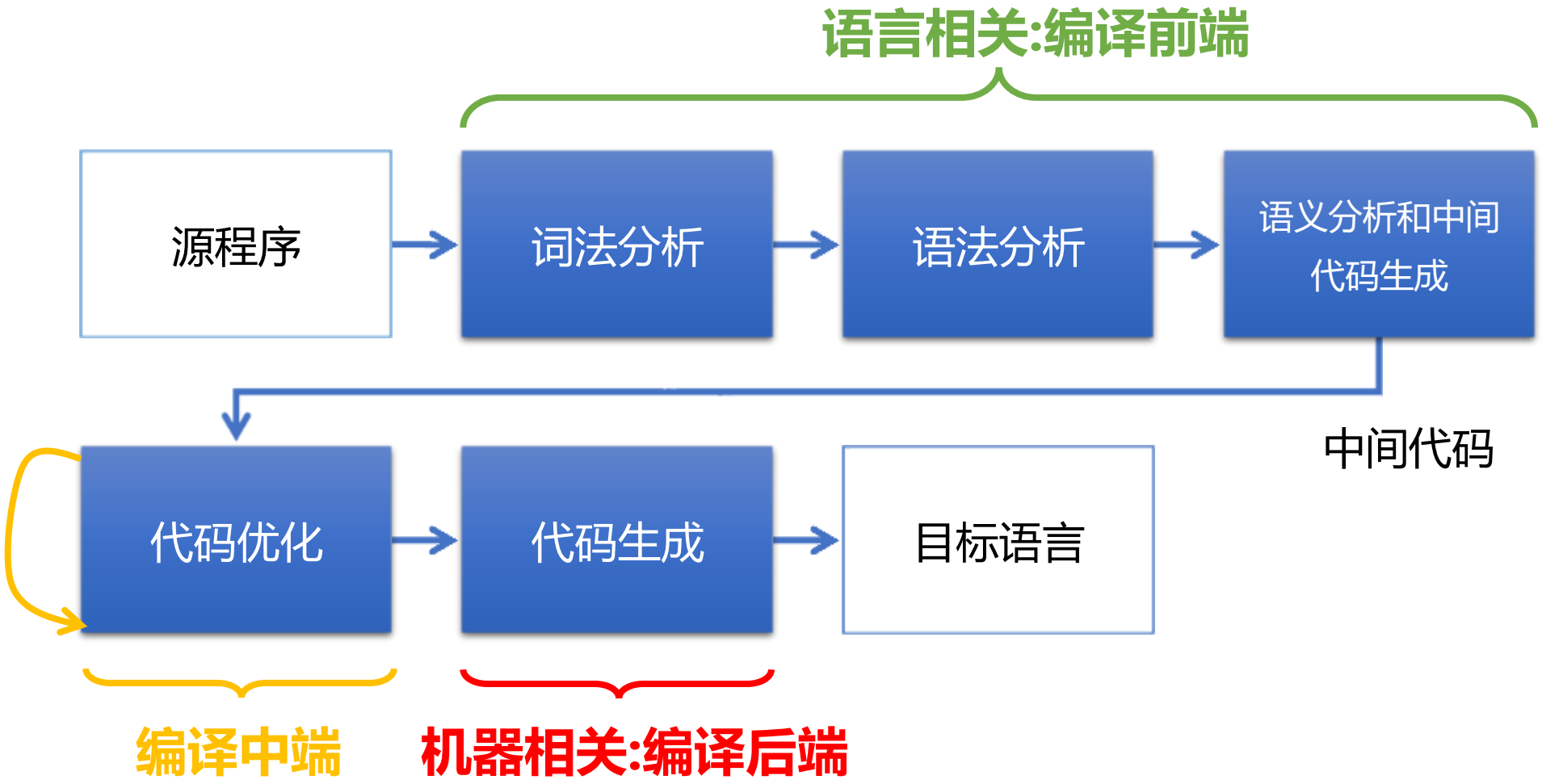
编译后端代码生成与优化介绍

国防科技大学计算机学院 沈洁

国防科技大学
NATIONAL UNIVERSITY
OF DEFENSE TECHNOLOGY

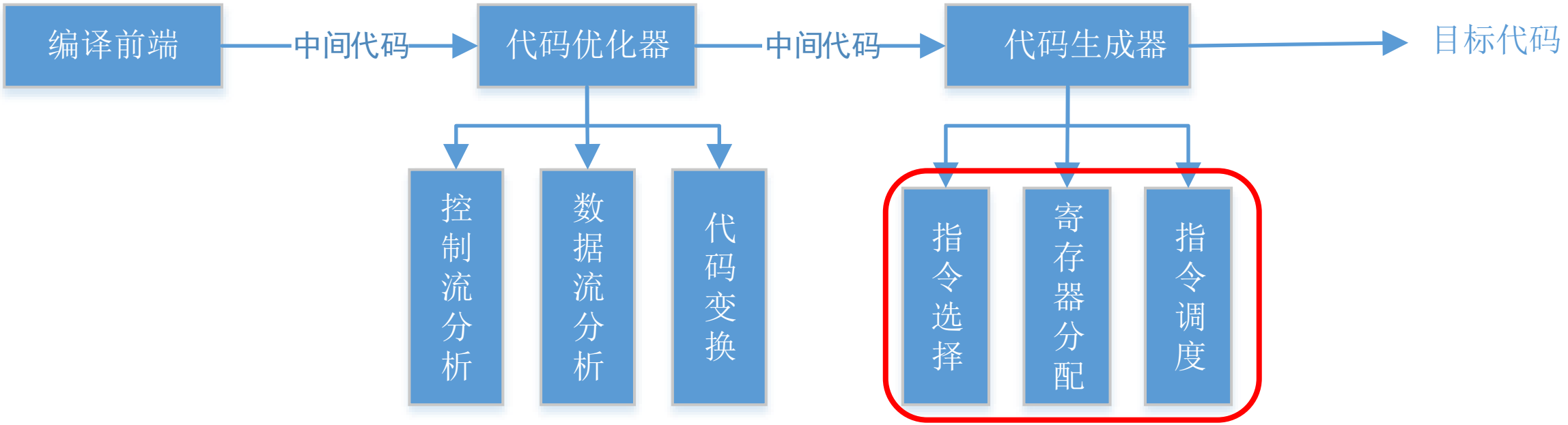
编译流程

总体流程


语言相关:编译前端
源程序
词法分析
语法分析
语义分析和中间
代码生成
中间代码
代码优化
代码生成
目标语言
编译中端
机器相关:编译后端

编译流程

- 编译中端
  - 与机器无关的代码优化(分析+变换)
- 编译后端
  - 与机器相关的代码优化

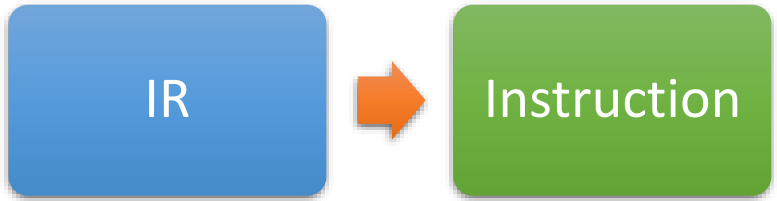


内容

- 编译后端基本介绍
  - 指令选择
  - 寄存器分配
  - 指令调度
- 面向ARM后端的代码生成介绍

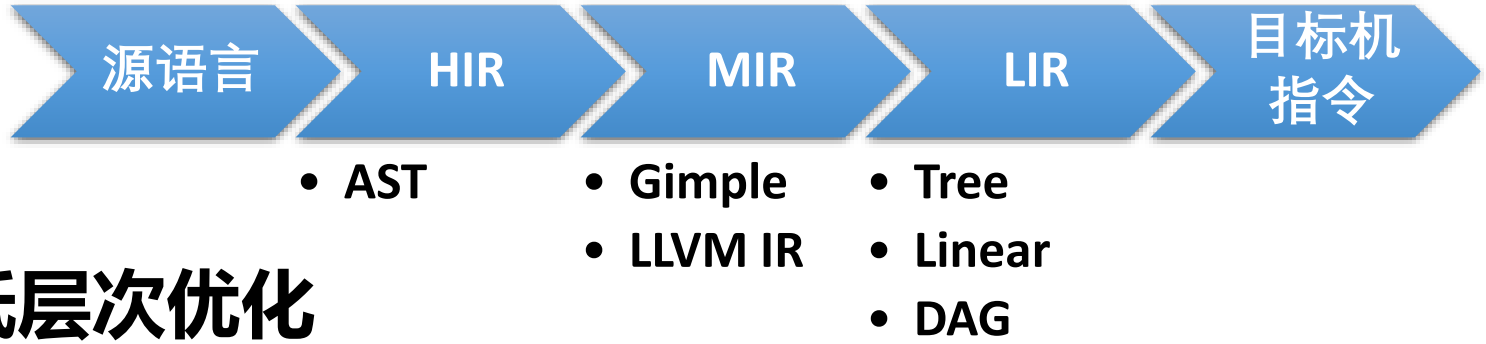
指令选择

- 将中间表示(IR)翻译成等价的目标机指令集(ISA)指令序列的过程
  - 编译中端代码优化器是运行在代码的IR形式上
  - IR代码必须翻译成ISA指令序列，才能在目标机上执行
  - 指令选择实现IR到目标机指令的翻译



低层次中间表示

- 高层次中间表示(HIR)
  - 靠近源语言，更多上下文信息用于进行高层次优化
- 中层次中间表示 (MIR)
  - 中端编译优化
- 低层次中间表示(LIR)
  - 靠近机器，用于进行低层次优化
  - 更容易翻译为目标机指令
- 在指令选择之前，可以将中间代码转换为更底层的表示



低层次中间表示

- LLVM在指令选择之前，将LLVM IR转换为SelectionDAG
  - 与目标机无关的LIR
  - 每个基本块对应一个DAG，DAG中的结点对应指令或者一个操作数，DAG中的边描述了指令间存在数据依赖关系
  - 基于SelectionDAG采用模式匹配进行指令选择
- 针对不同形式的低层次中间表示，有不同的指令选择方法

