fair is set to 0		R funct7		rs2 rs1	funct3 rd	Opcode	
5) Repli	cates i	ad instructions extend the sign the sign bit to fill in the letting	thit of data to fill the 64-bit register of hits of the result during right shift			111:		funct3 rd funct3 imm[4:0]	Opcode opcode	
					S imm[11:5		rs2 rs1	funct3 imm[4:0] funct3 imm[4:1]11]	opcode	
bit F	registe	version does a single-precision er	n operation using the rightmost 32 bits		SB imm[12]10:	2]	rs2 rs1 imm[31:12]	funct3 [mm[4:1[11]]	opcode	
8) Class	ifv wn	ites a 10-hit morek to chow whi	ch properties are true (c.g., -inf0,+0		U	L	imm[31:12] im[20 10:1 11 19:12]	rd rd	opeode	
					UJ	-100	[
hrite.	of the	mory operation; nothing else co	an interpose itself between the read and	I the						
The imme	diate f	memory tocation field is sign-extended in RISC-	V							
		3 11 1000								
18 by E.	16					2011		a. DISC-V Edition		
by Else	vier,	Inc. All rights reserved. Fro	m Patterson and Hennessy, Compu	ter Organi	ization and Design: The H	arch	vare/Software interfac	e. Arac-r banton		

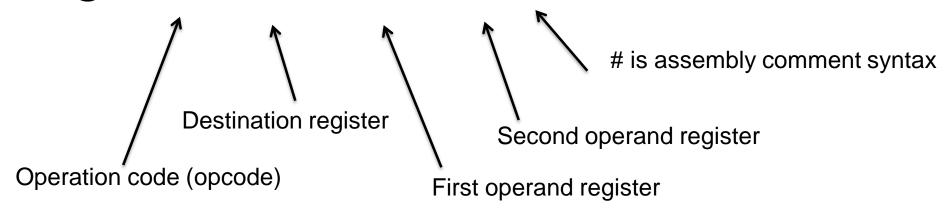
RISC-V RV64I Simple Green Card

In about the in	T			_	oreen caru
Instruction	Type	Opcode		Funct7/IMM	
add rd, rs1, rs2	4		0x0	0x00	$R[rd] \leftarrow R[rs1] + R[rs2]$
mul rd, rs1, rs2	-		0x0	0x01	R[rd] ← (R[rs1] * R[rs2])[31:0]
sub rd, rs1, rs2	1		0x0	0x20	R[rd] ← R[rs1] - R[rs2]
sll rd, rs1, rs2	1		0x1	0x00	R[rd] ← R[rs1] << R[rs2
mulh rd, rs1, rs2	1		0x1	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32]
slt rd, rs1, rs2	1		0x2	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0
xor rd, rs1, rs2	R	0x33	0x4	0x00	R[rd] ← R[rs1] ^ R[rs2]
div rd, rs1, rs2]		0x4	0x01	R[rd] ← R[rs1] / R[rs2]
srl rd, rs1, rs2			0x5	0x00	R[rd] ← R[rs1] >> R[rs2]
sra rd, rs1, rs2			0x5	0x20	R[rd] ← R[rs1] >> R[rs2]
or rd, rs1, rs2			0x6	0x00	R[rd] ← R[rs1] R[rs2]
rem rd, rs1, rs2			0x6	0x01	R[rd] ← (R[rs1] % R[rs2]
and rd, rs1, rs2			0x7	0x00	R[rd] ← R[rs1] & R[rs2]
lb rd, offset(rs1)			0x0		R[rd] ← SignExt(Mem(R[rs1] + offset, byte))
Ih rd, offset(rs1)	1	0x03	0x1		R[rd] ← SignExt(Mem(R[rs1] + offset, half))
lw rd, offset(rs1)	1	UXUS	0x2		R[rd] ← Mem(R[rs1] + offset, word)
ld rd, offset(rs1)	1		ox3		R[rd] ← Mem(R[rs1] + offset, doubleword)
addi rd, rs1, imm	1		0x0		$R[rd] \leftarrow R[rs1] + imm$
slli rd, rs1, imm	1		0x1	0x00	$R[rd] \leftarrow R[rs1] << imm$
slti rd, rs1, imm	1		0x2		$R[rd] \leftarrow (R[rs1] < imm) ? 1 : 0$
xori rd, rs1, imm	1		0x4		$R[rd] \leftarrow R[rs1] \land imm$
srli rd, rs1, imm	1 .	0x13	0x5	0x00	$R[rd] \leftarrow R[rs1] >> imm$
srai rd, rs1, imm	'		0x5	0x20	$R[rd] \leftarrow R[rs1] >> imm$
ori rd, rs1, imm	1		0x6		R[rd] ← R[rs1] imm
andi rd, rs1, imm	1		0x7		$R[rd] \leftarrow R[rs1] \& imm$
addiw rd, rs1, imm	1	0x1B	0x0		$R[rd] \leftarrow SignExt(R[rs1](31:0) + imm)$
Jalr rd, rs1, imm	1	0x67	0x0		R[rd] ← PC + 4
	1				PC ← R[rs1] + {imm, 1b'0}
ecall	1		0x0	0x000	(Transfers control to operating system)
	1	0x73			a0 = 1 is print value of a1 as an integer.
	1				a0 = 10 is exit or end of code indicator.
sb rs2, offset(rs1)			0x0		Mem(R[rs1] + offset) ← R[rs2][7:0]
sh rs2, offset(rs1)	1		0x1		Mem(R[rs1] + offset) ← R[rs2][15:0]
sw rs2, offset(rs1)	S	0x23	0x2		Mem(R[rs1] + offset) ← R[rs2][31:0]
sd rs2, offset(rs1)	1		0x3		Mem(R[rs1] + offset) ← R[rs2][63:0]
beg rs1, rs2, offset			0x0		if(R[rs1] == R[rs2])
beq 131, 132, 0113et	1		OAO		PC ← PC + {offset, 1b'0}
bne rs1, rs2, offset	1		0x1		if(R[rs1] != R[rs2])
blie 131, 132, 0113et	1		OXI		PC ← PC + {offset, 1b'0}
blt rs1, rs2, offset	SB	0x63	0x4		if(R[rs1] < R[rs2])
bit 131, 132, 0113et	1		0.4		PC ← PC + {offset, 1b'0}
bge rs1, rs2, offset	1		0x5		if(R[rs1] >= R[rs2])
uge 131, 152, Ullset	1		UXS		
auina rd. offsat		0x17			PC ← PC + {offset, 1b'0}
auipc rd, offset	U				R[rd] ← PC + {offset, 12'b0}
lui rd, offset		0x37			R[rd] ← {offset, 12'b0}
jal rd, imm	UJ	0x6f			R[rd] ← PC + 4
					PC ← PC + {imm, 1b'0}

