# 龙芯汇编自动生成

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年11月21日


# 龙芯架构介绍

# 龙芯架构简述

- **芯片产业作为全球最重要、最具战略价值的产业之一**

- **WSTS数据显示，在AI、HBM（高带宽内存）等需求带动下，2024年，全球芯片收入规模达6268亿美元，同比增长19%。**其中，中国芯片行业销售收入1.43万亿元人民币，同比增长16.7%；产量达4514亿块，同比增长22.2%。

# 龙芯架构简述

- 目前市场上的主流指令集由国外公司掌控，使用需授权许可，难以支撑自主的信息技术体系

- 2018年至2020年，龙芯中科基于MIPS指令系统，扩展数百条自定义指令，推出MIPS兼容指令系统LoongISA，并在多款CPU中应用落地

- 2020年，龙芯中科发布了完全自主设计的LoongArch指令系统，包含2000余条指令，涵盖向量计算、虚拟化、二进制翻译等扩展功能，彻底摆脱了对MIPS的依赖。2021年，龙芯中科推出搭载LoongArch的3A5000处理器，使龙芯处理器的性能接近主流产品

# 龙芯架构简述

LOONGSON 龙芯

龙芯来自中国科学院，为国产研制最早、科研实力最强的一支自主CPU研制队伍

**第一枚国产CPU**

龙芯1号是我国第一枚国产CPU。承载着二十余年的研发与产业化积累，龙芯中科于2022年6月于科创板上市，目前在党政军及各关键信息基础设施行业得到广泛应用。

**第三套生态体系**

龙芯致力于打造独立于X86与ARM架构的第三套生态体系，并得到得到国际社区广泛支持，LoongArch龙架构正在成为与X86、ARM并列的国际顶层开源生态系统。

龙芯在党的二十大报告中提及的十大科技成果中的七个应用中发挥了重要作用

# 龙芯发展历程

## 2001-2009年
### 技术积累、产业探索

- 2001年开始研发龙芯，得到中科院、863、核高基等大力支持
- 完成了底层核心技术的十年积累

## 2010-2020年
### 成为国内行业型CPU企业

- 2010年龙芯公司成立，产业化加速
- 产品性能大幅提升，在安全、嵌入式、党政领域得到广泛应用

## 2021—2030年
### 成为支撑产业型的CPU企业

- 发展进入快车道，自主生态体系加速完善
- 逐步走向开放市场、国际市场

**2001年** 中科院计算所龙芯课题组成立

**2002年** 我国首款通用CPU龙芯1号流片成功

**2006年** 我国首款主频超过1GHz龙芯2E流片成功

**2009年** 我国首款四核CPU龙芯3A流片成功

**2010年** 龙芯中科市场化，中科院和北京市政府共同出资

**2017年** 龙芯3A/B3000量产，在信创应用30万颗

**2019年** 龙芯3A4000量产，在信创市场累计应用超百万颗

**2020年** 龙芯公司完成股份制改造，销售收入过10亿元

**2021年** 龙芯推出完全自主的指令集系统LoongArch和3A5000 CPU

**2022年** 龙芯推出3C5000，性能逼近国际主流

**2023年** 龙芯推出3A6000，性能国内领先

坚持自主创新，坚持生态建设，坚持开放道路

# 龙芯技术路线

龙芯中科是国内唯一坚持基于自主指令系统构建

独立于Wintel和Arm+Android体系的信息技术体系和产业生态的CPU企业

1 基于自主IP核的芯片研发

与购买商业IP不同，龙芯中科坚持自主研发核心IP，包括CPU核、GPU核、内存控制器及PHY、高速总线控制器及PHY等上百种IP核。

2 基于自主工艺的芯片生产

与多数CPU企业依靠先进工艺提升性能不同，龙芯通过设计优化提升性能，摆脱对最先进工艺的依赖。

3 基于自主指令系统的软件生态

与融入国外指令系统不同，龙芯推出了自主指令系统LoongArch，掌握了基础软件核心能力，包括操作系统的核心模块、三大编译器（GCC、LLVM、GoLang）、三大虚拟机（Java、 JavaScript、.NET）、浏览器、图形系统等等。

龙芯基本不受国外制裁影响，自主研发、产品供货非常稳定

# 龙芯三大产品系列与路线图

## 龙芯3号芯片

信息化类：
政企、金融、教育等通用信息化
及工控类

LS3A1000

1GHz
4 GS464
（已停产）

LS3B1500

1.2GHz
8 GS464
（已停产）

LS3A/3B2000

1.0GHz
4 GS464E
（已停产）

LS3A/3B4000

2.0GHz
4 GS464V

LS3A/3B3000

1.5GHz
4 GS464E

LS7A1000

配套桥片

LS3A/3B5000

2.5GHz
4 LA464

LS7A2000

配套独显桥片

LS3C5000

2.2GHz
16 LA464

LS3D5000

2.2GHz
32 LA464

LS3A6000

2.5GHz
4 LA664

LS3C6000


## 龙芯2号芯片

工控类：
网络安全、电力控制、
轨道交通等

LS2F0800

800MHz
1 GS464
（已停产）

LS2H1000

1GHz
1 GS464
（已停产）

LS2k1000

1.0GHz
2 GS264

LS2K0500

500MHz
1 LA264

LS2K1500

1.0GHz
2 LA264

LS2K1000LA

1.0GHz
2 LA264

LS2K2000

1.5GHz
2 LA364

LS2K3000


LS2P0500


LS2K0300


## 龙芯1号芯片

工控类：
石油、电信、智能门锁、
水表、电表等工业互联网
等

LS1A0300

（已停产）

LS1B0200


LS1C0300


LS1D MCU


LS1H MCU

（已停产）

LS1C101


LS1C102


LS1C103


1号系列

2015 2020 2023

# 龙芯CPU大事记

- 2001年8月19日，龙芯1号FPGA验证系统研制成功
- 2002年8月10日，龙芯1号（代号XIA50）流片成功
  - 180nm工艺、8KICache+8KDCache、最高主频266MHz、单发射动态流水、7级流水、32位mips2 ISA
- 2003年10月17日，龙芯2B（代号MZD110）流片成功
  - 180nm工艺、32KL1I/D+片外L2、最高主频300MHz、**4发射超标量**、7-10级流水、**64位**MIPS3指令集、2定点2浮点1访存、性能比1号提升3-5倍
- 2004年9月28日，龙芯2C（代号DXP100）流片成功
  - **130nm**工艺，2B的升级版本，集成**片内L2缓存**，主频提升到500MHz，性能比2B提升3倍
- 2006年3月18日，龙芯2E（代号CZ70）流片成功
  - **90nm**工艺，主频800M-1GHz，优化**访存带宽**，性能再次提升3倍；使用FPGA桥片
- 2007年7月31日，龙芯2F（代号PLA80）流片成功
  - 90nm工艺、主频800M-1GHz、第一款**成功商业**芯片

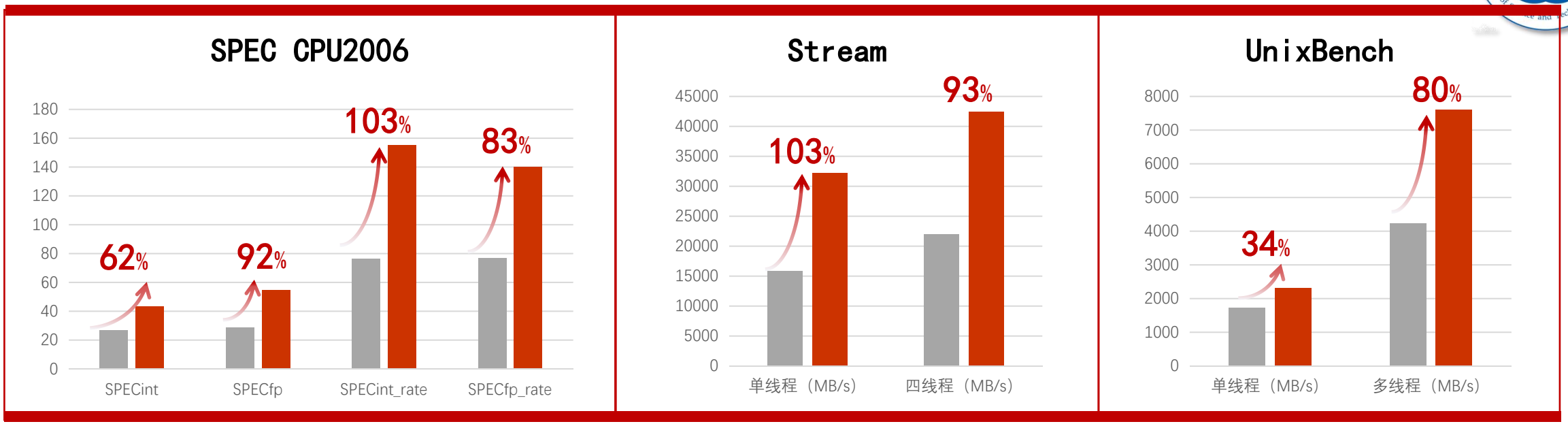
# 龙芯CPU大事记

- 2009年9月28日，龙芯3A1000（代号PRC60）流片成功
  - 65nm工艺，4M L2，**4xGS464核**，800M-1GHz，9级流水，MIPS64r1 + LoongISA扩展，集成DDR2/3-800控制器和HT1.0控制器 =》高度复杂的多核设计导致多次改版，2012年才正式商用
- 2012年4月，龙芯3B1000流片成功
  - 65nm工艺，**8x**GS464V核，800M-1GHz，向量和专用加速 =》高峰值性能(128GFlops)
- 2012年10月，龙芯3B1500流片成功
  - **32nm**工艺，8xGS464V核，64KL1 + **128K L2** + 8MB L3，1.5GHz，192GFLops，HT2.0，DDR2/3 800
- 2015年8月18日，龙芯3A2000/3B2000发布
  - 40nm工艺，4xGS464E核，1GHz，64K L1 + 256KB L2+4M L3，12级流水,HT3.0, DDR2/3-1333
  - 大幅度设计优化(分支预测、访存层次)，同频SPEC CPU性能为3A1000 2-3倍，Stream带宽提升20倍
- 2017年4月16日，龙芯3A3000发布
  - 28nm工艺，4xGS464E核，1.2-1.5GHz，8M L3，浮点定点发射队列从16/24提升到32/32

# 龙芯CPU大事记

- 2019年12月24日，龙芯3A4000发布
  - 28nm工艺，4xGS464EV核，1.8-2.0GHz，8M L3，DDR4-2400，支持虚拟化，内置安全处理
  - 同主频流水线效率比3A3000提升50%（**分支预测等**）；整体SPEC性能提升1倍
- 2021年7月23日，龙芯3A5000发布
  - **12nm工艺，2.5GHz主频，DDR4-3200，LoongArch指令集**
  - SPEC CPU性能较3A4000提升50%以上，功耗降低30%以上
- 2022年6月6日，龙芯3C5000发布
  - **16核**服务器芯片
- 2023年4月8日，龙芯3D5000发布
  - **32核**
- 2023年11月28日，龙芯3A6000发布
  - 4x**LS664核（2xSMT），预取**改进，单核比3A5000性能提升超过60%，多核超过80%

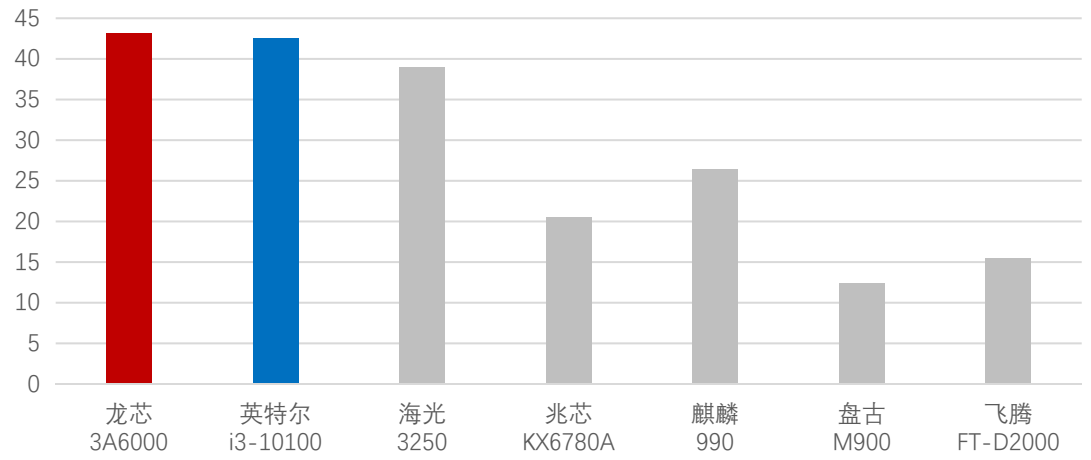
# 3A5000与3A6000

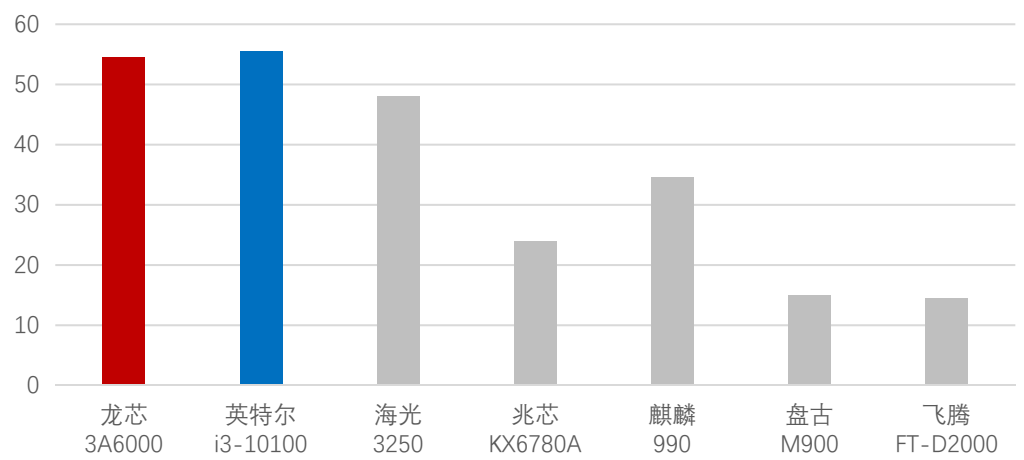



| CPU型号 | SPEC CPU2006 （base） | | | | Stream （Copy） | | UnixBench | |
|---|---|---|---|---|---|---|---|---|
| | SPECint | SPECfp | SPECint_rate | SPECfp_rate | 单线程（MB/s） | 四线程（MB/s） | 单线程 | 多线程 |
| 3A5000 | 26.6 | 28.5 | 76.4 | 76.7 | 15841.7 | 21968.8 | 1718.4 | 4223.5 |
| 3A6000 | 43.1 | 54.6 | 155 | 140 | 32210.8 | 42467.9 | 2302.8 | 7608.4 |