指令选择方法

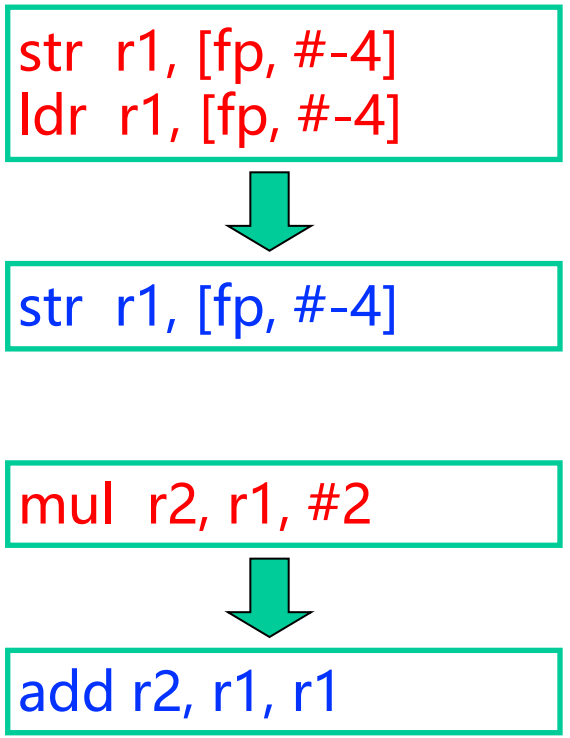
- 基于宏扩展(Macro-expansion)的指令选择
  - 自顶向下将LIR逐一翻译为指令序列 (one-by-one translation)
  - ☺ 简单，易于实现
  - ☹ 难以考虑代码整体质量，不能够利用指令集强大的寻址模式

一条指令可以同时完成地址计算、访存操作和寄存器算术运算

```
str  fp, [sp, #-4]!
```

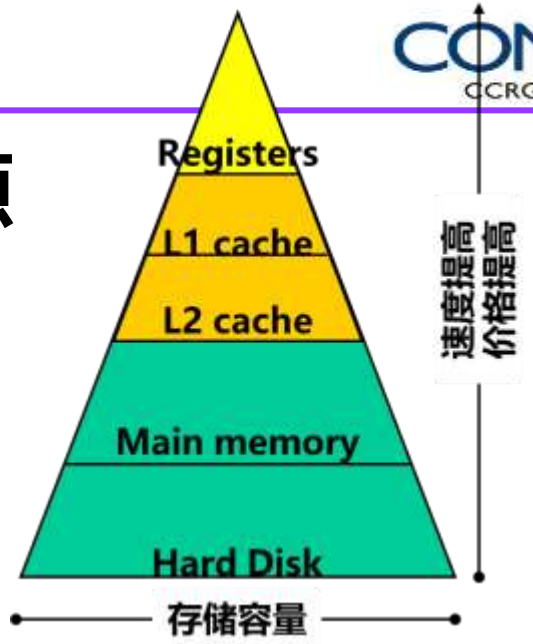
指令选择方法

- 基于模式模式匹配(Pattern-matching)的指令选择
  - 利用模式匹配技术选择与一段LIR匹配的指令，直到得到覆盖全部LIR的指令序列
  - 例如，树模式匹配方法
- 窥孔优化
  - 编译器使用滑动窗口(也称为窥孔)在代码上移动
  - 每次仅考察窗口中的指令序列(一小段相邻指令序列)
  - 寻找可以改进的特定模式
  - 识别出一个模式时，使用更好的指令序列重写该模式
  - ☺ 快速高效，有限的模式集合+有限的关注区域
- 指令选择阶段假设有足够多的符号寄存器
  - 到寄存器分配阶段再考虑符号寄存器到物理寄存器的分配

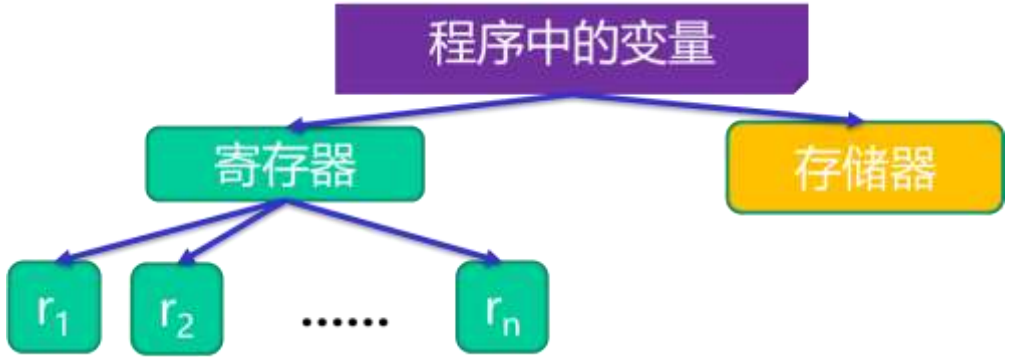


寄存器分配

- 目标：高效、合理地使用有限的寄存器资源
- 编译器中最重要的优化之一



- 确定哪些变量保存在寄存器中，哪些变量保存在存储器中 (寄存器分配)
- 保存在寄存器中的变量：具体放在哪个寄存器 (寄存器指派)



寄存器分配原则

- 程序变量数目 vs. 寄存器数目
  - 程序中有大量的变量，变量数一般大于寄存器数
- 寄存器分配原则一: 尽可能将更多的变量保存在寄存器中
  - 不能让一个变量占用寄存器的时间比实际需要的时间长
- 寄存器分配原则二: 尽可能将频繁使用的变量保存在寄存器中
  - 将那些使用较少的变量存放在存储器中
  - 尽可能减少溢出和读写指令次数