III RISC-V Instruction Assembly Syntax

Instructions have an opcode and operands

• E.g., add x1, x2, x3 # x1 = x2 + x3



Assembly

- Example: add x1, x2, x3 (in RISC-V)

- Equivalent to: a = b + c (in C)

where C variables \Leftrightarrow RISC-V registers are : a \Leftrightarrow x1, b \Leftrightarrow x2, c \Leftrightarrow x3

Summary of RISC-V Instruction Formats

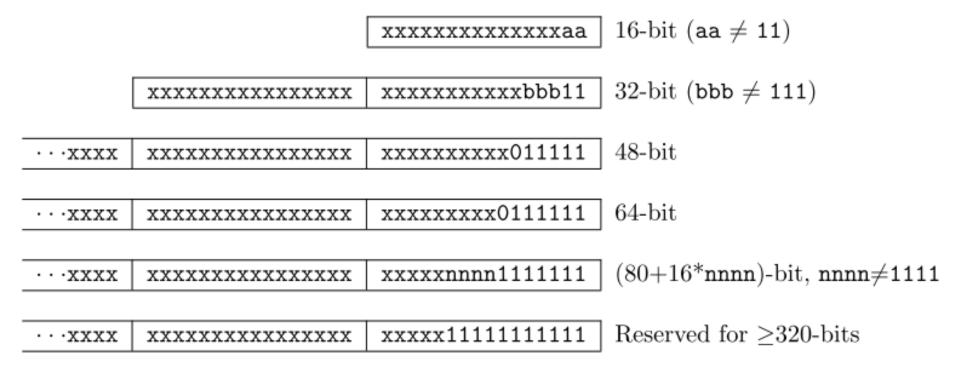
Additional opcode Source		Source Destina		ation 7-bit opco	de field	
bits/immediate	Reg. 2	Reg. 1	Reg	(but low 2 l	oits =11 ₂)	
31 30 2	5 24 21 20	19 18	5 14 12	2 11 8 7	6 0	/>
funct7	rs2	rs1	funct3	rd	opcode	R-type
74 <u>-</u>	V-10			20		i
imm[1	1:0]	rs1	funct3	rd	opcode	I-type
70 <u>-</u>		·	10	VC		
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
W			97	VO 00 2012		i
$imm[12] \mid imm[10:5]$	rs2	rs1	funct3	$ \operatorname{imm}[4:1] \operatorname{imm}[11]$	opcode	B-type
W	imm[31:12]					
	opcode	U-type				
W						I
[imm[20]] $[imm[1]$	[0:1] $[imm[11]]$	imm[1	.9:12]	rd	opcode	J-type
1.0 to 10 to	***			in.	:: <u>:</u>	ヽ ノ

- Aligned on a four-byte boundary in memory
- Sign bit of immediates always on bit 31 of instruction.
- Register fields never move
- Opcode fields is on the right

'|| Where is NOP?

