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

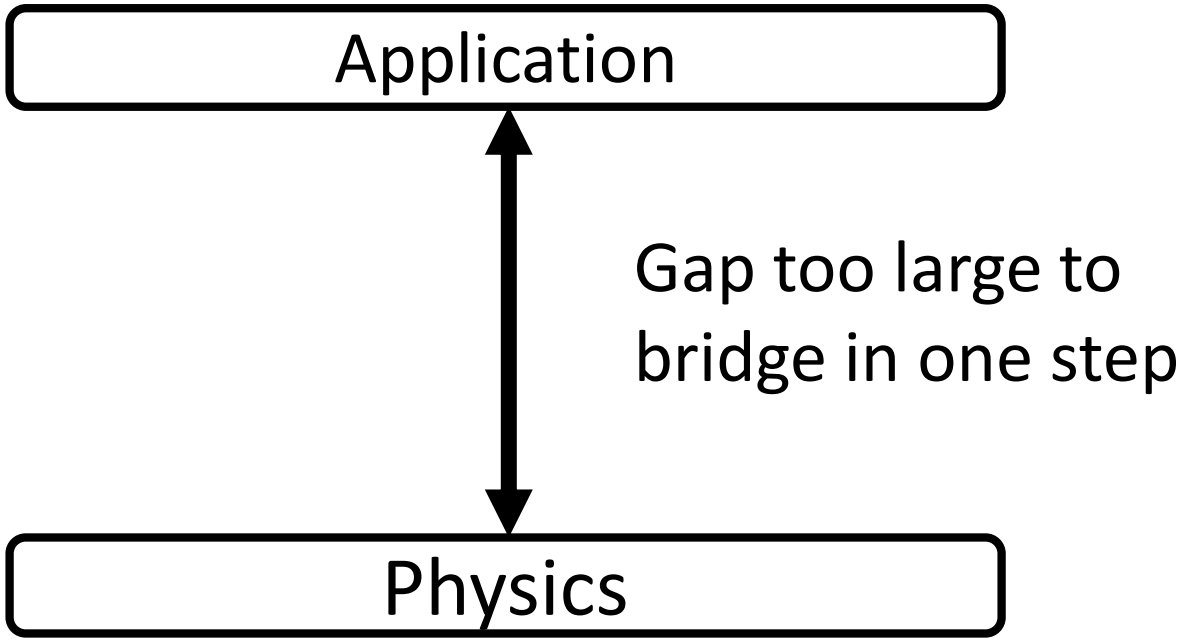
RISC-V ISA

北京大学微处理器研发中心

易江芳

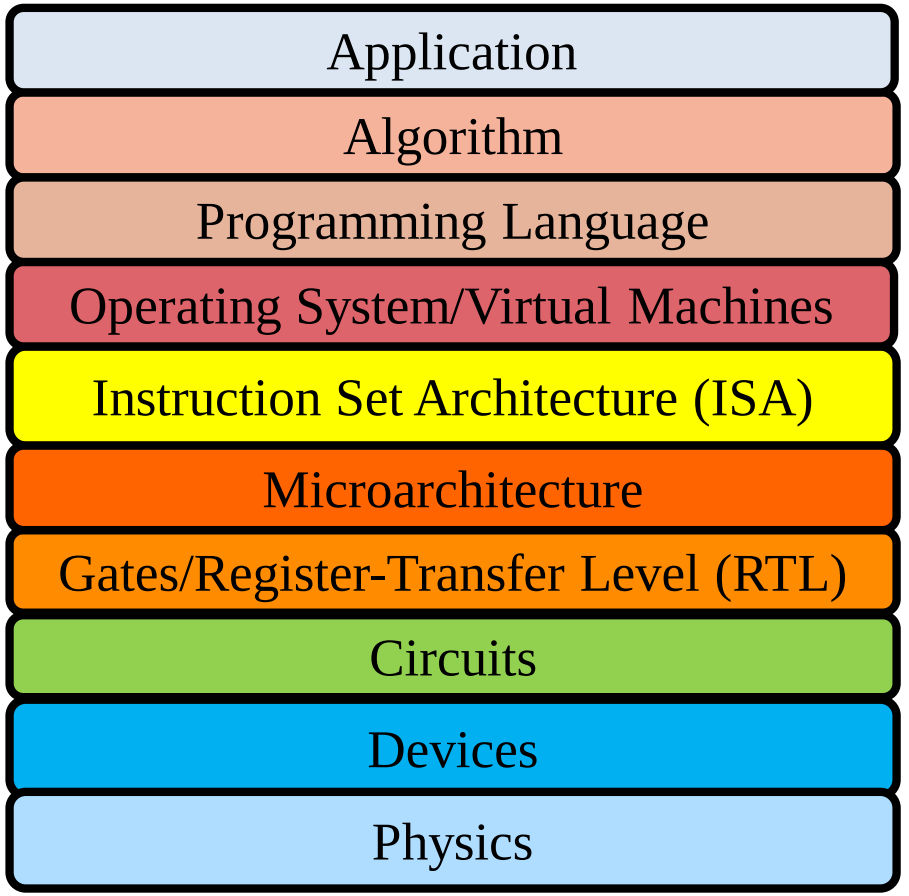
2022-10-10

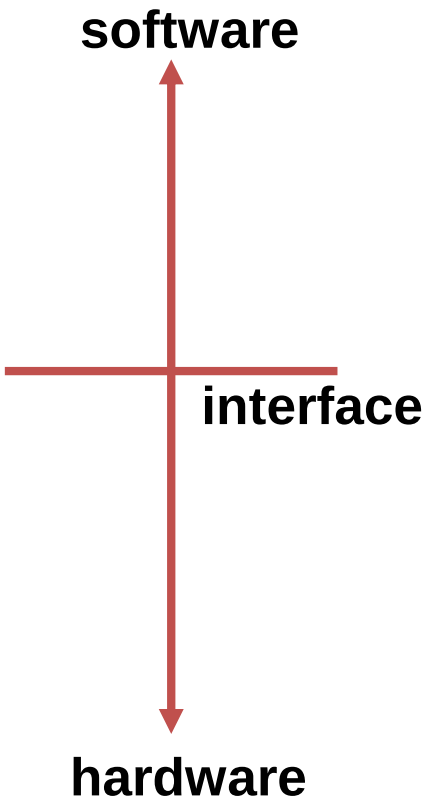
Computer Architecture



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

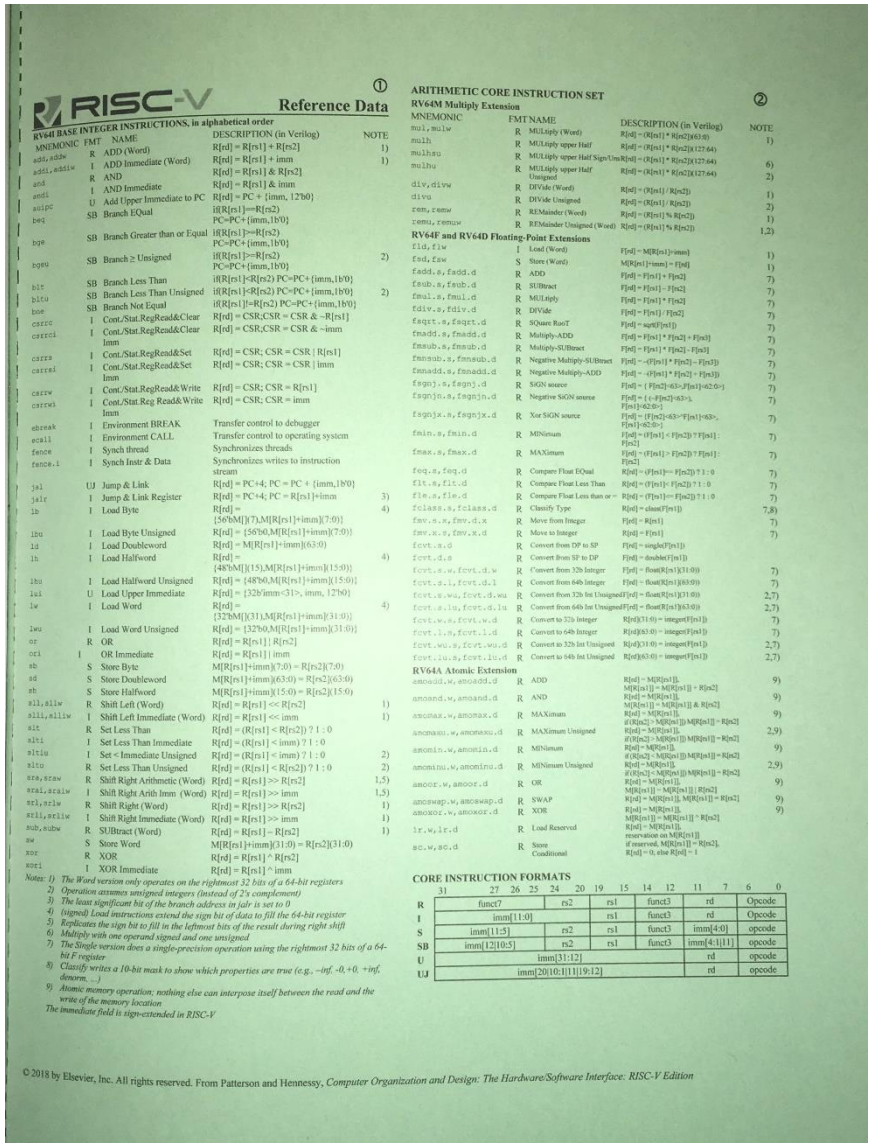
RISC ISA

- 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

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’ ll be using 64-bit RISC-V this semester in labs. Similar to MIPS you saw in Computer architecture.

RISC-V simple green card



RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add, addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi, addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2]) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2] PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2] PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2] PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll, slliw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli, slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra, sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai, sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl, srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli, srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub, subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) *The Word version only operates on the rightmost 32 bits of a 64-bit registers*
2) *Operation assumes unsigned integers (instead of 2's complement)*
3) *The least significant bit of the branch address in jalr is set to 0*
4) *(signed) Load instructions extend the sign bit of data to fill the 64-bit register*
5) *Replicates the sign bit to fill in the leftmost bits of the result during right shift*
6) *Multiply with one operand signed and one unsigned*
7) *The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register*
8) *Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)*
9) *Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location*

The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul, mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
div, divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem, remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu, remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
fld, flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd, fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s, fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s, fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s, fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s, fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s, fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s, fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s, fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmsub.s, fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = -(F[rs1] * F[rs2] + F[rs3])	7)
fnmadd.s, fnmadd.d	R	Negative Multiply-ADD	F[rd] = -(F[rs1] * F[rs2] - F[rs3])	7)
fsgnj.s, fsgnj.d	R	SiGN source	F[rd] = (F[rs2]<63>,F[rs1]<62:0>)	7)
fsgnjn.s, fsgnjn.d	R	Negative SiGN source	F[rd] = ({~F[rs2]<63>}, F[rs1]<62:0>)	7)
fsgnjx.s, fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s, fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s, fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s, feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s, flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s, fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
fclass.s, fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x, fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s, fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	F[rd] = double(F[rs1])	
fcvt.s.w, fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l, fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.w.s, fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s, fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s, fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s, fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
amoadd.w, amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w, amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w, amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if(R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w, amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if(R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w, amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w, amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w, amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w, amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w, amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w, lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w, sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