基于使用计数的寄存器分配

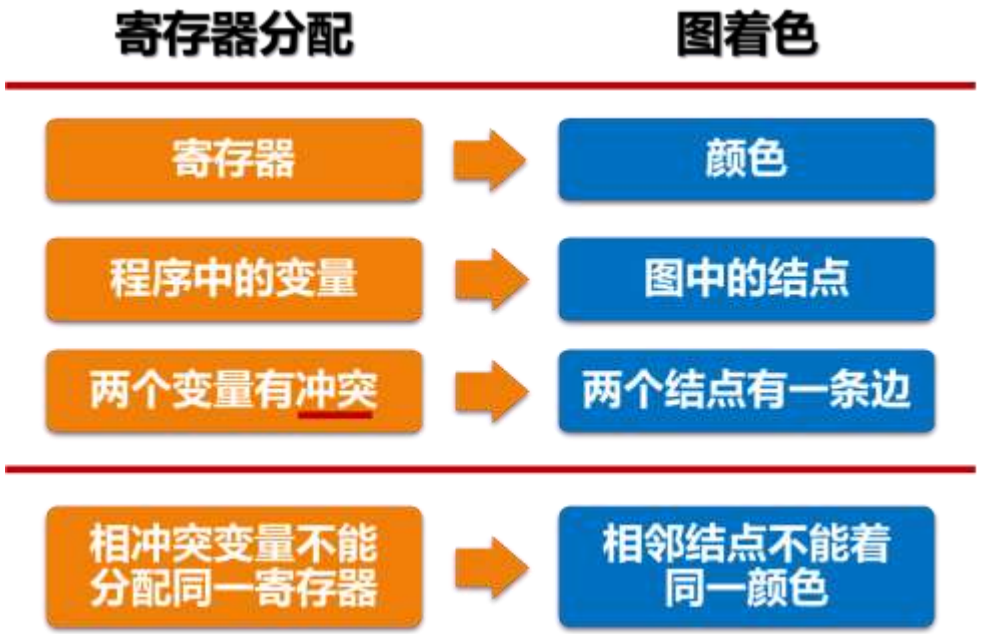
COMP CCRG Open MP

- 基本方法
  - 以基本块为单位，根据需要依次分配寄存器(需要一个则分配一个)
  - 遇到变量的一次使用，使用计数减1
  - 当使用计数为0时，寄存器便可再次分配给其它变量使用
  - 当寄存器不够时，根据启发式信息溢出一个变量
    - 溢出使用计数最小的
    - 溢出已经有副本在存储器中
- ☺ 简单，易于实现
- ☹ 使用计数不能真正反应变量的频繁使用情况
  - 循环内使用的变量 vs 循环外使用的变量

基于图着色的寄存器分配

COMP CCRG Open MP

- 全局方法
- 将寄存器分配转化为图着色问题
  - 基于活跃变量分析，获取程序每个点的活跃变量集合，构建冲突图
  - 两个变量在程序的某个点同时活跃，则对应两个结点之间有一条边相连不能分配同一寄存器
  - 两个变量在程序的任何一个点都不同时活跃，则对应两个结点之间没有边，可以分配同一寄存器



基于图着色的寄存器分配

- Chaitin算法: 奠定了图着色寄存器分配算法的基础
  - 构建冲突图、合并结点、化简冲突图、溢出结点的迭代过程

- 改进算法主要在合并和溢出部分
  - Briggs改进算法: 保守合并，乐观着色
  - George改进算法: 放松合并条件，引入迭代的合并过程和冻结结点
- ☺ 全局近似最优解
- ☹ 效率不高，比较耗时

基于线性扫描的寄存器分配

全局寄存器分配

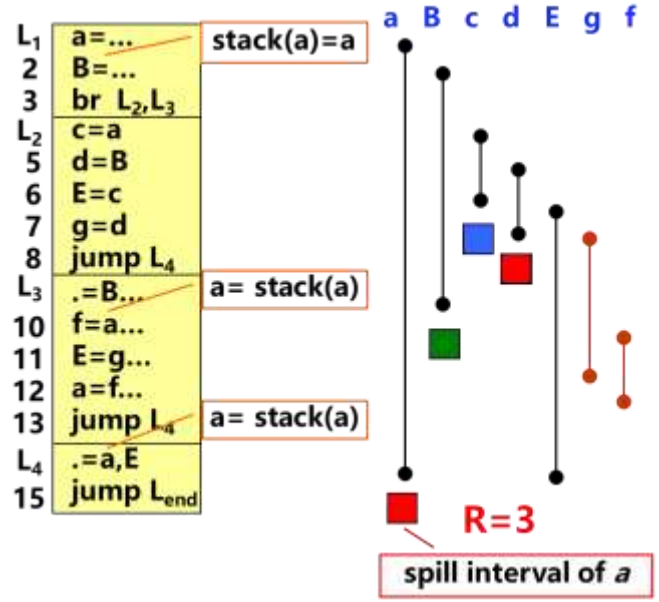
给一个由区间组成的有序序列指派颜色的问题

- **活跃区间(live interval)**
  - 对于变量v，如果指令序列中存在一段区间[i, j]，其中1≤i≤j≤N，使得变量v只在此区间内是活跃的，则称[i, j]是变量v的活跃区间
- **基本思想**
  - 如果两个活跃区间重叠，则称它们存在冲突
  - 分配寄存器给尽可能多的区间，保证冲突的区间不分配相同寄存器
  - 如果在程序某点重叠的活跃区间数n大于可分配的寄存器数R，则至少有n-R个活跃区间要分配到存储器中

基于线性扫描的寄存器分配

COMP CCRG Open MP

- 维护两张表
  - 一张表存放待分配的活跃区间，按活跃区间开始点增加的顺序存放(待分配表)
  - 一张表存放已分配寄存器但还未到达其结束点的活跃区间，按结束点增加的顺序存放(active表)
- 按开始点增加的顺序依次扫描待分配的活跃区间
- 在每一步
  - 如果有可用颜色，则给待分配的区间指派一个颜色
  - 如果没有可用的颜色，则尝试释放”已到期”的区间
  - 如果没有已到期的区间，则从所有已分配寄存器的区间和待分配的新区间中选择一个区间溢出

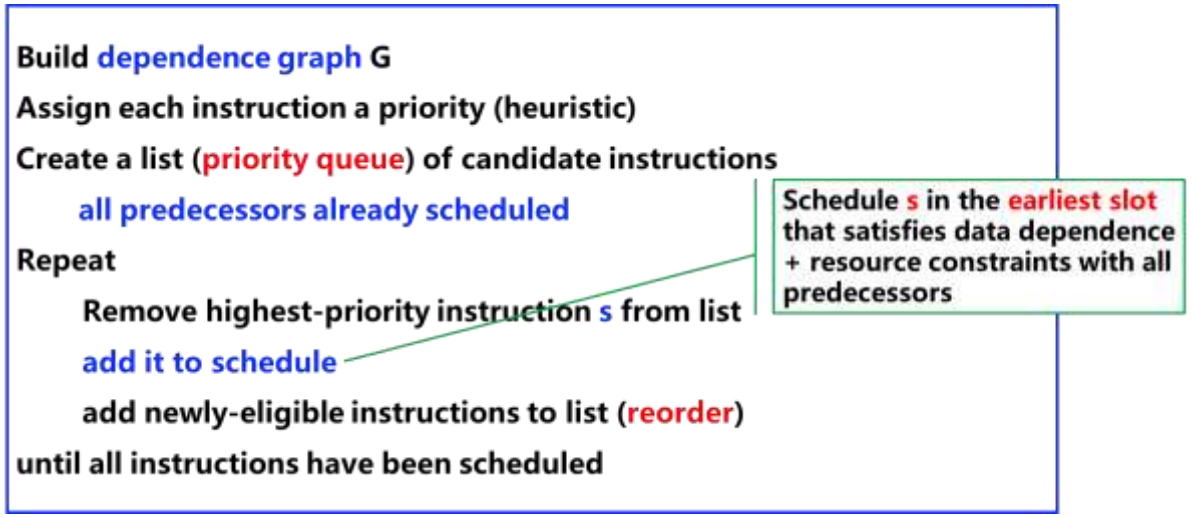


基于线性扫描的寄存器分配

- 在一个pass中扫描所有变量的活跃区间，以贪心的方式为变量分配寄存器
  - ☺ 实现简单，算法高效，生成的代码质量相对较高
  - ☹ 分配速度的提高是以一定程度对代码质量的影响为代价的
    - 基本块排序影响活跃区间的确定，影响寄存器分配质量，从而影响代码质量
    - 好的基本块排序使得变量的活跃区间更短、洞更少
  - LLVM Greedy分配方法可以看成线性扫描的一个改进(分隔活跃区间)