Addi x0, x0, #0

III RISC-V Instruction Encoding



- Base instruction set (RV32 and RV64) always has fixed 32-bit instructions lowest two bits = 11₂
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)
 - Still will cause a fault if fetching a 32-bit instruction

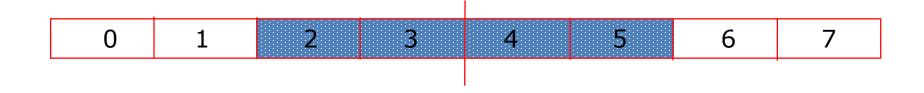
Data formats and addressing

Data formats

8b Bytes, 16b Half words, 32b words and 64b double words

Some issues

- Byte addressing
- Little endian
- Word alignment



Example: RISC-V rv32I/64I R-format instructions

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
		•	1		1	-

Different encoding in funct7 + funct3 selects different operations

III RISC-V I-Format Instruction

RISC-V Assembly Instruction:

addi x15, x1, -50

31	20 19	15 14	12 11	7 6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
		1	1	
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	ADD	rd=15	OP-Imm

- imm[11:0] can hold values in range [-2048₁₀, +2047₁₀]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation

Support 32-bit ops in rv64l

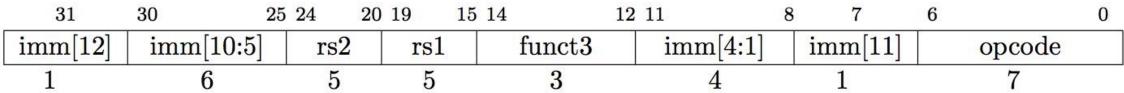
- Integer Register-Immediate Instructions
 - ADDIW
 - Rd = (sign extend)(rs1 + imm12)
 - Ignore overflow

31	. 20	19 15	5 14 12	2 11 7	6 0
	imm[11:0]	rs1	funct3	rd	opcode
	12	5	3	5	7
	I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

Shift instruction

31	26	25	24 20	19	15 14	12 11	7 6	0
imm[11:	6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode	
6		1	5	5	3	5	7	
00000	0	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM	
00000	0	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM	
01000	0	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM	
00000	0	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32	
00000	0	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32	
01000	0	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32	

III RISC-V B-Format for Branches



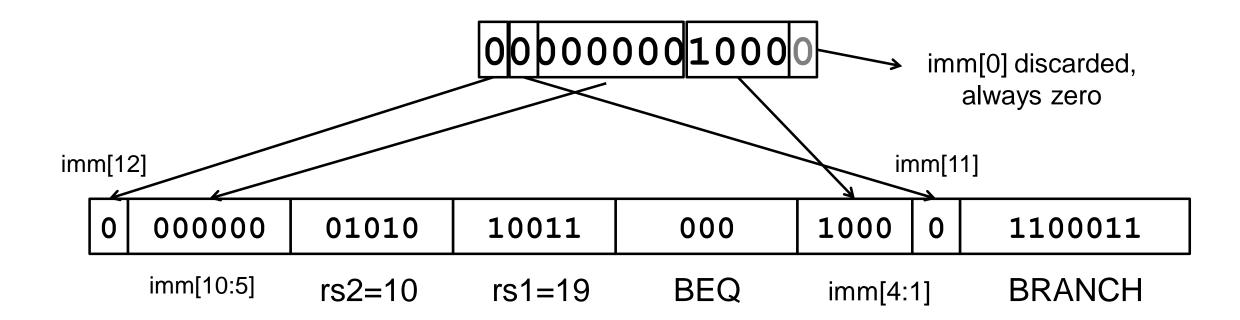
- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)



'|| Branch Example

beq x19,x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



** U-Format for "Upper Immediate" instructions

31		12 11	7 6	0
imm[31:1	[2]	rd	opcode	
20		5	7	
U-immediate	e[31:12]	dest	LUI	
U-immediate	e[31:12]	dest	AUIPC	

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - LUI Load Upper Immediate
 - AUIPC Add Upper Immediate to PC

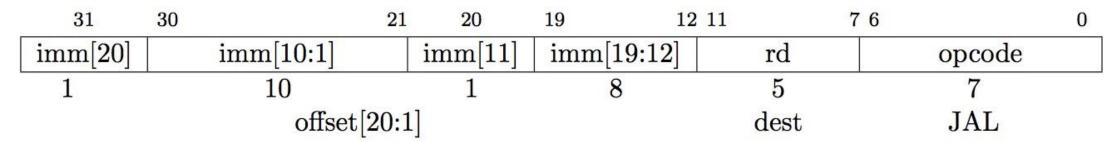
LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654 # x10 = 0x87654000
ADDI x10, x10, 0x321 # x10 = 0x87654321
```

思考:如果要得到OxDEADBEEF,如何?

III RISC-V J-Format for Jump Instructions



- JAL saves PC+4 in register rd (the return address)
 - Assembler "j" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - ±2¹⁸ 32-bit instructions
- Example:

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address
# Call function within 2<sup>18</sup> instructions of PC
jal ra, FuncName
```