注：以上3A5000、3A6000测试数据来自工信部电子四院测试报告

# 3A6000 ——国际主流、国内领先



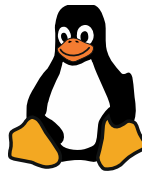

| CPU型号 | 指令架构 | 典型频率（GHz） | 物理核数 | 逻辑核数（线程数） | SPEC CPU整型单核性能 | SPEC CPU 浮点单核性能 |
|---|---|---|---|---|---|---|
| 龙芯3A6000 | LoongArch | 2.5 | 4 | 8 | 43.1 | 54.6 |
| 英特尔i3-10100 | X86 | 3.6 | 4 | 8 | 42.5 | 55.6 |
| 海光-3250 | X86 | 2.8 | 8 | 8 | 39 | 48 |
| 兆芯-KX6780A | X86 | 2.7 | 8 | 8 | 20.5 | 24 |
| 麒麟990 | ARM | 2.8 | 8 | 8 | 26.4 | 34.7 |
| 盘古M900 | ARM | 2.0 | 8 | 8 | 12.4 | 15.0 |
| 飞腾FT-D2000 | ARM | 2.3 | 8 | 8 | 15.4 | 14.5 |

龙芯3A6000性能超越国内其他所有桌面处理器性能，与国际上2020年Intel酷睿十代处理器性能相当

# 龙芯生态建设

目标愿景：LoongArch成为与X86/ARM并列的顶层开源生态架构

基于LoongArch建立起完整的开源软件技术体系，支撑操作系统和应用生态发展

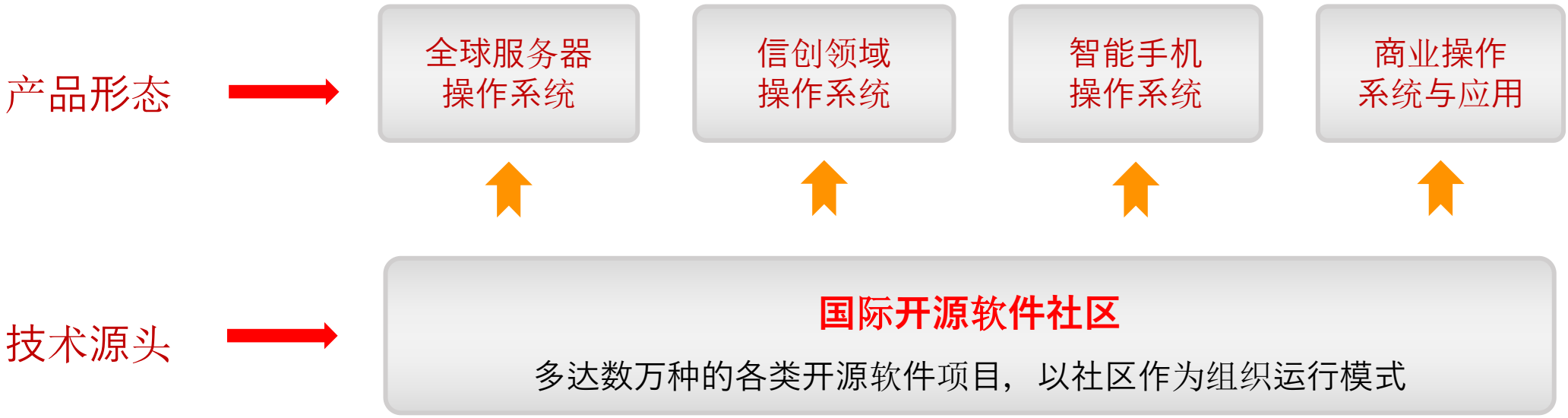



| X86 | ARM | LoongArch |
| --- | --- | --- |
| 商业指令集 | 商业指令集 | 国产指令集 |
| 一般不对外授权 | 指令集/IP授权 | 融入国际开源体系 |

# 龙芯生态建设

操作系统是产品形态，技术源头在国际开源软件社区，开源软件是IT产业之基、技术之根

- Linux is everywhere，全球服务器操作系统和云计算基础设施几乎全部采用Linux操作系统
- 信创领域桌面操作系统产品是开源操作系统的发行版
- 智能手机领域的安卓操作系统基于Linux内核等开源组件发展
- 商业闭源的桌面操作系统和应用也大量采用开源软件与运用开源技术




# 龙芯汇编介绍

# 龙芯架构简述

- 国产芯片的加速崛起也带动了高效编译技术的迫切需求

- 近年来，国产编译技术取得了显著进展，自主研发的编译工具和框架不断涌现，基于LLVM的编译技术体系在本地指令集适配、数据流分析优化和代码生成上有了深度提升

- 与国际先进水平相比，仍在复杂优化策略和跨语言兼容性方面存在差距，亟需在开拓技术边界和提升工具链完备性上不断创新，建立更具竞争力的国产编译生态

# 龙芯架构简述

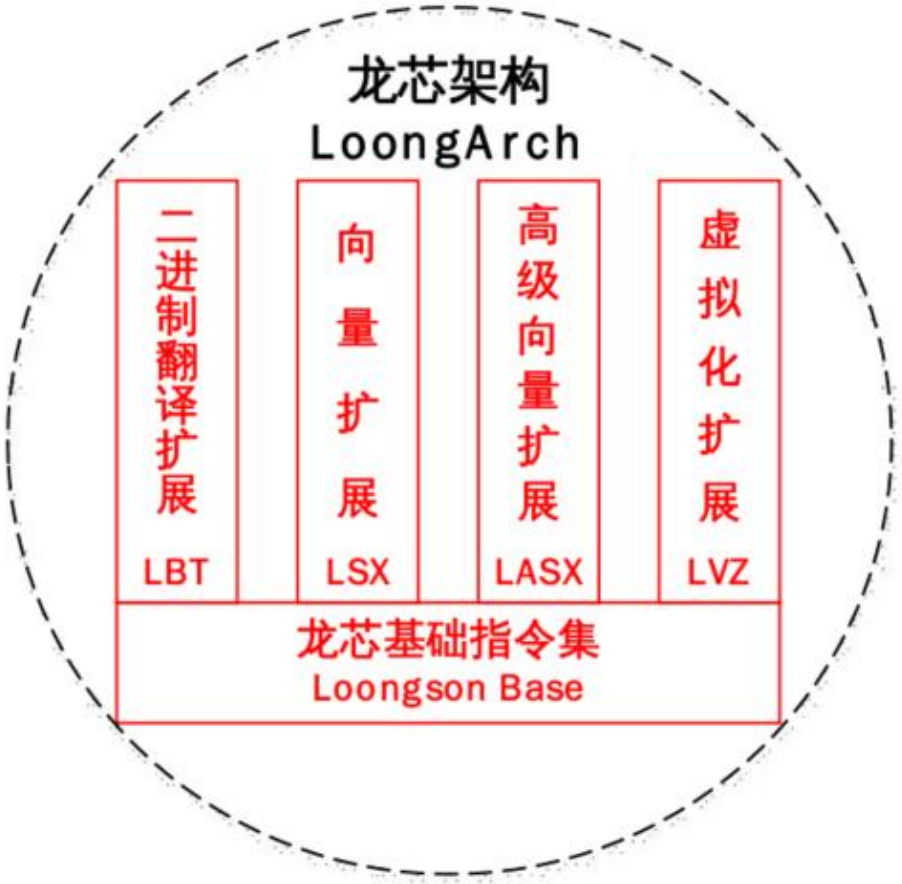
**龙芯架构 LoongArch（简称 LA）**

- 由龙芯中科技术股份有限公司研发，具有自主产权的国产指令集架构
- 精简指令集计算机（RISC）风格
- 分为 32 位和 64 位两个版本
  - 分别称为 LA32 架构和 LA64 架构
  - 本实验基于 LA64 架构
- 采用基础部分 + 拓展部分的组织形式

# 龙芯架构简述

## 龙芯架构 LoongArch（简称 LA）

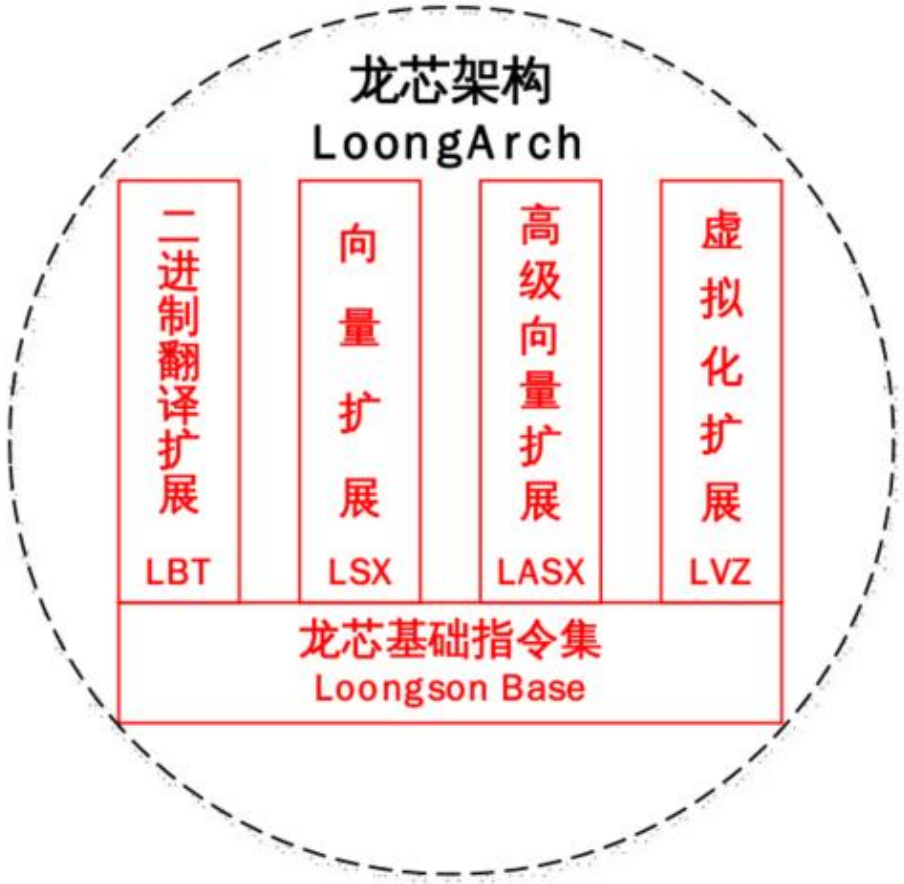
- 采用基础部分 + 拓展部分的组织形式
- 龙芯架构的基础部分包含非特权指令集和特权指令集两个部分
- 非特权指令集部分定义了常用的整数和浮点数指令，能够充分支持现有各主流编译系统生成高效的目标代码



# 龙芯架构简述

## 龙芯架构 LoongArch（简称 LA）

| 类别 | | 指令数 |
|---|---|---|
| 基础指令 | | 337 |
| 扩展指令 | 二进制翻译扩展 | 176 |
| | 虚拟化扩展 | 10 |
| | 向量扩展 | 1024 |
| | 高级向量扩展 | 1018 |
| 总计 | | 2565 |



# 寄存器介绍


**LoongArch 基础部分中的寄存器**

- 程序计数器 PC
- 32 个通用寄存器 GR
- 32 个浮点寄存器 FR
- 8 个浮点条件标志寄存器 FCR
- 一系列控制状态寄存器 CSR
- 4 个浮点控制状态寄存器 FCSR

# 寄存器介绍

**本实验编译器后端会使用到的寄存器**

- 程序计数器 PC
- 32 个通用寄存器 GR
- 32 个浮点寄存器 FR
- 8 个条件标志寄存器 CFR

**本实验编译器后端不会使用的寄存器**

- 一系列控制状态寄存器 CSR
- 4 个浮点控制状态寄存器 FCSR

# 寄存器介绍


**在龙芯架构的汇编代码中，寄存器用“$+寄存器名”标识，比如**

- 通用寄存器 $r0 $sp $zero
- 浮点寄存器 $f0 $ft0 $fa0
- 条件标志寄存器 $fcc0

# 寄存器介绍：程序计数器与通用寄存器


## 32 个通用寄存器 GR

- 记为 $r0 ~ $r31
- 零号寄存器 $r0 的值恒为 0
- 通用寄存器的位宽与架构的位宽一致
  - 位宽在 LA32 下为 32 比特，在 LA64 下为 64 比特
- 用于整数数据和指针的操作

## 程序计数器 PC

- 记录着当前指令的地址，不能被直接修改
- 位宽与架构的位宽一致

# 寄存器介绍：通用寄存器的使用规范

## 通用寄存器的使用应该遵循下面的规定

| 通用寄存器 | 别名 | 用途 | 在调用中是否保留 |
|---|---|---|---|
| $r0 | $zero | 常数 0 | （常数） |
| $r1 | $ra | 返回地址 | 否 |
| $r2 | $tp | 线程指针 | （不可分配） |
| $r3 | $sp | 栈指针 | 是 |
| $r4 - $r5 | $a0 - $a1 | 传参寄存器、返回值寄存器 | 否 |
| $r6 - $r11 | $a2 - $a7 | 传参寄存器 | 否 |
| $r12 - $r20 | $t0 - $t8 | 临时寄存器 | 否 |
| $r21 | | 保留寄存器 | （不可分配） |
| $r22 | $fp / $s9 | 栈帧指针、静态寄存器 | 是 |
| $r23 - $r31 | $s0 - $s8 | 静态寄存器 | 是 |

