Behind ICS-PA

王慧妍

why@nju.edu.cn

南京大学 软件学院 计算机软件研究所

本讲概述

- 回顾
- 关于riscv
  - 为什么PA要求大家走riscv-32线
- 从指令集看计算机系统的设计
  - In the middle of Software and Hardware
- 还有些什么？

本讲概述

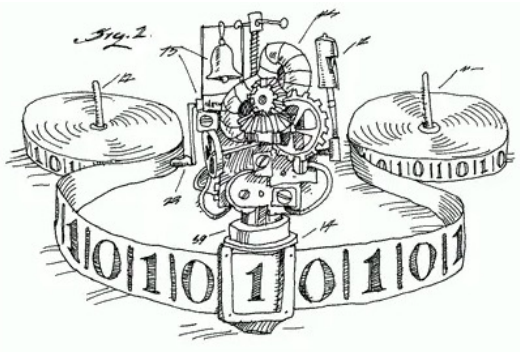
我们PA为什么要求大家做RISC-V线？

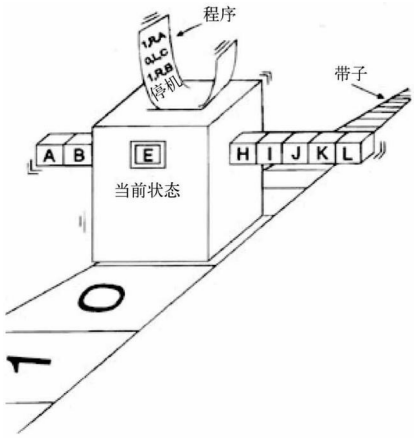
- PA所有可支持线路
  - x86
  - mips32
  - riscv32(64)
  - ……

	x86	mips32	riscv32(64)
PA1 - 简易调试器	与ISA选择关系不大		
PA2 - 冯诺依曼计算机系统	★☆☆☆☆	★★★☆☆	★★★★★
PA3 - 批处理系统	★★★★★	★★★☆☆	★★★★★
PA4 - 分时多任务	★★★★★	★☆☆☆☆	★★★★☆