III JALR Instruction (I-Format)

31		20 19	15 14 12	11	7 6	0
	imm[11:0]	rs1	funct3	rd	opcode	
2	12	5	3	5	7	
	offset[11:0]	base	. 0	dest	JALR	

- JALR rd, rs, immediate
 - Writes PC+4 to rd (return address)
 - Sets PC = rs + immediate
 - Uses same immediates as arithmetic and loads
 - no multiplication by 2 bytes

Example:

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit
absolute address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit
offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

III RISC-V Pseudo-instructions

PSEUDO INSTRUCTIONS



MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1] < 0)? -F[rs1]: F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = -F[rs1]	fsgnjn
j	Jump	$PC = \{imm, 1b'0\}$	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = -R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	$R[rd] = \sim R[rs1]$	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1] == 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]! = 0)?1:0	sltu

- 使用编译选项,可在编译用户程序时不生成伪指令:
 - riscv64-unknown-elf-objdump -Mno-aliases -D xxx

'| RISC-V 标准扩展集

- M集:标准整数乘除扩展集,在整数集的基础上增加了乘法和除法运算
- A集:标准原子指令扩展集,使得两个处理器之间同步原子地读,修改和写memory
- F集:标准单精度浮点扩展集,有浮点的运算指令和load/store指令
- D集:标准双精度浮点扩展集,有双精度的运算指令和load/store指令
- Q集:标准四倍精度浮点扩展集,有四倍精度的运算指令和load/store指令
- L集:标准十进制浮点扩展集,用于64位,128位的浮点运算
- C集:标准压缩指令扩展集,将32位的指令能与16位的指令相混合
- B集:标准位操作扩展集,能插入,删除,测试比特域
- T集:标准内存交易扩展集,用于内存交易
- P集:标准Packed-SIMD指令集,用于打包的SIMD指令