- 本实验编译器后端不会用到 $r2、$r21 和 $r23 ~ $r31

# 寄存器介绍：通用寄存器的使用规范

## 通用寄存器的使用应该遵循下面的规定

| 通用寄存器 | 别名 | 用途 | 在调用中是否保留 |
|---|---|---|---|
| $r0 | $zero | 常数 0 | （常数） |
| $r1 | $ra | 返回地址 | 否 |
| $r3 | $sp | 栈指针 | 是 |
| $r4 - $r5 | $a0 - $a1 | 传参寄存器、返回值寄存器 | 否 |
| $r6 - $r11 | $a2 - $a7 | 传参寄存器 | 否 |
| $r12 - $r20 | $t0 - $t8 | 临时寄存器 | 否 |
| $r22 | $fp / $s9 | 栈帧指针、静态寄存器 | 是 |

- 在汇编代码中，可以直接使用寄存器名，也可以使用寄存器的别名
- 若“在调用中是否保留”这一栏为“是”，则在函数调用过程中，被调用函数在返回时需要恢复寄存器原来的值

# 寄存器介绍：浮点相关寄存器


**32 个浮点寄存器 FR**

- 记为 $f0 ~ $f31
- 不论什么架构，位宽均为 64 比特
- 用于单/双精度浮点数的操作

**8 个条件标志寄存器 CFR**

- 记为 $fcc0 ~ $fcc7
- 位宽为 1 比特

# 寄存器介绍：条件标志寄存器

## 8 个条件标志寄存器 CFR

- 用于存储浮点比较指令的结果
  - 比较结果为真时值为 1，否则为 0
  - *fcmp.slt.s $fcc0, $f0, $f1*
    - 将 *$f0* 和 *$f1* 中的单精度浮点数进行比较，
      如果前者小于后者则置 *$fcc0* 为 1，否则置 *$fcc0* 为 0
- 浮点分支指令的判断条件源于 CFR
  - *bcnez $fcc0, label_true*
    - 若 *$fcc0* 不为 0，则跳转到 *label_true*，否则不跳转

# 寄存器介绍：浮点寄存器的使用规范

## 浮点寄存器的使用应该遵循下面的规定

| 通用寄存器 | 别名 | 用途 | 在调用中是否保留 |
|---|---|---|---|
| $f0 - $f1 | $fa0 - $fa1 | 传参寄存器、返回值寄存器 | 否 |
| $f2 - $f7 | $fa2 - $fa7 | 传参寄存器 | 否 |
| $f8 - $f23 | $ft0 - $ft15 | 临时寄存器 | 否 |
| $f24 - $f31 | $fs0 - $fs7 | 静态寄存器 | 是 |

- 本实验编译器后端不会用到 $f24 - $f31
- 在汇编代码中，可以直接使用寄存器名，也可以使用寄存器的别名
- 若“在调用中是否保留”这一栏为“是”，则在函数调用过程中，被调用函数在返回时需要恢复寄存器原来的值

# 指令介绍：指令编码格式


## 指令编码格式

- 所有指令均采用 32 位固定长度
- 指令的地址都要求 4 字节边界对齐
  - 指令地址不对齐时将触发地址错例外

# 指令介绍：指令助记格式

**为了方便汇编编程人员和编译器开发人员，龙芯架构对指令名的前、后缀进行了统一考虑**

- 前缀：用于指示指令类型
  - 整数指令：无前缀，如 add.w ld.d lu12i.w
  - 浮点指令：前缀为 f，如 fadd.s fst.d

# 指令介绍：指令助记格式

**为了方便汇编编程人员和编译器开发人员，龙芯架构对指令名的前、后缀进行了统一考虑**

- 前缀：用于指示指令类型
  - 整数指令：无前缀，如 *add.w ld.d lu12i.w*
  - 浮点指令：前缀为 *f*，如 *fadd.s fst.d*
- 后缀：用于指示指令的操作对象类型

| 后缀 | 整数指令 | | | | 浮点指令 | | | |
|---|---|---|---|---|---|---|---|---|
| | *.b* | *.h* | *.w* | *.d* | *.s* | *.d* | *.w* | *.l* |
| 操作对象类型 | 字节 | 半字 | 字 | 双字 | 单精度浮点数 | 双精度浮点数 | 字 | 双字 |
| 位宽 | 1 byte | 2 bytes | 4 bytes | 8 bytes | 4 bytes | 8 bytes | 4 bytes | 8 bytes |

# 指令介绍：整数指令

- **二元运算指令**
  - **无立即数**
    - 算术指令：add/sub/mul/div（后缀 .w/.d）
    - 比较指令：slt/sltu
    - 位运算指令：and/or/nor/xor/andn/orn
  - **有立即数**
    - 算术指令：addi（后缀 .w/.d，12 位有符号立即数）
    - 比较指令：slti/sltui（12 位有符号立即数）
    - 位运算指令：andi/ori/xori（12 位无符号立即数）
  - **示例**
    - 无立即数：`add.w $t2, $t0, $t1  # $t2 = $t0 + $t1`
    - 有立即数：`addi.w $t0, $zero, -12 # $t0 = -12`

# 指令介绍：整数指令

- **立即数加载指令**
  - *lu12i.w*：用于设置寄存器的 [31:12] 位
  - *lu32i.d*：用于设置寄存器的 [51:32] 位
  - *lu52i.d*：用于设置寄存器的 [63:52] 位
  - **示例**：加载不同长度的立即数
    - 加载长度小于等于 12 位的立即数
      - 使用 *ori* 加载 12 位无符号立即数（0 ~ 4095）
        将 2255 写入 $r0：

```
ori $r2, $zero, 2255          # $r2 = 0x0000_0000_0000_08CF
```

      - 使用 *addi* 加载 12 位有符号立即数（-2048 ~ 2047）
        将 -1841 写入 $r0：

```
addi.w $r2, $zero, -1841      # $r2 = 0xFFFF_FFFF_FFFF_F8CF
```

# 指令介绍：整数指令

- **立即数加载指令**
  - *lu12i.w*：用于设置寄存器的 [31:12] 位
  - *lu32i.d*：用于设置寄存器的 [51:32] 位
  - *lu52i.d*：用于设置寄存器的 [63:52] 位
  - **示例**：加载不同长度的立即数
    - 加载长度大于 12 位，小于等于 32 位的有符号立即数（$-2^{31}$ ~ $2^{31}-1$）
      - 使用 *ori* 和 *lu12i.w*
        将 *0xFFFF_FFFF_8765_4321* 写入 *$t0*：

```
lu12i.w $t0, -0x789AC          # $t0 = 0xFFFF_FFFF_8765_4000
ori     $t0, $t0, 0x321 # $t0 = 0xFFFF_FFFF_8765_4321
```

        - 因为 *lu12i.w* 指令中的立即数是 20 位有符号立即数，
          所以汇编代码中使用 *-0x789AC* 而不是 *0x87654*

# 指令介绍：整数指令

- **立即数加载指令**
  - *lu12i.w*：用于设置寄存器的 [31:12] 位
  - *lu32i.d*：用于设置寄存器的 [51:32] 位
  - *lu52i.d*：用于设置寄存器的 [63:52] 位
  - **示例**：加载不同长度的立即数
    - 加载长度大于 32 位，小于等于 52 位的有符号立即数（$-2^{51}$ ~ $2^{51}-1$）
      - 使用 *ori*、*lu12i.w* 和 *lu32i.d*
        将 0x0003_4567_0123_4567 写入 $t0：

```
lu12i.w $t0, 0x1234          # $t0 = 0x0000_0000_0123_4000
ori     $t0, $t0, 0x567      # $t0 = 0x0000_0000_0123_4567
lu32i.d $t0, 0x34567         # $t0 = 0x0003_4567_0123_4567
```

# 指令介绍：整数指令

- **立即数加载指令**
  - *lu12i.w*：用于设置寄存器的 [31:12] 位
  - *lu32i.d*：用于设置寄存器的 [51:32] 位
  - *lu52i.d*：用于设置寄存器的 [63:52] 位
  - **示例**：加载不同长度的立即数
    - 加载长度大于 52 位，小于等于 64 位的有符号立即数（$-2^{63}$ ~ $2^{63}-1$）
      - 使用 *ori*、*lu12i.w*、*lu32i.d* 和 *lu52i.d* 将 0x1234_5678_1234_5678 写入 $t0：

```
lu12i.w $t0, 0x12345          # $t0 = 0x0000_0000_1234_5000
ori     $t0, $t0, 0x678       # $t0 = 0x0000_0000_1234_5678
lu32i.d $t0, 0x45678          # $t0 = 0x0004_5678_1234_5678
lu52i.d $t0, $t0, 0x123               # $t0 = 0x1234_5678_1234_5678
```

# 指令介绍：整数指令

- **分支指令**
  - **有条件跳转**
    - *beq/bne/blt/bge/bltu/bgeu*：寄存器之间比较
      - **示例**：
        - *blt $t0, $t1, label_true*

          将 *$t0* 和 *$t1* 视作有符号数进行比较，如果前者小于后者，则跳转到 *label_true*，否则不跳转
        - *bltu $t0, $t1, label_true*

          将 *$t0* 和 *$t1* 视作无符号数进行比较，如果前者小于后者，则跳转到 *label_true*，否则不跳转
    - *beqz/bnez*：寄存器与 0 比较
      - **示例**：
        - *beqz $t0, label_true*

          如果 *$t0* 中的值为 0 则跳转到 *label_true*，否则不跳转

# 指令介绍：整数指令

- **分支指令**
  - **无条件跳转**
    - *b*
      - 无条件跳转（类似于 C 语言 *goto*）
      - **示例**：*b label_xxx # 跳转到 label_xxx 处*
    - *bl*
      - 无条件跳转，同时保存 *PC+4* 的值到 *$ra*
      - 用于调用函数（类似于 X86 汇编 *call* 指令）
      - **示例**：*bl func_xxx # 跳转到 func_xxx 处，并将 PC+4 写入 $ra*
    - *jirl*：
      - 无条件跳转，目标地址由其中一个操作数寄存器加上偏移量计算得到，同时保存 *PC+4* 的值到目的寄存器
      - **示例**：*jirl $t0, $t1, 0 # 跳转到地址 $t1+0 处，并将 PC+4 写入 $t0*
      - 一般使用 *jirl $zero, $ra, 0* 来实现返回语句（类似于 X86 汇编 *ret* 指令）
        - 等价于宏指令 *jr $ra*，返回地址为 *$ra*
        - 无条件跳转回 *bl* 语句的下一条语句

# 指令介绍：整数指令

- **访存指令**
  - **读取内存**
    - *ld.b*：读取字节（8 位）
    - *ld.h*：读取半字（16 位）
    - *ld.w*：读取字（32 位）
    - *ld.d*：读取双字（64 位）
  - **写入内存**
    - *st.b*：写入字节（8 位）
    - *st.h*：写入半字（16 位）
    - *st.w*：写入字（32 位）
    - *st.d*：写入双字（64 位）

# 指令介绍：整数指令

- **访存指令**
  - **读取内存**
    - *ld.b/ld.h/ld.w/ld.d*：读取字节/半字/字/双字到寄存器
  - **写入内存**
    - *st.b/st.h/st.w/st.d*：将寄存器中的字节/半字/字/双字写入存储器
  - **示例**
    - *ld.w $t0, $fp, -24*

      从地址 ($fp - 24) 处读取一个字（32 位），符号拓展后写入 $t0 中
    - *st.b $t0, $fp, -28*

      将 $t0 的低 8 位写入地址 ($fp - 28)

# 指令介绍：整数指令

- **宏指令**
  - *la.local*
    - 将标签的地址写入寄存器
    - 常用于加载全局变量
    - **示例**：

```
label_xxx:        # 假设地址为 0x1000_0000
    .space 4 # 为 label_xxx 分配四字节空间
    # ...
    la.local $t0, label_xxx # $t0 = &label_xxx = 0x1000_0000
```

  - *jr*
    - *jr* $rj 等价于 *jirl* $zero, $rj, 0
    - *jr* $ra 常用于函数返回

# 指令介绍：浮点指令

- **浮点算术指令**
  - fadd.s：单精度浮点数加法
  - fsub.s：单精度浮点数减法
  - fmul.s：单精度浮点数乘法
  - fdiv.s：单精度浮点数除法
  - 对于单精度浮点算术指令，结果浮点寄存器的高 32 位可以是任意值
  - **示例**：

```
fadd.s $ft2, $ft0, $ft1        # $ft2 = $ft0 + $ft1
fsub.s $ft2, $ft0, $ft1        # $ft2 = $ft0 - $ft1
fmul.s $ft2, $ft0, $ft1        # $ft2 = $ft0 * $ft1
fdiv.s $ft2, $ft0, $ft1        # $ft2 = $ft0 / $ft1
```

# 指令介绍：浮点指令

- **浮点搬运指令**
  - movgr2fr.w
    - 将通用寄存器的低 32 位搬运到浮点寄存器的低 32 位
    - 浮点寄存器的 [63:32] 位值不确定
    - **示例**：$t0 中的值为 0x4108_0000，
      执行 movgr2fr.w $ft0, $t0 后，
      $ft0 的低 32 位变为 0x4108_0000 (8.5 的单精度表示)
  - movfr2gr.s
    - 将浮点寄存器的低 32 位符号扩展后搬运到通用寄存器
    - **示例**：$ft0 的低 32 位中的值为 -8.5 (0xC108_0000)，
      执行 movfr2gr.s $t0, $ft0 后，
      $t0 中的值变为 0xFFFF_FFFF_C108_0000

# 指令介绍：浮点指令

- **浮点转换指令**
  - ffint.s.w
    - 将 4 字节整数值转换为单精度浮点数
    - **示例**：$ft0 的低 32 位值为 0x0000_0008，
      执行 ffint.s.w $ft1, $ft0 后，
      $ft1 的低 32 位值为 0x4100_0000 (8.0 的单精度表示)
  - ftintrz.w.s
    - 将单精度浮点数转换为 4 字节整数值
    - 采用“向零方向舍入”作为舍入模式
    - **示例**：
      - $ft0 的低 32 位值为 8.5 (0x4108_0000)，执行 ftintrz.w.s $ft1, $ft0 后，
        $ft1 中的值为 0x0000_0000_0000_0008
      - $ft0 的低 32 位值为 -8.5 (0xC108_0000)，执行 ftintrz.w.s $ft1, $ft0 后，
        $ft1 中的值为 0x0000_0000_FFFF_FFF8 (低 32 位为 -8)

