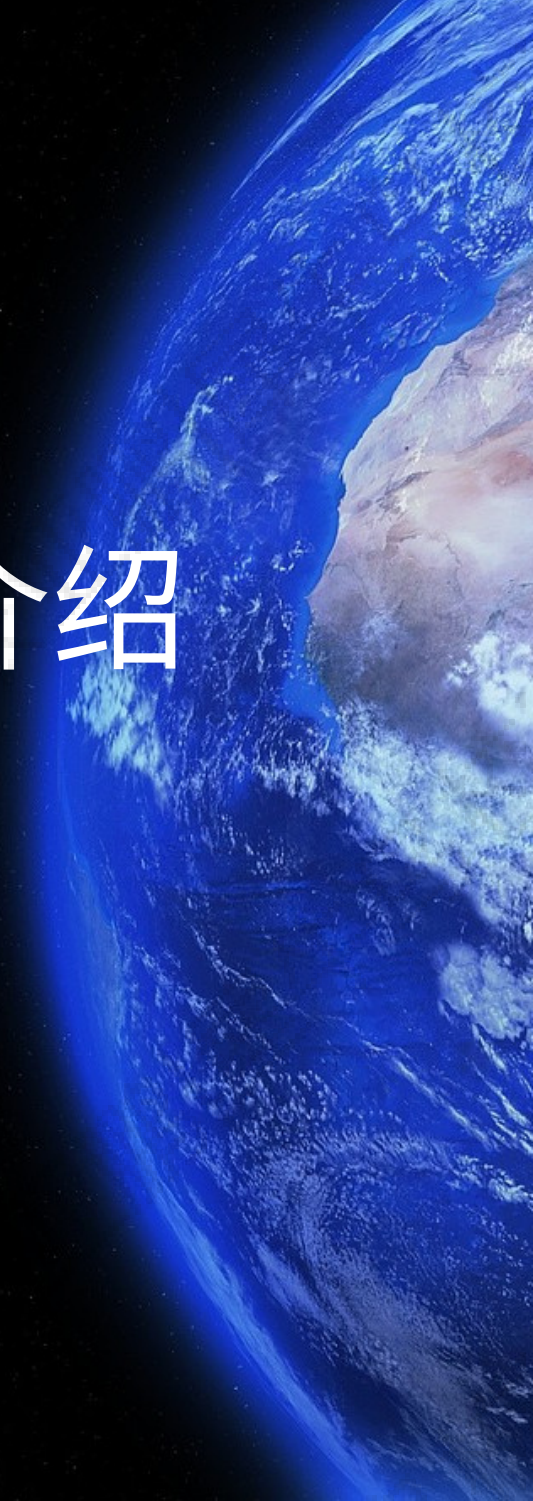
LoongArch 指令集介绍



龙芯实验室

2021/04/04

1 指令集基础知识

2 龙芯指令集简介

指令集基础知识

什么是指令集?

- ISA
  - Instruction Set Architecture
- 软硬件之间的 '合同'
  - 硬件所支持的操作(指令)、运行模式和存储界面等的功能定义
  - 上述硬件资源的访问方式
- 不约定具体实现方式和效果
  - 指令快慢
  - 功耗大小等