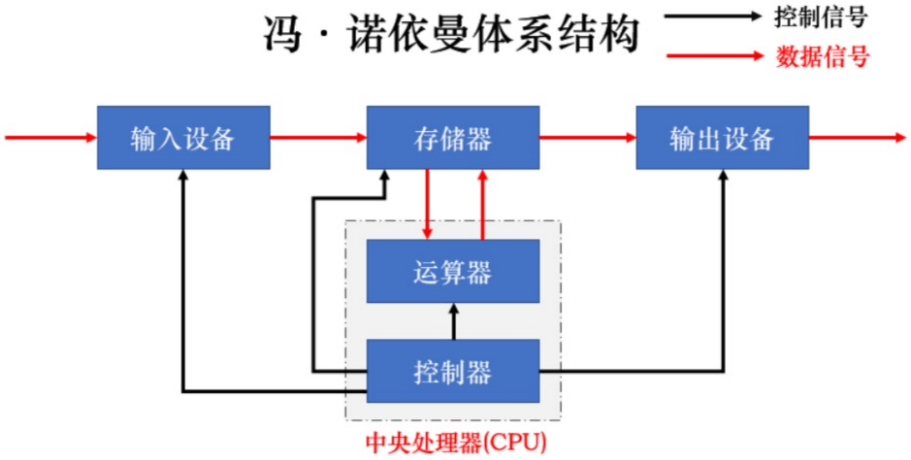
一切背后的故事

- 什么可计算

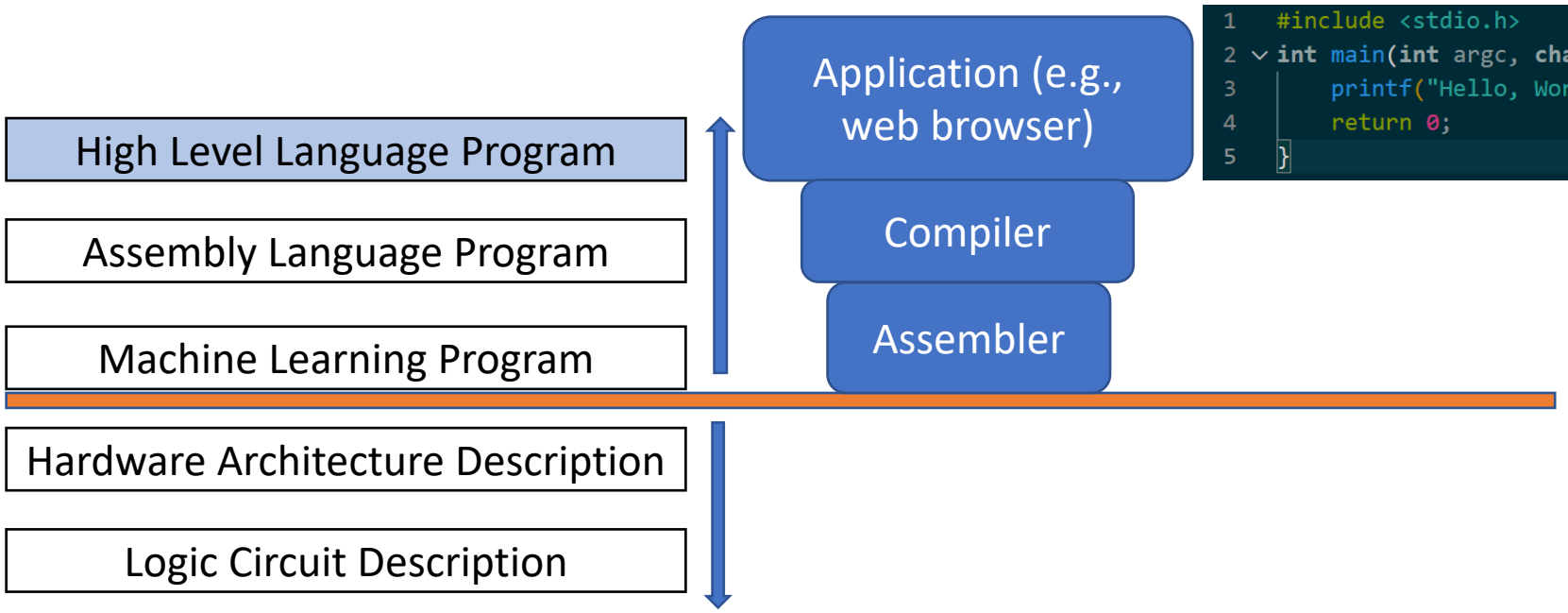




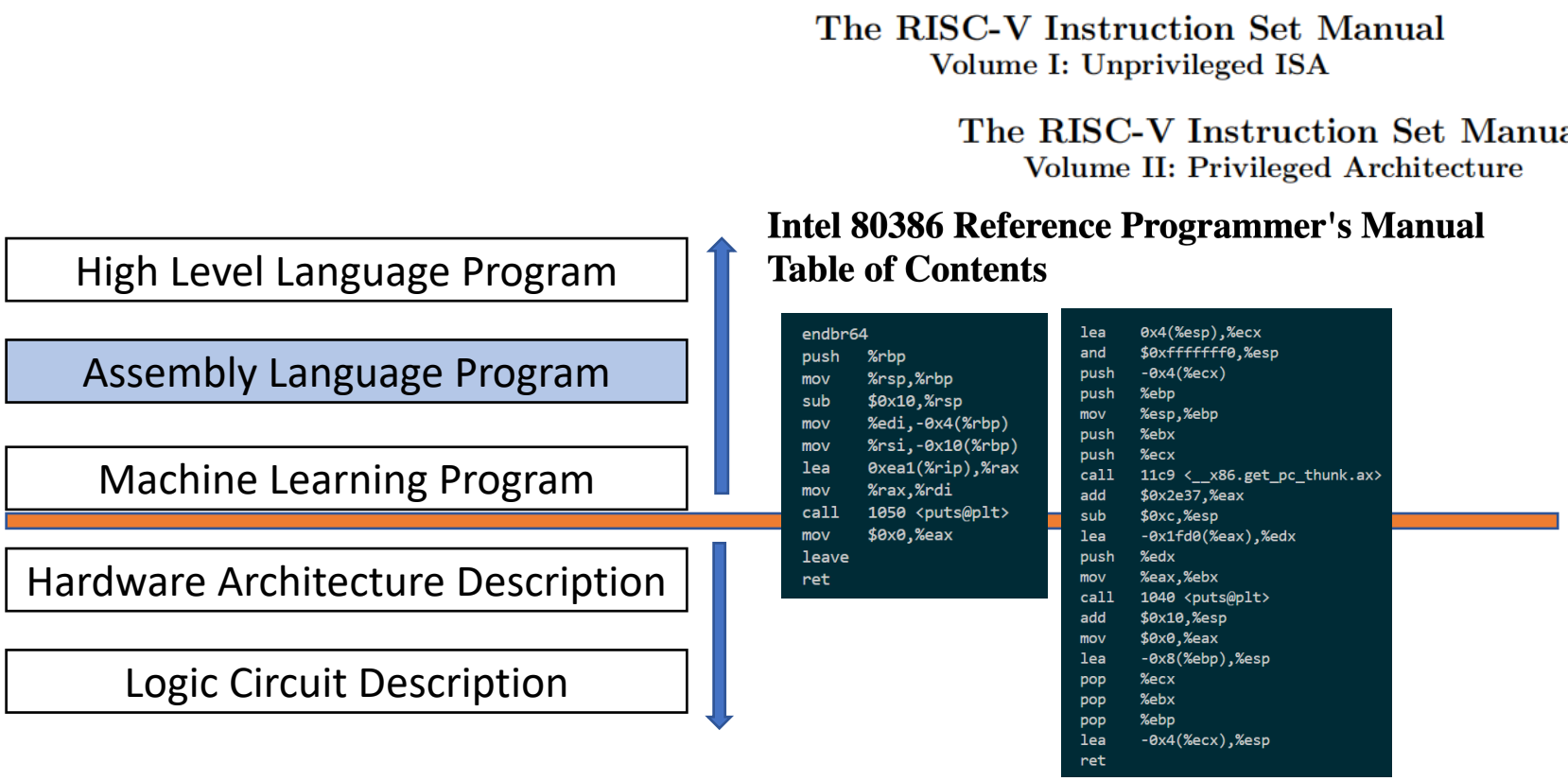
- 如何进行计算



语言的抽象


High Level Language Program
Assembly Language Program
Machine Learning Program
Hardware Architecture Description
Logic Circuit Description
Application (e.g., web browser)
Compiler
Assembler
#include <stdio.h>
int main(int argc, char** argv){
printf("Hello, World!\n");
return 0;
}