'|| Lab2.1中涉及的指令

RISC-V RV64I Simple Green Card

Description	Instruction	Type	Opcode	Funct3	Funct7/IMM	Operation
mul rd, rs1, rs2 sub rd, rs2, sub rd, rs1, rs2 sub rd, rs2, sub rd, rs1, rs2 sub rd, rs2, sub	add rd, rs1, rs2			0x0	0x00	R[rd] ← R[rs1] + R[rs2]
Sub rd, rs1, rs2 Sil rd, rs1, rs2 Sit rd, rs1, rs2 Sir rd, rs2, rs2 Sir rd, rs2, rs3 Sir rd, rs2, rs3 Sir rd, rs4, rs4 Sir rd, rs4, rs5 Sir rd, rs6, rs5 Sir rd, rs6, rs6 Sir rd, rs6, rs6, rs6 Sir rd, rs6, rs6, rs6, rs6, rs6, rs6, rs6, rs6						
Sil rd, rs1, rs2 mulh rd, rs2, rs2 mulh rd, rs3, rs2 mulh rd, rs4, rs5, rs5, rs5, rs5, rs5, rs5, rs5, rs5						
mulh rd, rs1, rs2 sit rd, rs1, rs2 sit rd, rs1, rs2 div rd, rs1, rs2 sr rd, rs1, rs2 or rd, rs1, rs2 ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] ox 6 ox 00 R[rd] + R[rs1] × R[rs2] radius ox 00 R[rd] + R[rs1] × R[rs2] radius						
Sit rd, rs1, rs2 Xor rd, rs3 Xor rd, rs4 Xor rd, rs5 Xor rd, rs5						
No. No.						
div rd, rs1, rs2 srl rd, rs1, rs2 srl rd, rs1, rs2 srl rd, rs1, rs2 or rd, rs1, rs2 0x5 0x00 R[rd] - R[rs1] >> R[rs2] or rd, rs1, rs2 0x5 0x00 R[rd] - R[rs1] >> R[rs2] or rd, rs1, rs2 0x6 0x00 R[rd] - R[rs1] >> R[rs2] ox6 0x01 R[rd] - R[rs1] >> R[rs2] ox6 0x01 R[rd] - R[rs1] >> R[rs2] ox6 0x01 R[rd] - R[rs1] >> R[rs2] ox7 0x00 R[rd] - R[rs1] >> R[rs2] ox7 0x00 R[rd] - R[rs1] >> R[rs2] ox8 0x13 R[rd] - R[rs1] >> R[rs2] ox9 R[rd] - R[rs1] >> R[rs2] ox0 R[rd] - R[rs1] >> R[rs2] ox1 R[rd] - R[rs1] >> R[rs2] ox2 R[rd] - R[rs1] >> R[rs2] ox3 R[rd] - Mem(R[rs1] + offset, byte)) ox1 R[rd] - Mem(R[rs1] + offset, byte) ox3 R[rd] - Mem(R[rs1] + offset, byte) ox4 R[rd] - R[rs1] + R[rs2] ox5 0x20 R[rd] - R[rs1] + mm ox1 0x3	, ,	R	0x33			
Sri rd, rs1, rs2 Sra rd, rs1, rs2 Sra rd, rs1, rs2 Ox5 Ox00 R[rd] + R[rs1] > R[rs2] Ox5 Ox20 R[rd] + R[rs1] > R[rs2] Ox6 Ox00 R[rd] + R[rs1] R[rs2] Ox6 Ox00 R[rd] + R[rs1] R[rs2] Ox7 Ox10 Ox11 Ox00 R[rd] + R[rs1] R[rs2] Ox13 Ox13 R[rd] + R[rs1] R[rs2] R[rs2] Ox13 R[rd] + R[rs1] R[rs2] R[rs2] Ox13 R[rd] + R[rs1] R[rs2] R[rs2] Ox14 R[rs1] R[rs2] R[rs2]						
Star dr, rs1, rs2 Or rd, rs1, rs2 Or rd, rs1, rs2 Ox6 Ox00 R[rd] - R[rs1] >> R[rs2] Ox6 Ox00 R[rd] - R[rs1] R[rs2] Ox6 Ox01 R[rd] - R[rs1] R[rs2] Ox6 Ox01 R[rd] - R[rs1] R[rs2] Ox7 Ox00 R[rd] - R[rs1] Ox11 Ox00 R[rd] - R[rs1] Ox12 Ox3 R[rd] - Mem(R[rs1] + offset, word) Ox3 R[rd] - R[rs1] Cx12 Ox33 R[rd] - R[rs1] Cx13 Ox13 Ox13 R[rd] - R[rs1] Cx13 Ox13 Ox14 R[rd] - R[rs1] Cx13 Ox14 R[rd] - R[rs1] Cx13 Ox15 Ox15 Ox16 R[rd] - R[rs1] Cx13 Ox14 R[rd] - R[rs1] Cx13 Ox15 Ox15						
or rd, rs1, rs2 0x6 0x00 R[rd] - R[rs1] R[rs2] ox do, rs1, rs2 0x6 0x01 R[rd] - R[rs1] & R[rs2] lb rd, offset(rs1) 0x7 0x00 R[rd] - SignExt(Mem(R[rs1] + offset, byte)) lb rd, offset(rs1) 0x1 R[rd] - SignExt(Mem(R[rs1] + offset, byte)) lb rd, offset(rs1) 0x2 R[rd] - Mem(R[rs1] + offset, word) ox1 0x00 R[rd] - R[rs1] * imm slir rd, rs1, imm 0x1 0x00 R[rd] - R[rs1] + offset, doubleword) ox1 0x00 R[rd] - R[rs1] + imm 0x1 0x0 R[rd] - R[rs1] + imm ox1 0x00 R[rd] - R[rs1] + imm 0x1 0x0 R[rd] - R[rs1] + imm ox1 0x00 R[rd] - R[rs1] + imm 0x1 0x2 R[rd] - R[rs1] + imm ox1 0x00 R[rd] - R[rs1] + imm 0x2 R[rd] - R[rs1] + imm ox1 0x1 0x0 R[rd] - R[rs1] + imm ox2 0x2 R[rd] - R[rs1] + imm ox3 R[rd] - R[rs1] + imm ox4 R[rd] - R[rs1] + imm ox						
Description						
Dark						
In rd, offset(rs1) Iw rd, offset(rs1) Iw rd, offset(rs1) Iw rd, offset(rs1) Id rd, offset(rs1) addird, rs1, imm slli rd, rs1, imm slli rd, rs1, imm slli rd, rs1, imm srli rd, rs1, imm addiw rd, rs1, imm srli rd, rs1, i					0.00	
No. No.						
	- , ,		0x03			
Document Document	. , ,					
Sili rd, rs1, imm Siti	. ,					
Siti rd, rs1, imm Nori rd, rs1, imm Srii rd, rs1, imm Srii rd, rs1, imm Srii rd, rs1, imm Srii rd, rs1, imm Ox5	, ,				0×00	
Note					UXUU	
Seri rd, rs1, imm Srai rd, rs1, imm Srai rd, rs1, imm Ori rd, rs1, imm Ori rd, rs1, imm Ori rd, rs1, imm Ox6 R[rd] + R[rs1] > imm Ox6 R[rd] + R[rs1] imm Ox7 R[rd] + R[rs1] imm Ox8 R[rd] + R[rs1] imm Ox9 R[rd] + R[rs1] imm Ox8 R[rd] +						
sraird, rs1, imm 0x5 0x20 R[rd] + R[rs1] > imm orird, rs1, imm 0x6 R[rd] + R[rs1] imm addiw rd, rs1, imm 0x1B 0x0 R[rd] + R[rs1] & imm 0x67 0x0 R[rd] - PC + 4 ecall 0x0 0x00 (Transfers control to operating system) 0x73 0x0 0x000 (Mem(R[rs1] + offset) + R[rs2][7:0] 0x1 0x1 0x1 0x1 0x1 0x1 0x1 0x1 0x1 0x2 0x3 0x4		1	0x1B		000	