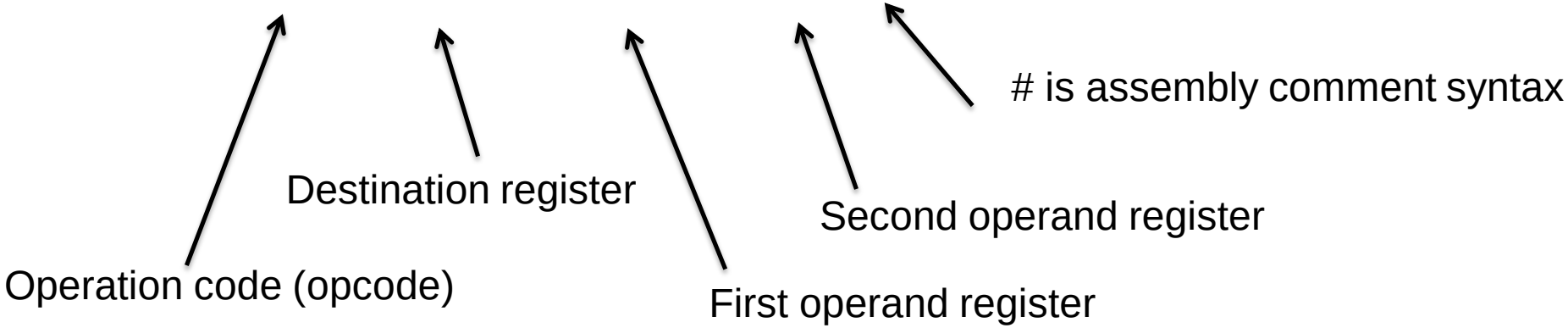
	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

RISC-V RV64I Simple Green Card

Instruction	Type	Opcode	Funct3	Funct7/IMM	Operation
add rd, rs1, rs2	R	0x33	0x0	0x00	R[rd] ← R[rs1] + R[rs2]
mul rd, rs1, rs2			0x0	0x01	R[rd] ← (R[rs1] * R[rs2])[31:0]
sub rd, rs1, rs2			0x0	0x20	R[rd] ← R[rs1] - R[rs2]
sll rd, rs1, rs2			0x1	0x00	R[rd] ← R[rs1] << R[rs2
mulh rd, rs1, rs2			0x1	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32]
slt rd, rs1, rs2			0x2	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0
xor rd, rs1, rs2			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)	I	0x03	0x0		R[rd] ← SignExt(Mem(R[rs1] + offset, byte))
lh rd, offset(rs1)			0x1		R[rd] ← SignExt(Mem(R[rs1] + offset, half))
lw rd, offset(rs1)			0x2		R[rd] ← Mem(R[rs1] + offset, word)
ld rd, offset(rs1)			ox3		R[rd] ← Mem(R[rs1] + offset, doubleword)
addi rd, rs1, imm		0x13	0x0		R[rd] ← R[rs1] + imm
slli rd, rs1, imm			0x1	0x00	R[rd] ← R[rs1] << imm
slti rd, rs1, imm			0x2		R[rd] ← (R[rs1] < imm) ? 1 : 0
xori rd, rs1, imm			0x4		R[rd] ← R[rs1] ^ imm
srli rd, rs1, imm			0x5	0x00	R[rd] ← R[rs1] >> imm
srai rd, rs1, imm			0x5	0x20	R[rd] ← R[rs1] >> imm
ori rd, rs1, imm			0x6		R[rd] ← R[rs1] | imm
andi rd, rs1, imm			0x7		R[rd] ← R[rs1] & imm
addiw rd, rs1, imm		0x1B	0x0		R[rd] ← SignExt(R[rs1](31:0) + imm)
Jalr rd, rs1, imm		0x67	0x0		R[rd] ← PC + 4
					PC ← R[rs1] + {imm, 1b'0}
ecall		0x73	0x0	0x000	(Transfers control to operating system)
					a0 = 1 is print value of a1 as an integer.
					a0 = 10 is exit or end of code indicator.
sb rs2, offset(rs1)	S	0x23	0x0		Mem(R[rs1] + offset) ← R[rs2][7:0]
sh rs2, offset(rs1)			0x1		Mem(R[rs1] + offset) ← R[rs2][15:0]
sw rs2, offset(rs1)			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	SB	0x63	0x0		if(R[rs1] == R[rs2])
					PC ← PC + {offset, 1b'0}
bne rs1, rs2, offset			0x1		if(R[rs1] != R[rs2])
					PC ← PC + {offset, 1b'0}
blt rs1, rs2, offset			0x4		if(R[rs1] < R[rs2])
					PC ← PC + {offset, 1b'0}
bge rs1, rs2, offset			0x5		if(R[rs1] >= R[rs2])
					PC ← PC + {offset, 1b'0}
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
					PC ← PC + {imm, 1b'0}

RISC-V Instruction Assembly Syntax

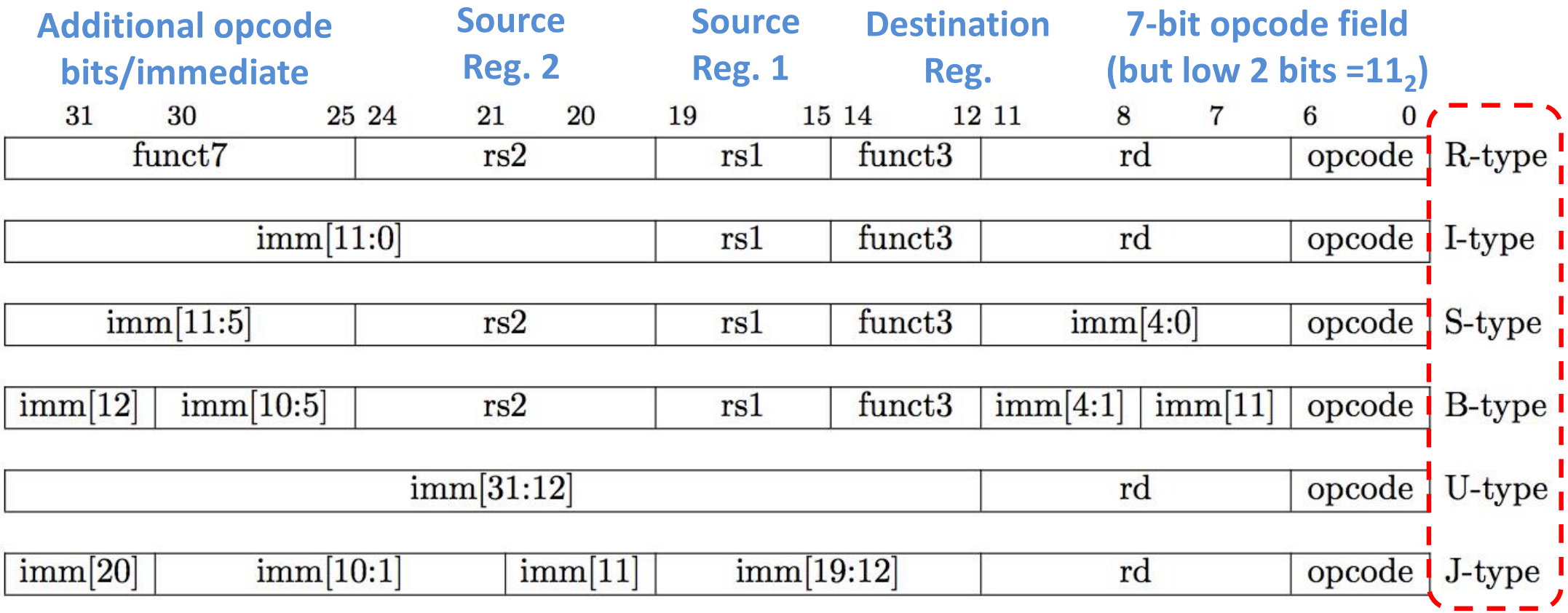
- Instructions have an opcode and operands
- E.g., **`add x1, x2, x3 # x1 = x2 + x3`**



- Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: `a = b + c` (in C)

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

Summary of RISC-V Instruction Formats

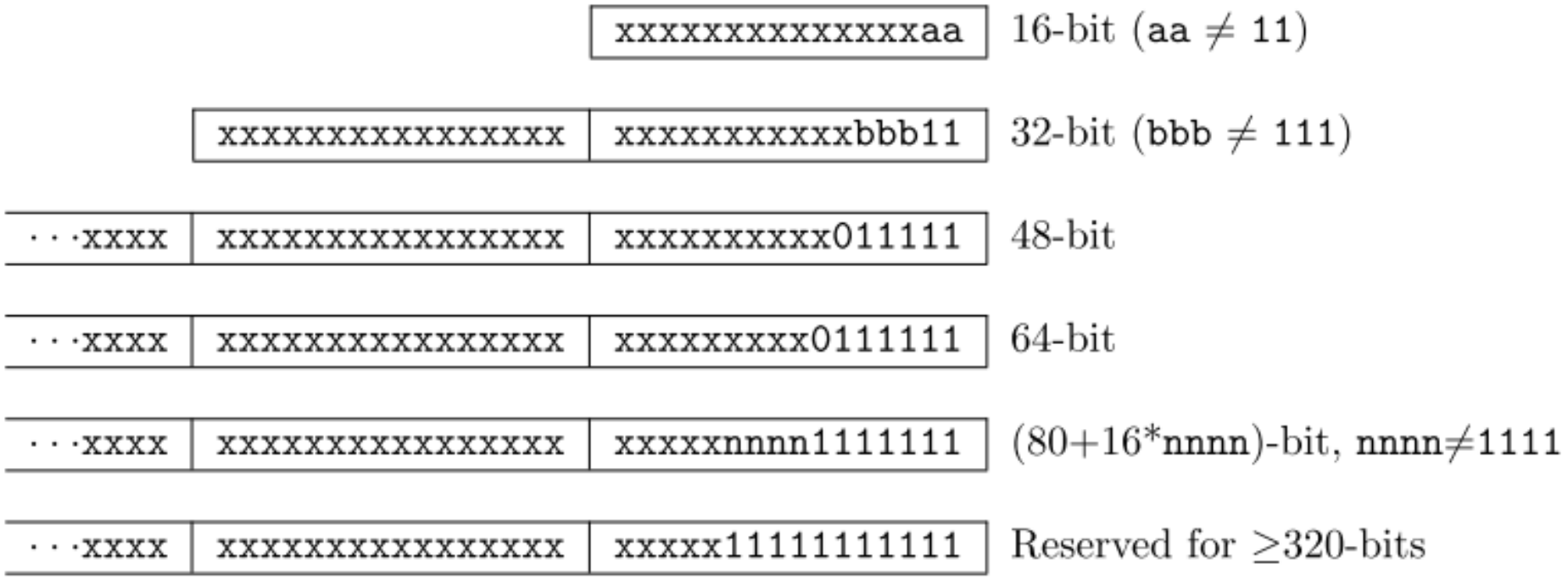


- Aligned on a four-byte boundary in memory
- Sign bit of immediates always on bit 31 of instruction.
- Register fields never move
- Opcode fields is on the right

Where is NOP?

- Addi x0, x0, #0

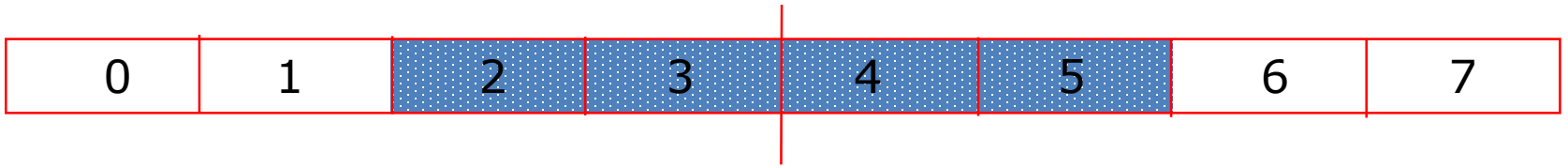
RISC-V Instruction Encoding



- Base instruction set (RV32 and RV64) always has fixed 32-bit instructions lowest two bits = 11_2
- 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

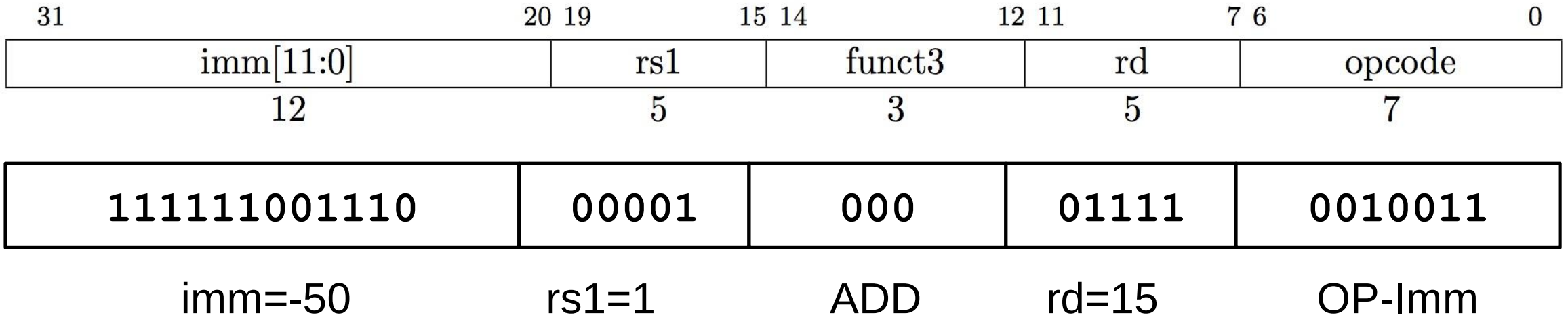
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110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

Different encoding in funct7 + funct3 selects different operations

RISC-V I-Format Instruction

- RISC-V Assembly Instruction:

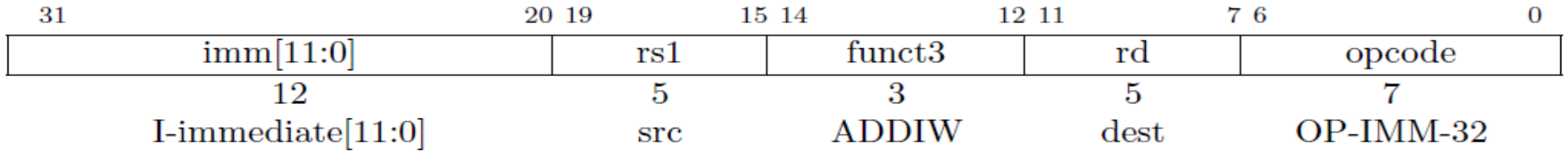
```
addi  x15,x1,-50
```



- imm[11:0] can hold values in range [-2048_{10} , $+2047_{10}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation

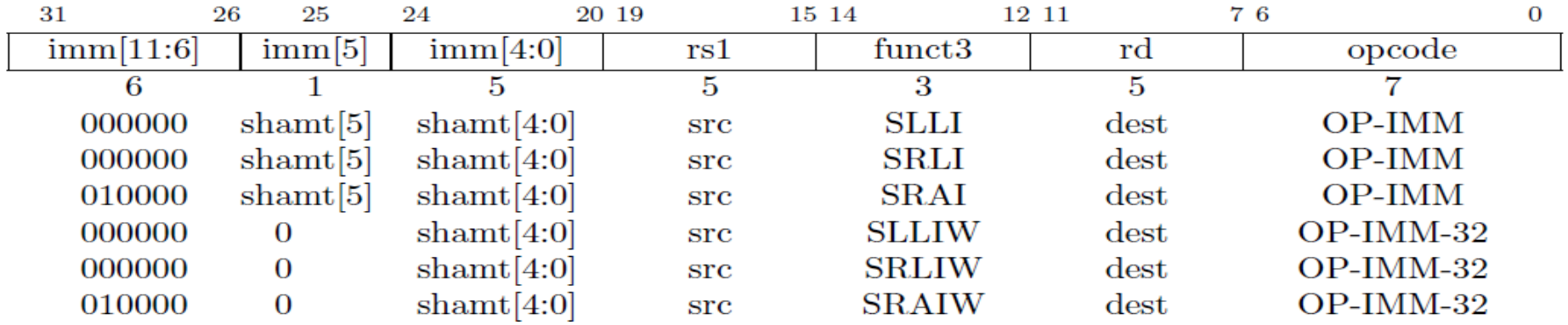
Support 32-bit ops in rv64I

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



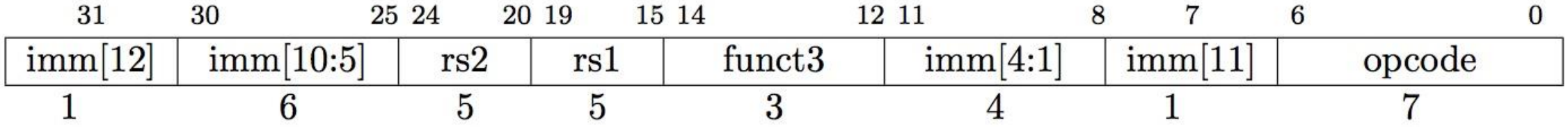
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

  - Shift instruction

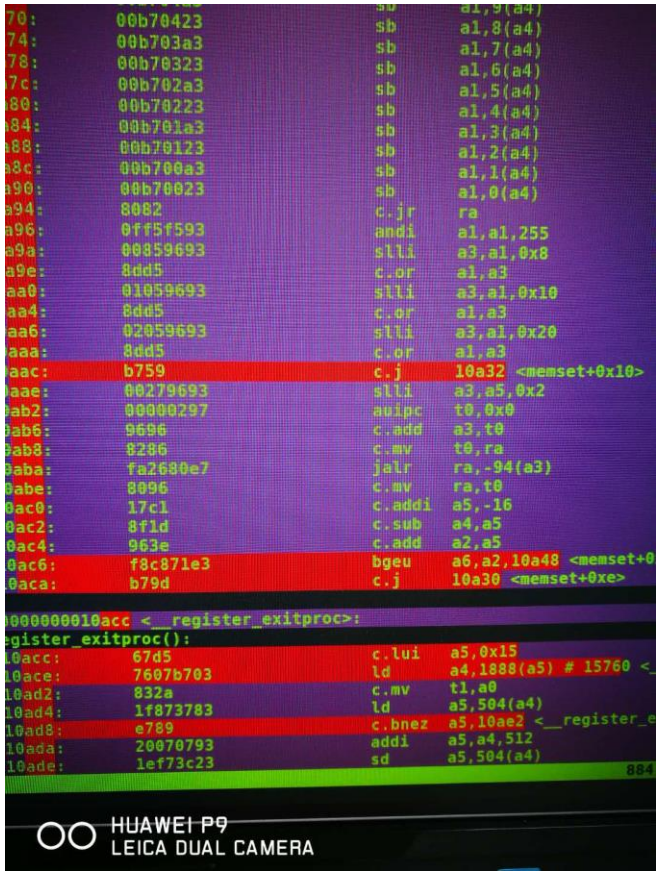


31 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

RISC-V B-Format for Branches



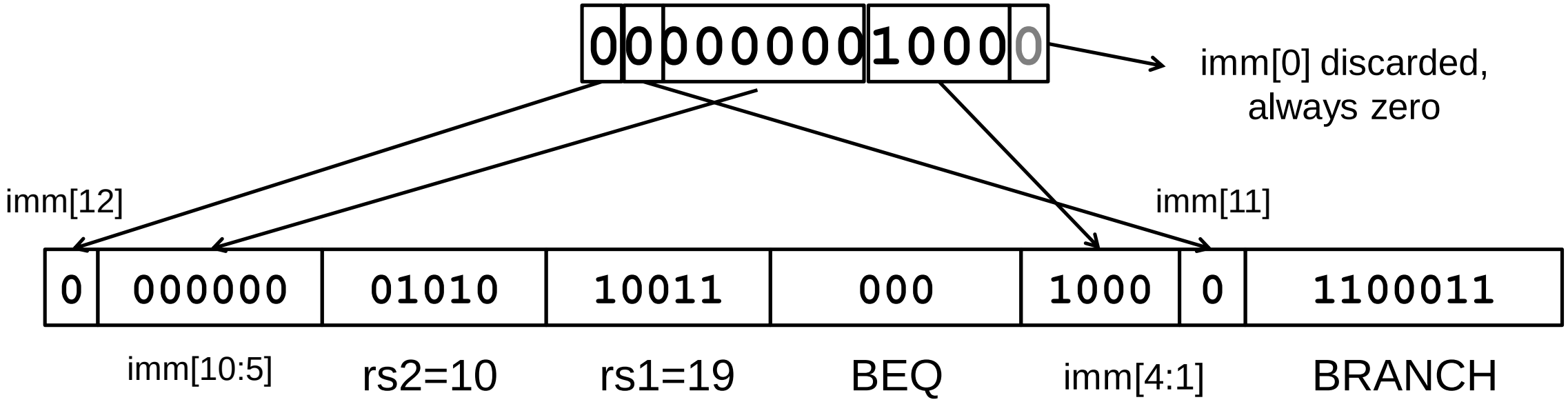
- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)



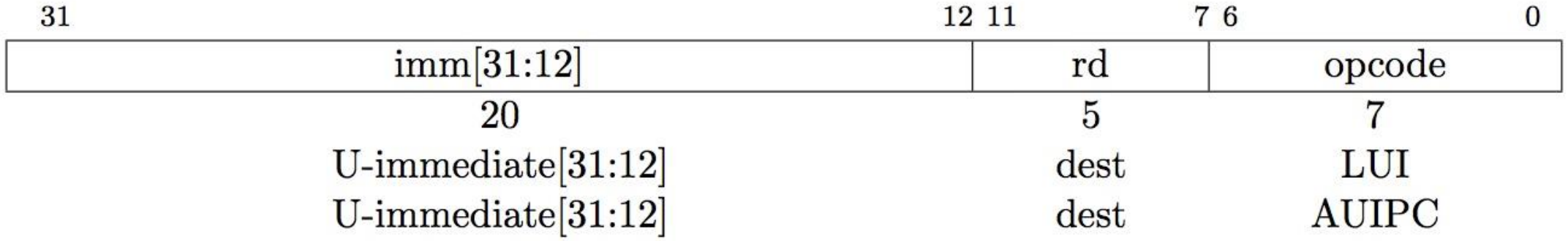
Branch Example

```
beq   x19,x10, offset = 16 bytes
```

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



U-Format for "Upper Immediate" instructions



- 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

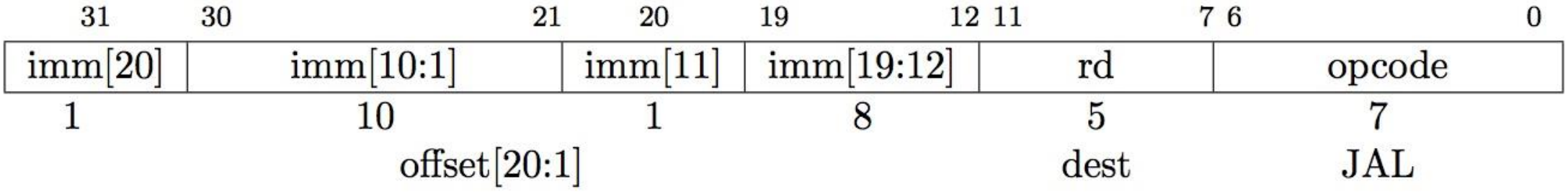
LUI to create long immediates

- 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0xDEADBEEF，如何？

RISC-V J-Format for Jump Instructions

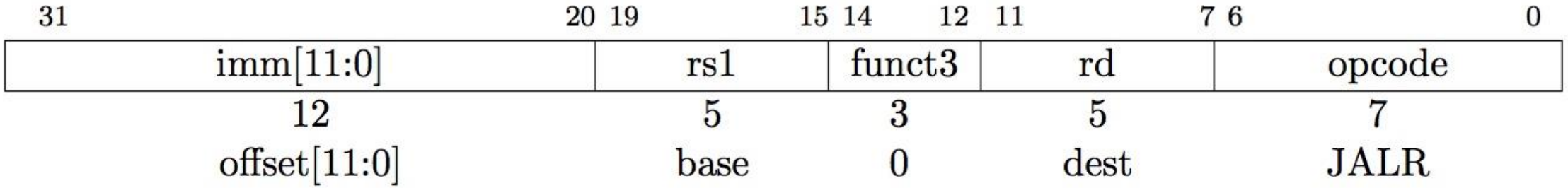


- JAL saves PC+4 in register rd (the return address)
  - Assembler "`j`" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Example:

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 2^18 instructions of PC
jal ra, FuncName
```

JALR Instruction (I-Format)



- 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>
```

RISC-V Pseudo-instructions

PSEUDO INSTRUCTIONS ③

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] = ~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

Instruction	Type	Opcode	Funct3	Funct7/IMM	Operation
add rd, rs1, rs2	R	0x33	0x0	0x00	R[rd] ← R[rs1] + R[rs2]
mul rd, rs1, rs2			0x0	0x01	R[rd] ← (R[rs1] * R[rs2])[31:0]
sub rd, rs1, rs2			0x0	0x20	R[rd] ← R[rs1] - R[rs2]
sll rd, rs1, rs2			0x1	0x00	R[rd] ← R[rs1] << R[rs2
mulh rd, rs1, rs2			0x1	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32]
slt rd, rs1, rs2			0x2	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0
xor rd, rs1, rs2			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)	I	0x03	0x0		R[rd] ← SignExt(Mem(R[rs1] + offset, byte))
lh rd, offset(rs1)			0x1		R[rd] ← SignExt(Mem(R[rs1] + offset, half))
lw rd, offset(rs1)			0x2		R[rd] ← Mem(R[rs1] + offset, word)
ld rd, offset(rs1)			ox3		R[rd] ← Mem(R[rs1] + offset, doubleword)
addi rd, rs1, imm		0x13	0x0		R[rd] ← R[rs1] + imm
slli rd, rs1, imm			0x1	0x00	R[rd] ← R[rs1] << imm
slti rd, rs1, imm			0x2		R[rd] ← (R[rs1] < imm) ? 1 : 0
xori rd, rs1, imm			0x4		R[rd] ← R[rs1] ^ imm
srli rd, rs1, imm			0x5	0x00	R[rd] ← R[rs1] >> imm
srai rd, rs1, imm			0x5	0x20	R[rd] ← R[rs1] >> imm
ori rd, rs1, imm			0x6		R[rd] ← R[rs1] | imm
andi rd, rs1, imm			0x7		R[rd] ← R[rs1] & imm
addiw rd, rs1, imm		0x1B	0x0		R[rd] ← SignExt(R[rs1](31:0) + imm)
Jalr rd, rs1, imm		0x67	0x0		R[rd] ← PC + 4
					PC ← R[rs1] + {imm, 1b'0}
ecall		0x73	0x0	0x000	(Transfers control to operating system)
					a0 = 1 is print value of a1 as an integer.
					a0 = 10 is exit or end of code indicator.
sb rs2, offset(rs1)	S	0x23	0x0		Mem(R[rs1] + offset) ← R[rs2][7:0]
sh rs2, offset(rs1)			0x1		Mem(R[rs1] + offset) ← R[rs2][15:0]
sw rs2, offset(rs1)			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	SB	0x63	0x0		if(R[rs1] == R[rs2])
					PC ← PC + {offset, 1b'0}
bne rs1, rs2, offset			0x1		if(R[rs1] != R[rs2])
					PC ← PC + {offset, 1b'0}
blt rs1, rs2, offset			0x4		if(R[rs1] < R[rs2])
					PC ← PC + {offset, 1b'0}
bge rs1, rs2, offset			0x5		if(R[rs1] >= R[rs2])
					PC ← PC + {offset, 1b'0}
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
					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/

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

RISC-V calling convention register usage

Register	ABI Name	Description	Saver
x0	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:

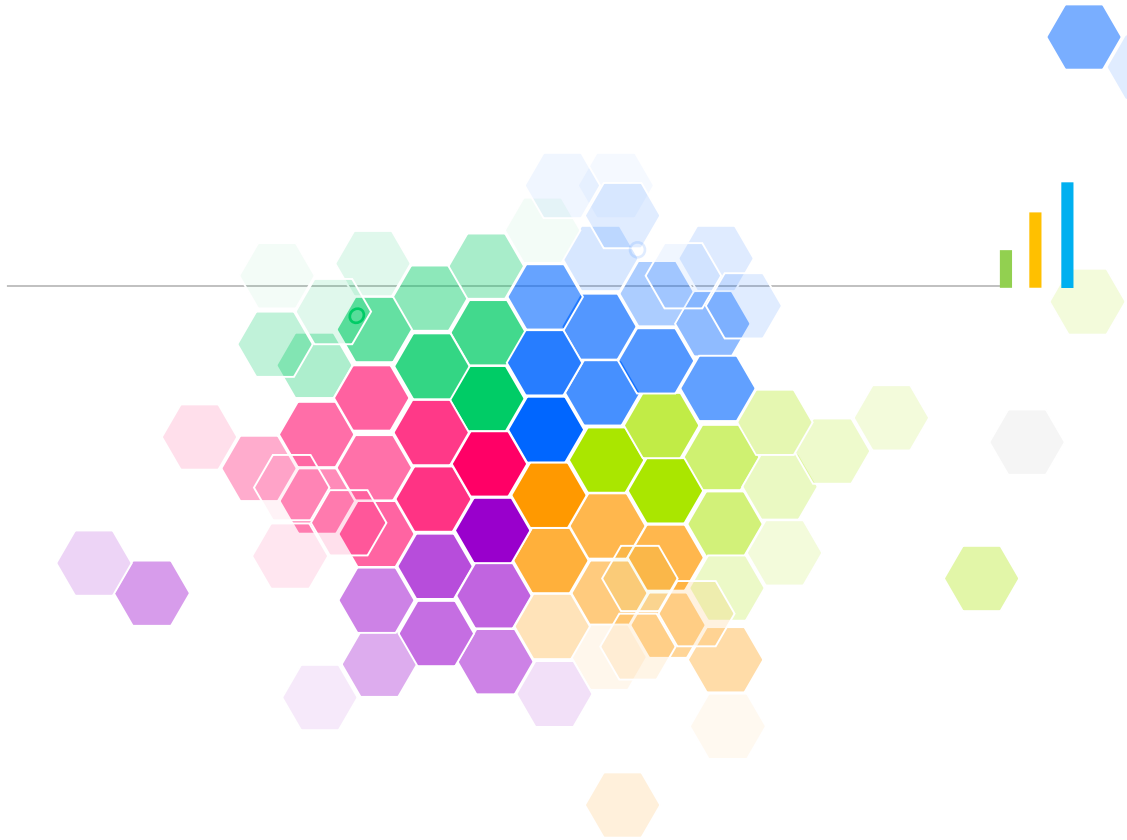
1. Preserved across function call
   - Caller can rely on values being unchanged
   - **`sp`**, **`gp`**, **`tp`**, "saved registers" **`s0`**- **`s11`** (**`s0`** is also **`fp`**)
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Argument/return registers **`a0-a7`**,**`ra`**, "temporary registers" **`t0`**-**`t6`**

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

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



欢迎提问

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

RISC-V Function Call Conventions

- **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

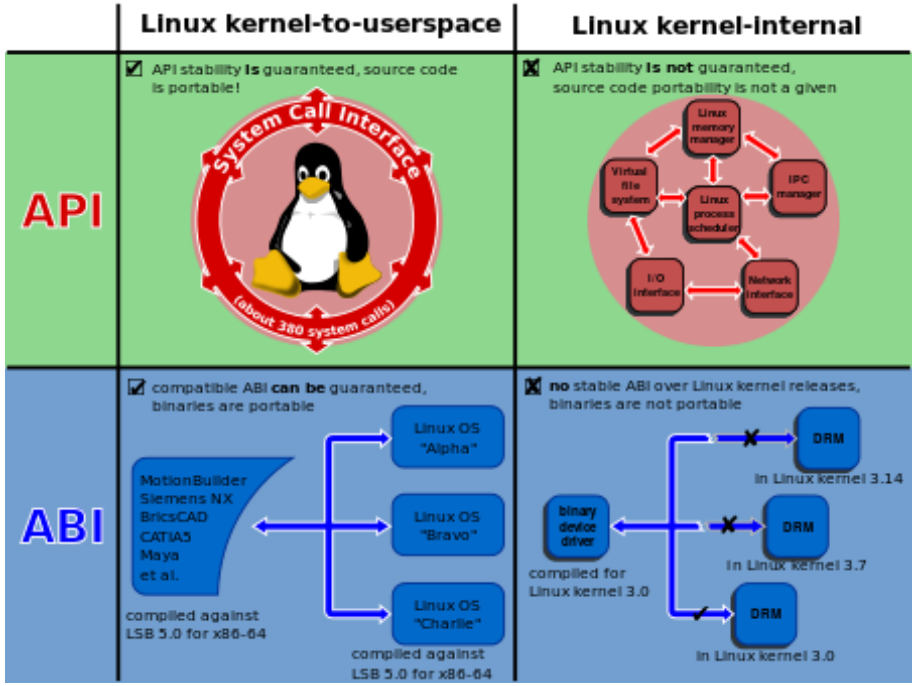
```
static inline long
__internal_syscall(long n, long _a0, long _a1, long _a2, long _a3)
{
  register long a0 asm("a0") = _a0;
  register long a1 asm("a1") = _a1;
  register long a2 asm("a2") = _a2;
  register long a3 asm("a3") = _a3;
  register long a7 asm("a7") = n;

  asm volatile ("scall\n"
        "bltz a0, __syscall_error"
        : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3), "r"(a7));

  return a0;
}
```

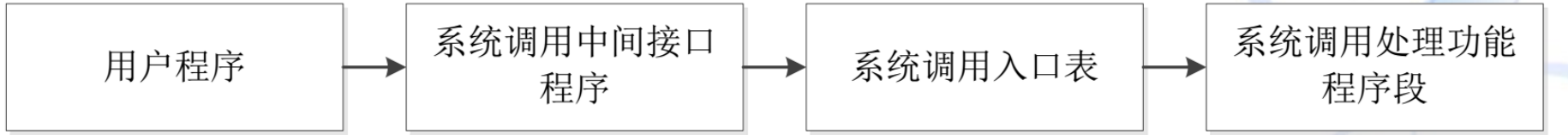
系统调用（system call）

- 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

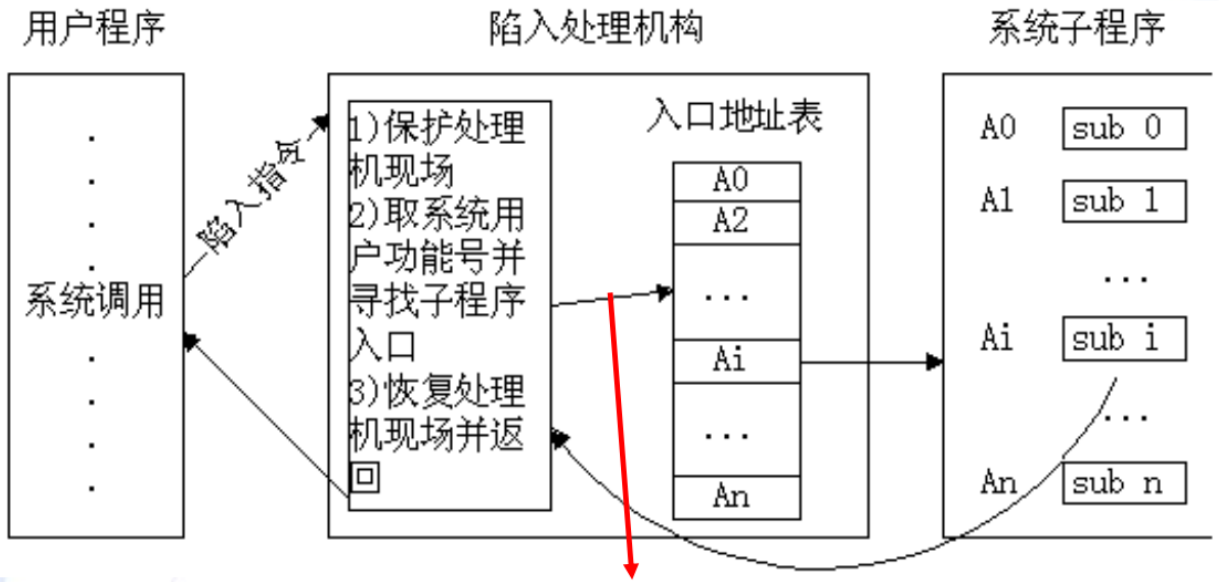


系统调用是用户态进入内核态的唯一入口

一般的系统调用处理过程



系统调用处理过程[1]



系统调用号*4+入口地址表首地址