指令调度

- 通过重排指令序列，试图减少程序总执行时间
- 目标：最大化指令级并行，减少流水线中的气泡
- 基本块内的局部调度: 表调度方法
  - 基于依赖关系分析构建依赖图
  - 根据启发式信息给结点设置优先级
    - 关键路径
    - 结点后继数
    - 资源需求
  - 根据优先级选择指令进行调度



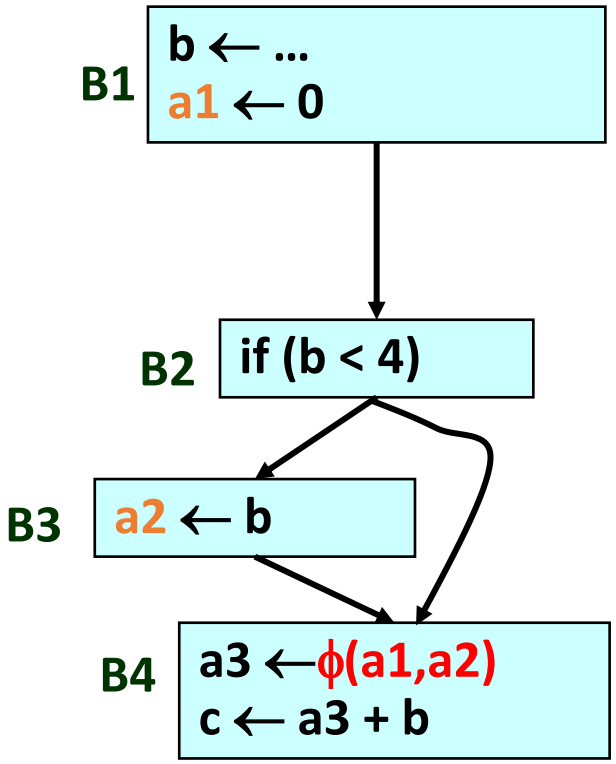
- 跨基本块的全局调度: 轨迹调度, 软件流水等

指令调度中的优化技术

- 寄存器重命名 (Register renaming)
  - 消除反向依赖(WAR)和输出相关(WAW) 两种“伪相关性”
- 平衡调度 (Balanced scheduling)
  - 在load指令后插入与load无关的指令来隐藏访存延迟
- 循环展开 (Loop unrolling)
  - 增加基本块的大小，从而有更多指令可以用于调度

SSA IR

- 插入Phi函数，将stack形式的IR转换为SSA形式的IR

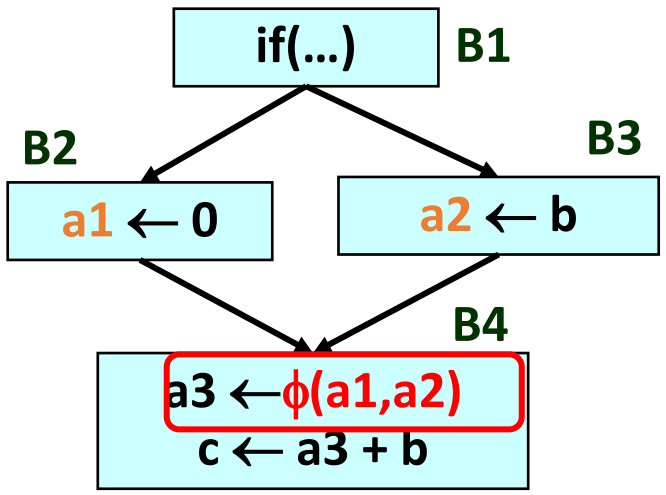


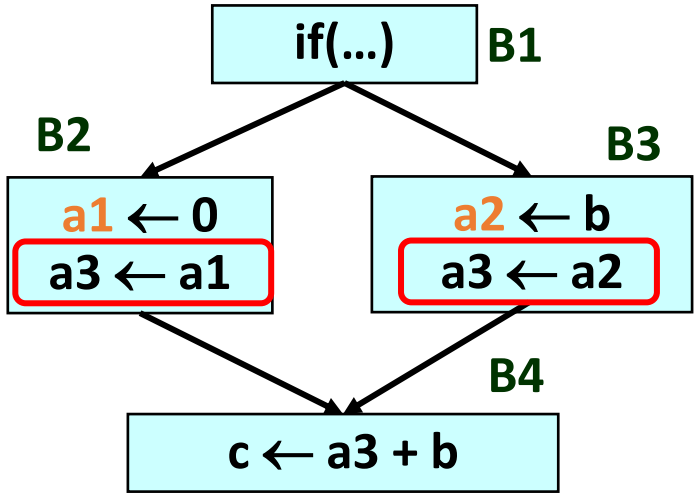
$$\phi(a1,a2) = \begin{cases} a1 \text{ if arriving at B4 from B2} \\ a2 \text{ if arriving at B4 from B3} \end{cases}$$

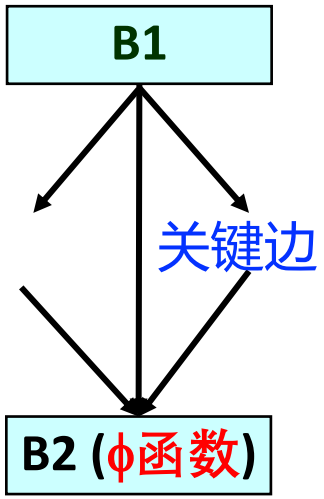
针对Phi函数，如何做代码生成?

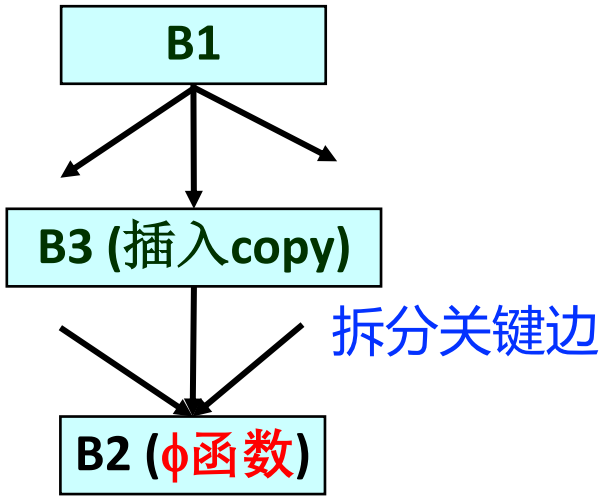
删除Phi函数

- 没有Phi函数对应的指令，需删除Phi函数，转回Non-SSA形式
  - Phi的前驱均只有汇合结点这一个后继: 在前驱的定值点后，插入copy或move指令
  - Phi的前驱有多个后继: 在Phi和前驱之间插入一个新基本块，在新基本块中插入copy或move指令