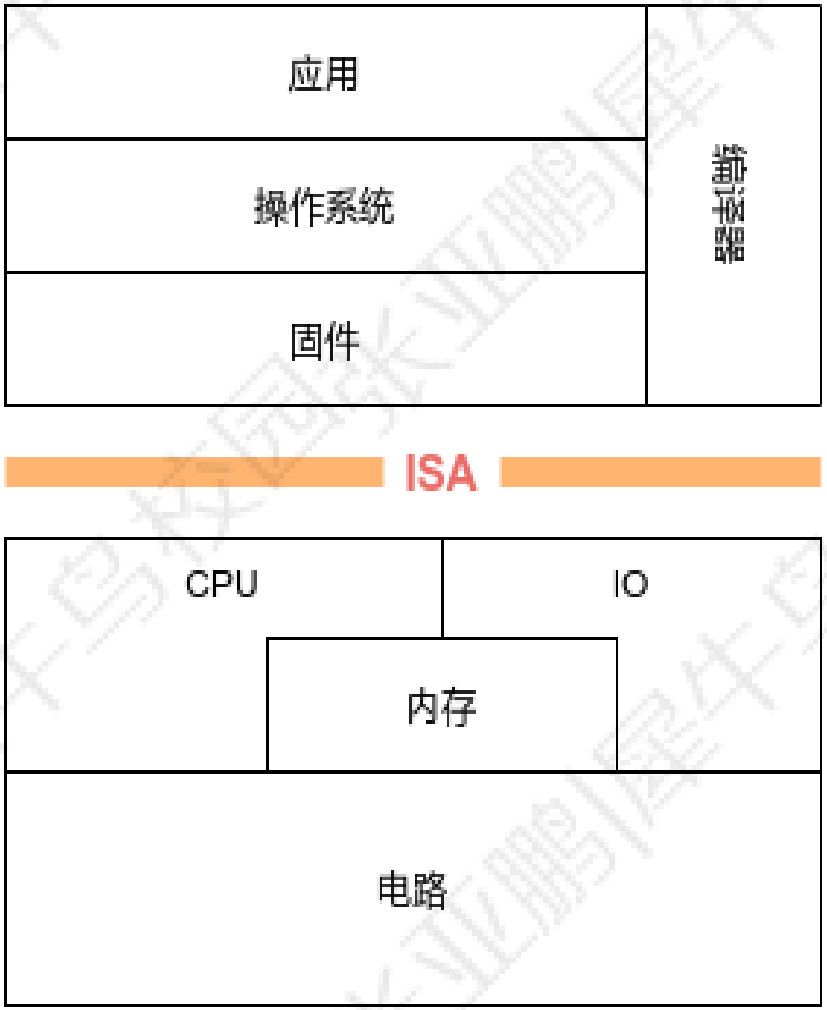


Figure: 指令集

指令集和人类语言的对比

- 相似之处
  - 都是用于沟通
    - 指令集：硬件和软件沟通
    - 人类语言：人和人
  - 有许多可类比的特性
    - 单词 vs 指令
    - 语法 vs ABI
    - 不同的指令集 vs 不同的语言
- 差异之处
  - 指令集是精确描述，没有二义性

指令集的设计考虑

- 指令实现
  - 指令格式
  - 操作数模型
  - 数据类型和寻址模式
- 硬件与软件实现的权衡
  - 应用领域、设计目标
- 二进制程序接口（ABI）
  - 寄存器等资源使用方式
- 系统资源管理
  - 虚拟化、安全、扩展等需求

指令格式

- 定长
  - 优点：实现简单，下一个 PC 易计算
  - 缺点：代码密度
    - 相同源代码，RISC 二进制常常比 CISC 大 20-30%
- 变长
  - 优点：可实现高代码密度
  - 缺点：实现复杂
- 折中：两个长度
  - MIPS16 和 Arm Thumb
- 编码：MIPS 3 种格式 vs. LoongArch 11 种 vs. X86 很多种

操作数模型: memory only

- 操作数来自哪里?
  - 如何指定它们?
  - 例: A = B + C 的实现
- Memory Only
  - add B,C,A
  - mem[A] = mem[B] + mem[C]

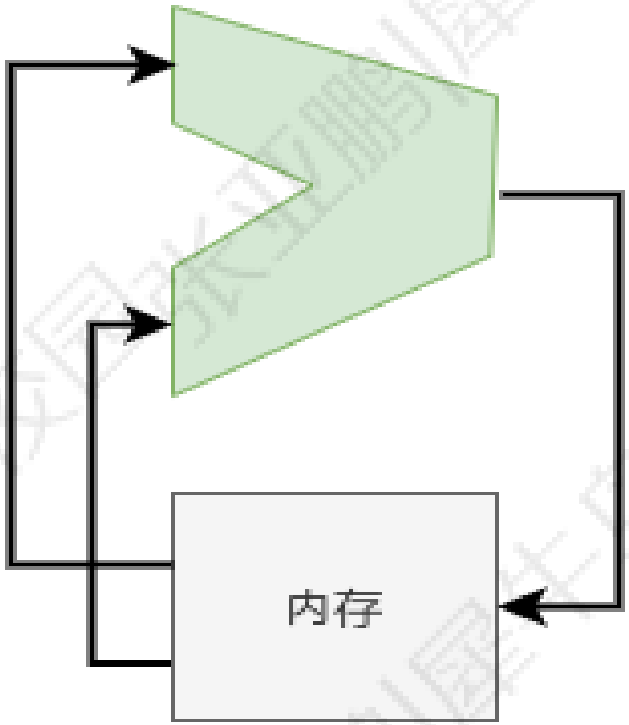


Figure: memory-only

操作数模型: 累加器

- Accumulator
  - load B; add C; store A
  - ACC = mem[B];
    ACC+=mem[C];
    mem[A]=ACC

