RISC-V 指令系统

2022年秋

主要内容

- RISC-V指令系统概述
- RISC-V指令集与汇编语言概述
  - 算术指令、逻辑指令、移位指令
  - 数据传输指令（访存指令）
  - 比较指令、有条件跳转指令、无条件跳转指令
  - 伪指令
  - 函数调用

RISC-V指令系统概述

RISC-V指令集历史

- 加州大学伯克利分校Krste Asanovic教授、Andrew Waterman和Yunsup Lee等开发人员于2010年提出。
  - 其中"RISC"表示精简指令集，而其中"V"表示伯克利分校从RISC I开始设计的第五代指令集。
- 基于BSD协议许可的免费开放的指令集架构
- 适合多层次计算机系统
  - 从 微控制器 到 超级计算机
  - 支持大量定制与加速功能
  - 32bit, 64bit, 128bit
- 规范由RISC-V非营利性基金会维护
  - RISC-V基金会负责维护RISC-V指令集标准手册与架构文档

RISC-V架构的特点

- 指令集架构简单
  - 指令集的文档约200页，特权级编程手册也不到150页
  - 作为对比，Intel的处理器手册有5000多页
  - 新的体系结构设计吸取了经验和最新的研究成果
  - 指令数量少，基本的RISC-V指令数目仅有40多条，加上其他的模块化扩展指令总共几十条指令。
- 模块化的指令集设计
  - 不同的部分还能以模块化的方式组织在一起
  - ARM的架构分为A、R和M三个系列，分别针对于Application（应用操作系统）、Real-Time（实时）和Embedded（嵌入式）三个领域，彼此之间并不兼容
  - RISC-V嵌入式场景，用户可以选择RV32IC组合的指令集，仅使用Machine Mode（机器模式）；而高性能操作系统场景则可以选择譬如RV32IMFDC的指令集，使用Machine Mode（机器模式）与User Mode（用户模式）两种模式，两种使用方式的共同部分相互兼容

RISC-V的模块化设计

- RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示
- RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集，使用该整数指令子集，便能够实现完整的软件编译器
- 其他的指令子集部分均为可选的模块，具有代表性的模块包括M/A/F/D/C

RISC-V模块化设计（基本指令集）

指令集名称	描述	版本	状态
基本指令集			
RV32I	基本整数指令集，32位	2.0	冻结
RV32E	基本整数指令集(嵌入式系统)，32位，16 寄存器	1.9	开放
RV64I	基本整数指令集，64位	2.0	冻结
RV128I	基本整数指令集，128位	1.7	开放

RISC-V模块化设计（扩展指令集）

标准扩展指令集			
M	整数乘除法标准扩展	2.0	冻结
A	不可中断指令(Atomic)标准扩展	2.0	冻结
F	单精确度浮点运算标准扩展	2.0	冻结
D	双倍精确度浮点运算标准扩展	2.0	冻结
G	所有以上的扩展指令集以及基本指令集的总和的简称	不适用	不适用
Q	四倍精确度浮点运算标准扩展	2.0	冻结
L	十进制浮点运算标准扩展	0.0	开放
C	压缩指令标准扩展	2.0	冻结
B	位操作标准扩展	0.36	开放
J	动态指令翻译标准扩展	0.0	开放
T	顺序存储器访问标准扩展	0.0	开放
P	单指令多资料流（SIMD）运算标准扩展	0.1	开放
V	向量运算标准扩展	0.2	开放
N	用户中断标准扩展	1.1	开放

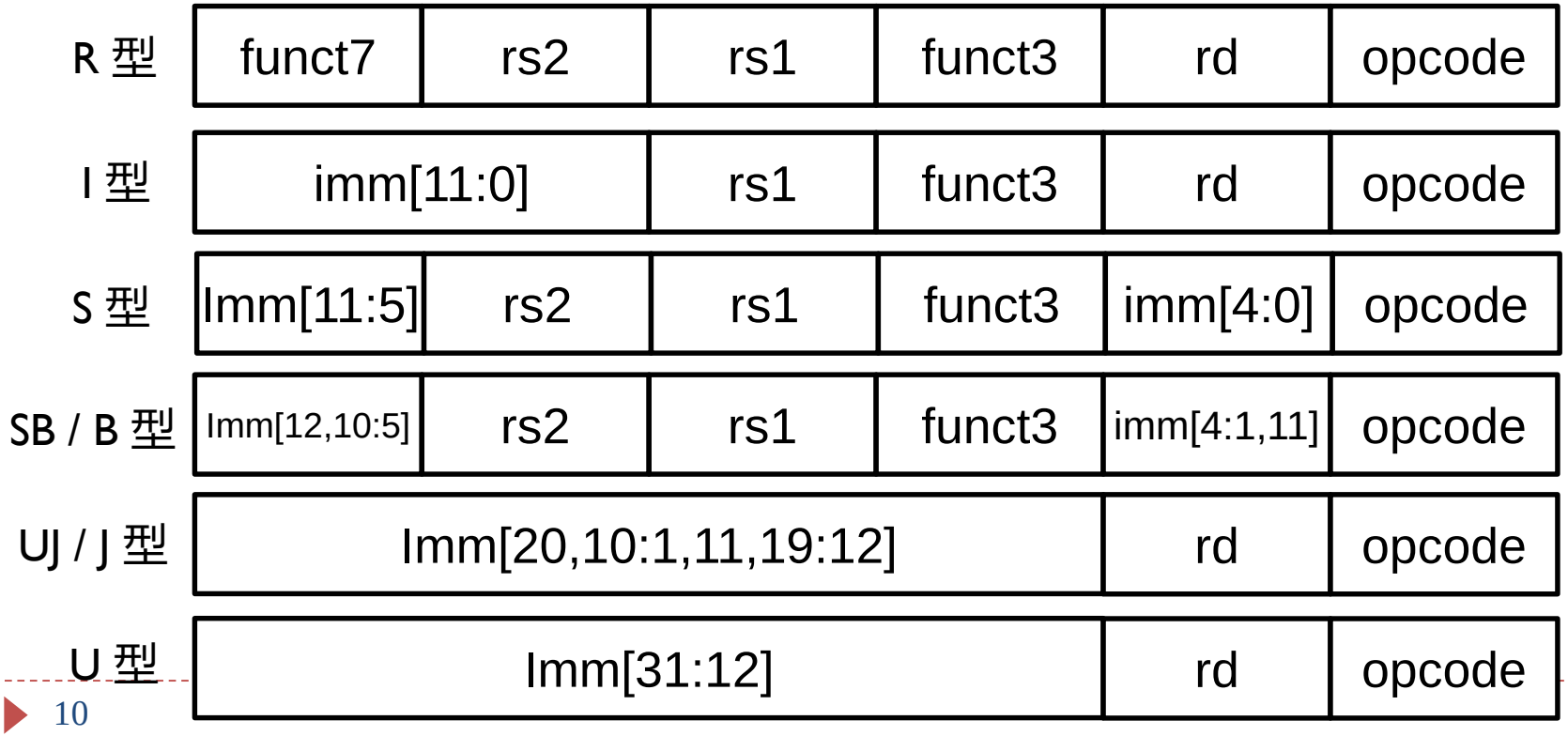
用户可以扩展自己的指令子集，RISC-V预留了大量的指令编码空间用于用户的自定义扩展，同时，还定义了四条Custom指令可供用户直接使用，每条Custom指令都有几个比特位的子编码空间预留，因此，用户可以直接使用四条Custom指令扩展出几十条自定义的指令。