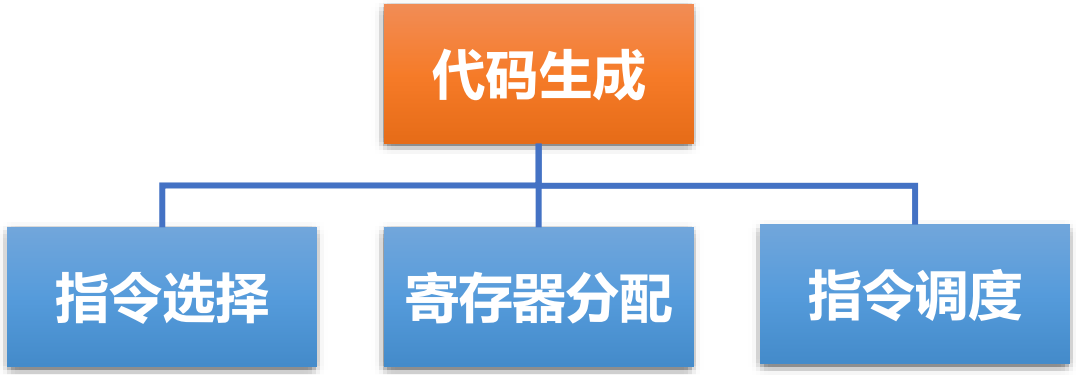




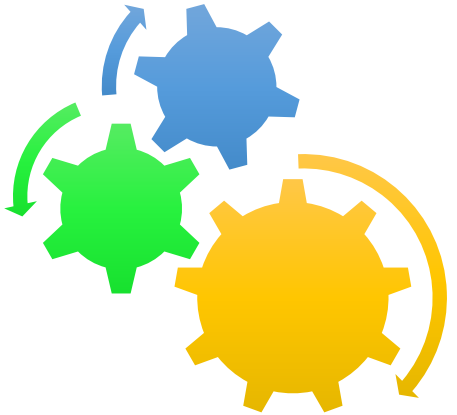


小结

- 编译后端：生成高质量目标代码 (与目标机相关的优化)



- 三个部分会相互影响
  - 指令选择影响寄存器分配和指令调度
  - 寄存器分配要重用寄存器，减少访存
  - 指令调度提高指令级并行，会增大寄存器分配压力
  - 需在降低访存延迟和提高指令级并行性之间折中
  - 指令选择→前指令调度→寄存器分配→后指令调度



先后问题?

小结

- 为什么不在寄存器分配后才做指令调度?

virtual registers	physical registers
r1 = load(r10)	R1 = load(R1)
r2 = load(r11)	R2 = load(R2)
r3 = r1 + 4	R5 = R1 + 4
r4 = r1 – r12	R1 = R1 – R3
r5 = r2 + r4	R2 = R2 + R1
r6 = r5 + r3	R2 = R2 + R5
r7 = load(r13)	R5 = load(R4)
r8 = r7 * 23	R5 = R5 * 23
store (r8, r6)	store (R5, R2)

Too many artificial ordering constraints!!!

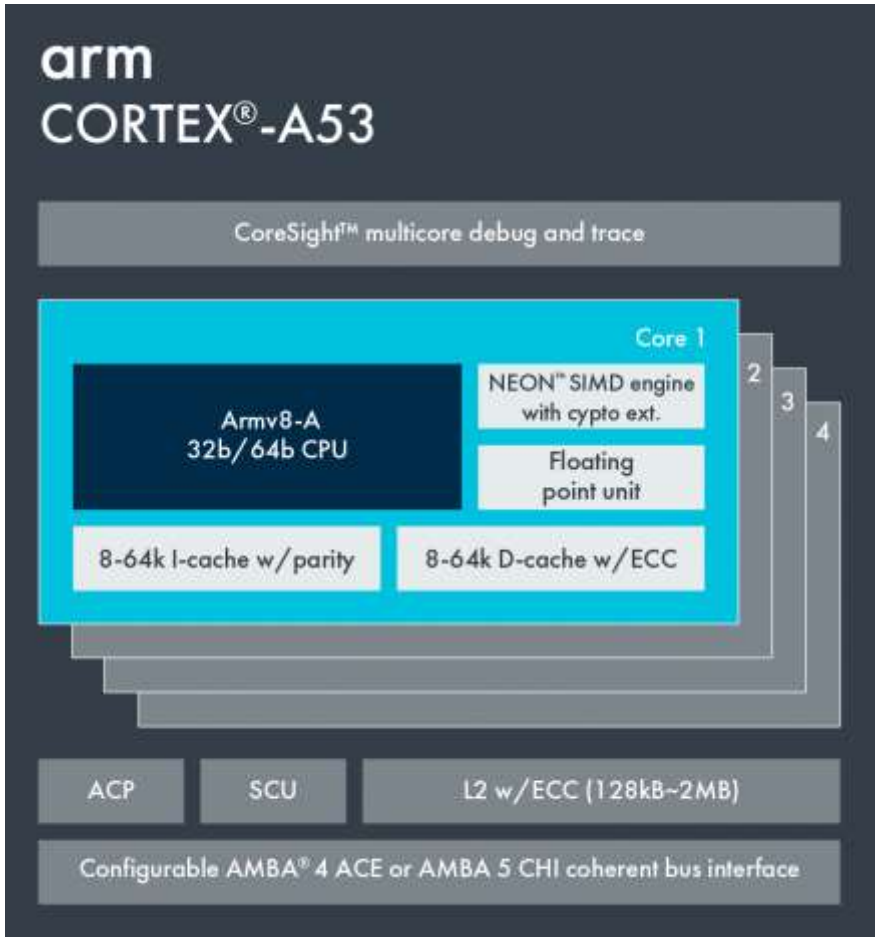
内容

- 编译后端功能介绍
  - 指令选择
  - 寄存器分配
  - 指令调度
- 面向ARM后端的代码生成介绍

比赛ARM平台

COMP CCRG Open MP

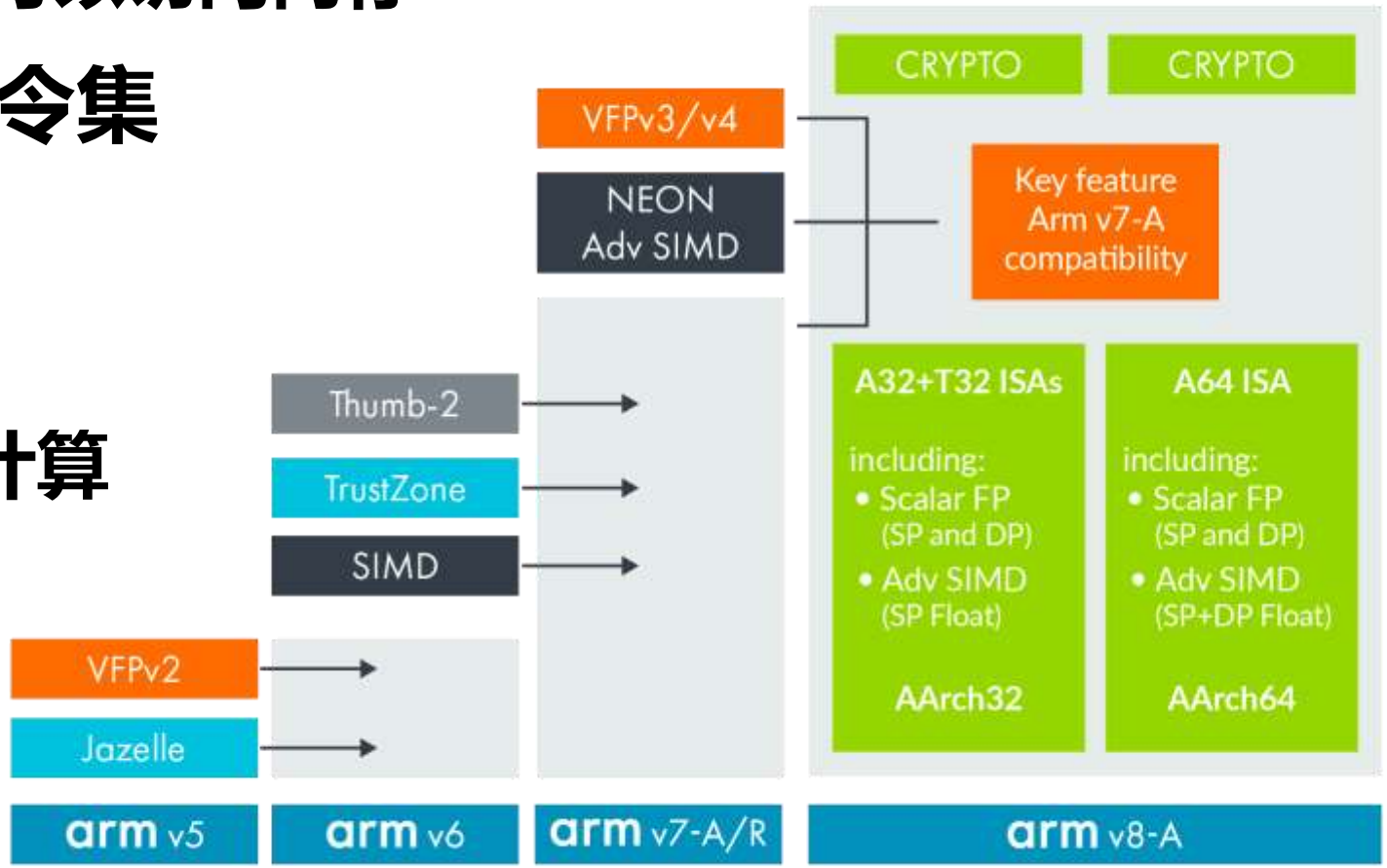
- 赛灵思 XCZU15EG ARM Cortex-A53 MPCore
  - ARMv8架构，4核心（隔离核心2、3用于性能评测）
  - 8-stage流水线，顺序执行
  - L1D：32KB，4路组相联
  - L1I：32KB，2路组相联
  - L2：1MB，所有核心共享，16路组相联
  - 内存：4GB DDR4
  - OS：Ubuntu 22.04，64位