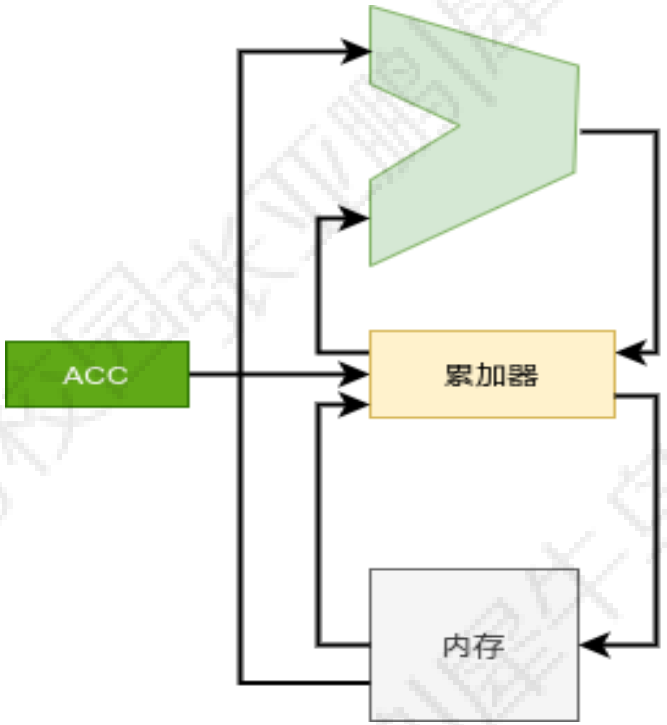


Figure: accumulator

操作数模型: 栈式

- Stack
  - push B; push C; add; pop A;
  - stk[tos++]=mem[B];
  - stk[tos++]=mem[C];
  - stk[pos+
    +]=stk[–tos]+stk[–tos];
  - mem[A]=stk[–tpos]

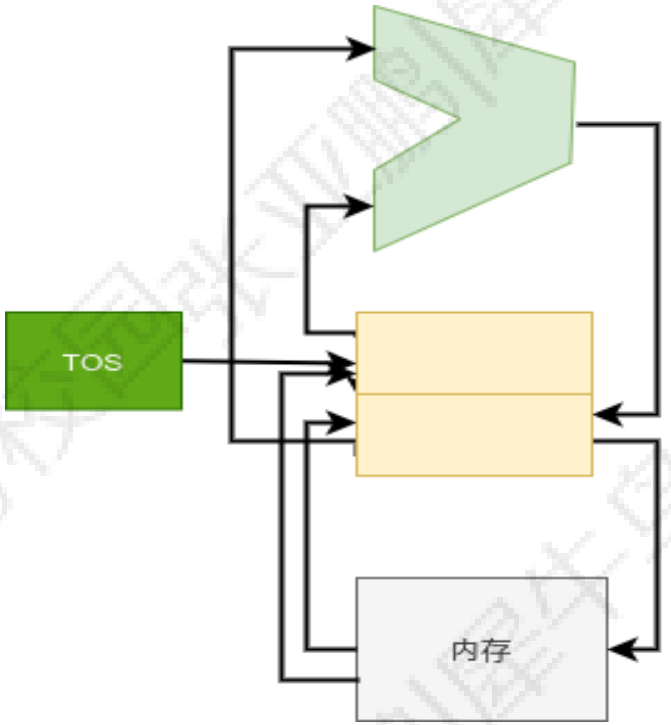


Figure: stack

操作数模型: 寄存器 load-store

- register load-store
  - load R1,B; load R2,C;
  - add R1, R1, R2; store R1, A;
  - R1=mem[B]; R2=mem[C];
  - R1 = R1 + R2; mem[A]=R1;