可配置的通用寄存器组

- 寄存器组主要包括通用寄存器（General Purpose Registers）和控制状态寄存器（Control and Status Registers）
  - 32位架构(RV32I)32个32位的通用寄存器，64位架构(RV64I)32个64位的通用寄存器
  - 嵌入式架构RV32E有16个32位的通用寄存器
  - 支持单精度浮点数（F），或者双精度浮点数（D），另外增加一组独立的通用浮点寄存器组，f0~f31
- CSR寄存器用于配置或记录一些运行的状态（后续异常和中断处理中会详细描述）
  - CSR寄存器是处理器核内部的寄存器，使用专有的12位地址码空间

规整的指令编码

- 所有通用寄存器在指令码的位置是一样的，方便译码阶段的使用
- 所有的指令都是32位字长，有 6 种指令格式：寄存器型，立即数型，存储型，分支指令、跳转指令和大立即数



RISC-V的数据传输指令

- 专用内存到寄存器之间传输数据的指令，其它指令都只能操作寄存器
  - 简化硬件设计
  - 支持字节（8位），半字（16位），字（32位），双字（64位，64位架构）的数据传输
  - 推荐但不强制地址对齐
  - 小端机结构

RISC-V的特权模式

- RISC-V架构定义了三种工作模式，又称特权模式（Privileged Mode）：
  - Machine Mode：机器模式，简称M Mode
  - Supervisor Mode：监督模式，简称S Mode
  - User Mode：用户模式，简称U Mode
- RISC-V架构定义M Mode为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统
- 在异常和中断处理中会详细讨论各个特权模式的机制