https://developer.arm.com

ARMv8指令集架构

- RISC架构 (load/store架构)
  - 大部分指令处理寄存器中的数据，结果写回寄存器
  - 只有load/store指令可以访问内存
- ARMv8 AArch64指令集
  - 64位指令集
  - 支持FMA和NEON
  - 支持单、双精度浮点计算



ARMv8通用寄存器

- 通用寄存器: 31个，64位
  - 64位通用寄存器：X0~X30
  - 32位通用寄存器：W0~W30
  - 特殊用途寄存器
    - X29/W29(FP)： 64位/32位栈帧指针寄存器，指向栈底
    - X30/W30(LR)： 64位/32位链接寄存器，保存返回地址
    - XZR/WZR：64位/32位零寄存器
    - SP/WSP： 64位/32位栈指针寄存器，指向栈顶
    - PC：程序计数器
    - X8：间接结果地址寄存器（保存大型结构体在栈中的地址）
    - X16~X17：内部过程调用临时寄存器 (可以被函数破坏，调用者保护)
    - X18：平台寄存器（保留供平台ABI使用）

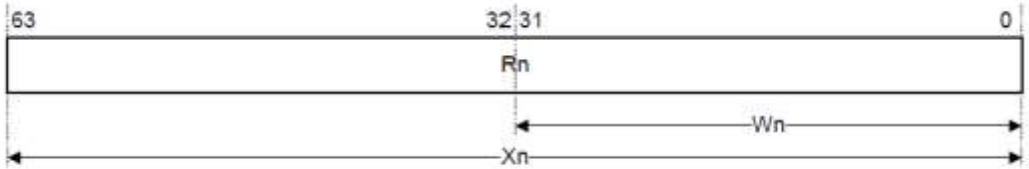


Figure B1-1 General-purpose register naming

ARMv8 FP/SIMD寄存器

COMP CCRG Open MP

- SIMD&FP寄存器: 32个, 128位
  - FP寄存器
    - 128位：Q0~Q31
    - 64位：D0~D31
    - 32位：S0~S31
    - 16位：H0~H31
    - 8位：B0~B31
  - SIMD寄存器
    - 128位向量: Vn.{2D, 4S, 8H, 16B}
    - 64位向量: Vn.{1D, 2S, 4H, 8B}

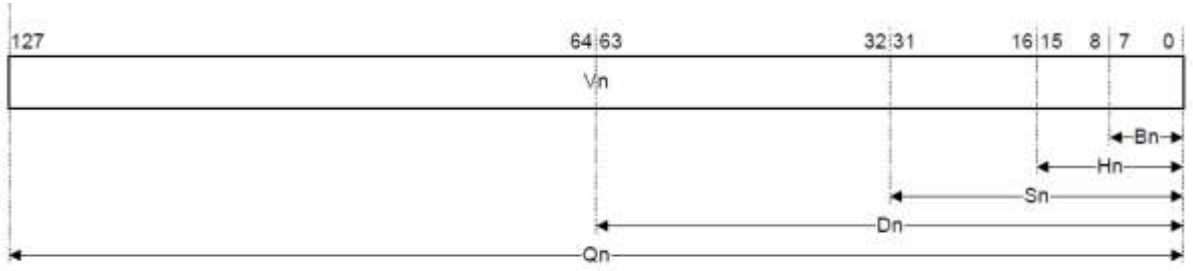
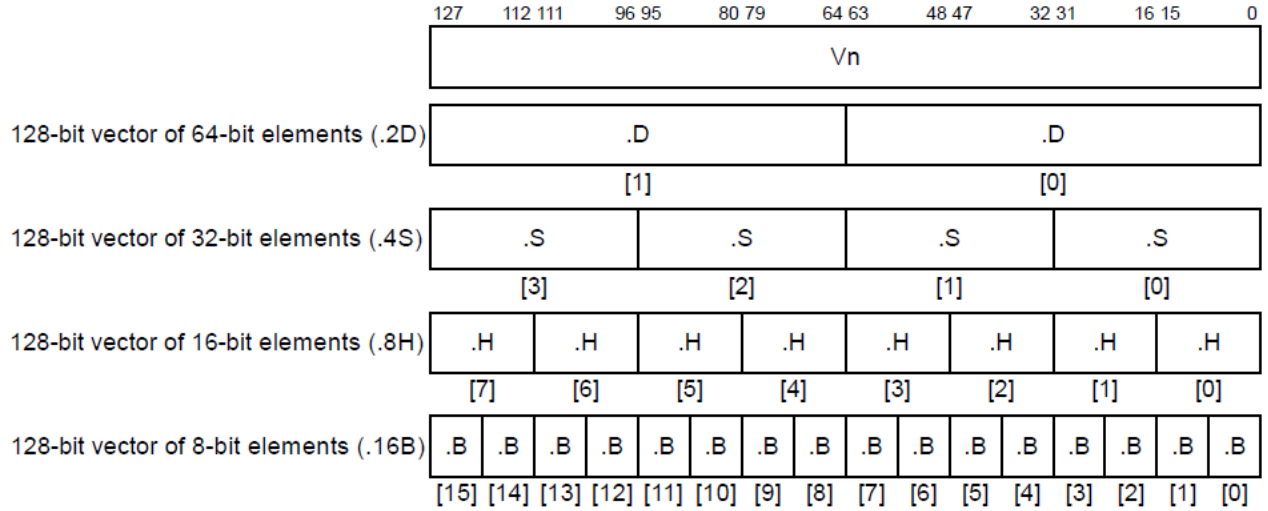


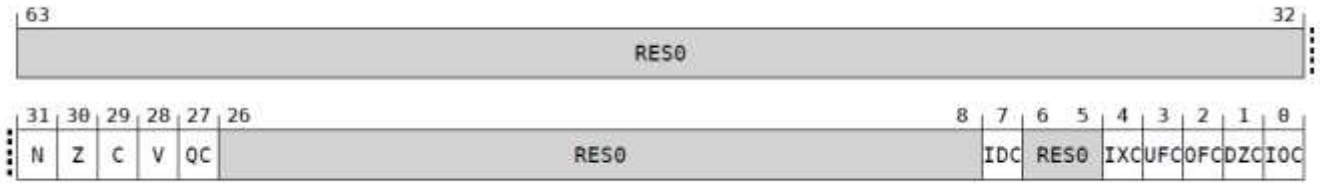
Figure B1-2 SIMD and floating-point register naming



ARMv8 NZCV条件标志

COMP CCRG Open MP

- PSTATE(处理器状态) / FPSR(浮点状态寄存器)
  - 管理整型指令/浮点指令的条件标志



  - 条件标志位
    - N：正负标志，N=1表示运算结果为负数，N=0表示运算结果为正数或零
    - Z：零标志，Z=1表示运算结果为零，Z=0表示运算结果为非零
    - C：进位标志，产生进位C=1，否则C=0
    - V：溢出标志，V=1表示有溢出，V=0表示无溢出