# 指令介绍：浮点指令

- **浮点转换指令**
  - **示例**：翻译类型转换语句
    - 将 $t0 中的整数转换为浮点数写入 $ft0（$ft0 = (float)$t0）

```
movgr2fr.w $ft0, $t0   # 将 $t0 中的整数搬运到 $ft0
ffint.s.w $ft0, $ft0   # 将 $ft0 中的整数转换为浮点数，写入 $ft0
```

    - 将 $ft0 中的浮点数转换为整数写入 $t0（$t0 = (int)$ft0）

```
ftintrz.w.s $ft0, $ft0 # 将 $ft0 中的浮点数转换为整数，写入 $ft0
movfr2gr.s $t0, $ft0   # 将 $ft0 中的数据（整数）搬运到 $t0
```

    - 将 $ft0 中的浮点数转换为整数写入存储器（Mem[addr] = (int)$ft0）

```
# 假设地址为 $fp - 24
ftintrz.w.s $ft0, $ft0 # 将 $ft0 中的浮点数转换为整数，写入 $ft0
fst.s $ft0, $fp, -24   # 将 $ft0 中的数据（整数）写入存储器
```

# 指令介绍：浮点指令

- **浮点比较指令**
  - fcmp.cond.s：比较单精度浮点数值，将结果写入条件标志寄存器
    - cond 可以为 seq/sne/slt/sle/...
      - 对应的指令为 fcmp.seq.s/fcmp.sne.s/...
    - **示例**：

      fcmp.slt.s $fcc0, $ft0, $ft1

      将 $f0 和 $f1 中的单精度浮点数进行比较，如果前者小于后者则置 $fcc0 为 1，否则置为 0
- **浮点分支指令**
  - bceqz/bcnez：根据条件标志寄存器的值决定是否跳转
  - **示例**：

    bceqz $fcc0, label_true

    若 $fcc0 为 0，则跳转到 label_true，否则不跳转

# 指令介绍：浮点指令

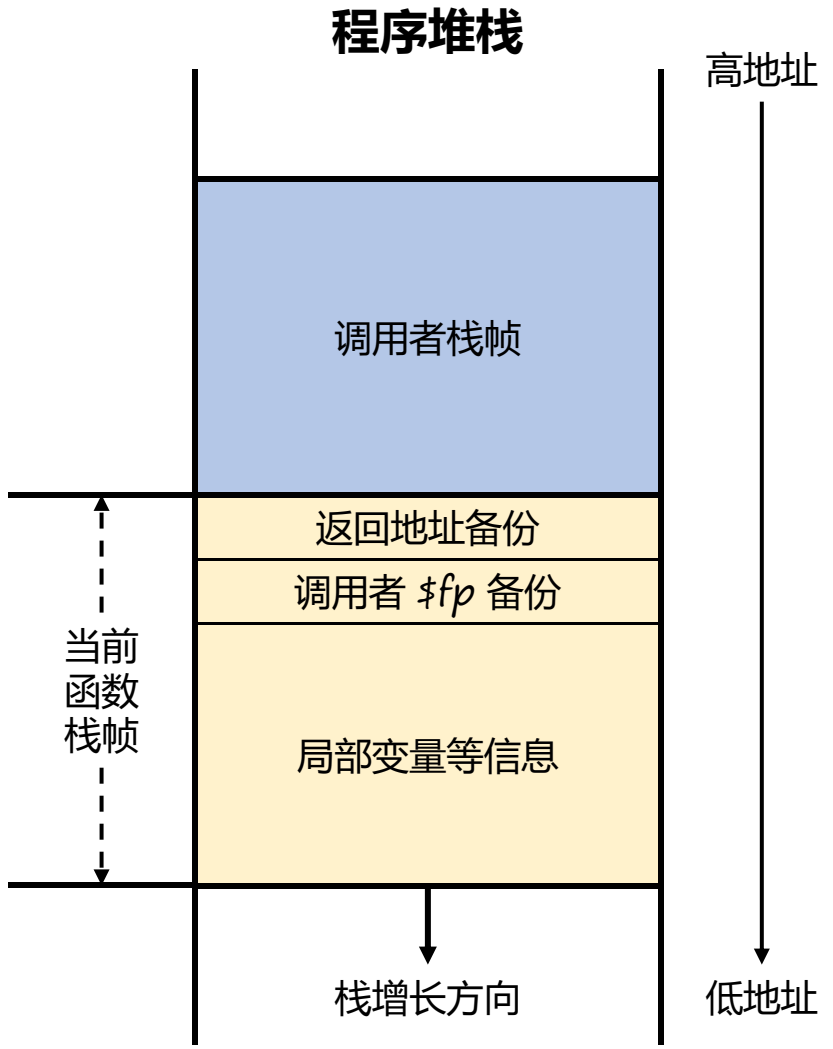
- **浮点访存指令**
  - fld.s/fld.d：读取内存中的单/双精度浮点数
  - fst.s/fst.d：将浮点寄存器中的单/双精度浮点数写入内存
  - **示例**：
    - fld.s $ft0, $fp, -24

      从地址 ($fp - 24) 处读取单精度浮点数写入 $ft0 中
    - fst.d $ft0, $fp, -28

      将 $ft0 中的双精度浮点数写入地址 ($fp - 28)

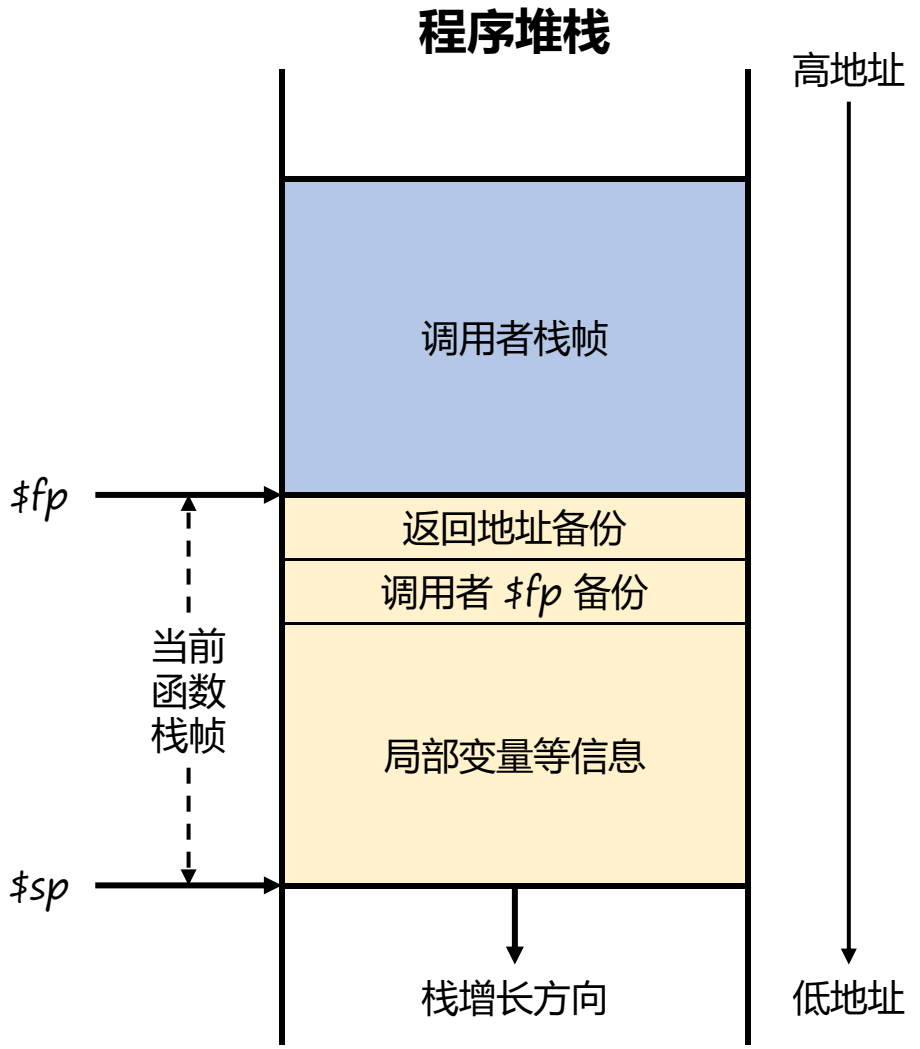
# 栈帧与函数调用

- **栈帧**
  - 每个未完成函数在堆栈中占用的一段连续区域
  - 保存了函数的上下文信息
  - 用于支持函数的调用与返回



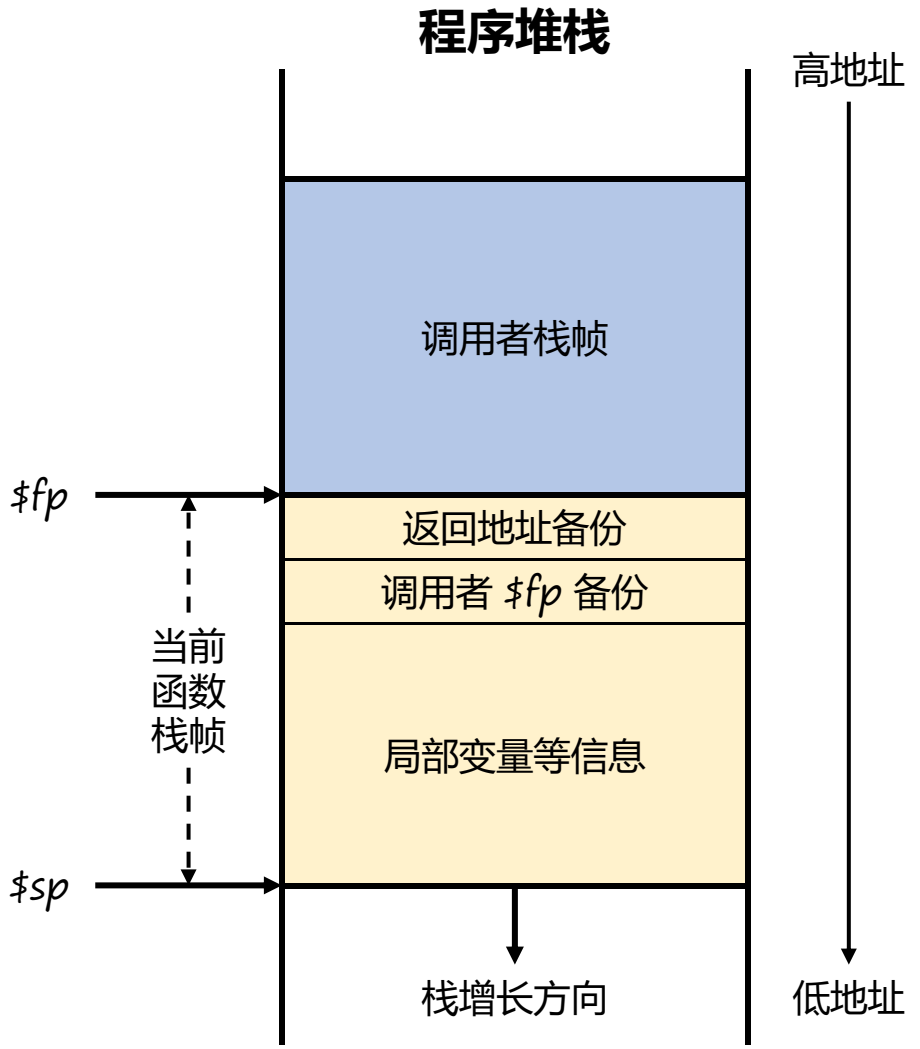
# 栈帧与函数调用：栈帧

- **栈帧**
  - 每个未完成函数在堆栈中占用的一段连续区域
  - 保存了函数的上下文信息
  - 用于支持函数的调用与返回
  - 龙芯架构 ABI 规定使用两个寄存器来访问堆栈：
    - 栈帧指针 *$fp*(*$r22*) 指向栈帧的底部
    - 堆栈指针 *$sp*(*$r3*) 指向栈帧的顶部
    - 这两个寄存器的值应该对齐到 16 字节，即龙芯架构中栈帧大小必须是 128 位的整数倍



# 栈帧与函数调用：函数调用

- **函数调用过程**
  - 函数调用的过程由调用者和被调用函数共同完成
  - 函数调用过程中使用到的寄存器：
    - $sp($r3) 和 $fp($r22)用于维护程序堆栈
    - $ra($r1) 用于记录返回地址
    - $a0 – $a7($r4 – $r11) 和 $fa0 – $fa7($f0 – $f7) 用于传递参数
    - $a0($r4)、$a1($r5)、$fa0($f0) 和 $fa1($f1) 用于传递返回值
      - 本实验编译器后端只关心 $a0($r4) 和 $fa0($f1)