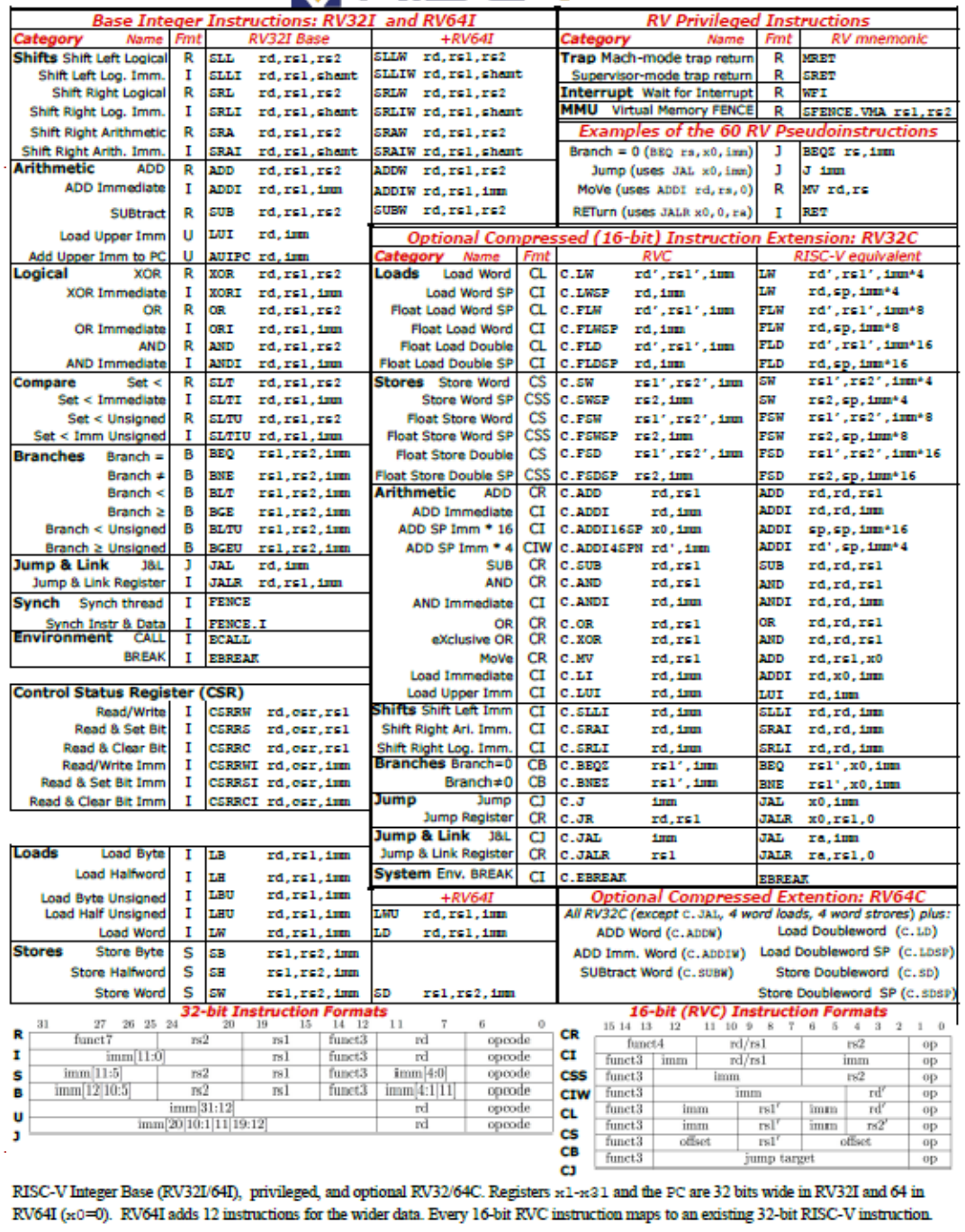
RISC-V 的指令集

- RISC-V官方指令集手册

https://riscv.org/specifications/isa-spec-pdf/

- 中文简化版

http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf



Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	B	BEQ rs1,rs2,imm	
	Branch ≠	B	BNE rs1,rs2,imm	
	Branch <	B	BLT rs1,rs2,imm	
	Branch ≥	B	BGE rs1,rs2,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm	
Jump & Link	J&L	J	JAL rd,imm	
	Jump & Link Register	I	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

Control Status Register (CSR)

Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Float Load Word SP	CL	C.FLW rd',rs1',imm	FLW rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8
	Float Load Double	CL	C.FLD rd',rs1',imm	FLD rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Float Store Word	CS	C.FSW rs1',rs2',imm	FSW rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8
	Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
	AND	CR	C.AND rd,rs1	AND rd,rd,rs1
	AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm
	OR	CR	C.OR rd,rs1	OR rd,rd,rs1
	eXclusive OR	CR	C.XOR rd,rs1	AND rd,rd,rs1
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,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

+RV64I			
LWU	rd,rs1,imm		
LD	rd,rs1,imm		
SD	rs1,rs2,imm		

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:

- ADD Word (C.ADDW)
- ADD Imm. Word (C.ADDIW)
- SUBtract Word (C.SUBW)
- Load Doubleword (C.LD)
- Load Doubleword SP (C.LDSP)
- Store Doubleword (C.SD)
- Store Doubleword SP (C.SDSP)

32-bit 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
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

16-bit (RVC) Instruction Formats

	15 14 13 12	11 10	9 8 7	6 5 4 3 2	1 0
CR	funct4	rd/rs1		rs2	op
CI	funct3 | imm	rd/rs1		imm	op
CSS	funct3	imm		rs2	op
CIW	funct3	imm		rd'	op
CL	funct3	imm	rs1' | imm	rd'	op
CS	funct3	imm	rs1' | imm	rs2'	op
CB	funct3	offset	rs1'	offset	op
CJ	funct3	jump target			op

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 in RV64I (x0=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

RISC-V指令集与汇编语言概述

算术指令、逻辑指令、移位指令

RISC-V 汇编

- 汇编指令格式

```
op dst, src1, src2
```

  - 1个操作码，3个操作数
  - op 操作的名字
  - dst 目标寄存器
  - src1 第一个源操作数寄存器
  - src2 第二个源操作数寄存器
- 通过一些限制来保持硬件简单

RISC-V 汇编格式

- 每一条指令只有一个操作，每一行最多一条指令
- 汇编指令与C语言的操作相关（=, +, -, *, /, &, |, 等）
  - C语言中的操作会被分解为一条或者多条汇编指令
  - C语言中的一行会被编译为多行RISC-V汇编程序

RISC-V 中的寄存器

- 在RISC-V中有32个寄存器（x0-x31）
  - 每个寄存器的长度为32位
  - X0是一个特殊的寄存器，只用于全零
  - 每一个寄存器都有自己的别名，用于软件的使用的惯例，但是实际的硬件并没有任何区别

RISC-V中的寄存器

寄存器	ABI名字	描述	保存者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	t0	Temporary/alternative link register	Caller
x6-7	t1-2	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

算术指令

- 变量a，b和c被分别放置在寄存器x1, x2,和x3中
- 整数的加法（add）
- C: a = b + c
- RISC-V: add x1, x2, x3
- 整数的减法（sub）
- C: a = b – c
- RISC-V: sub x1, x2, x3

RISC-V 程序举例

- 假设a→x1, b→x2, c→x3, d→x4, e→x5。下面会将一段C语言程序编译成RISC-V汇编指令

a = (b + c) – (d + e);

```
add x6, x4, x5          # tmp1 = d + e
add x7, x2, x3          # tmp2 = b + c
sub x1, x7, x6          # a = (b + c) – (d + e)
```

- 指令执行顺序反映了源程序的计算过程
- 指令中可以看到如何使用临时寄存器
- #符号后面是程序的注释

特殊的寄存器zero

- 0在程序中很常见，拥有一个自己的寄存器
- x0，或者zero是一个特殊的寄存器，只拥有值0，并且不能被改变
  - 注意，在任意的指令中，如果使用x0作为目标寄存器，将没有任何效果，仍然保持0不变
- 使用样例
  - `add  x3,  x0,  x0  # c=0`
  - `add  x1, x2,  x0  # a=b`
  - `add  x0,  x0,  x0  #nop`

RISC-V 中的立即数

- 数值常数被称为是立即数（immediates）
- 立即数有特殊的指令语法：

```
opi dst, src, imm
```

  - 操作码的最后一个字符为i的，会将第二个操作数认为是一个立即数（经常用后缀来指明操作数的类型，例如无符号数unsigned的后缀为u）
- 指令举例
  - `addi x1, x2, 5   # a=b+5`
  - `addi x3, x3, 1   # c++`
- 问题：为何没有subi指令？

算术操作的溢出

- 溢出是因为计算机中表达数本身是有范围限制的
  - 计算的结果没有足够多的位数进行表达
- RISC-V 忽略溢出问题，高位被截断，低位写入到目的寄存器中

RISC-V 乘法与除法指令

- 积的长度是乘数和被乘数长度的和。将两个32位数相乘得到的是64位的乘积。
- 为了正确地得到一个有符号或无符号的64位积，RISC-V中带有四个乘法指令。
  - 要得到整数32位乘积（64位中的低32位）就用mul指令。
  - 要得到高32位，如果操作数都是有符号数，就用mulh指令；

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

除法指令举例

```
# mod using div: x5 = x6 mod x7
mod:
div   x5, x6, x7        # x5 = x6/x7
rem   x5, x6, x7        # x5 = x6 mod x7
```