ARMv8条件执行

COMP CCRG Open MP

- 如果N|Z|C|V位与指令条件码匹配，则执行指令，否则不执行

Table F1-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

默认为AL(无条件执行)

ARMv8函数调用约定

- 遵循AAPCS (ARM Architecture Procedure Call Standard)
  - 整型参数传递
    - 前8个参数通过X0~X7传递，后续参数通过栈传递
    - 相比ARMv7增加了4个参数寄存器
    - 32位整型使用64位X0~X7的低32位，零扩展到64位
  - 整型返回值传递
    - 64位及以下：通过X0传递
    - 128位：通过X0(低64位)，X1(高64位)传递

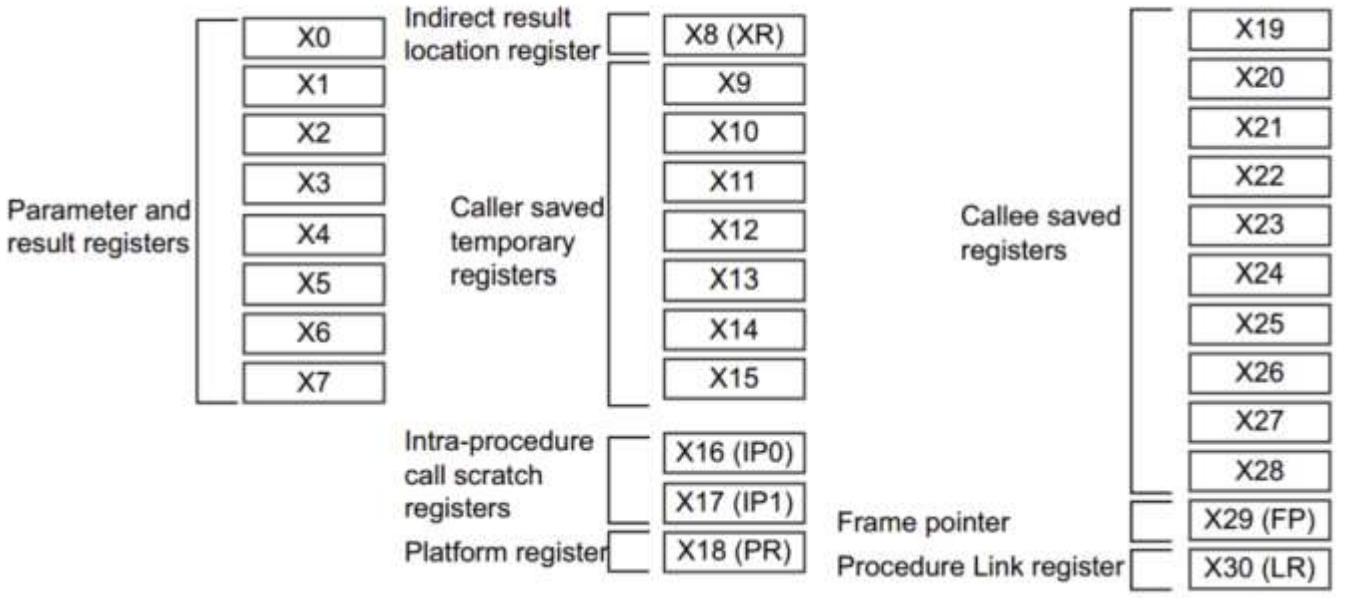
ARMv8函数调用约定

COMP CCRG Open MP

寄存器保护

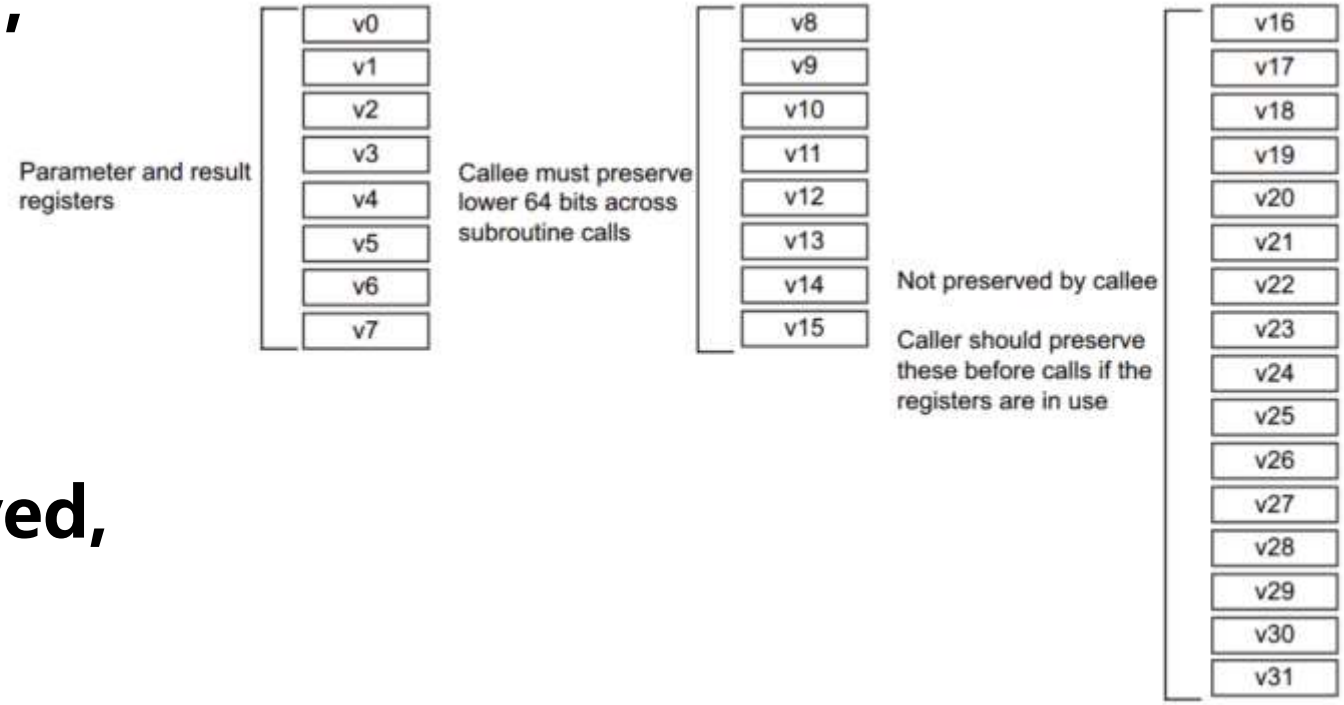
- 调用者保护 (caller-saved, call-clobbered)
  - 参数和返回值寄存器：X0~X7
  - 调用者保护临时寄存器：X9~X15
- 被调用者保护 (callee-saved, call-preserved)
  - X19~X29
  - FP(X29), LR(X30), SP

如Callee函数内部还有子函数调用，则需要保护LR，Callee返回到Caller函数时才能回到正确的返回地址



ARMv8函数调用约定

- 浮点参数和返回值传递
  - 通过V0~V7传递
- 寄存器保护
  - 调用者保护 (caller-saved, call-clobbered)
    - 参数和返回值寄存器：V0~V7
    - 调用者保护临时寄存器：V16~V31
  - 被调用者保护 (callee-saved, call-preserved)
    - V8~V15



汇编指令格式

- 格式和助记符 (mnemonic)

```
<opcode>{cond}{s} <Rd>, <Rn>, {,<op2>}
```

  - <opcode>: 操作码
  - {<cond>}: 指令条件执行的条件域，满足条件则执行指令，否则不执行 (可选)
  - {s}: s后缀，依据指令执行的结果更新CPSR，否则不更新 (可选)
  - <Rd>: 目的寄存器
  - <Rn>: 第一操作数，为寄存器
  - {<op2>}：第二操作数，可以是立即数、寄存器和寄存器移位操作数(可选)