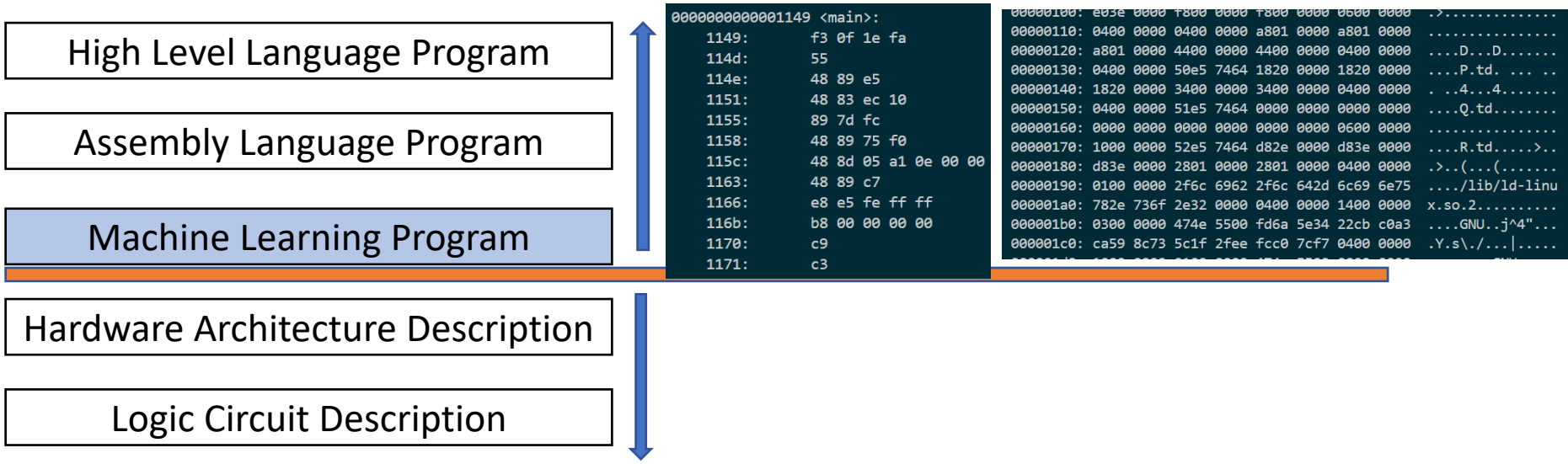
语言的抽象



Each assembly language is just a human readable version of machine language

Tie to a specific ISA

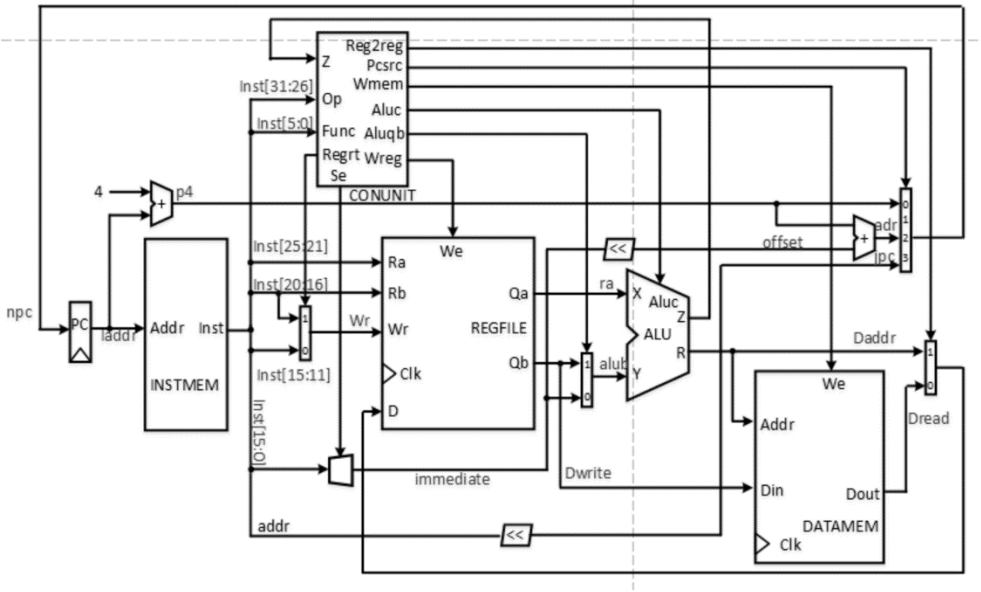
语言的抽象



Each assembly language is just a human readable version of machine language

Tie to a specific ISA

语言的抽象



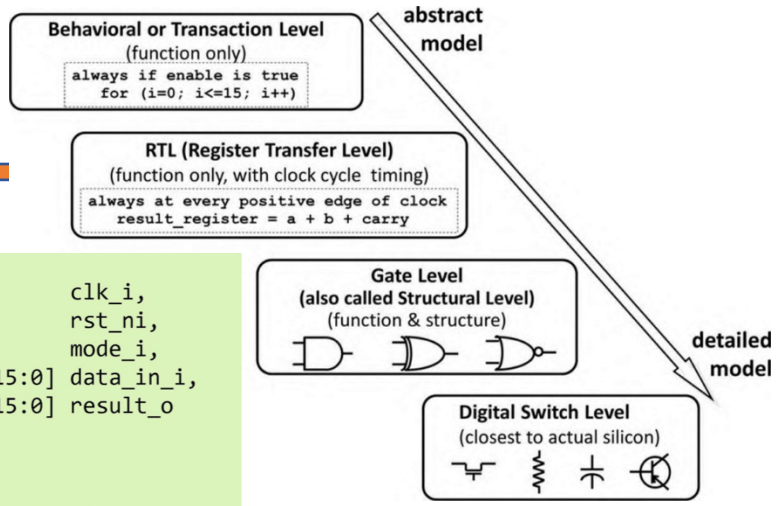
High Level Language Program

Assembly Language Program

Machine Learning Program

Hardware Architecture Description

Logic Circuit Description




```
module top (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic        mode_i,
  input  logic [15:0] data_in_i,
  output logic [15:0] result_o
);


// instantiate one block
  ffs #(.Width(16)) i_reg_1 (
      .clk_i(clk_i),
      .rst_ni(rst_ni),
      .in_i(data_in_i),
      .out_o(first));

endmodule // top
```

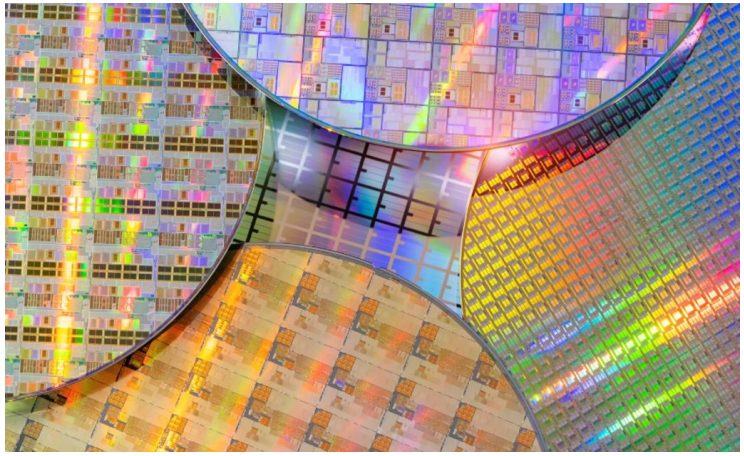
再看性能公式

$$\frac{time}{program} = \frac{instruction}{program} * \frac{cycle}{instruction} * \frac{time}{cycle}$$

再看性能公式

$$\frac{time}{program} = \frac{instruction}{program} * \frac{cycle}{instruction} * \frac{time}{cycle}$$

- 简单微结构 v.s. 复杂微结构
  - 简单微结构
    - 一个周期完成更多事情 $\Rightarrow \frac{cycle}{instruction}$ ⇩
    - 关键路径较长 $\frac{time}{cycle}$ ⇧
  - 复杂微结构
    - 事情拆分到多个周期完成 $\Rightarrow \frac{cycle}{instruction}$ ⇧
    - 关键路径较短 $\frac{time}{cycle}$ ⇩



再看性能公式

$$\frac{time}{program} = \frac{instruction}{program} * \frac{cycle}{instruction} * \frac{time}{cycle}$$

- CISC v.s. RISC
  - CISC
    - 包含行为复杂的指令，编译器可以选择更优的指令
    - 但是复杂的指令执行时间较长
  - RISC
    - 指令简单，编译器可选方案较少
    - 简单指令执行实际较短