位操作指令

Note: a→x1, b→x2, c→x3

Instruction	C	RSC-V
And	`a = b & c;`	`and  x1,x2,x3`
And Immediate	`a = b & 0x1;`	`andi x1,x2,0x1`
Or	`a = b | c;`	`or  x1,x2,x3`
Or Immediate	`a = b | 0x5;`	`ori  x1,x2,0x5`
Exclusive Or	`a = b ^ c;`	`xor  x1,x2,x3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori x1,x2,0xF`

移位指令

- 左移相当于乘以2
  - 左移右边补0
  - 左移操作更快
- 逻辑右移：在最高位添加0
- 算术右移：在最高位添加符号位
- 移位的位数可以是立即数或者寄存器中的值

移位指令

Instruction Name	RISC-V
Shift Left Logical	`sll rd, rs1, rs2`
Shift Left Logical Imm.	`slli rd, rs1, shamt`
Shift Right Logical	`srl rd, rs1, rs2`
Shift Right Logical Imm.	`srli rd, rs1, shamt`
Shift Right Arithmetic	`sra rd, rs1, rs2`
Shift Right Arithmetic Imm.	`srai rd, rs1, shamt`

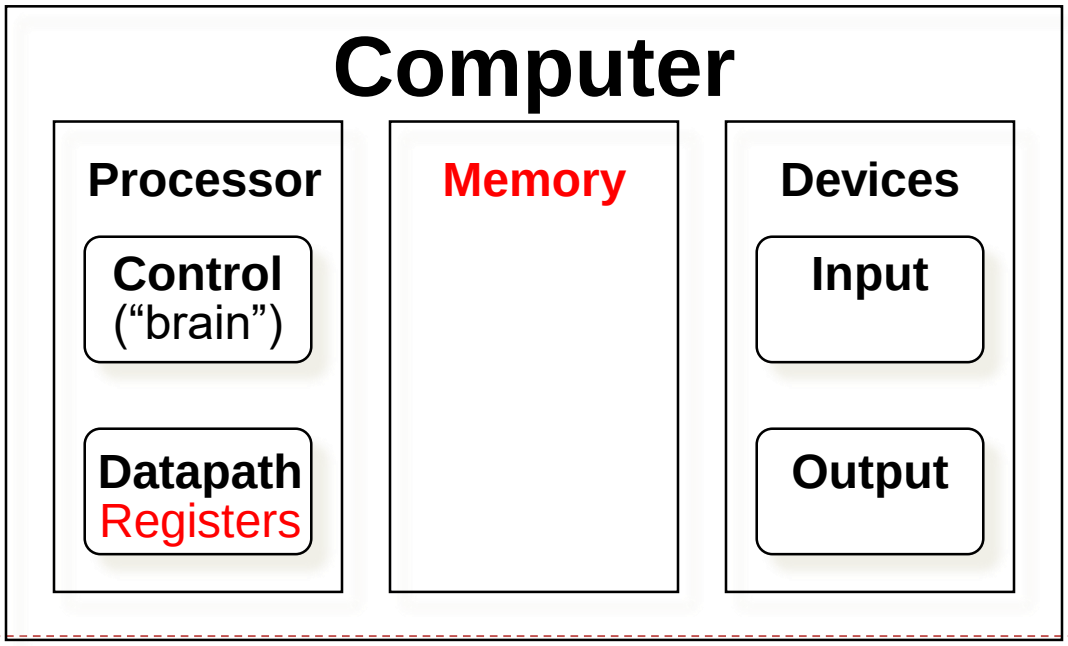
- slli, srli, srai只需要最多移动63位 (对64位寄存器)，只会使用immediate低6位的值 （I类型指令）

RISC-V指令集与汇编语言概述

数据传输指令（访存指令）

数据传输指令

- 数据传输指令在寄存器（数据通路）和内存之间传输数据
  - 从内存中取出操作数或者将操作数保存到内存中



数据传输

- C语言中的变量会映射到寄存器中；而其它的大量的数据结构，例如数组会映射到内存中
- 内存是一维的数组，地址从0开始
- 所有的RISC-V的指令操作（除load/store）只会在寄存器中操作
- 特殊的数据传输指令在寄存器和内存之间传输数据
  - Store指令：从寄存器到内存
  - Load指令：从内存到寄存器

数据传输指令的格式

- 数据传输指令的格式

```
memop reg, off(bAddr)
```

- memop = 操作的名字（load或者store）
- reg = 寄存器的名字，源寄存器或者目标寄存器
- bAddr = 指向内存的基地址寄存器 (base address)
- off = 地址偏移，字节寻址，为立即数 (offset)
- 访问的内存地址为 bAddr + off
- 必须指向一个合法的地址

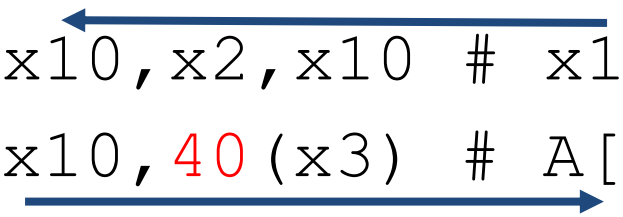
内存的字节寻址方式

- 在现代计算机中操作以8bits为单位，即一个字节
  - 一个word的定义依据不同的体系结构定义不同，这里定义1 word = 4 bytes
  - 内存是按照字节进行编址的，不是按照字进行编址的
- 字地址之间有4个字节的距离
  - 字的地址为其最低位的字节的地址
  - 按字对齐的话地址最后两位为0（地址为4的倍数）
- C语言会自动按照数据类型来计算地址，在汇编中需要程序员自己计算