汇编指令格式

- cond后缀
  - 测试条件标志位：测试指令执行前的标志位
- S后缀
  - 更新条件标志位：依据指令执行的结果改变标志位
  - 既有条件后缀又有S后缀，书写时S排在后面
- !后缀
  - 指令中的地址表达式有!后缀，基址寄存器中的地址值更新
  - 指令执行后基址寄存器中的地址值 = 指令执行前的值 + 偏移量

```
ADDEQS   X0, X1, X2     @当Z=1时执行指令(X1+X2->X0)，同时更新条件标志位
BNE   .Loop             @当Z=0时跳转到标号.Loop
STR   FP, [SP, #-4]!    @执行STR指令后，SP=SP-4
```

GNU汇编语法

- 汇编由一系列语句组成，每条语句包括三个可选部分

```
label: instruction @ comment
```

- label
  - 标号指示指令或数据(如const变量)的地址
  - 由点、字母、数字、下划线等组成
- instruction
  - 可以是汇编指令或伪指令
- 书写规范
  - ARM指令、伪指令、寄存器名可以全部为大写字母或全部为小写字母，但不可大小写混用

GNU汇编语法

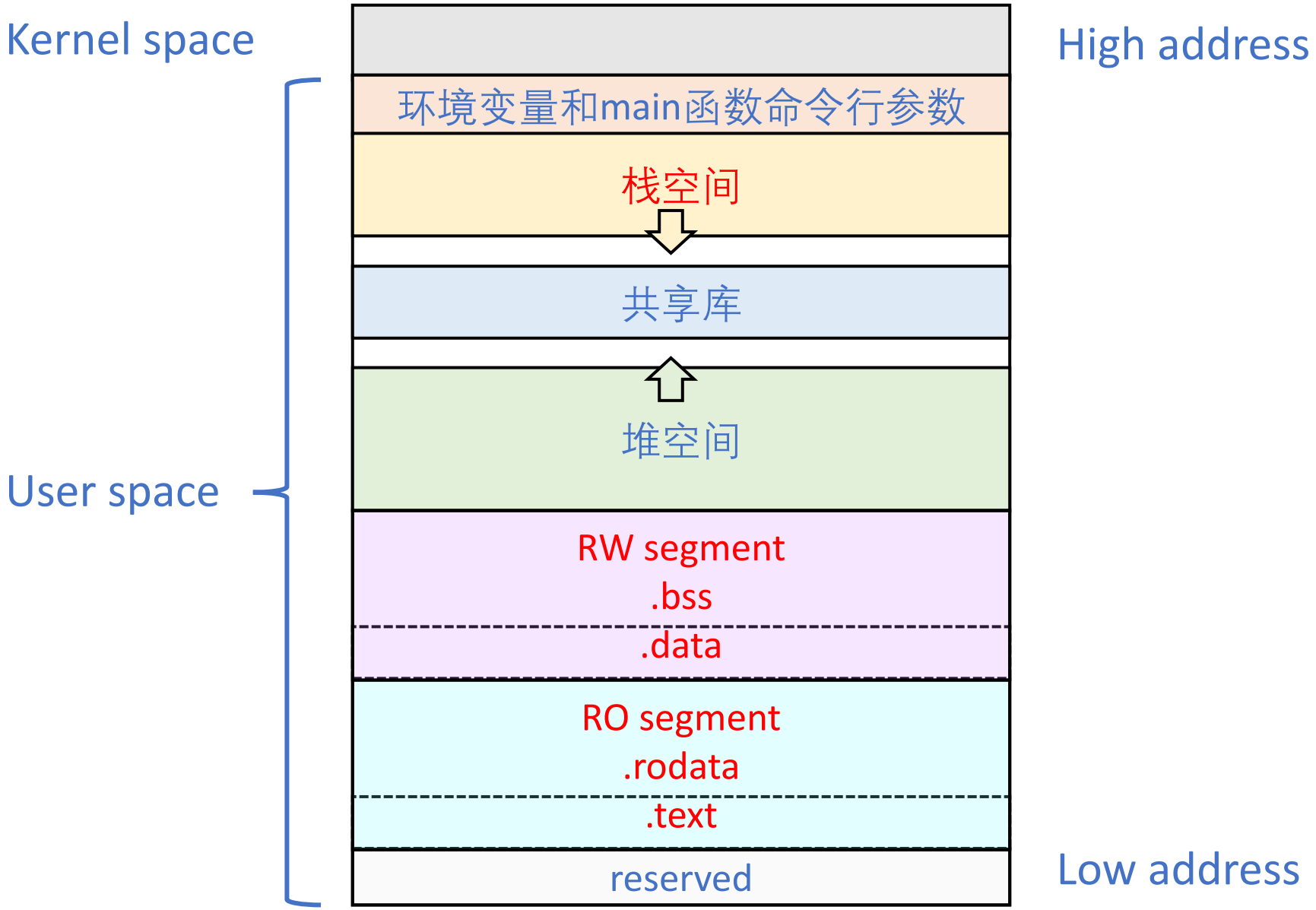
- 伪指令(assembler directives)
  - .arch：指示目标架构
  - .byte, .word, .long, .float, .string/.asciz/.ascii <expr>: 定义某种类型的数据
  - .align <n>, .p2align <n>: 通过填充字节，使当前位置按2^n字节对齐
  - .global <symbol>: 定义一个全局的符号
  - .local <symbol>: 定义一个局部的符号(未声明为.global的符号默认为局部的)
  - .type <symbol> <@function/@object>:指定一个符号的类型是函数类型或者是数据对象类型
  - .size <symbol> <size>: 指定一个符号的大小

GNU汇编语法

- 段 (sections)
  - 每个段以段名开始，以下一段名或者文件结尾结束
  - .text: 代码段
  - .data: 初始化数据段
    - 存放初始化的全局变量和静态变量
  - .bss: 未初始化数据段
    - 存放未初始化的全局变量和静态变量，以及初始化为0的全局变量和静态变量
    - 编译器会默认初始化为0
  - .rodata: 只读数据段
    - 存放const修饰的全局变量
  - .section <section_name> {,"<flag>" }: 定义一个段，指定段属性

```
14      .text
15      .global a
16      .data
17      .align  2
18      .type   a, %object
19      .size   a, 4
20  a:
21      .word   1
22      .text
```

进程内存空间


Kernel space
High address
环境变量和main函数命令行参数
栈空间
共享库
堆空间
User space
RW segment
.bss
.data
RO segment
.rodata
.text
reserved
Low address

函数调用栈

- ARM采用满减栈FD(Full Descending)，栈由高地址向下增长
- 每个函数对应一个栈帧，使用两个指针维护
  - FP：指向栈底
  - SP：指向栈顶 (指向最后一个入栈的数据)
- 函数调用时
  - 保存父函数 FP 和 LR（如果被调用函数还有子函数调用，则保存LR)
  - 设置被调用函数 FP 和 SP，开辟栈空间
    - 进入被调用函数后，通过父函数SP设置本函数的FP
    - 通过SP做减法开辟栈空间: SUB SP, SP #N
    - 本函数中FP保持不变，因此后续基于栈的访问操作可以基于FP实现
  - 恢复父函数FP, SP, LR，返回父函数

相关指令

- 现场保护和恢复指令
  - 多存储和多加载指令(可一次操作多个寄存器数据)
  - PUSH <{regs1, 2, 3}>: 将寄存器按regs3, 2, 1的顺序压入栈中
  - POP <{regs1, 2, 3}>: 按regs1, 2, 3的顺序从栈中恢复寄存器
- 流控指令
  - BL <Label>: 带返回的分支指令，实现函数调用
    - 跳转到标号处，并将返回地址(函数调用后下一条指令的地址，即当前PC值)保存在LR中
    - Caller在调用Callee时生成指令 BL <calleefunc>
  - BX LR: 跳转到LR指定返回地址处
    - Callee在返回Caller时生成指令 BX LR
  - B指令：实现基本块之间的跳转，如if-then-else
    - 无条件跳转：使用B指令
    - 条件跳转：B指令与条件码的结合(如BEQ, BNE等)

文档和工具

- ARM指令集手册、Programmer's guide
- GCC交叉编译工具链
  - https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads
- Qemu模拟器
- llvm lli解释执行

预祝取得好成绩!

 j.shen@nudt.edu.cn