ori rd, rs1, imm and rd, rs1, imm 0x6 R[rd] ← R[rs1] imm addiw rd, rs1, imm 0x1B 0x0 R[rd] ← R[rs1] & imm Jalr rd, rs1, imm 0x0 R[rd] ← PC + 4 ecall 0x0 0x000 (Transfers control to operating system) 0x73 0x0 0x000 (Transfers control to operating system) 0x0 0x0 0x000 (Transfers control to operating system) 0x1 0x0 0x0 0x000 (Mem(R[rs1] + offset) ← R[rs2][7:0] 0x1 0x2 0x2						
Dark					0x20	
addiw rd, rs1, imm Jalir rd, rs1, imm 0x0 R[rd] → SignExt(R[rs1](31:0) + imm) lecall 0x0 R[rd] → PC + 4 PC ← R[rs1] + (imm, 1b'0) ecall 0x0 0x0000 (Transfers control to operating system) 0x73 a0 = 1 is print value of a1 as an integer. a0 = 10 is exit or end of code indicator. sb rs2, offset(rs1) S 0x0 Mem(R[rs1] + offset) ← R[rs2][7:0] ox1 Mem(R[rs1] + offset) ← R[rs2][15:0] 0x2 Mem(R[rs1] + offset) ← R[rs2][31:0] ox3 Mem(R[rs1] + offset) ← R[rs2][63:0] 0x0 if(R[rs1] = R[rs2]) beq rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) 0x0 if(R[rs1] = R[rs2]) bbe rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) 0x0 if(R[rs1] = R[rs2]) bbe rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) 0x0 if(R[rs1] = R[rs2]) 0x1 if(R[rs1] = R[rs2]) 0x1 if(R[rs1] = R[rs2]) 0x4 if(R[rs1] = R[rs2]) 0x4 if(R[rs1] = R[rs2]) 0x5 if(R[rs1] = R[rs2]) 0x5 if(R[rs1] = R[rs2]) 0x5 </td <td></td> <td></td> <td></td> <td></td> <td></td>						
Jair rd, rs1, imm						
DX67 PC R[rs1] + (imm, 1b'0)						
0x0	Jair rd, rs1, imm			0x0		
0x73						
Second	ecall		0.70	0x0	0x000	
Sb rs2, offset(rs1) S S Ox0 Mem(R[rs1] + offset) \(R[rs2][7:0] \) Ox1 Mem(R[rs1] + offset) \(R[rs2][5:0] \) Ox2 Mem(R[rs1] + offset) \(R[rs2][15:0] \) Ox3 Mem(R[rs1] + offset) \(R[rs2][31:0] \) Ox3 Mem(R[rs1] + offset) \(R[rs2][31:0] \) Ox3 Mem(R[rs1] + offset) \(R[rs2][63:0] \) Ox4 if(R[rs1] = R[rs2]) Ox6 if(R[rs1] = R[rs2]) Ox7 if(R[rs1] + R[rs2]) Ox7 if(R[rs1] + R[rs2]) Ox7 R[rs2] \) Ox6 Ox7 R[rs2] \(PC + (offset, 1b'0) \) Ox5 if(R[rs1] = R[rs2]) Ox7 R[rs2] \(PC + (offset, 1b'0) \) Ox5 R[rs2] \(PC + (offset, 1b'0) \) Ox7 R[rs2] \(0x73			·
sh rs2, offset(rs1) S 0x1 Mem(R[rs1] + offset) + R[rs2][15:0] ox rs2, offset(rs1) 0x2 Mem(R[rs1] + offset) + R[rs2][31:0] beq rs1, rs2, offset 0x0 if(R[rs1] + offset) + R[rs2][63:0] bne rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) blt rs1, rs2, offset 0x1 if(R[rs1] + R[rs2]) pc - pc + C + (offset, 1b'0) 0x4 if(R[rs1] + R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] > R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] > R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] - P(rs2)) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] - R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] - R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] - R[rs2]) pc - pc + (offset, 1b'0) 0x5 if(R[rs1] - R[rs2]) pc - pc + (offset, 1b'0) 0x5 R[rd] - Pc + (offset, 1b'0) pc - pc + (offset, 1b'0) 0x6 R[rd] - Pc + (offset, 1b'0)						
sw rs2, offset(rs1) S 0x2 Mem(R[rs1] + offset) ← R[rs2][31:0] sd rs2, offset(rs1) 0x3 Mem(R[rs1] + offset) ← R[rs2][63:0] beq rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) bne rs1, rs2, offset 0x63 if(R[rs1] != R[rs2]) bt rs1, rs2, offset 0x63 if(R[rs1] != R[rs2]) bge rs1, rs2, offset 0x64 if(R[rs1] <= R[rs2])						
sd rs2, offset(rs1) 0x3 Mem(R[rs1] + offset) ← R[rs2][63:0] beq rs1, rs2, offset 0x0 if(R[rs1] = R[rs2]) bne rs1, rs2, offset 0x63 PC ← PC + {offset, 1b'0} blt rs1, rs2, offset 0x63 if(R[rs1] != R[rs2]) blt rs1, rs2, offset PC ← PC + {offset, 1b'0} bge rs1, rs2, offset 0x4 if(R[rs1] <= R[rs2])		s	0x23			
beq rs1, rs2, offset Data						
Description						
bne rs1, rs2, offset Ox1 if(R[rs1] != R[rs2]) blt rs1, rs2, offset PC ← PC + {offset, 1b'0} bge rs1, rs2, offset PC ← PC + {offset, 1b'0} bge rs1, rs2, offset PC ← PC + {offset, 1b'0} auipc rd, offset PC ← PC + {offset, 1b'0} lui rd, offset V Ox37 R[rd] ← PC + {offset, 12'b0} jal rd, imm UJ Ox6f R[rd] ← PC + 4	beq rs1, rs2, offset			0x0		
Dit rs1, rs2, offset						
$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$	bne rs1, rs2, offset			0x1		
Dot		SB	0x63			
	blt rs1, rs2, offset			0x4		
PC ← PC + {offset, 1b'0} auipc rd, offset						
auipc rd, offset U 0x17 R[rd] ← PC + {offset, 12'b0} lui rd, offset 0x37 R[rd] ← {offset, 12'b0} jal rd, imm UJ 0x6f R[rd] ← PC + 4	bge rs1, rs2, offset			0x5		
lui rd, offset 0 0x37 R[rd] ← {offset, 12'b0} jal rd, imm UJ 0x6f R[rd] ← PC + 4						
lui rd, offset		U				
	lui rd, offset		0x37			
	jal rd, imm	UJ	0x6f			
PC ← PC + {imm, 1b'0}						PC ← PC + {imm, 1b'0}