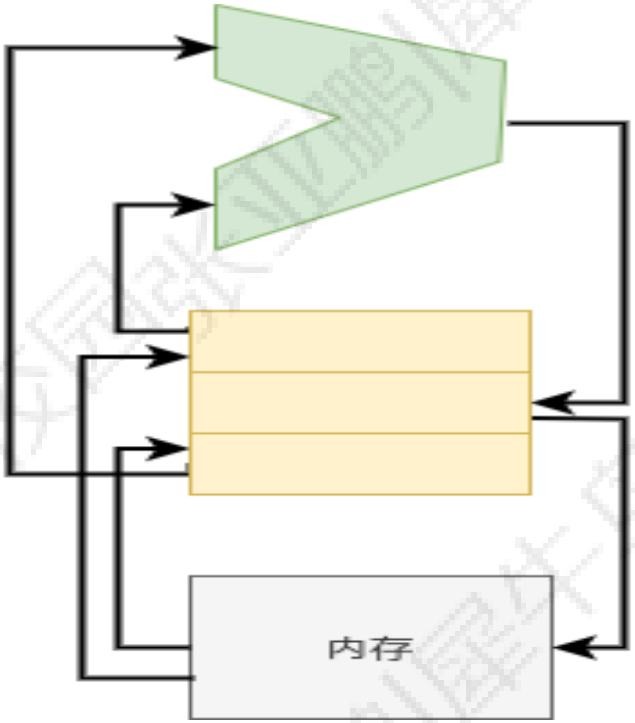


Figure: register

数据类型和寻址模式

- 寄存器数量、位宽、虚地址空间大小
  - xx 位处理器
- 寻址模式: 指定地址的方式
  - register-indirect: 地址 = 寄存器值
  - displacement: 地址 = 寄存器值 + 立即数偏移
  - index-base: 地址 = 寄存器 1 值 + 寄存器 2 值
  - PC-relative: 地址 =PC + 立即数偏移
  - Scaled: 地址 = 寄存器 1 值 + 寄存器 2 值 * 立即数 1+ 立即数 2
  - auto-increasement: 寄存器存的地址用完自动增加
  - 等等

硬件和软件实现的权衡

- 与场景需求、设计目标相关
  - 高性能 vs. 低功耗
- 用简单指令合成还是硬件提供?
  - 硬件做比软件有效益才合算，如字节逆序、数 0 数 1 等指令
  - X86 的字符串拷贝指令被认为是 bad，不比软件高效
  - 慎重添加可能影响实现效率的复杂指令
- 总体趋势是硬件指令不断增加
  - 晶体管数量持续增加
  - 向量、DSP、AI 等负载的有效加速

‘好’ 指令集的三要素

- 可编程性
  - 是不是能够有效地表达程序?
- 可实现性
  - 是不是易于实现高性能?
  - 逐步发展到
    - 是不是易于实现低功耗?
    - 是不是易于实现低成本?
    - 是不是易于实现高可靠?
- 兼容性
  - 软硬件技术发展时是否易于保持可编程性和可实现性?

可编程性

- 1985 年之前主要是面向人类
  - 尽量接近高级语言?
  - X86 call/rep movsb, Sparc save/restore, VAX insqueue 等
- 1985 之后主要面向编译器
  - 提供完备的底层原语而不是综合方案
    - 编译器擅长把复杂结构分解为简单原语
    - 从一堆复杂指令选择最优不容易
  - 原则
    - 规整
    - 正交、可组合

可实现性

- 不是每种指令集都能高效实现
- 某些指令集特性使得一些实现技术的应用变得困难
  - 变长指令、复杂格式：译码复杂
- 可实现性随着技术发展变化
  - 延迟槽技术在单发射静态调度时能有效提升性能，在超标量动态调度时变成负担

兼容性

- 兼容性非常重要
  - Intel 获胜的关键之一
  - Intel 推安腾失败，无法动摇自己的生态
- 兼容性需要指令集设计进行前瞻思考
  - 为了有限的收益引入某些基础性的特性可能带来巨大负担
    - 贴近高级语言：SPARC 寄存器窗口
    - 面向当时流水线过度优化：延迟槽
- 兼容性可以通过技术弥补
  - 二进制翻译

RISC vs. CISC

- CISC
  - VAX/IBM 360&370/X86/Motorola 68000...
- RISC
  - MIPS/SPARC/POWER/PA-RISC/ALPHA/ARM/ RISC-V...
- 80 年代中期开始的 '圣战'
  - RISC 赢了技术战争
  - CISC 赢了商业战争
- RISC/CISC 界限趋于模糊
  - 现代商业 RISC 指令集大都超过 1000 条指令
  - X86 内部实现 RISC 核心

迷思 1：指令集决定功耗

- Arm 天生低功耗?
  - AMD 的 Arm 处理器功耗
  - Nvidia 的几款处理器
- X86 太复杂，功耗肯定高
  - 对于现代高性能处理器，译码逻辑的功耗占比很有限
- 微结构实现对功耗的影响远大于指令集
  - E. Blem 等, “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures,” 2013 HPCA.
  - We find that ARM and x86 processors are simply engineering design points optimized for different levels of performance, and there is nothing fundamentally more energy efficient in one ISA class or the other. The ISA being RISC or CISC seems irrelevant

迷思 2：面向高级语言的指令集?

- VAX 等早期指令集有大量面向高级语言特性的设计
  - 缩小语义鸿沟?
- JAVA 处理器
  - 不时有人提出类似想法
- 编译器能够很好地用基础指令实现高级语言的一些特性
  - 面向对象?
  - 定制特性的收益常常比较有限
- 部分基础性的支持被采纳
  - 边界检查

重要的指令集：X86

- 以收入算的赢家
- CISC 结构
  - 变长指令: 1-15 字节
  - 1000+ 指令
  - 8-16 个通用定点寄存器
  - 支持丰富的寻址模式
  - 大量指令使用隐式操作数
  - 运算指令同时生成结果和条件码
  - X87 浮点采用栈式操作数
  - 16/32/64 位向后兼容

重要的指令集：ARM

- 以出货量算的赢家
- RISC 结构
  - 定长指令 (2/4 字节两种长度)
  - 1000+ 指令，53 种指令格式
  - load/store 结构
    - 八种寻址模式
  - 32 个通用定点寄存器
  - SP 和 PC 专用寄存器
  - 64 位不直接兼容 32 位
    - armv8 抛弃了一些不利于实现的特性

重要的指令集：RISC-V

- 新星
- 典型 RISC 结构
  - 开源
  - 基础整数部分 47 条，定长，无延迟槽
  - 基础部分 6 种格式
  - 32 个通用定点寄存器 (嵌入版本 16 个)
  - load/store
  - 12 位立即数
  - 仅提供一种寻址模式（寄存器 + 立即数）和 PC 相对寻址的分支
  - 扩展组织为多个相对独立的模块

龙芯指令集简介

龙芯指令集 (LoongArch)

- 设计考虑
- 主要特点
- 性能分析与优化
- 二进制翻译支持
- 进展情况
- 下一步工作

设计考虑

- 先进性
  - 积极吸收现代研究成果，摒弃一些过时的技术包袱
- 兼容性
  - 融合 X86/MIPS/Arm 等主流指令系统的主要特点，高效支持二进制翻译
- 扩展性
  - 预留足够的指令槽用于后续扩展

LoongArch 概况

- 充分考虑兼容需求的自主指令系统
- 约 2000 条指令
  - 基础指令 337 条
  - 虚拟机扩展 10 条
  - 二进制翻译扩展 170+
  - 128 位向量扩展 700+
  - 256 位向量扩展 700+

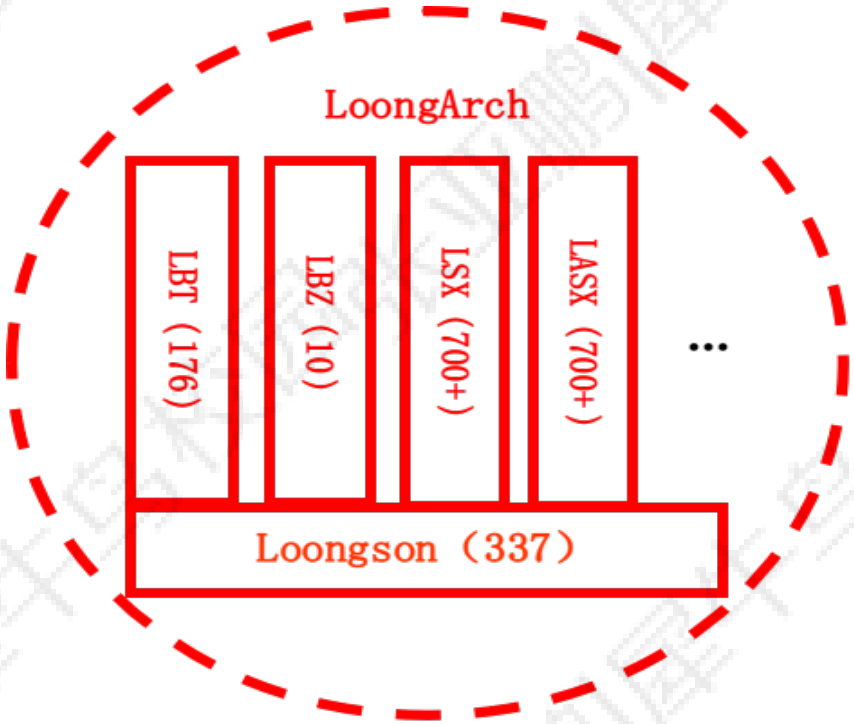


Figure: LoongArch 指令集

LoongArch 特点

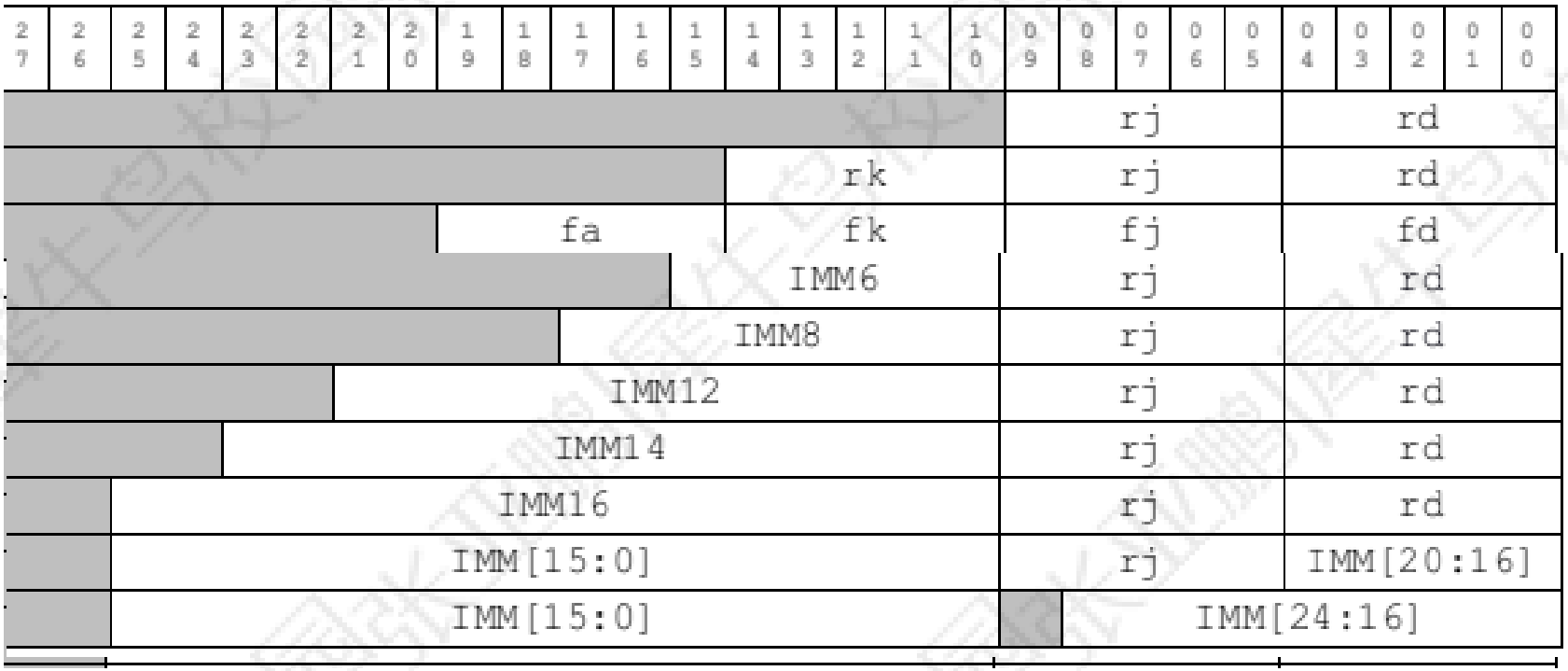
- 用户态指令
  - 保持典型 RISC 风格: 定长、32 通用定/浮点、load-store 结构
  - 提升效率
    - 取消延迟槽
    - 新增以 PC 为源操作的运算指令
    - 实现位操作等多数指令集选择增加的指令
- ABI
  - 更多的可用寄存器
  - 相对 PC 的重定位
- 系统态
  - 吸取实践经验，完全重新设计：规整、可扩展

LoongArch 和 MIPS 格式

- MIPS 格式

R-type	OP(6)	RS1(5)	RS2(5)	RD(5)	SA(5)	OPX(6)
I-type	OP(6)	RS(5)	RD(5)	Immediate		
J-type	OP(6)	target				

- LA 指令格式: 精打细算，预留足够的扩展空间



LoongArch 和 MIPS ABI(更多可用寄存器)

MIPS N64

R0:	永远为0
R1:	at临时寄存器
R2–R3:	v0/v1返回值
R4–R11：	参数a0–a7
R12–R15:	t0–t3临时寄存器
R16–R23:	s0–s7 callee save
R24–R25:	t8/t9
R26–R27:	k0/k1,内核用
R28:	gp,一般指向GOT表
R29:	sp,栈指针
R30:	s8/fp
R31:	ra,返回地址

LA64 ABI

R0:	永远为0
R1:	ra返回地址
R2:	tp,线程指针
R3:	sp,栈指针
R4–R11：	参数a0–a7,a0/a1返回
R12–R20:	t0–t8临时寄存器
R21:	reserve
R22:	fp
R23–R31:	s0–s8 callee save

LoongArch 和 MIPS ABI(重定位)

```c
int a; int test(void) { return a; }
```

MIPS code

```
 0: daddiu   sp,sp,-16
 4: sd   s8,8(sp)
 8: move     s8,sp
 c: lui  v1,0x0
10: daddu    v1,v1,t9
14: daddiu   v1,v1,0
18: ld   v0,0(v1)
1c: lw   v0,0(v0)
20: move     sp,s8
24: ld   s8,8(sp)
28: daddiu   sp,sp,16
2c: jr   ra
```

LA code

```
 0: addi.d   $r3,$r3,-16
 4: st.d     $r22,$r3,8
 8: addi.d   $r22,$r3,16
 c: pcaddu12i    $r12,0

10: ld.d     $r12,$r12,0
14: ldptr.w  $r12,$r12,0
18: move     $r4,$r12
1c: ld.d     $r22,$r3,8
20: addi.d   $r3,$r3,16
24: jirl     $r0,$r1,0
```

LoongArch vs. MIPS 系统资源管理

项目	MIPS	LoongArch
运行模式	user/supervisor/kernel	PLV0-PLV3 四级
计时系统	计时频率随主频变化	计时频率恒定
例外/中断	27 种，多数共用入口	25 种，独立入口
内存管理	固定分段	单一全映射 flat 空间
控制寄存器	最多 2^8 个， 无原子域修改支持， 无核外寄存器规范	最多 2^{14} 个， 支持原子域修改， 核外规范

LoongArch 和 MIPS 性能比较

- 相同的微结构
- LA 比 MIPS 快 12%
  - 指令数减少
  - ABI 优势

Benchmark	MIPS	LA	MIPS/LA
164.gzip	2063	1695	122%
175.vpr1	575	466	123%
175.vpr2	606	503	120%
176.gcc	156	162	97%
181.mcf	1474	1554	95%
186.crafty	766	761	101%
197.parser	394	354	111%
252.eon1	38	32	119%
252.eon2	54	46	117%
252.eon3	207	179	115%
253.perlbmk1	1014	923	110%
253.perlbmk2	1032	634	163%
253.perlbmk3	613	618	97%
255.vortex	416	437	95%
256.bzip2	1292	1413	91%
300.twolf	691	549	126%
average	687	625	112%

动态指令数对比

	X86_64	MIPS	LoongArch	LA/X86	LA/MIPS
动态指令数	98,974,167,562	116,349,250,572	96,022,813,068	97%	83%

Coremark 1.0, 300000 iterations

程序执行时间 = 执行的动态指令数 * 平均每条指令的执行周期数 * 每周期的时间

软件和指令集的迭代优化：案例 1

- 案例 1：OpenJDK 优化
  - 解释器 (235 条字节码)：
    - 优化前：51 个 LA 翻译用了比 MIPS 多的指令，95 个相等，89 个更少，总体略有优势
    - 优化后：没有一个比 MIPS 多用指令，196 条用更少，总体提升近 20%
    - 方法：优化立即数生成代码、采用 LA 的基址变址等指令
  - 翻译器
    - 充分利用 LA 更大的跳转范围和相对 PC 跳转等优势
    - 定位缺乏条件传送指令导致个别基准程序性能大幅低于 MIPS 问题并找出解法

软件和指令集的迭代优化: 案例 2

- Unixbench 优化
  - 通过对比分析定位了数十项性能优化点，性能超越竞争者
    - dhrystone 指令数显著多于 X86/Arm => 全局变量的访问模式 (ABI) 优化
    - clear_page/memcpy 等优化 => 借鉴 Arm64 等的 cacheline 清零支持等
    - 原子操作效率, percpu 变量的原子访问等等 => 更丰富的原子操作支持
  - 提出了多个后续指令集优化建议

对二进制翻译的支持

- 指令融合技术
  - X86/Arm eflags
  - 浮点栈访问模式
  - RISC-V 同步指令
  - **并非简单的并集!**
- 高效二进制翻译过程
  - 地址翻译加速
  - 便签寄存器、专用插桩转移指令等加速翻译中边角情况处理
  - 间接跳转加速（3A6000 实现）等等
- 目标：“十九八七”
  - MIPS/Linux 应用, Android/Arm 应用, X86/Linux 应用, X86/windows 系统
    - 翻译性能为原生的 100%、90%、80% 和 70% 以上

二进制翻译加速案例

0		SUB	ECX	EDX		
1		JE	X86_target			

- EFLAGS 翻译加速
  - X86/Arm 运算指令同时产生多个标志位
  - RISC 需要 40 多条指令完全模拟
  - 定义一条指令产生标志
    - RISC 风格

0.00		SUBU	Result	Recx	Redx	
0.01		SRL	Rsf	Result	31	/*SF=Result[31]*/
0.02		BEQ	Result	R0	L1	
0.03		ADD	Rzf	R0	R0	/*ZF=0*/
0.04		B	L2			
0.05		NOP				
0.06	L1:	ADDI	Rzf	R0	1	/*ZF=1*/
0.07	L2:	SRL	Rtmp1	Result	31	
0.08		SRL	Rtmp2	Redx	31	
0.09		SRL	Rtmp3	Recx	31	
0.10		BEQ	Rtmp1	Rtmp3	L3	
0.11		NOP				
0.12		BEQ	Rtmp2	Rtmp3	L3	
0.13		NOP				
0.14		ADDI	Rof	R0	1	/*OF=1*/
0.15		B	L4			
0.16		NOP				
0.17	L3:	ADD	Rof	R0	R0	/*OF=0*/
0.18	L4:	SRL	Rhigh2	Recx	16	
0.19		SRL	Rhigh1	Redx	16	
0.20		SUBU	Rtmp	Rhigh2	Rhigh1	
0.21		BEQ	Rtmp	R0	L5	
0.22		NOP				
0.23		BLTZ	Rtmp	L7		
0.24		NOP				
0.25		B	L6			
0.26		NOP				
0.27	L5:	SLL	Rlow2	Recx	16	
0.28		SRL	Rlow2	Rlow2	16	
0.29		SLL	Rlow1	Redx	16	
0.30		SRL	Rlow1	Rlow1	16	
0.31		SUBU	Rtmp	Rlow2	Rlow1	
0.32		BLTZ	Rtmp	L7		
0.33		NOP				
0.34	L6:	ADD	Rcf	R0	R0	/*CF=0*/
0.35		B	L8			
0.36		NOP				
0.37	L7:	ADDI	Rcf	R0	1	/*CF=1*/
0.38	L8	ADD	Recx	Result	R0	
1.00		BNE	Rzf	R0	MIPS_target	
1.01		NOP				

0.0		SUB	Result	Recx	Redx	/*Generating Sub result*/
0.1		X86SUB	Reflag	Recx	Redx	/*Generating EFLAGS*/
1.0		X86JE	Reflag	MIPS_target		/*Branch on EFLAGS*/

进展情况

- 2019 年之前 loongisa 扩展 MIPS，2019 年转向自主指令集
- 2020 年
  - 完成 3A5000 流片
  - 基本完成 Loongnix 移植
    - GCC/LLVM/OpenJDK/Golang/Javascript...
    - 2 万 + 软件包
  - 二进制翻译原型开发
    - MIPS2LA
- 2021 年
  - 深度磨合优化
  - 商业软件移植
  - 指令系统知识产权分析
  - 即将发布指令集、6 月 5000 系列芯片发布

下一步工作

- Upstream 开源软件
  - 向各类软件上游社区推送 LA 架构支持
- 指令集持续完善
  - 软硬件磨合优化
  - 二进制翻译效率提升
    - 消灭指令集!
  - 新指令集特性开发
    - 32 位子集和更多扩展等

融合型体系结构

- 指令融合
  - 提取主流指令集的主要特征，实现高效的 '并集'
- 加速器融合
  - 各类领域专用处理与 CPU 的融合
- 软硬件跨层融合
  - 打破传统软硬界面寻求更高效率

小结

- 指令集设计
  - 根据场景和需求定位来权衡
  - 可编程性、可实现性和兼容性
- 龙芯指令集
  - 面向通用 CPU 发展需求
    - 先进、兼容、可扩展
  - 持续迭代优化
- 欢迎大家参与 LA 生态建设！


Thank
You!