$\frac{instruction}{program}$ ⇩

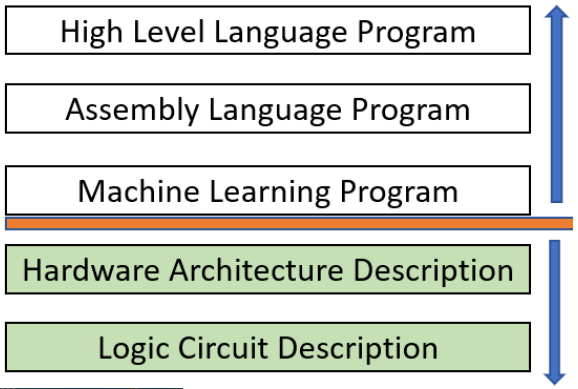
$\frac{cycle}{instruction}$ ⇧

$\frac{instruction}{program}$ ⇧

$\frac{cycle}{instruction}$ ⇩

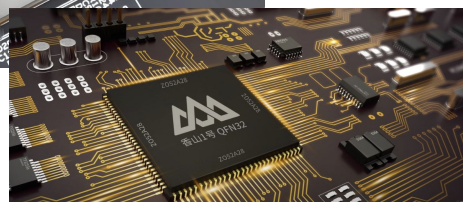
ISA和汇编

- 指令集
  - 沟通软件与硬件
  - 对于软件：ISA是一个抽象接口
  - 对于硬件：ISA是一个功能规约



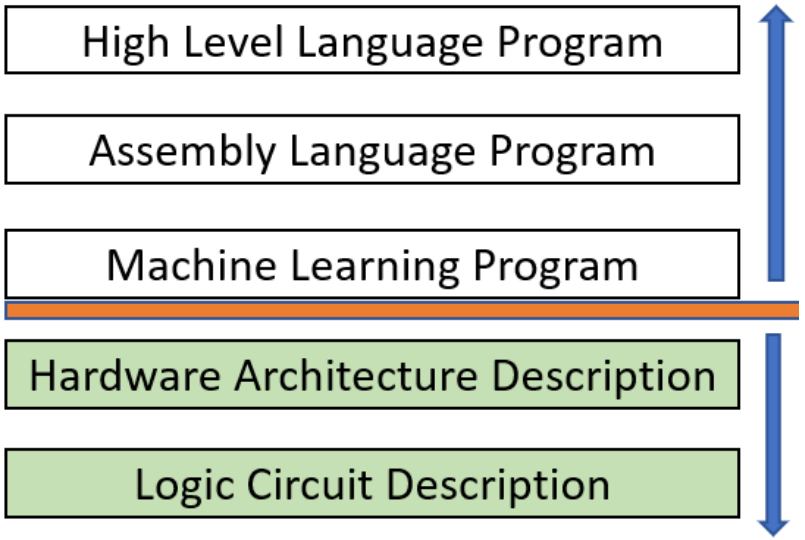
- CISC（1960~1970兴起）
  - x86为主（8086处理器、~300条指令）

- RISC（1980理念）
  - Patterson，Hennessy，1980s
  - ARM、MIPS、RISC-V（2010）
  - LoogArch



ISA和汇编

- 指令集
  - 沟通软件与硬件
  - 对于软件：ISA是一个抽象接口
  - 对于硬件：ISA是一个功能规约



- 主流指令集
  - x86
    - 从1978年80条指令增加到2015年3600条（平均每4天增加一条）
  - ARM
    - v7（整数计算/乘除/原子）：> 278条
  - RISC-V
    - RV32I: 47条
    - RV32IMA：68条

精简or复杂

- x86
  - enter-创建栈帧
    - enter 0, 0等价于：push ebp； mov ebp, esp
    - 但是uop层面复杂度不同

精简or复杂

- x86
  - enter-创建栈帧
    - enter 0, 0等价于：push ebp； mov ebp, esp
    - 但是uop层面复杂度不同
  - rep movsb
    - 不是一条指令，更像是一个uop loop
  - cpuid
    - 各种复杂查询的接口（根据eax取值不同查询功能）

精简o

- x86
  - ente
    - e
    -
  - rep
    - 47 694 7706

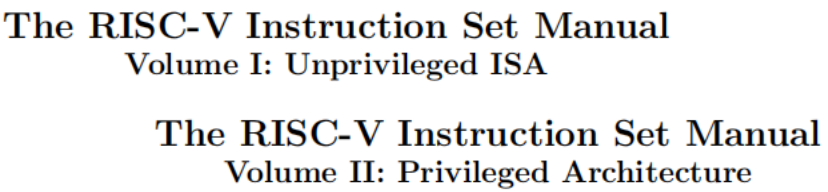
精简or复杂

- x86
  - enter-创建栈帧
    - enter 0, 0等价于：push ebp； mov ebp, esp
    - 但是uop层面复杂度不同
  - rep movsb
    - 不是一条指令，更像是一个uop loop
  - cpuid
    - 各种复杂查询的接口（根据eax取值不同查询功能）
- ARM
  - crc32：计算循环冗余校验码（Cyclic Redundancy Check）
  - ldmiaeq SP!, {R4-R7, PC}指令（v8去除）
- MIPS

RISC-V

- 官方手册
  - RISC-V Instruction Set Manual
    - Volume 1: Unprivileged ISA
    - Volume 2: Privileged Architecture

- 特色
  - 简单、干净、无历史包袱
  - 与微结构设计解耦
  - 模块化：可以根据需要扩展
    - 变长的指令编码
    - 预留扩展空间
    - 扩展相互独立、新版本兼容旧版本
  - 开放稳定（RISC-V基金会所有）
    - MIPS公司宣布转入RISC-V阵营



模块化的场景

- 基础指令集
  - RV32I、RV64I、RV128I、RV32E
    - RV32E是16寄存器的RV32I变种
    - 基础指令只有40+条

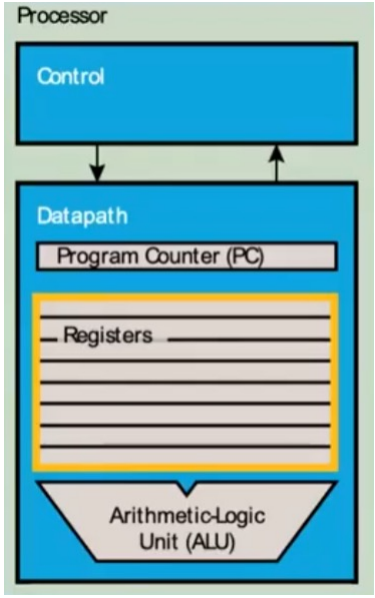
- 标准扩展
  - M-整数乘除、F-单精度浮点、G=IMAFD
  - A-原子操作、D-双精度浮点、C=压缩指令