数据传输指令

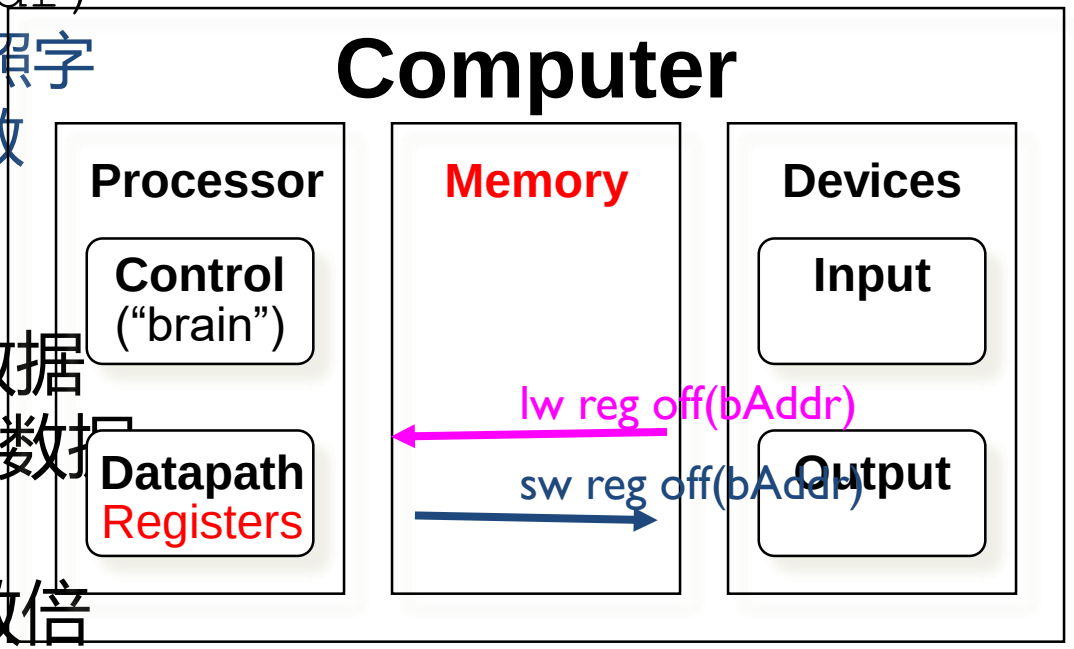
- 装入一个字(lw)
- 写出一个字(sw)
- 指令举例

```
# addr of int A[] -> x3, a -> x2
  lw   x10,12(x3)  # x10=A[3]
  add  x10,x2,x10  # x10=A[3]+a
  sw   x10,40(x3)  # A[10]=A[3]+a
```



内存与变量的大小（variable size）

- 数据传输指令
  - `lw reg, off(bAddr)`
  - `sw reg, off(bAddr)`
  - `off+bAddr` 必须按照字进行对齐，即4的倍数
  - 例如整数的数字每个整数32位=4字节
- 如何传输1个字节的数据或者传输一个short的数据（2个字节）都不是4个字节的整数倍



传输一个字节数据

- 还是使用字类型指令，配合位的掩码来达到目的

```
lw   x11,0(x1)
andi x11,x11,0xFF # lowest byte
```

- 或者，使用字节传输指令

```
lb   x11,1(x1)
sb   x11,0(x1)
```

- 上述指令无需字对齐

```
*(x0) = 0x00000180
```

00	00	01	80

字节排布

- 大端机：最高的字节在最低的地址，字的地址等于最高字节的地址
- 小端机：最低的字节在最低的地址，字的地址等于最低字节的地址

高地址 `*($s0) = 0x00000180` 低地址

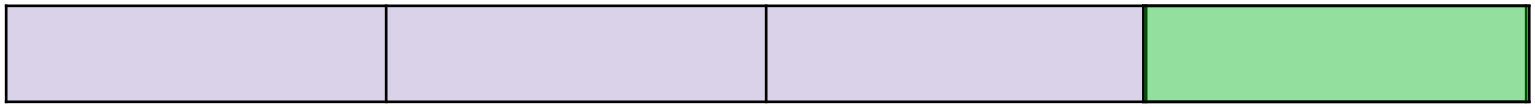
Big Endian

Little Endian

RISC-V 是 小端机

字节数据传输指令

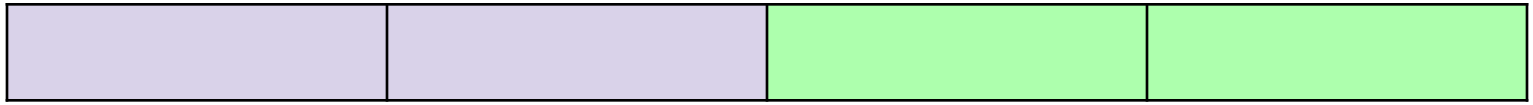
- lb/sb使用的是最低的字节
- 如果是sb指令，高24位被忽略
- 如果是lb指令，高24位做符号扩展



- 例如：let *(x1) = 0x00000180:

```
lb x11,1(x1)          # x11=0x00000001
lb x12,0(x1)          # x12=0xFFFFFF80
sb x12,2(x1)          # *(x1)=0x00800180
```

半字数据传输指令



- `lh reg, off(bAddr)` “load half”
- `sh reg, off(bAddr)` “store half”
  - `off(bAddr)`必须是 2 的倍数
  - `sh`指令中高`16`位忽略
  - `lh`指令中高`16`位做符号扩展

无符号的版本

- `lhu reg, off(bAddr)` “load half unsigned”
- `lbu reg, off(bAddr)` “load byte unsigned”
  - `l(b/h)u`指令, 高位都做0扩展

RISC-V指令集与汇编语言概述

分支与跳转指令

比较指令

- Set Less Than (`slt`)
  - `slt dst,reg1,reg2`
  - if value in `src1` < value in `src2`, `dst` = 1, else 0
- Set Less Than Immediate (`slti`)
  - `slti dst,reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0

- 如何完成无符号数的比较?