For further reference, here are the bit lengths of the instruction components

R-TYPE	funct7	rs2	rs1	funct3	rd	opcode
Bits	7	5	5	3	5	7

I-TYPE	imm[11:0]	rs1	funct3	rd	opcode
Bits	12	5	3	5	7

S-TYPE	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
Bits	7	5	5	3	5	7

SB-TYPE	imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
Bits	1	6	5	5	3	4	1	7

U-TYPE	imm[31:12]	rd	opcode
Bits	20	5	7

UJ-TYPE	imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
Bits	1	10	1	8	5	7

执行结果参考:

https://kvakil.github.io/venus/

III RISC-V Functional Call

- 1. Put parameters in a place where function can access them
- 2. Transfer control to function
- 3. Acquire (local) storage resources needed for function
- 4. Perform desired task of the function
- 5. Put result value in a place where calling code can access it and restore any registers you used
- 6. Return control to point of origin, since a function can be called from several points in a program

III RISC-V calling convention register usage

Register	ABI Name	Description	Saver
х0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

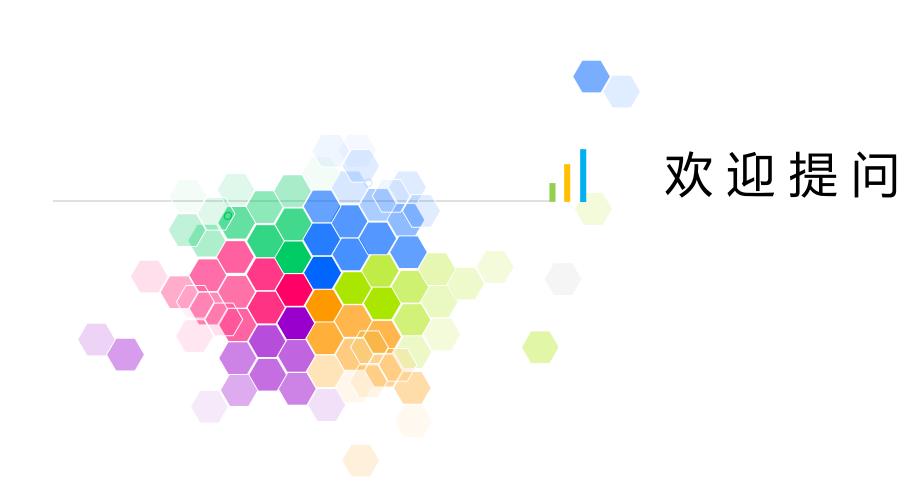
- 1. Preserved across function call
 - Caller can rely on values being unchanged
 - sp, gp, tp, "saved registers" s0-s11(s0 is also fp)
- 2. Not preserved across function call
 - Caller cannot rely on values being unchanged
 - Argument/return registers a0-a7,ra, "temporary registers" t0t6