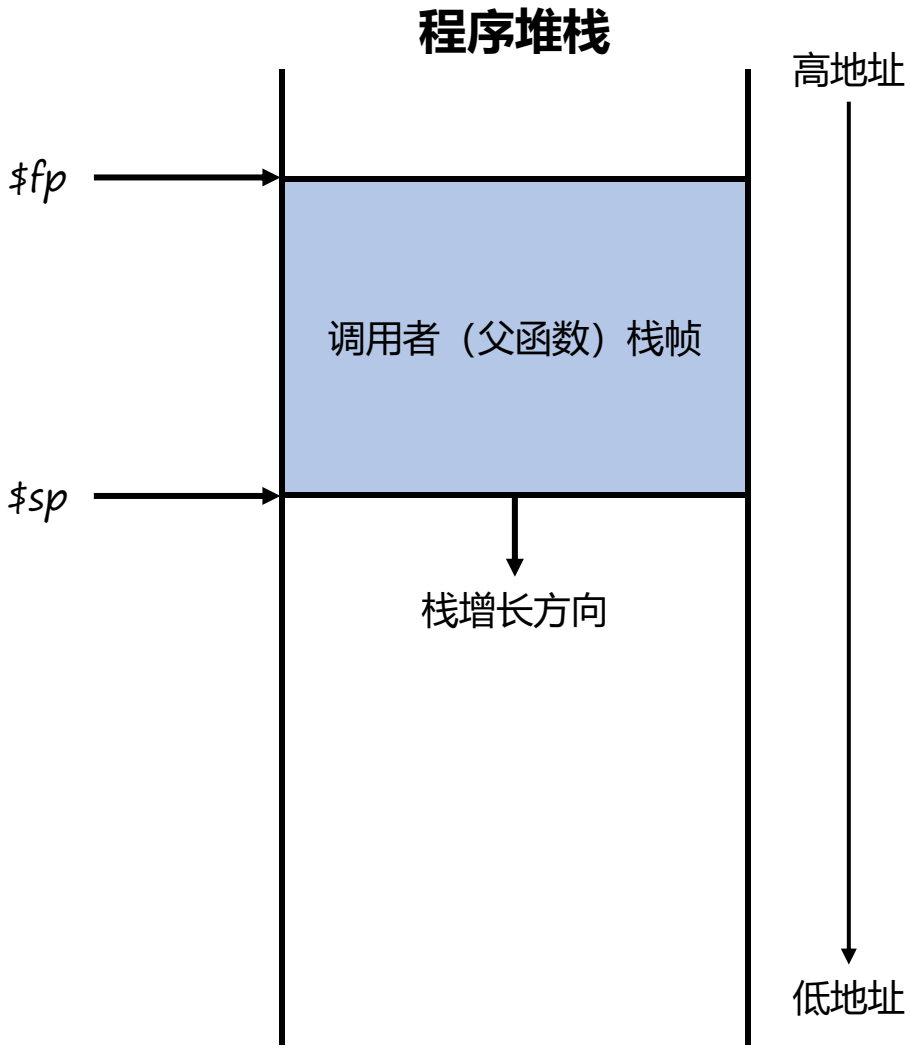
# 栈帧与函数调用：函数调用

- **示例：**父函数 caller() 调用子函数 callee()
  1. 父函数调用子函数
     1. 调用方（父函数）将参数加载到适当的寄存器和堆栈位置
     2. 执行跳转指令：`bl callee # 跳转到子函数`
  2. 子函数进入
     1. 初始化堆栈并保存上下文信息
  3. 子函数退出
     1. 将返回值放入对应寄存器（$a0/$a1/$fa0/$fa1）中
     2. 释放栈帧并恢复各个保留寄存器
     3. 返回到父函数：`jirl $zero, $ra, 0`（等价于 `jr $ra`）
  4. 父函数恢复
     1. 如果调用方在调用时为参数分配了额外的堆栈空间，则需要将这些空间释放
     2. 函数调用的返回值位于 $a0/$a1/$fa0/$fa1 中

# 栈帧与函数调用：函数调用 – 示例

## 1. 父函数调用子函数

```
caller: # 父函数
  caller_entry:
    ...
  caller_labels_before_call:
    ...
  caller_call_callee:
```

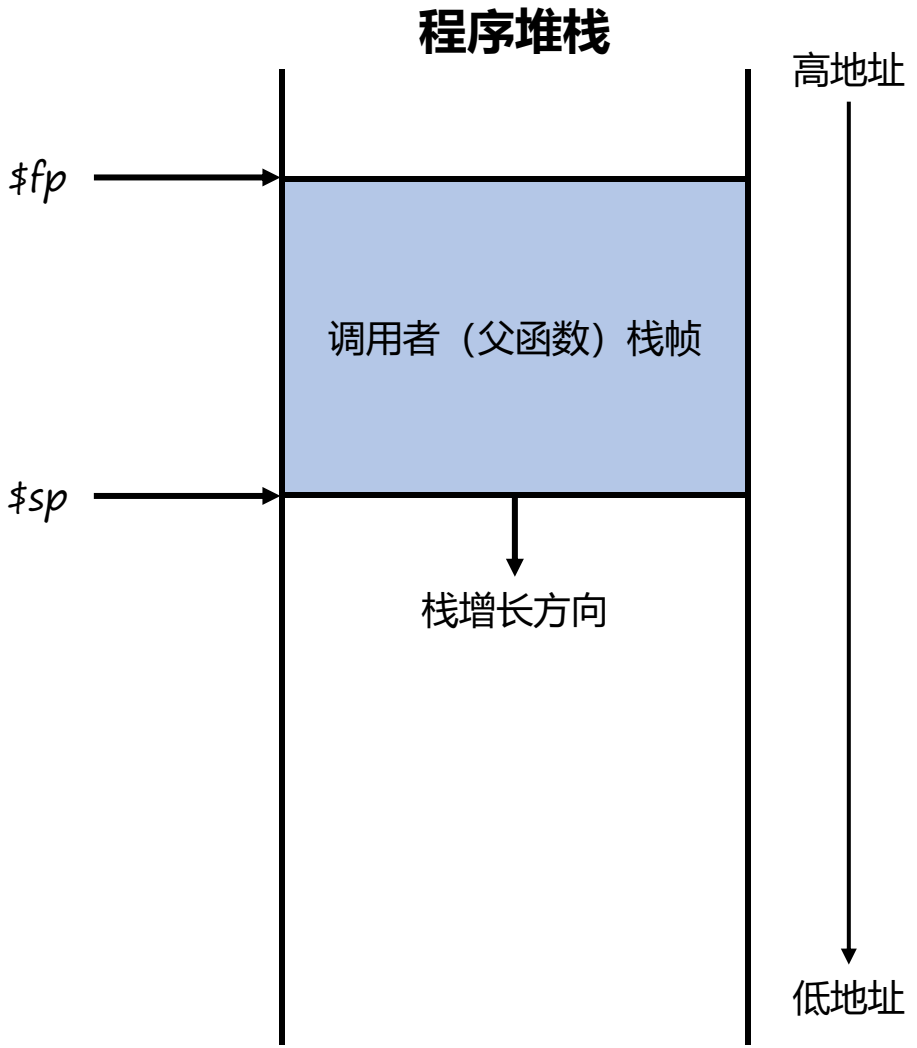


# 栈帧与函数调用：函数调用 – 示例

## 1. 父函数调用子函数

1. 调用方（父函数）将参数加载到适当的寄存器和堆栈位置

```
caller: # 父函数
  caller_entry:
    ...
  caller_labels_before_call:
    ...
  caller_call_callee:
    # 加载参数
    ld.w   $a0, $fp, -ARG1_OFFSET
    ld.w   $a1, $fp, -ARG2_OFFSET
    addi.w $a3, $zero, ARG3
```

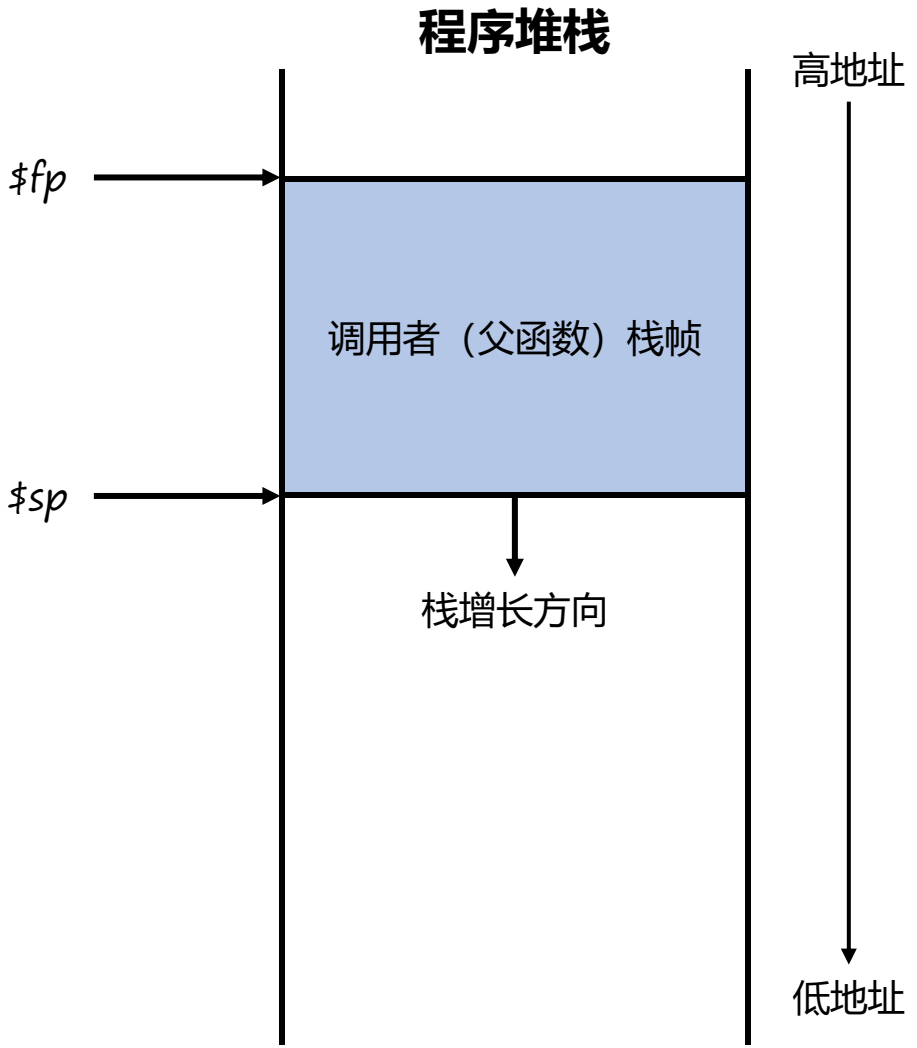


# 栈帧与函数调用：函数调用 – 示例

## 1. 父函数调用子函数

1. 调用方（父函数）将参数加载到适当的寄存器和堆栈位置
2. 执行跳转指令：*bl callee # 跳转到子函数*

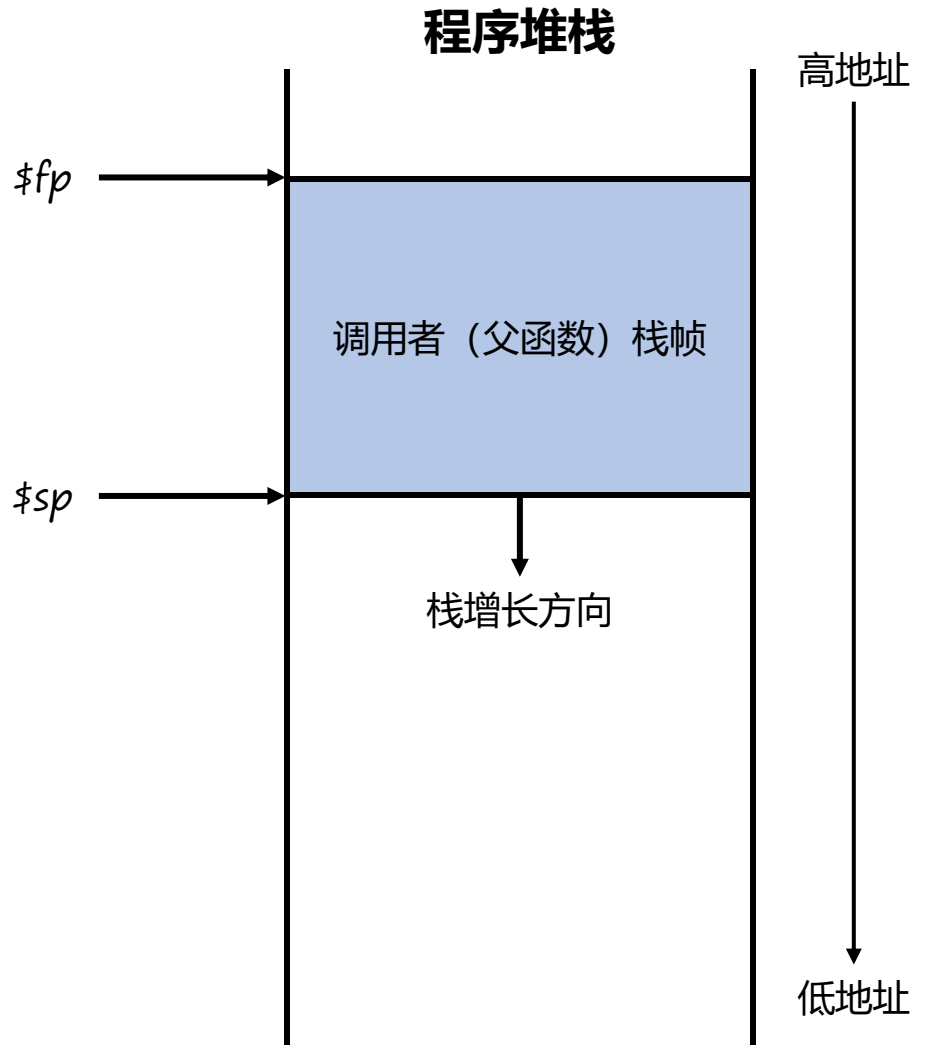
```
caller: # 父函数
  caller_entry:
    ...
  caller_labels_before_call:
    ...
  caller_call_callee:
    # 加载参数
    ld.w   $a0, $fp, -ARG1_OFFSET
    ld.w   $a1, $fp, -ARG2_OFFSET
    addi.w $a3, $zero, ARG3
    # 调用子函数
    bl callee
```