在 RISC-V 指令中有无符号数比较

- Unsigned versions of `slt(i)`:
  - `sltu  dst,src1,src2`: unsigned comparison
  - `sltiu dst,src,imm`: unsigned comparison against constant
- 例子:

```
addi  x10,x0,-1   # x10=0xFFFFFFFF
slti  x11,x10,1   # x11=1 (-1 < 1)
sltiu x12,x10,1   # x12=0 (2^32-1 >>> 1)
```

RISC-V 中的有符号与无符号

- 有符号和无符号在3个上下文环境中
  - Signed vs. unsigned bit extension 符号扩展
    - `lb, lh`
    - `lbu, lhu`
  - Signed vs. unsigned comparison 比较
    - **`slt, slti`**
    - **`sltu, sltiu`**
  - Signed vs. unsigned branch 比较
    - **`blt, bge`**
    - **`bltu, bgeu`**

条件跳转指令

- C语言中有控制流
  - 比较语句/逻辑语句确定下一步执行的语句块
- RISC-V 汇编无法定义语句块，但是可以通过标记（Label）的方式来定义语句块起始
  - 标记后面加一个冒号（main: ）
  - 汇编的控制流就是跳转到标记的位置
  - 在C语言中也有类似的结构，但是被认为是坏的编程风格（C语言有goto语句，跳转到标记所在的位置）

条件跳转指令

- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
  - 注意没有依据标志位的跳转（与`x86`不同）

无条件跳转指令 (jal, jalr)

- `jal` 将某一条指令的地址放到寄存器`ra`
- RISC-V: 指令是4字节长度
  - 内存是按照字节编址的

0x0040061C `jal newMoney`

0x00400620 (add 4)

RISC-V指令集与汇编语言概述

伪指令

汇编中的伪指令

- 伪指令可以给程序员更加直观的指令，但不是直接通过硬件来实现
- 通过汇编器来翻译为实际的硬件指令
- 例子：

  ```
  move dst,src
  ```

  并没有实际的数据移动指令，被翻译为下面的指令

  ```
  addi dst,src,0 or add dst,src,x0
  ```

其它的伪指令

- Load Immediate (`li`) 装入一个立即数
  - `li dst,imm`
  - 装入一个32位的立即数到 `dst`
  - 被翻译为: `addi dst x0 imm`
- Load Address (`la`) 装入一个地址
  - `la dst,label`
  - 装入由Label指定的地址到 `dst`
  - (思考一下如何翻译)
- 指令手册

Pseudo	Real
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs, rt, offset`	`blt rt, rs, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`
`call offset` (if too big for just a `jal`)	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`
`tail offset` (if too far for a `j`)	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`

伪指令 vs. 硬件指令

- 硬件指令
  - 所有指令都是硬件可以直接执行的指令，在硬件中直接实现了
- 伪指令
  - 汇编语言程序员可以使用的指令（加上了伪指令）
  - 每一条伪指令指令会被翻译为1条或者多条TAL指令
- 硬件指令 ⊂ 伪指令

RISC-V指令集与汇编语言概述

函数调用

函数调用

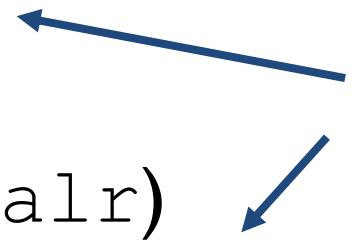
1. 将参数放置在函数可以访问到的地方
2. 将控制流转到函数中
3. 函数获取任何其所需要的存储资源
4. 执行函数体，完成功能
5. 函数放置返回值，清理函数调用信息
6. 控制流返回给函数调用者

对函数调用的支持

- 如果有可能，尽可能使用寄存器，寄存器要比内存快得多
- x10-x17：可以用来传递参数或返回值
- x1：返回地址寄存器，用于返回到起始点
- 传递参数的时候，顺序是有用的，代表了程序中的参数的顺序
- 如果寄存器空间不够，则需要借助于在内存中的栈

RISC-V 中的函数调用

- Jump and Link (`jal`)
  - `jal label`
- Jump and Link Register (`jalr`)
  - `jalr src`



用来调用一个函数

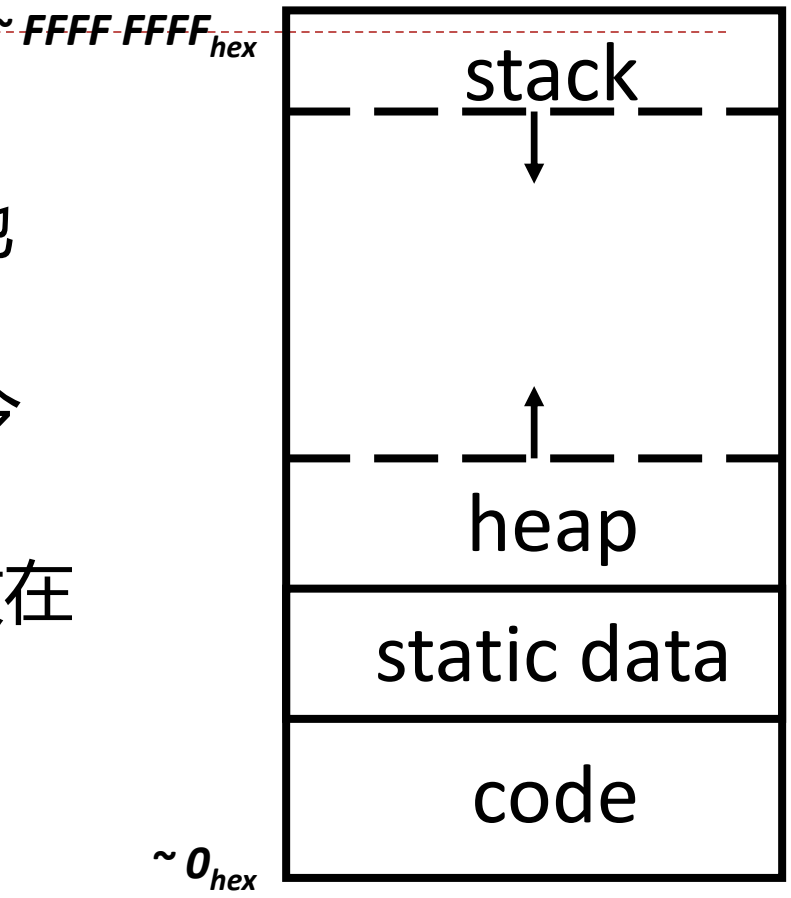
- "and Link": 在调到对应函数内部之前，将下一条指令的地址放置在寄存器`x1` 中
- `x1`: `ra` 返回地址寄存器