'I Allocating Space on Stack

- C has two storage classes: automatic and static
 - Automatic variables are local to function and discarded when function exits
 - Static variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- Procedure frame or activation record: segment of stack with saved registers and local variables

III RISC-V RV32 memory map

- RV64 convention (RV32 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
 - Hexadecimal (base 16) : bfff_fff0_{hex}
 - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV64 programs (text segment) in low end
 - $-0001_{0000_{hex}}$
- static data segment (constants and other static variables) above text for static variables
 - RISC-V convention global pointer (gp) points to static
 - $RV64 gp = 1000_0000_{hex}$
- Heap above static for data structures that grow and shrink; grows up to high addresses



'| 系统调用的具体实现

- 内核中为每个系统调用定义了唯一的编号,就是系统调用号
- 内核中保存了一张系统调用表,该表中包含了系统调用号和其对应的服务例程地址。第n个表项包含系统调用号为n的服务例程地址。
- 系统调用陷入内核前,需要把系统调用号一起传入内核,而该号实际上是系统调用表的下标。
 - 在RISC-V模拟器中,这个传递动作可以通过在执行 scall 前把系统调用号装入寄存器a[7]实现,这样系统 调用处理程序一旦运行,就可以从a[7]中得到系统调用 号,然后再去系统调用表中中寻找相应服务例程
 - 通过什么方式传递系统调用号,由ABI决定

'| 系统调用的返回

- 以x86为例
- system_call()从eax获得系统调用的返回值,并把这个值存放在曾保存用户态eax寄存器栈单元的那个位置上,然后跳转到ret_from_sys_call(),终止系统调用处理程序的执行

'|| Acknowledgements

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252

- a0-a7 (x10-x17): eight *argument* registers to pass parameters and two return values (a0-a1)
- ra: one return address register to return to the point of origin (x1)
- As a special function call, system calls also use a0-a7 to transfer arguments.

不同指令集	系统调用指令	系统调用号 保存寄存器	系统调用参数保 存寄存器
X86(linux)	int \$0x80	eax	ebx, ecx, edx, esi, edi
RISCV(linux)	scall	a[7]	a[0]~a[3]

#define __NR_read 3 #define __NR_write 4

系统调用号举例(x86)

#define SYS_read 63
#define SYS write 64

系统调用号举例 (RISCV)

syscall_riscv.h

```
static inline long
 _internal_syscall(long n, long _a0, long _a1, long _a2, long _a3)
 register long a0 asm("a0") = a0;
 register long al asm("a1") = a1;
 register long a2 asm("a2") = a2;
 register long a3 asm("a3") = a3;
 register long a7 asm("a7") = n;
 asm volatile ("scall\n"
        "bltz a0, syscall error"
        : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3), "r"(a7));
 return a0;
```

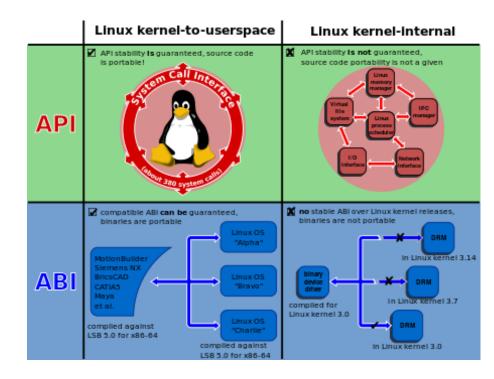
'| 系统调用 (system call)

• 功能概述

位于用户空间的应用程序与位于内核空间的操作系统内核程序之间的功能接口

作用

- 使用户程序与内核程序相分离,保护内核,提高系统的安全性
- 为用户提供有关设备管理、输入输出系统、文件系统和进程 控制、通信及存储管理等方面 的功能
- 对用户隐藏系统程序的内部结构
- 编程容易, 从硬件设备的编程中解脱出来



系统调用是用户态进入内核态的唯一入口

'| 一般的系统调用处理过程