# 栈帧与函数调用：函数调用 – 示例

## 2. 子函数进入

```
callee: # 子函数
```

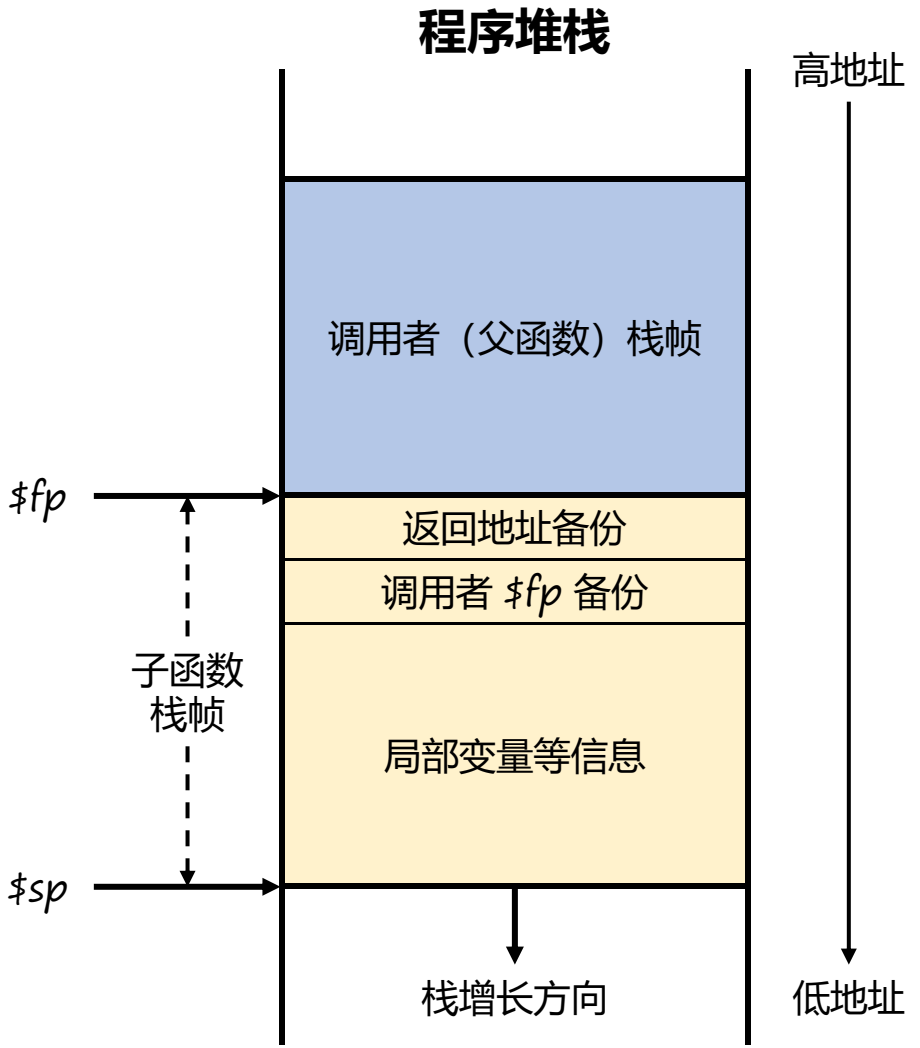


# 栈帧与函数调用：函数调用 – 示例

## 2. 子函数进入

- 初始化堆栈并保存上下文信息
  - 栈帧大小必须为 16 字节的整数倍

```
callee: # 子函数
  callee_entry:
    st.d   $ra, $sp, -8   # 保存返回地址
    st.d   $fp, $sp, -16  # 保存栈帧指针的值
    addi.d $fp, $sp, 0    # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N   # 分配新栈帧
```



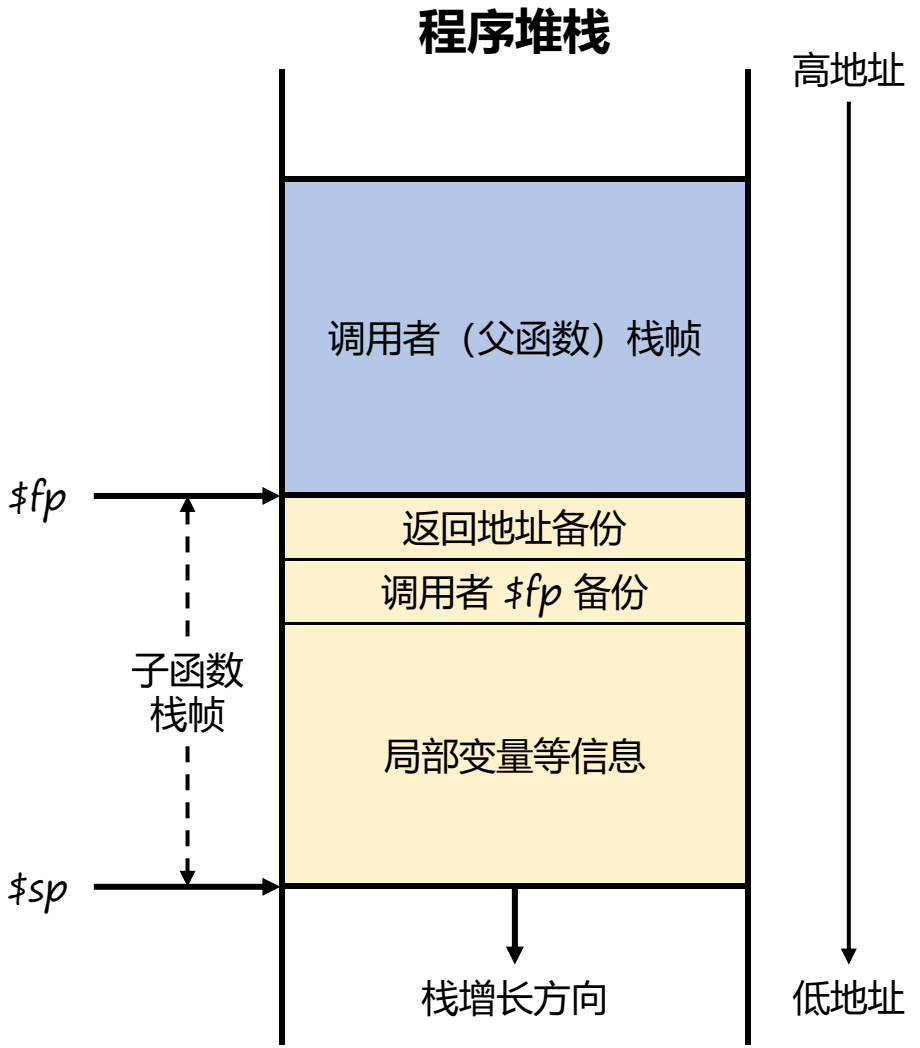
# 栈帧与函数调用：函数调用 – 示例

## 2. 子函数进入

- 初始化堆栈并保存上下文信息
  - 栈帧大小必须为 16 字节的整数倍

然后子函数开始执行计算需要的指令

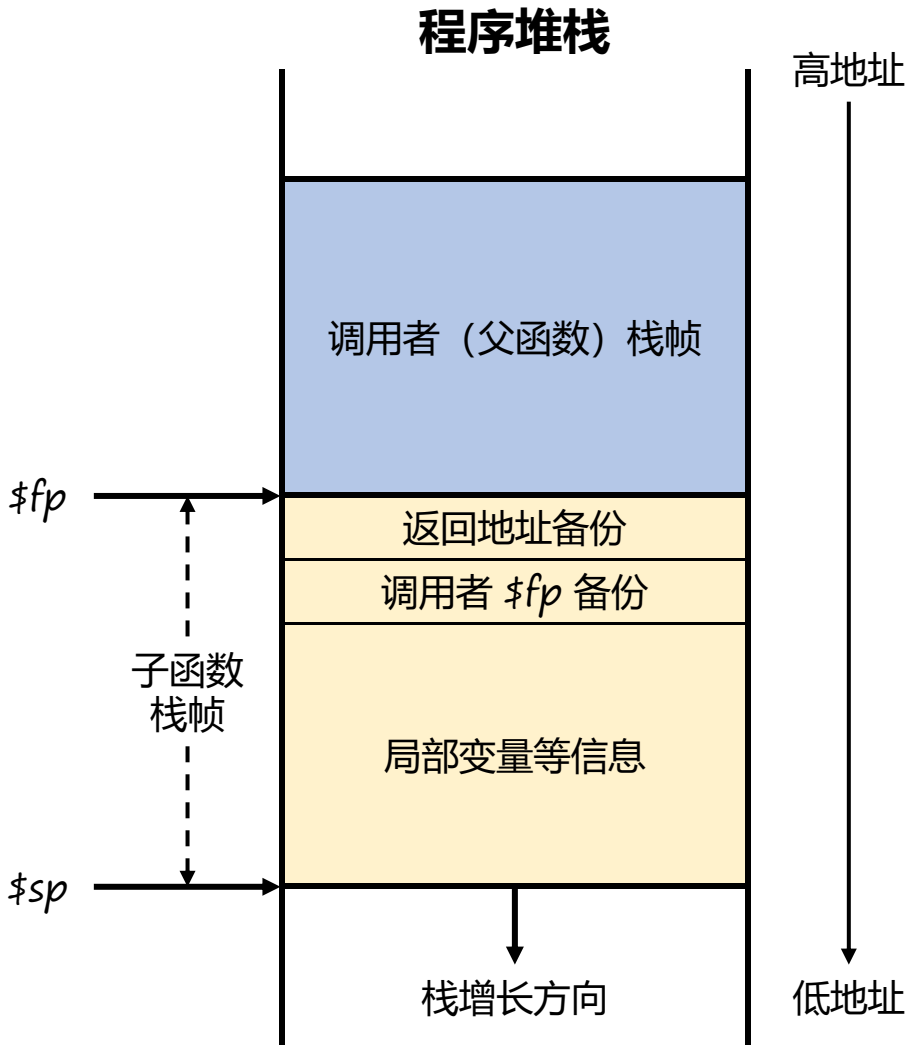
```
callee: # 子函数
  callee_entry:
    st.d   $ra, $sp, -8   # 保存返回地址
    st.d   $fp, $sp, -16  # 保存栈帧指针的值
    addi.d $fp, $sp, 0    # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N   # 分配新栈帧
  callee_compute_labels:
    ...
```



# 栈帧与函数调用：函数调用 – 示例

## 3. 子函数退出

```
callee: # 子函数
  callee_entry:
    st.d   $ra, $sp, -8   # 保存返回地址
    st.d   $fp, $sp, -16  # 保存栈帧指针的值
    addi.d $fp, $sp, 0    # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N   # 分配新栈帧
  callee_compute_labels:
    ...
```

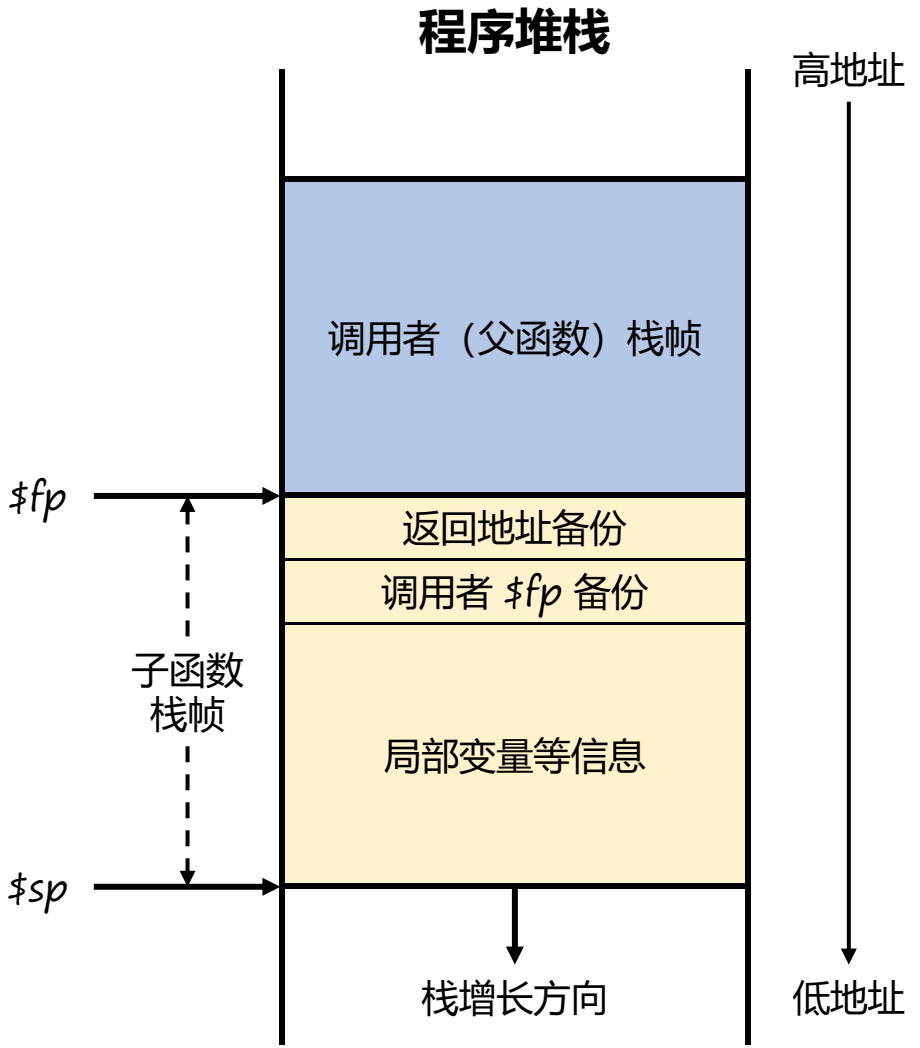


# 栈帧与函数调用：函数调用 – 示例

## 3. 子函数退出

1. 将返回值放入对应寄存器（$a0 等寄存器）中

```
callee: # 子函数
  callee_entry:
    st.d    $ra, $sp, -8    # 保存返回地址
    st.d    $fp, $sp, -16   # 保存栈帧指针的值
    addi.d $fp, $sp, 0      # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N     # 分配新栈帧
  callee_compute_labels:
    ...
  callee_return_value:
    ori     $a0, $t0, 0     # 将返回值写入 $a0
    b callee_exit
```