RISC-V 中的寄存器

- x10-x17：用以传递参数和返回值
- x1：ra返回地址寄存器
- x2：sp栈指针

指令地址

- 指令和数据都存放在同一个地址空间中
- 标记Label会被翻译为一个指令地址
- jal指令会把一条指令的地址放在寄存器`ra`



程序计数器

- 程序计数器（PC）指向的是当前正在执行的指令（上下文不同的环境下，有时候也会说明为指向下一条指令，PC经过更新后指向下一条将执行的指令）
- 值在指令执行第一个阶段就会被更新）
- PC值对于程序员是不可见的，但是可以被jal指令访问到
- 所有的分支指令(`beq`, `bne`, `jal`, `jalr`)，跳转指令都是通过更新PC来完成功能

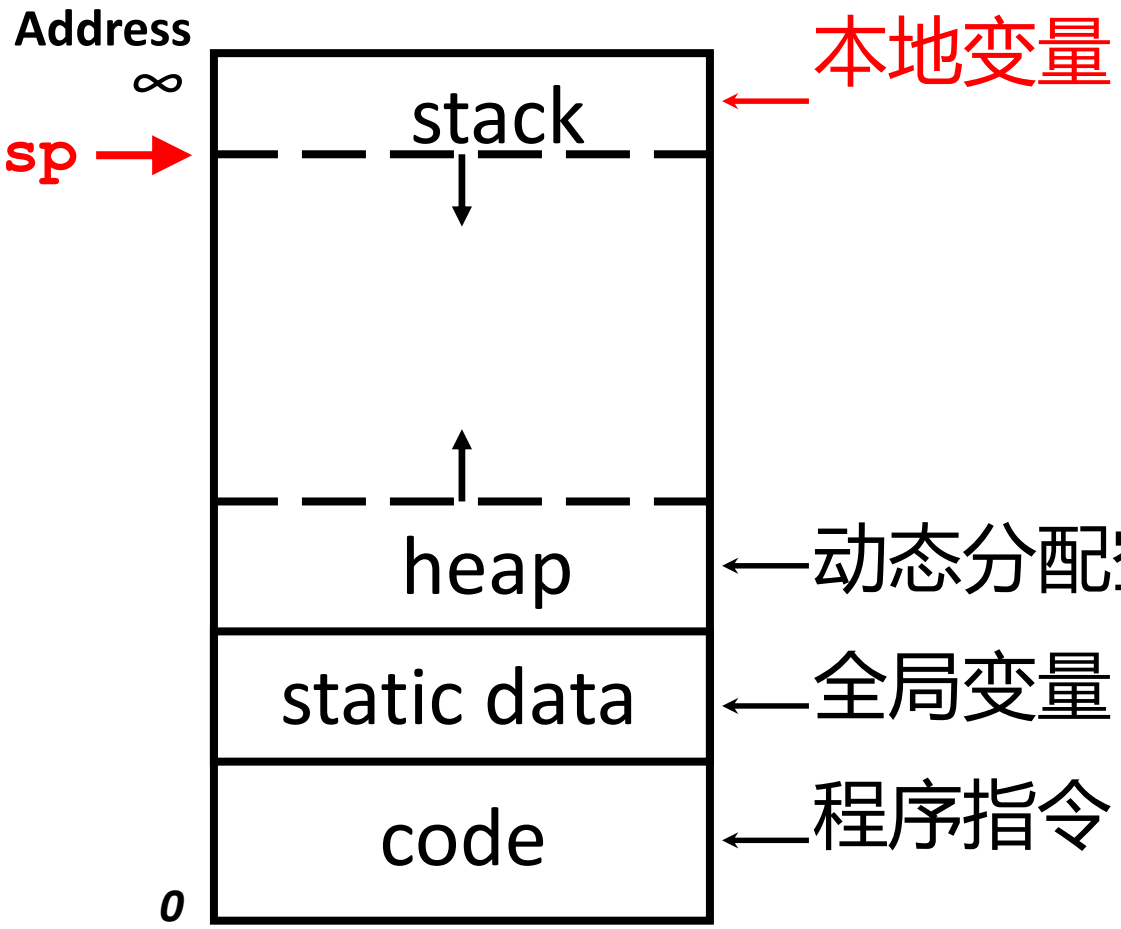
寄存器惯例

- **CalleR**：调用者函数

- **CalleE**：被调用函数

- 寄存器使用惯例：寄存器约定的方案，调用者保存的寄存器在函数调用前后可能会被改变；被调用者保存的寄存器在调用前后不会被改变（jal）

数据的内存排布情况


Address
∞
sp
stack
本地变量
heap
动态分配空间
static data
全局变量，字符串常量
code
程序指令
0

被调用者保存寄存器（Callee Saved Registers）

- `s0-s11`: x8-x9 + x18-x27 (callee *saved* registers)
- `sp` (stack pointer)
  - 必须要指向相同的位置，否则调用者就找不到当前的栈帧
- 如果寄存器不够用，则可以将原值保存在栈上，待函数返回的时候恢复

调用者保存寄存器（caller saved registers, Volatile Registers）

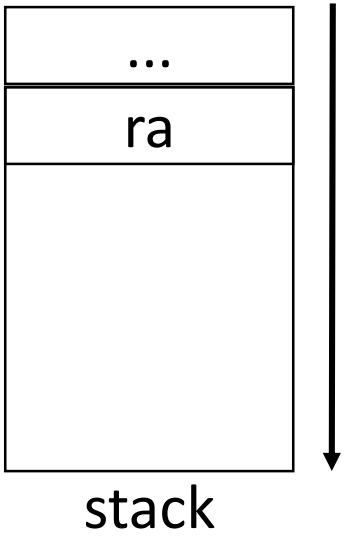
- 被调用的函数可以自由使用
  - 调用者如果需要使用这些值的话，调用者必须要自己去保存

- `X5-x7 + x28-x31: t0-t6` (*temporary* registers，临时寄存器)
- `X10-x11: a0-a1` (return values，返回值)
  - 保存需要传回来的返回值
- `x1: ra` (return address)

Register	ABI Name	Description	Saved By Callee?
x0	zero	Always Zero	N/A
x1	ra	Return Address	No
x2	sp	Stack Pointer	Yes
x3	gp	Global Pointer	N/A
x4	tp	Thread Pointer	N/A
x5-7	t0-2	Temporary	No
x8	s0/fp	Saved Register/Frame Pointer	Yes
x9	s1	Saved Register	Yes
x10-x17	a0-7	Function Arguments/Return Values	No
x18-27	s2-11	Saved Registers	Yes
x28-31	t3-6	Temporaries	No