  - 自由组合
    - 桌面：RV64GC
    - 高性能：RV64GCBV
    - 嵌入式：RV32E、RV32IC

Base	Version
RVWMO	2.0
RV32I	**2.1**
RV64I	**2.1**
RV32E	*1.9*
RV128I	*1.7*
Extension	**Version**
Zifencei	**2.0**
Zicsr	**2.0**
M	**2.0**
A	*2.0*
F	**2.2**
D	**2.2**
Q	**2.2**
C	**2.0**
Ztso	*0.1*
Counters	*2.0*
L	*0.0*
B	*0.0*
J	*0.0*
T	*0.0*
P	*0.2*
V	*0.7*
N	*1.1*
Zam	*0.1*

Register

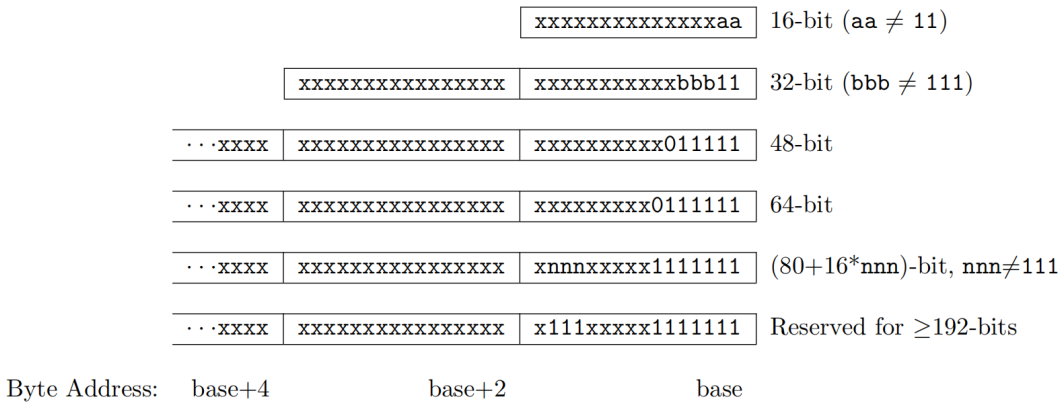
- 寄存器数量
  - RISC-V：32个GPR（**x0**~x31、PC）
  - MIPS：32个GPR，有$zero，PC
  - ARM-v7：16个GPR，无零寄存器
    - PC甚至也是个通用寄存器
  - x86(32位)只有8个：8个GPR，无零寄存器
    - 取立即数- mov eax, 0
    - 类似xor指令清零- xor eax, eax



指令格式

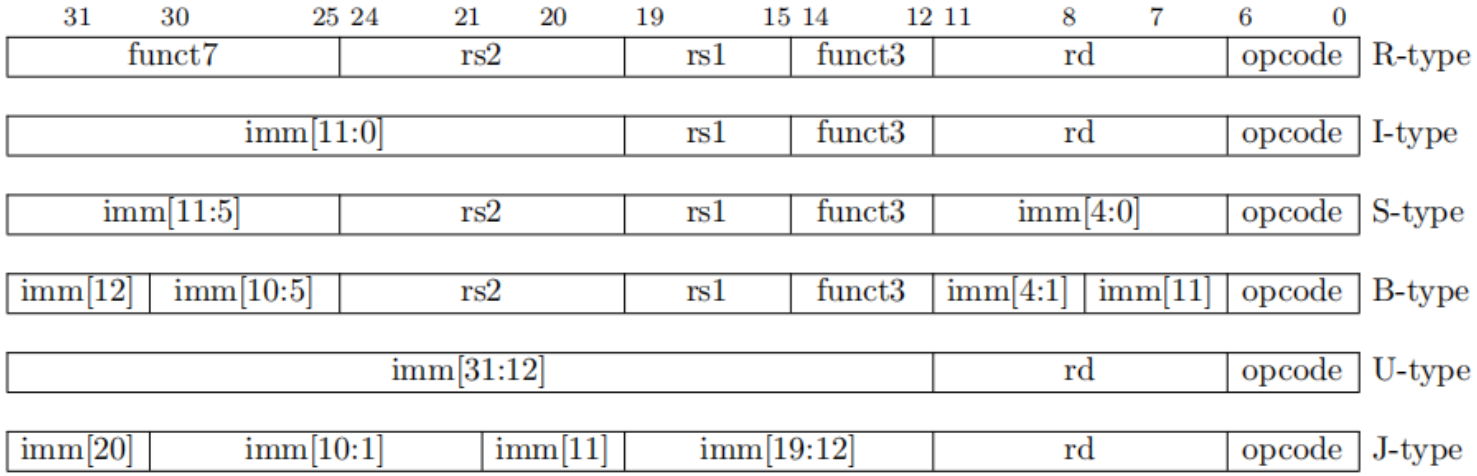
```
0000000000001149 <main>:
    1149:       f3 0f 1e fa
    114d:       55
    114e:       48 89 e5
    1151:       48 83 ec 10
    1155:       89 7d fc
    1158:       48 89 75 f0
    115c:       48 8d 05 a1 0e 00 00
    1163:       48 89 c7
    1166:       e8 e5 fe ff ff
    116b:       b8 00 00 00 00
    1170:       c9
    1171:       c3
```

- 选择：
  - x86：变长指令集，有无限的操作码空间
  - MIPS/ARM：定长指令集，有限的操作码空间（总有用完的一天）
    - ARM增加**模式位**扩展操作码范围，设计Thumb和Thumb-2指令集
  - RISCV：基础和标准扩展大多为4字节定长指令集
    - 扩展可以更加需要选择，支持变长指令集
    - 并且不影响现有RISC-V处理器



指令格式

- 统一指令长度
  - 简化译码器实现
    - 越复杂→成本越高 + 性能影响越大
    - 译码逻辑相似



  - MIPS中：目的寄存器
    - R型：[15:11]
    - I型：[20:16]

基础指令

- RISC-V采用三地址指令
  - a = b + c ;

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

  - 大部分x86采用二地址指令

```
1199:       05 3f 2e 00 00          add    $0x2e3f,%eax
119e:       8b 45 08                mov    0x8(%ebp),%eax
```

  - 全0、全1指令都是非法指令
    - x86全0代表 `add %al, (%eax)`
    - MIPS全0代表空指令
    - MIPS全1代表 `sdc3 $31, -1(ra)`

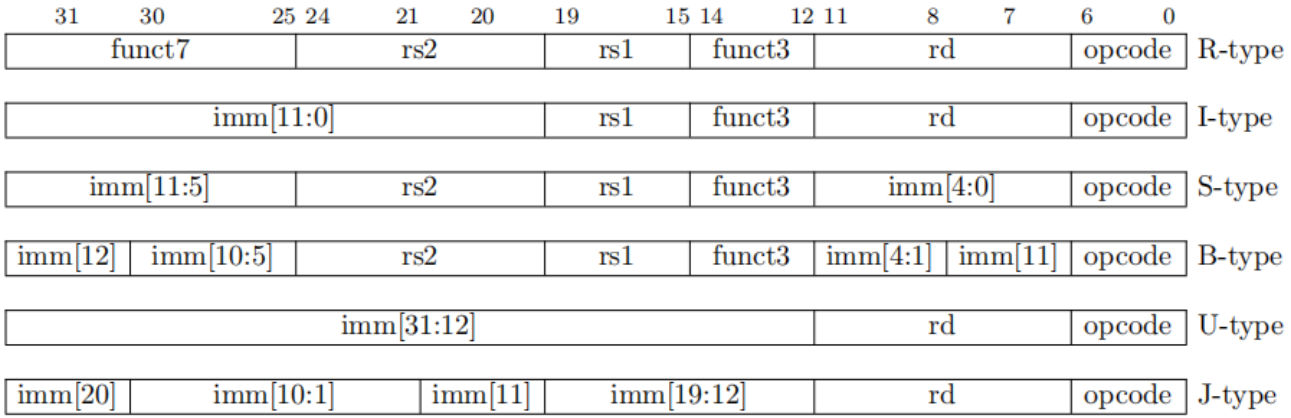
基础指令

- Addition/Subtraction
  - 没有subi指令（I-type）

Integer Register-Immediate Instructions

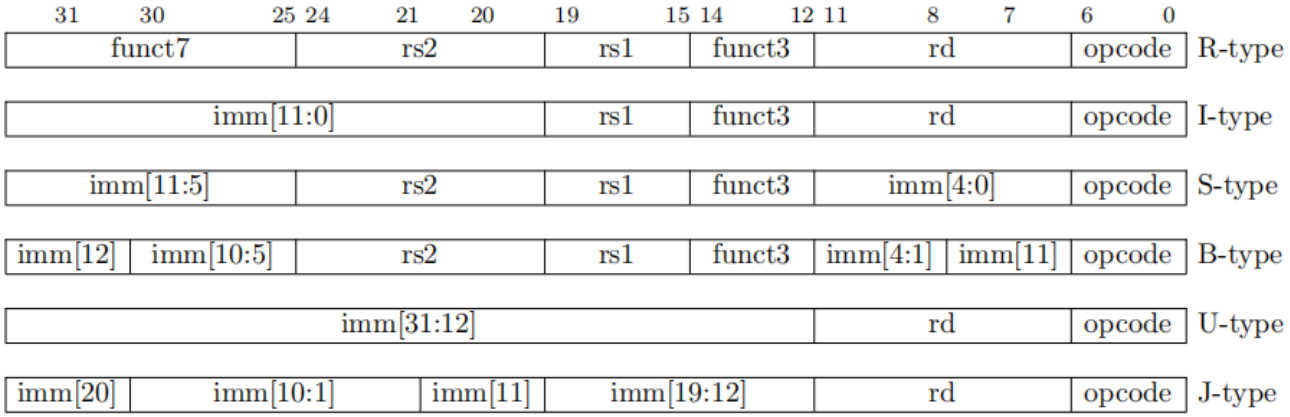
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.



基础指令

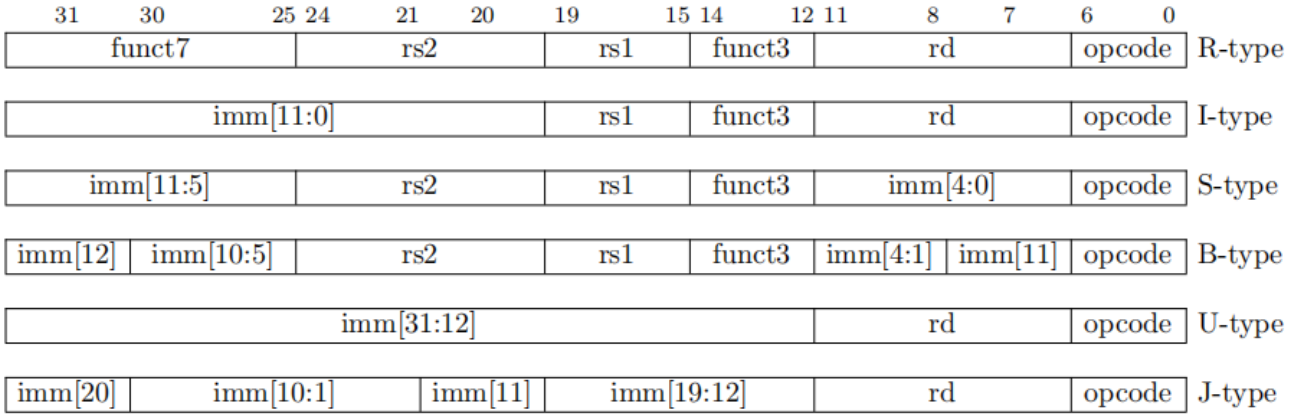
- Immediate number



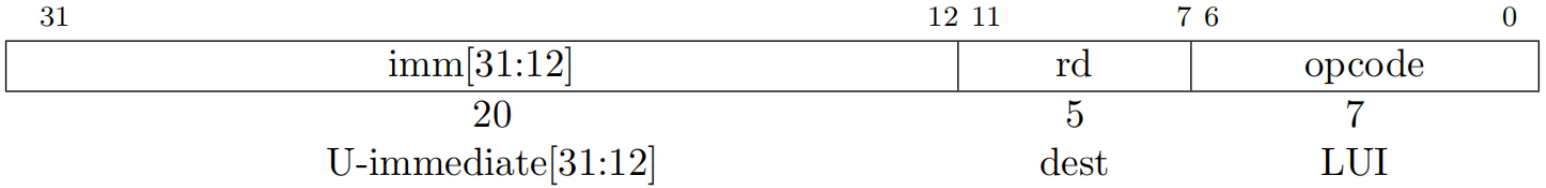
  - 减少立即数每一位可能来源于指令对应位的差别
    - `imm[31]`只可能来源于`inst[31]`，无需选择器
    - `imm[5]`只可能来源于`inst[25]`或`0`(U型)，只需2选1选择器
    - 编译时需要分段放立即数，但是代价可忽略不计

基础指令

- Immediate number



- 减少立即数每一位可能来源于指令对应位的差别
  - U型指令和其他指令的组合（U+I）
    - auipc + lw：支持PIC的关键核心
  - 立即数关注20+12的位数
    - lui(riscv): 7 + 5 + 20
    - MIPS: 6 + 5 + 5(rs = 0) + 16

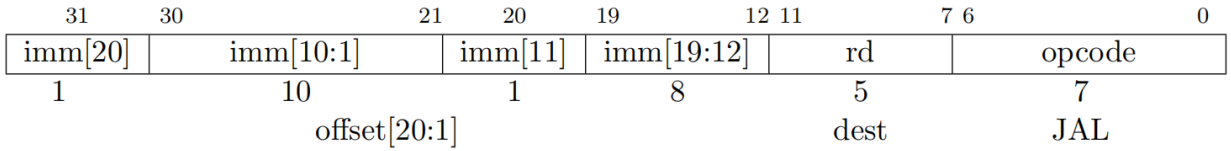


跳转指令

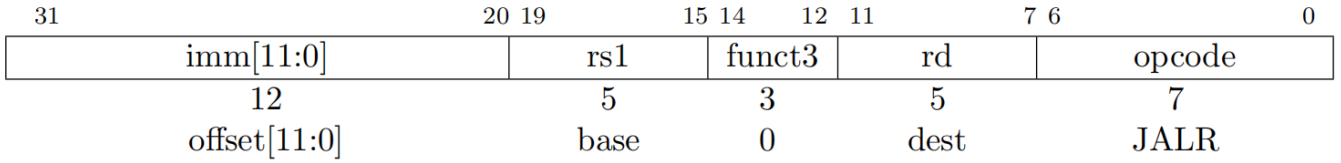
```
+------+-----+-----+-----+-----+------+
|000000|  rs |00000|00000|00000|001000|  jr   (浪费较多比特)
+------+-----+-----+-----+-----+------+
+------+-----+-----+-----+-----+------+
|000000|  rs |00000|  rd |00000|001001|  jalr (浪费较多比特)
+------+-----+-----+-----+-----+------+
+------+-------------------------------+
|000010|           offset              |  j
+------+-------------------------------+
+------+-------------------------------+
|000011|           offset              |  jal
+------+-------------------------------+
```

- 无条件跳转
  - MIPS有jr、jalr、j、jal
  - RISC-V只需要jal和jalr
    - `jal rd, imm` – 返回地址保存到rd，跳转到PC+imm
    - `rd = x0`实现j
    - j和jr为伪指令，不占用操作码空间

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a JAL with *rd*=x0.



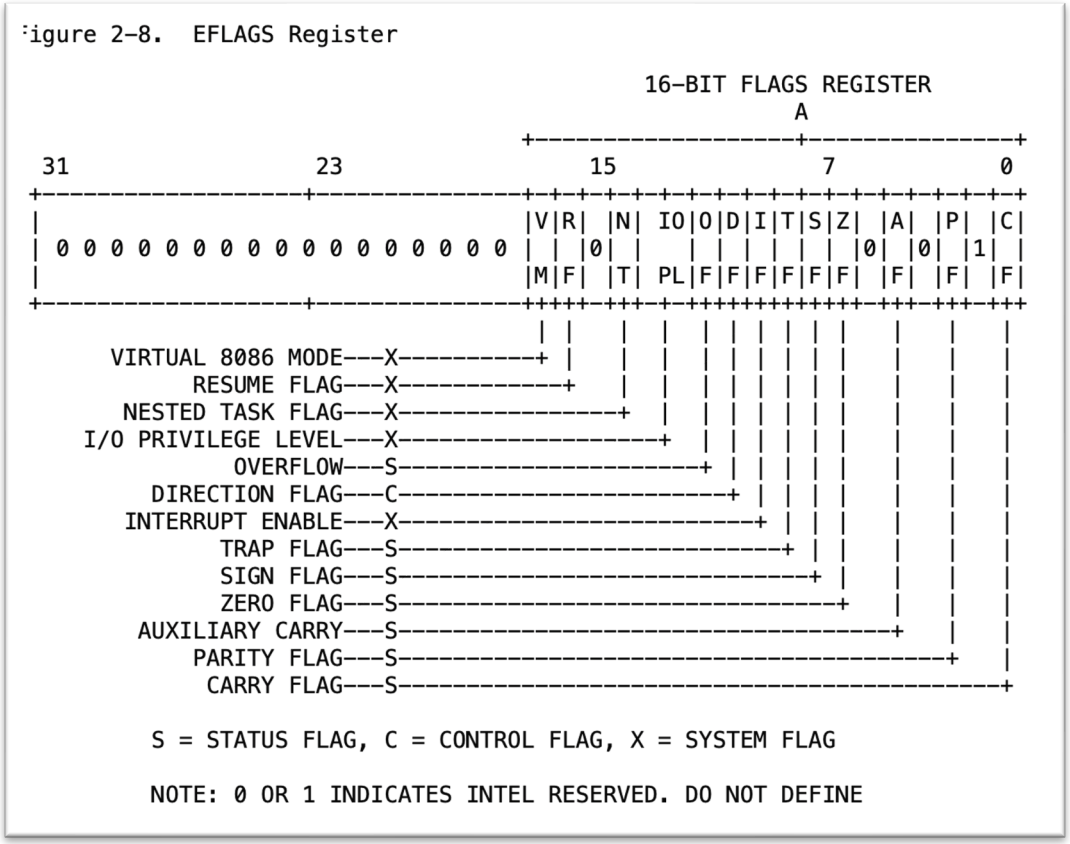
- 甚至ret也是一条伪指令
  - `jalr x0, x1, 0`



跳转指令

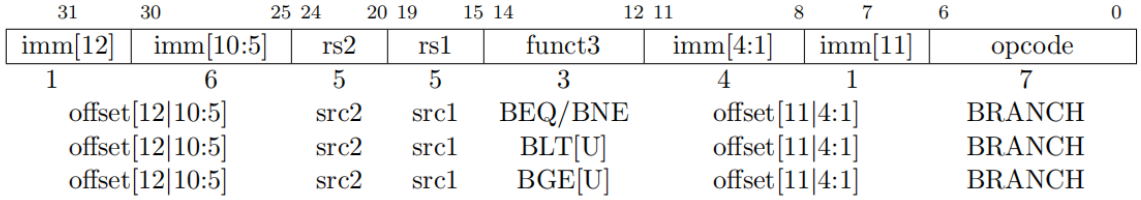
```
je    label      # ZF=1            (相等) A == B
jne   label      # ZF=0            (不等) A != B
jl    label      # SF≠OF           (小于) A < B
jle   label      # (SF≠OF)或ZF=1   (小于等于) A <= B
jg    label      # (SF=OF)且ZF=0   (大于) A > B
jge   label      # SF=OF           (大于等于) A >= B
```

- 有条件跳转
  - x86：隐式比较，需按要求设置标志位（Eflags）
    - `cmp eax, ebx;  jl label;`



跳转指令

- 有条件跳转
  - x86：隐式比较，需按要求设置标志位（Eflags）
  - MIPS：即时比较，提供较多条件指令
    - 颇具争议的延迟槽设计
    - `bne; addi; sw;`
  - RISC-V：即时比较，更加精简（ 6条）
    - 没有ble、bgt、bltz、bgtz等（丰富伪指令）
    - 无分支延迟槽设计



31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

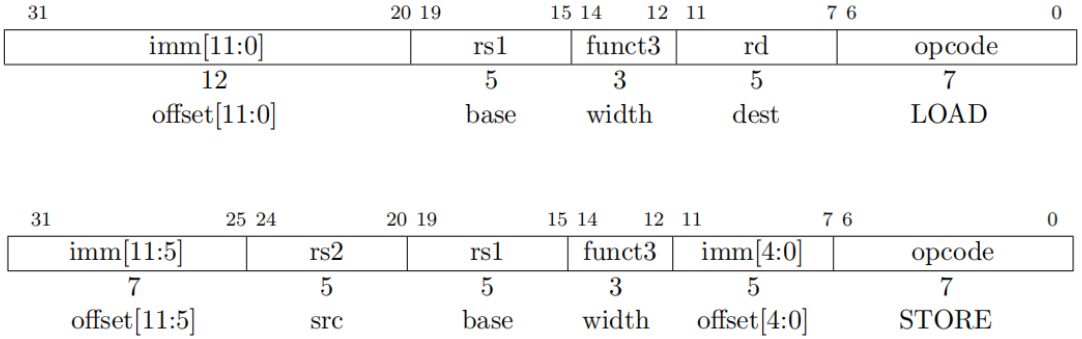
延迟槽设计

- 设计与微结构关联
  - MIPS延迟槽指令
    - 改变分支执行的顺序
    - 可能带来编译器/处理器设计处理的复杂性和无意义的nop
    - 延迟槽为1？超标量十几级流水线怎么弄

    - MIPS的历史负担，已在release 6中被移除（ –mips32r6 ）
    - LoongArch作为基于MIPS设计的指令集，不采用延迟槽设计

内存访问指令

- RISC-V的一大特点是精简
  - 只有专门的加载和存储指令才能够进行内存访问（RISC理念）
  - 寻址方式简单
    - `lw rd, offset(rs1)`



内存访问指令

- RISC-V的一大特点是精简
  - 只有专门的加载和存储指令才能够进行内存访问（RISC理念）
  - 寻址方式简单
    - 除了lw/sw，还有half-word/byte data transfer：lh/sh, lb/sb

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

  - 特例：有lbu为什么没有sbu？
    - sbu需要保持“读-修改-写”的原子性
  - 类似的不对称性恰恰提醒了RISC-V的精简设计思想

逻辑指令

- RISC-V：仅支持6条基础指令
  - AND、OR、XOR、ANDI、ORI、XORI
  - 没有NOT、XNOR等

  - B扩展（`-march=rv64imazbb_zbs`）
    - 计算32位整数的1的个数（beqz+andi+add+srli+j组合）
    - `cpop rd, rs` (R-TYPE)
    - 还有`clz`、`ctz`等

设计考量

- 基础集指令译码精简、降低复杂度
- 扩展集增强自由度，适配不同需求与场景
- 系统设计，考虑实际低/高频事件后的指令设计
- 作为开放标准指令集架构，正在接受时代的检验
  - 碎片化和生态依然是推广中的普遍忧虑
- 推荐课外读物
  - 《The RISC-V Reader》和官方手册
  - UC Berkeley CS61C
    - RISC-V诞生处（2010年）

Don’t forget reality!

- CISC与RISC的相互靠近

- 多核时代+异构硬件+超大规模数据中心+AI
  - 并行化、效率瓶颈、安全性、能耗

学习≠获得分数

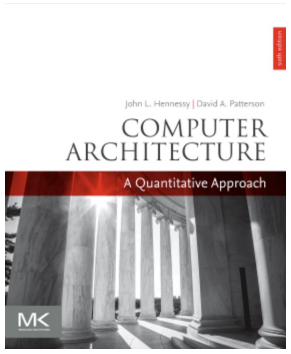
High Level Language Program

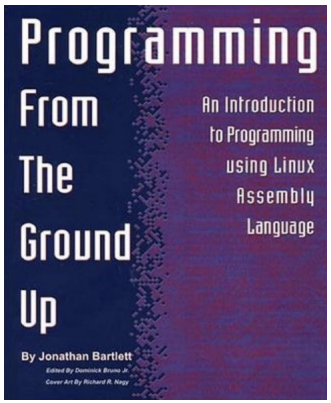
Assembly Language Program

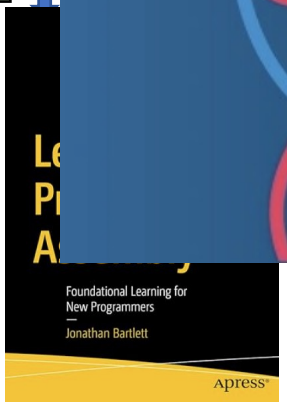
Machine Learning Program

Hardware Architecture Description

Logic Circuit Description







机器永远是对的

没什么是 RTFM/RTFSC 解决不了的

知道了计算机系统这个 “状态机” 是如何工作的

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
			SD	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

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word strores) 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

	15–12	11–10	9–7	6–5	4–3	2–0	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.