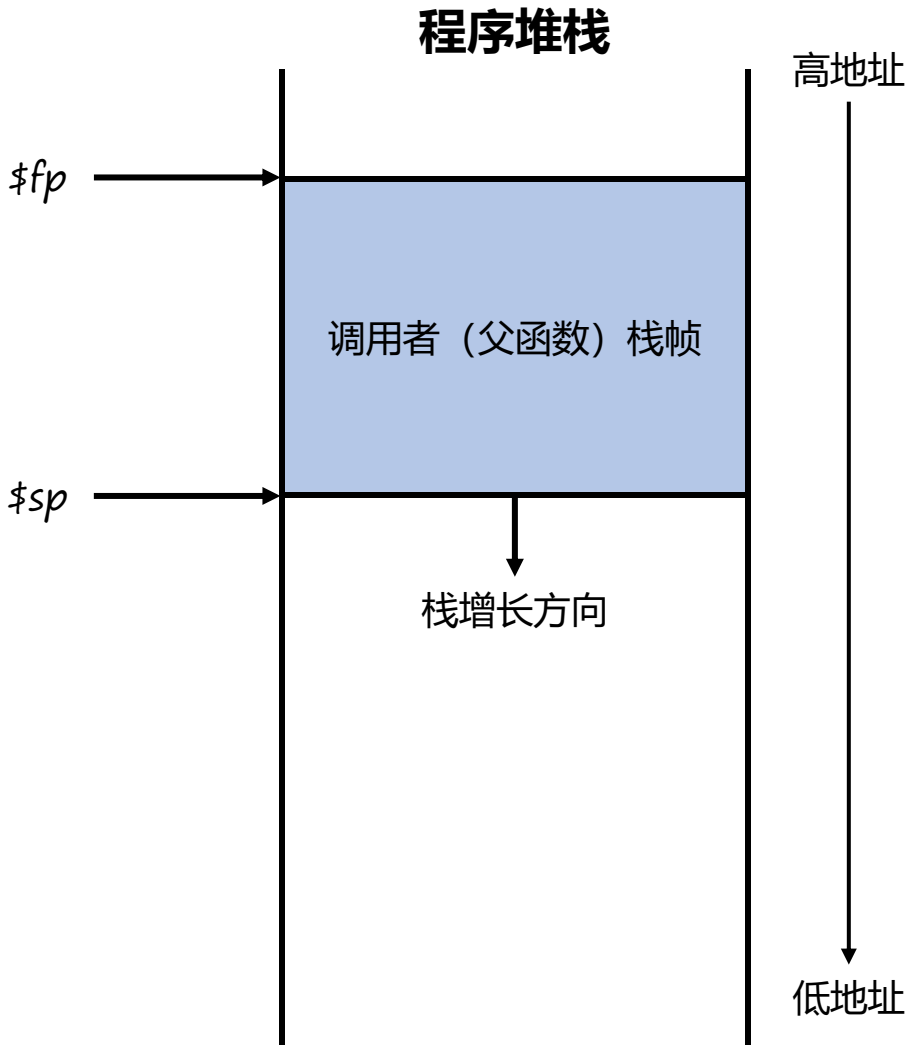


# 栈帧与函数调用：函数调用 – 示例

## 3. 子函数退出

1. 将返回值放入对应寄存器（$a0 等寄存器）中
2. 释放栈帧并恢复各个保留寄存器

```
callee: # 子函数
  callee_entry:
    st.d    $ra, $sp, -8    # 保存返回地址
    st.d    $fp, $sp, -16   # 保存栈帧指针的值
    addi.d $fp, $sp, 0      # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N     # 分配新栈帧
  callee_compute_labels:
    ...
  callee_return_value:
    ori     $a0, $t0, 0     # 将返回值写入 $a0
    b callee_exit
  callee_exit:
    addi.d $sp, $sp, N      # 恢复 $sp
    ld.d    $fp, $sp, -16   # 恢复 $fp
    ld.d    $ra, $sp, -8    # 恢复 $ra
```

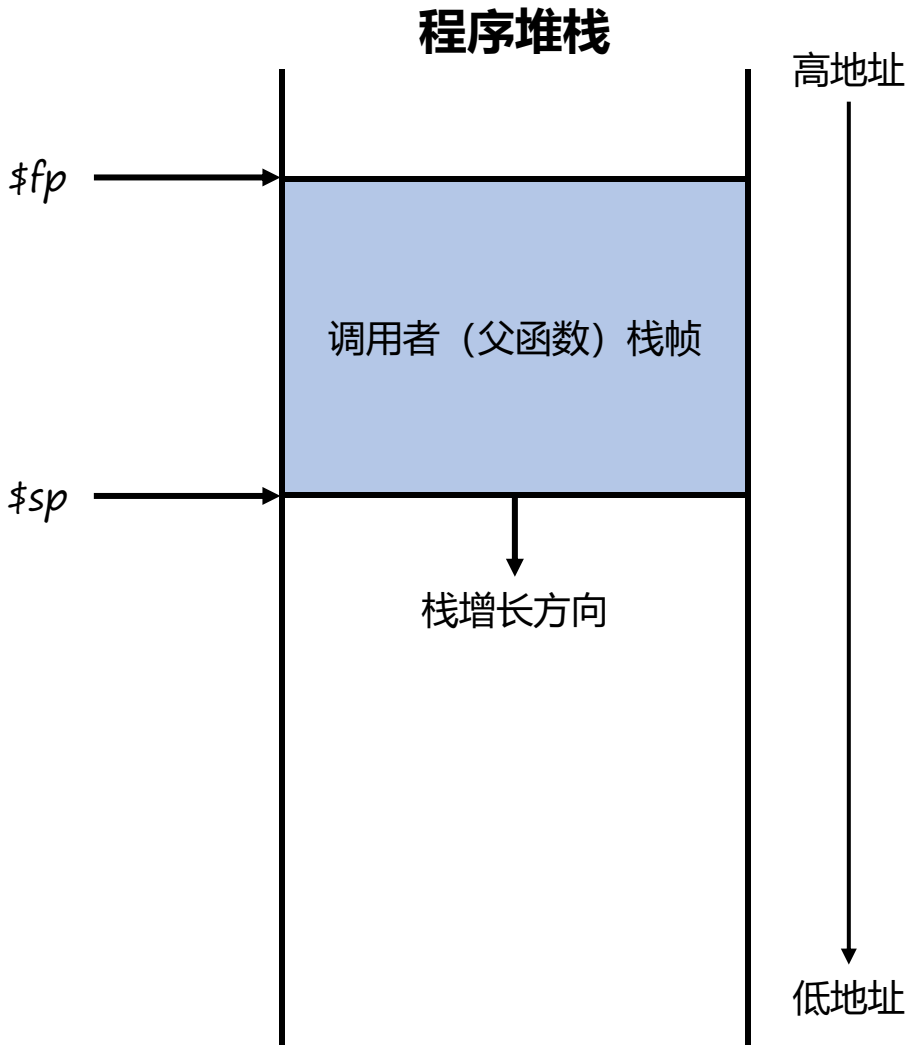


# 栈帧与函数调用：函数调用 – 示例

## 3. 子函数退出

1. 将返回值放入对应寄存器（$a0 等寄存器）中
2. 释放栈帧并恢复各个保留寄存器
3. 返回到父函数：jr $ra

```
callee: # 子函数
  callee_entry:
    st.d   $ra, $sp, -8   # 保存返回地址
    st.d   $fp, $sp, -16  # 保存栈帧指针的值
    addi.d $fp, $sp, 0    # 将新的栈底值写入 $fp
    addi.d $sp, $sp, -N   # 分配新栈帧
  callee_compute_labels:
    ...
  callee_return_value:
    ori    $a0, $t0, 0    # 将返回值写入 $a0
    b callee_exit
  callee_exit:
    addi.d $sp, $sp, N    # 恢复 $sp
    ld.d   $fp, $sp, -16  # 恢复 $fp
    ld.d   $ra, $sp, -8   # 恢复 $ra
    jr     $ra            # 返回
```

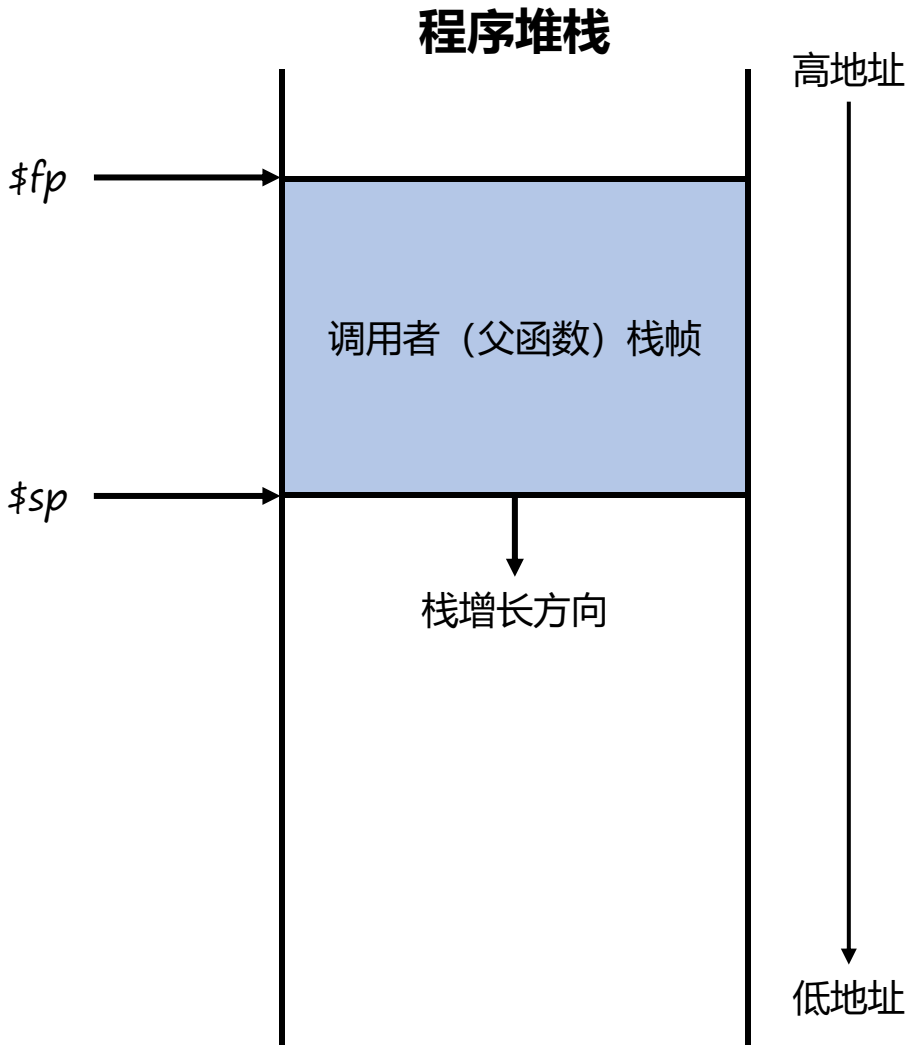


# 栈帧与函数调用：函数调用 – 示例

## 4. 父函数恢复

- 如果调用方在调用时为参数分配了额外的堆栈空间，则需要将这些空间释放
- 函数调用的返回值位于 $a0 等寄存器中

```
caller: # 父函数
  caller_entry:
    ...
  caller_labels_before_call:
    ...
  caller_call_callee:
    # 加载参数
    ld.w   $a0, $fp,  -ARG1_OFFSET
    ld.w   $a1, $fp,  -ARG2_OFFSET
    addi.w $a3, $zero, ARG3
    # 调用 callee
    bl     callee
    # 返回值在 $a0 中
```



# 栈帧与函数调用：参数传递

- **汇编程序怎么在函数调用时传递参数?**
  - 使用规定的寄存器：$a0 - $a7，$fa0 - $fa7
  - 寄存器不够用时，调用者在堆栈上开辟一块空间用于传参

# 栈帧与函数调用：参数传递


- **汇编程序怎么在函数调用时传递参数?**
  - 使用规定的寄存器：$a0 - $a7，$fa0 - $fa7
  - 寄存器不够用时，调用者在堆栈上开辟一块空间用于传参
- **完整的参数传递规范比较复杂，本实验编译器后端使用简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| | |
| | |
| | |
| | |
| | |

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| 整型 | $a0 |
| | |
| | |
| | |
| | |

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| 整型 | $a0 |
| 指针 | $a1 |
| | |
| | |
| | |

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| 整型 | $a0 |
| 指针 | $a1 |
| 浮点型 | $fa0 |
| | |
| | |

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| 整型 | $a0 |
| 指针 | $a1 |
| 浮点型 | $fa0 |
| 整型 | $a2 |
| | |

# 栈帧与函数调用：参数传递

- **简化后的参数传递规范**
  - 仅使用 $a0 - $a7 和 $fa0 - $fa7 用于参数传递
  - 整数类型（指针/整型）参数使用通用寄存器 $a0 - $a7
  - 浮点类型参数使用浮点寄存器 $fa0 - $fa7
  - 各个参数按照顺序依次放入对应的寄存器中
- **示例：**

```
int function(
    int a,
    int b[],
    float c,
    int d,
    float e
)
```

| 参数类型 | 参数寄存器 |
| --- | --- |
| 整型 | $a0 |
| 指针 | $a1 |
| 浮点型 | $fa0 |
| 整型 | $a2 |
| 浮点型 | $fa1 |


# 栈式分配介绍

# 栈式分配：What

将IR翻译为汇编指令：

```
%op2 = add i32 %op0, %op1
```

```
add.w $rd, $rj, $rk
```

*如何得到%op0与%op1的值?*

# 栈式分配：What


## 栈式分配的定义

1. 程序的所有变量都保存在栈上
2. 只在参与计算时提取到寄存器中

## 栈式分配的步骤

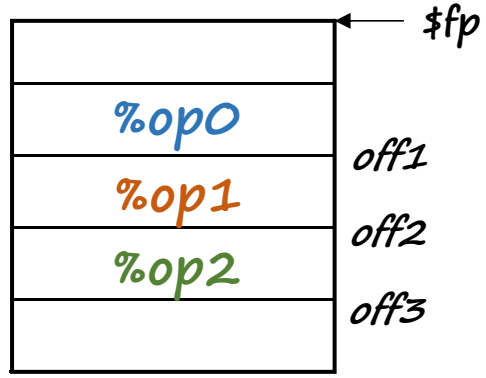
1. 变量分配：为每个变量分配栈帧位置
2. 指令选择
   a. load：提取源操作数至寄存器
   b. 指令选择：根据指令类型选择合适的汇编指令
   c. store：将结果从寄存器写回栈帧

# 栈式分配：What

将IR翻译为汇编指令：

```
%op2 = add i32 %op0, %op1
```

```
ld.w  $t0, $fp, off1
ld.w  $t1, $fp, off2
add.w $t0, $t0, $t1
st.w  $t0, $fp, off3
```



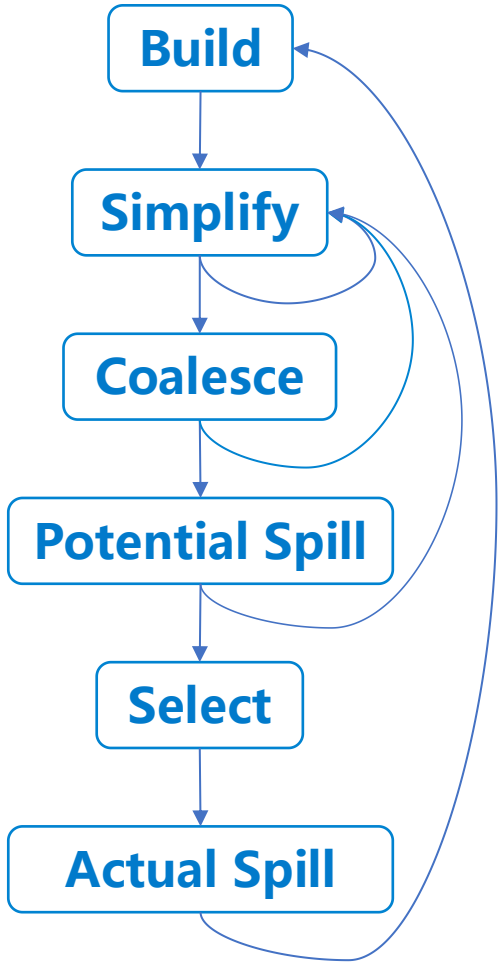

程序部分栈帧

# 栈式分配：Why

## 寄存器分配

- 目标：变量保存在寄存器中
- 难以用算法实现最优解
- 需要构建冲突图、循环迭代
- 需要处理复杂的寄存器溢出
- ……

**实现复杂！**




*寄存器分配*


# 汇编程序示例

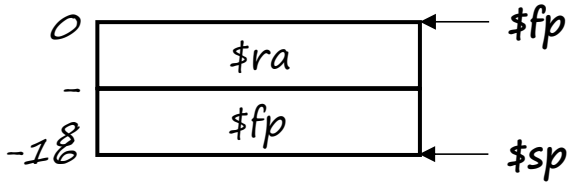
# 程序1：返回值

```
define i32 @main() {
label_entry:
  ret i32 0
}
```

*IR程序*

```
    .text
    .globl main
    .type main, @function
main:
    st.d    $ra, $sp, -8
    st.d    $fp, $sp, -16
    addi.d $fp, $sp, 0
    addi.d $sp, $sp, -16
.main_label_entry:
    addi.w $a0, $zero, 0
    b       main_exit
main_exit:
    addi.d $sp, $sp, 16
    ld.d    $ra, $sp, -8
    ld.d    $fp, $sp, -16
    jr $ra
```

*汇编代码*




*程序栈帧*

# 程序2：局部变量赋值

```
define void @main() {
label_entry:
  %op0 = alloca i32
  store i32 1234, i32* %op0
  ret void
}
```

*IR程序*

```
        .text
        .globl main
        .type main, @function
main:
        st.d    $ra, $sp, -8
        st.d    $fp, $sp, -16
        addi.d $fp, $sp, 0
        addi.d $sp, $sp, -32
.main_label_entry:
        addi.d $t0, $fp, -28
        st.d    $t0, $fp, -24
        ld.d    $t0, $fp, -24
        addi.w $t1, $zero, 1234
        st.w    $t1, $t0, 0
        addi.w $a0, $zero, 0
        b       main_exit
main_exit:
        addi.d $sp, $sp, 32
        ld.d    $ra, $sp, -8
        ld.d    $fp, $sp, -16
        jr      $ra
```

*汇编代码*

# 程序2：局部变量赋值

```
define void @main() {
label_entry:
  %op0 = alloca i32
  store i32 1234, i32* %op0
  ret void
}
```

IR程序

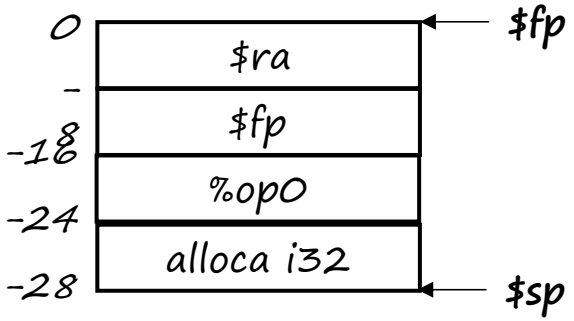
```
.....

addi.d $t0, $fp, -28
st.d   $t0, $fp, -24

ld.d   $t0, $fp, -24
addi.w $t1, $zero, 1234
st.w   $t1, $t0, 0

.....
```

汇编代码




程序栈帧

# 程序3：分支

```
define i32 @main() {
label_entry:
  %op0 = icmp slt i32 1, 2
  br i1 %op0, Label %L1, %L2
L1:
  ret i32 0
L2:
  ret i32 1
}
```

***IR程序***

# 程序3：分支

```
        .text
        .globl main
        .type main, @function
main:
        st.d    $ra, $sp, -8
        st.d    $fp, $sp, -16
        addi.d $fp, $sp, 0
        addi.d $sp, $sp, -32
.main_label_entry:
        addi.w $t0, $zero, 1
        addi.w $t1, $zero, 2
        slt     $t2, $t0, $t1
        st.b    $t2, $fp, -17
        ld.b    $t0, $fp, -17
        bstrpick.w $t0, $t0, 0, 0
        st.w    $t0, $fp, -21
        ld.w    $t0, $fp, -21
        addi.w $t1, $zero, 0
        xor     $t2, $t0, $t1
        sltu    $t2, $zero, $t2
```

汇编代码

# 程序3：分支

```
define i32 @main() {
label_entry:
  %op0 = icmp slt i32 1, 2
  br i1 %op0, Label %L1, %L2
L1:
  ret i32 0
L2:
  ret i32 1
}
```

*IR程序*

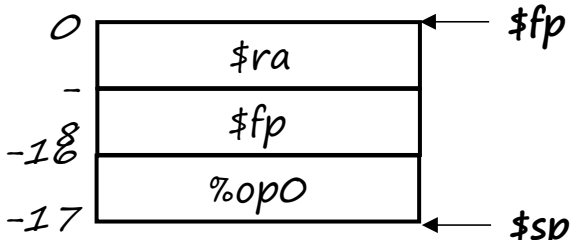
```
.....

addi.w $t0, $zero, 1
addi.w $t1, $zero, 2
slt    $t2, $t0, $t1
st.b   $t2, $fp, -17

ld.b $t0, $fp, -17
bne  $t0, $zero, .L1
b    .L2

.....
```

*汇编代码*




*程序栈帧*

# 程序4：全局变量

```
@a = global i32 zeroinitializer
define void @main() {
label_entry:
  store i32 10, i32* @a
  ret void
}
```

IR程序

```
        .text
        .section .bss, "aw", @nobits
        .globl   a
        .type    a, @object
        .size    a, 4
a:
        .space   4
        .text
        .globl   main
        .type    main, @function
main:
        st.d     $ra, $sp, -8
        st.d     $fp, $sp, -16
        addi.d   $fp, $sp, 0
        addi.d   $sp, $sp, -16
.main_label_entry:
        la.local $t0, a
        addi.w   $t1, $zero, 10
        st.w   $t1, $t0, 0
        addi.w $a0, $zero, 0
        b      main_exit
main_exit:
        addi.d $sp, $sp, 16
        ld.d   $ra, $sp, -8
        ld.d   $fp, $sp, -16
```

汇编代码

# 程序4：全局变量

```
@a = global i32 zeroinitializer
define void @main() {
label_entry:
  store i32 10, i32* @a
  ret void
}
```

IR程序

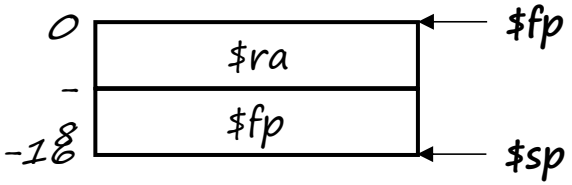
```
.text
.section .bss, "aw", @nobits
.globl a
.type a, @object
.size a, 4
a:
.space 4

......

.main_label_entry:
la.local $t0, a
addi.w   $t1, $zero, 10
st.w     $t1, $t0, 0

......
```

汇编代码




程序栈帧

# 程序5：函数调用

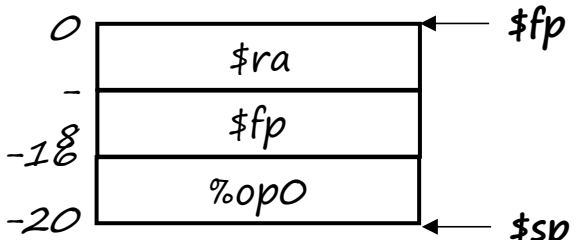
```
define i32 @func(i32 %arg0) {
label_entry:
  ret i32 %arg0
}
define void @main() {
label_entry:
  %op0 = call i32 @func(i32 1)
  ret void
}
```

IR程序

```
        .text
        .globl func
        .type func, @function
func:
        jr $ra

        .globl main
        .type main, @function
main:
        st.d    $ra, $sp, -8
        st.d    $fp, $sp, -16
        addi.d $fp, $sp, 0
        addi.d $sp, $sp, -32
.main_label_entry:
        addi.w $a0, $zero, 1
        bl      func
        st.w    $a0, $fp, -20
        addi.w $a0, $zero, 0
        b       main_exit
main_exit:
```

汇编代码




程序栈帧


# 实验代码框架

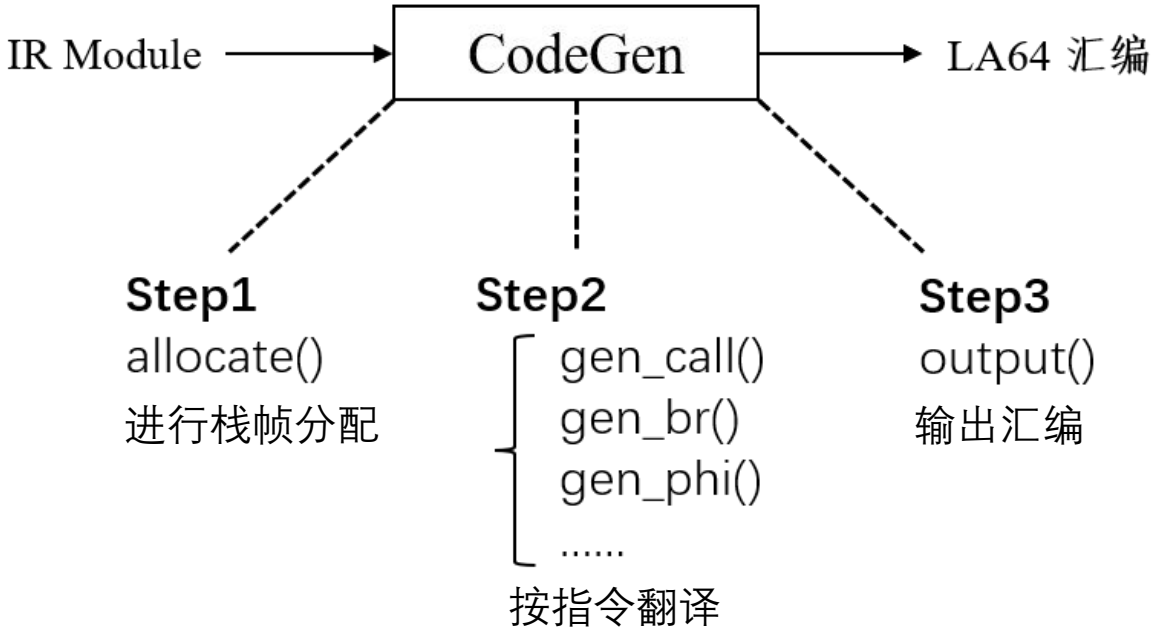
# CodeGen类概述

```
class CodeGen {
private:
  void allocate(); // 变量分配

  /*=== 以下为辅助函数 ===*/
  // 将数据在寄存器和栈帧间搬移
  void load_xxx(...);
  void store_xxx(...);
  // 添加汇编指令
  void append_inst(...);
  // 基本块在汇编程序中的名字
  static std::string label_name(BasicBlock*);
  /*=== 以上为辅助函数 ===*/

  // 需要补全的部分，进行代码生成的各个环节
  void gen_xxx(...);
};
```

# 程序流程




***程序流程***

**代码框架导读**

# 代码示例：添加指令

以如下目标汇编代码为例：

```
    .text
    .globl main
    .type main, @function
main:
    addi.w $a0, $zero, 0
    jr $ra
```

# 代码示例：添加指令


定义不同的 gen_manually()，以生成不同的汇编程序

```
void gen_manually (CodeGen *codegen);

int main() {
  auto *codegen = new CodeGen(nullptr);

  gen_manually(codegen);

  std::cout << codegen->print();
  delete codegen;
  return 0;
}
```

```
    .text
    .globl main
    .type main, @function
main:
    addi.w $a0, $zero, 0
    jr $ra
```

# 代码示例：添加指令

使用 CodeGen::append_inst() 添加指令：

```
void gen_manually(CodeGen *codegen) {
  codegen->append_inst(".text", ASMInstruction::Atrribute);
  codegen->append_inst(".globl main", ASMInstruction::Atrribute);
  codegen->append_inst(".type main, @function", ASMInstruction::Atrribute);

  codegen->append_inst("main", ASMInstruction::Label); // main 函数标签
  codegen->append_inst("addi.w $a0, $zero, 0");
  codegen->append_inst("jr $ra");
}
```

```
    .text
    .globl main
    .type main, @function
main:
    addi.w $a0, $zero, 0
    jr $ra
```

# 代码示例：辅助函数


**栈式分配的 load 环节：**

*void CodeGen::load_to_greg(Value *v, const Reg &reg);*

将 IR 变量 v 的值加载至寄存器 reg 中

**栈式分配的 store 环节：**

*void CodeGen::store_from_greg(Value *v, const Reg &reg);*

将寄存器 reg 中的值存入 IR 变量 v 的栈帧位置

# 代码示例：辅助函数

## 例：对二元 IR 指令的翻译

- context.inst 代表当前翻译指令，比如 *%op2 = add i32 %op0, %op1*
- Reg::t(i) 定义了寄存器 $ti

```
void CodeGen::gen_binary() {
  load_to_greg(context.inst->get_operand(0), Reg::t(0));
  load_to_greg(context.inst->get_operand(1), Reg::t(1));
  switch (context.inst->get_instr_type()) {
  case Instruction::add:
    append_inst("add.w $t2, $t0, $t1");
    break;
  ...
  }
  store_from_greg(context.inst, Reg::t(2));
}
```

从栈上提取两个操作数

按照IR指令的类型生成对应的汇编

将结果保存回栈上

# 一起努力
# 打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年11月21日