栈帧结构

- Prologue
  - func_label:
  - addi sp, sp, -framesize
  - sw ra, <framesize-4>(sp)
  - save other regs if needed
- Body (call other functions…)
  - ……
- Epilogue
  - restore other regs if needed
  - lw ra, <framesize-4>(sp)
  - addi sp, sp, framesize
  - jalr x0, 0(ra)



小结

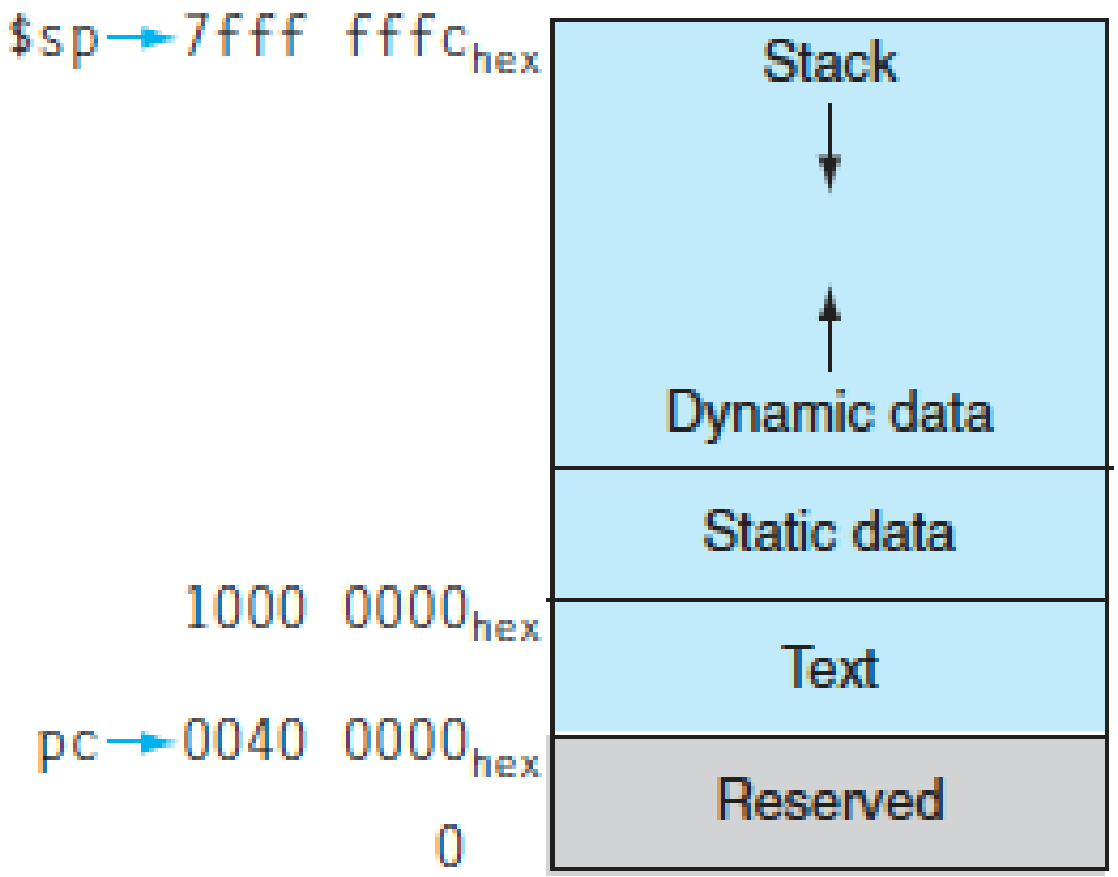
- 计算机理解对应ISA中的指令
- RISC的设计原则：更小更快，保持简单
- 指令类型
  - 算术指令、逻辑指令、移位指令
  - 数据传输指令（访存指令）
  - 比较指令、有条件跳转指令、无条件跳转指令
- 函数调用之间通过调用惯例来指导参数的放置以及寄存器的使用
  - 调用者（caller）和被调用者（callee）都有自己的可直接使用寄存器和需要保存的寄存器
  - 寄存器被分类为被保存的寄存器和易失的寄存器

阅读和思考

- 阅读
- 思考
  - 计算机指令系统中哪些是必备指令?为什么?
  - 指令寻址方式有哪些?这些寻址方式可以在高级语言程序中找到哪些影子?
  - 分析ThinPAD RISCV指令系统的在寻址方式和指令格式方面的特点
  - 根据ThinPAD RISCV指令系统要求，确定ALU应具备的功能

谢谢

Linux操作系统对于进程的内存排布



pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol` (*non-PIC*)	`auipc rd, delta[31:12]+delta[11]` `addi rd, rd, delta[11:0]`	Load absolute address, where `delta = symbol − pc`
`la rd, symbol` (*PIC*)	`auipc rd, delta[31:12]+delta[11]` `l{w|d} rd, rd, delta[11:0]`	Load absolute address, where `delta = GOT[symbol] − pc`
`lla rd, symbol`	`auipc rd, delta[31:12]+delta[11]` `addi rd, rd, delta[11:0]`	Load local address, where `delta = symbol − pc`
`l{b|h|w|d} rd, symbol`	`auipc rd, delta[31:12]+delta[11]` `l{b|h|w|d} rd, delta[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, delta[31:12]+delta[11]` `s{b|h|w|d} rd, delta[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, delta[31:12]+delta[11]` `fl{w|d} rd, delta[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, delta[31:12]+delta[11]` `fs{w|d} rd, delta[11:0](rt)`	Floating-point store global

The base instructions use `pc`*-relative addressing, so the linker subtracts* `pc` *from* `symbol` *to get* `delta`*. The linker adds* `delta[11]` *to the 20-bit high part, counteracting sign extension of the 12-bit low part.*

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned

pseudoinstruction	Base Instruction	Meaning
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset[31 : 12] + offset[11]` `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31 : 12] + offset[11]` `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`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 rd, rs`	`csrrw rd, fcsr, rs`	Swap 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
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags