并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室

Lab three Code Generation

实验三：代码生成

内容

1. 实验介绍
2. ARMv8指令集架构
3. ARM汇编基础
4. 函数调用
5. 实验步骤

1 实验介绍

- 实验任务
  - 实现SysY编译器后端(基础实现)，完成代码生成功能
  - 从中间语言生成ARMv8汇编指令
- 实验目标
  - 通过实验，掌握编译后端代码生成的基本方法
- 实验方法
  - 基于宏扩展的指令选择方法
  - 自顶向下逐条翻译

1 实验介绍

实验内容

- 实现SysY编译器后端的代码生成
  - 从IR Module开始，自顶向下遍历
  - 构建相关符号表，逐条翻译生成ARM汇编代码

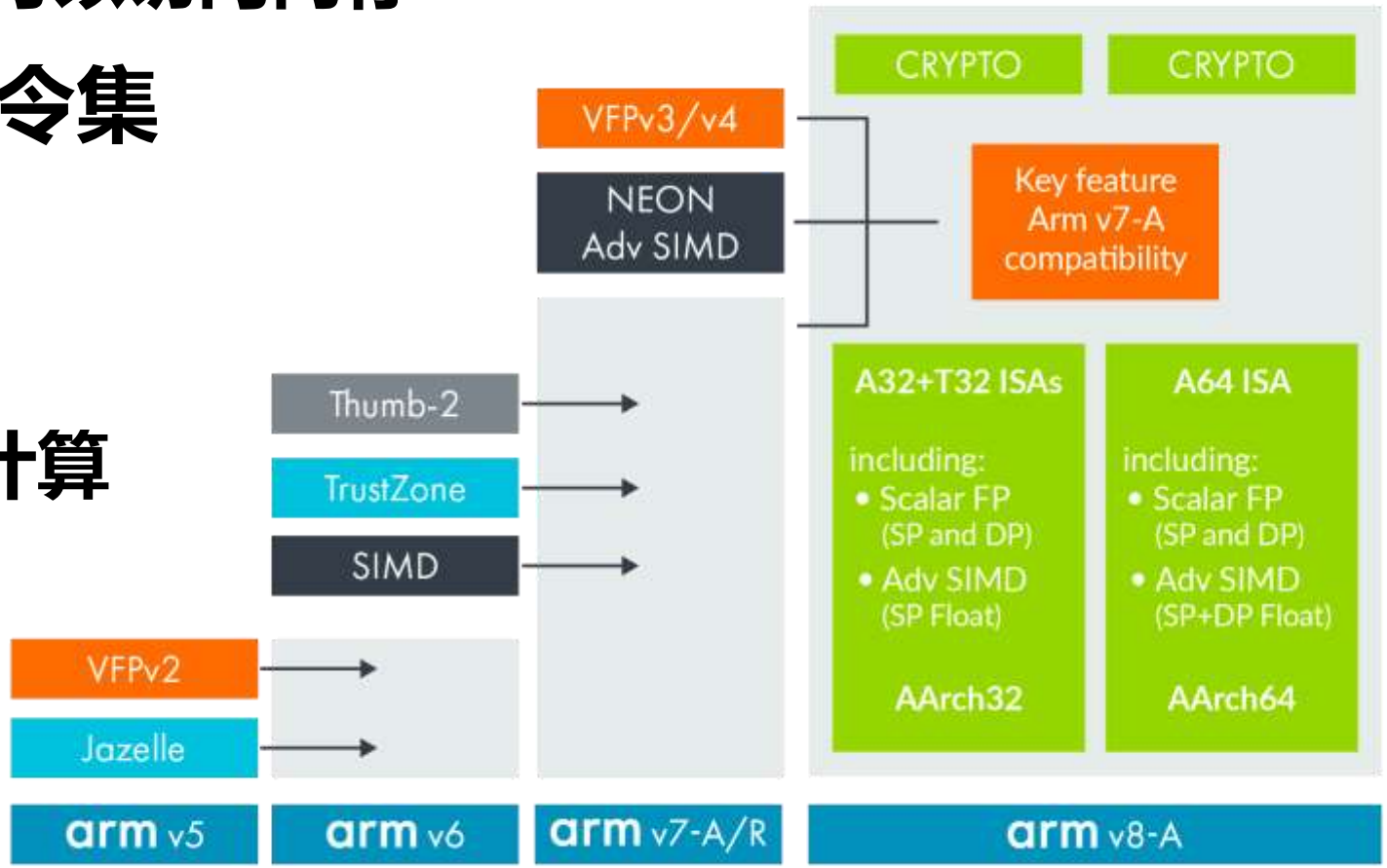
实验要求

- 能够成功生成SysY测试程序对应的汇编文件
- 利用GCC交叉编译工具链生成二进制文件，在Qemu模拟器上运行，得到正确的结果

内容

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



2.1 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

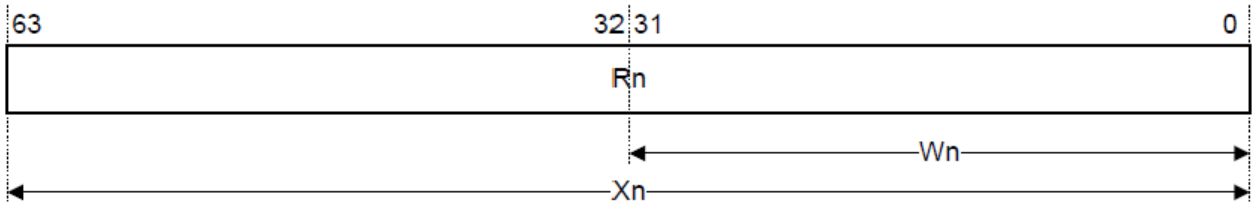


Figure B1-1 General-purpose register naming

  - x0-x7：参数/结果寄存器
  - x8：间接结果地址寄存器（保存大型结构体在栈中的地址，无此需求则可用于临时寄存器）
  - x9-x15：临时寄存器
  - x16~x17：内部过程调用临时寄存器 (无此需求则可用于临时寄存器)
  - x18：平台寄存器(供平台ABI使用，大部分编译器不使用其作为通用寄存器)
  - x19~x28：被调用者保护寄存器
  - x29(FP)： 帧指针寄存器
  - x30(LR)： 链接寄存器，保存返回地址

2.1 ARMv8通用寄存器

特殊用途寄存器

- 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，64位

2.2 ARMv8 FP/SIMD寄存器

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

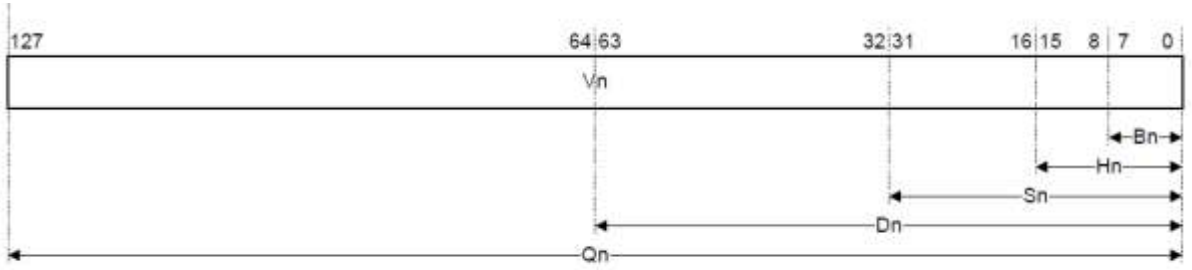
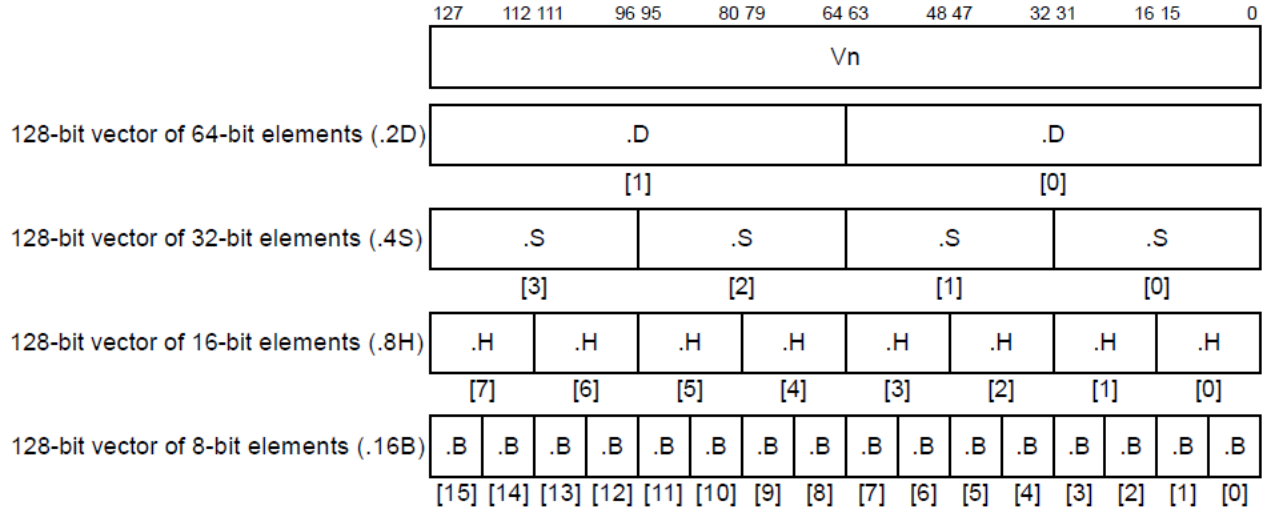


Figure B1-2 SIMD and floating-point register naming

SIMD寄存器

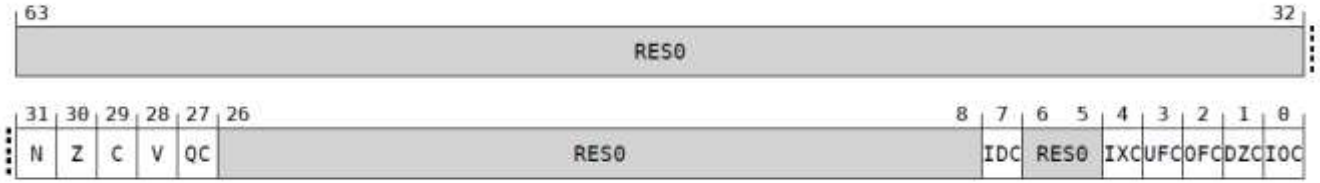
- 128位向量: vn.{2d, 4s, 8h, 16b}
- 64位向量: vn.{1d, 2s, 4h, 8b}



2.3 ARMv8 NZCV条件标志

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

2.4 ARMv8条件执行

- 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 C1-1 Condition codes

cond	Mnemonic	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 or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	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	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
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 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

默认为AL(无条件执行)

2.5 ARMv8函数调用约定

- 遵循AAPCS64 (ARM Architecture Procedure Call Standard AArch64)
  - 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

2.5 ARMv8函数调用约定

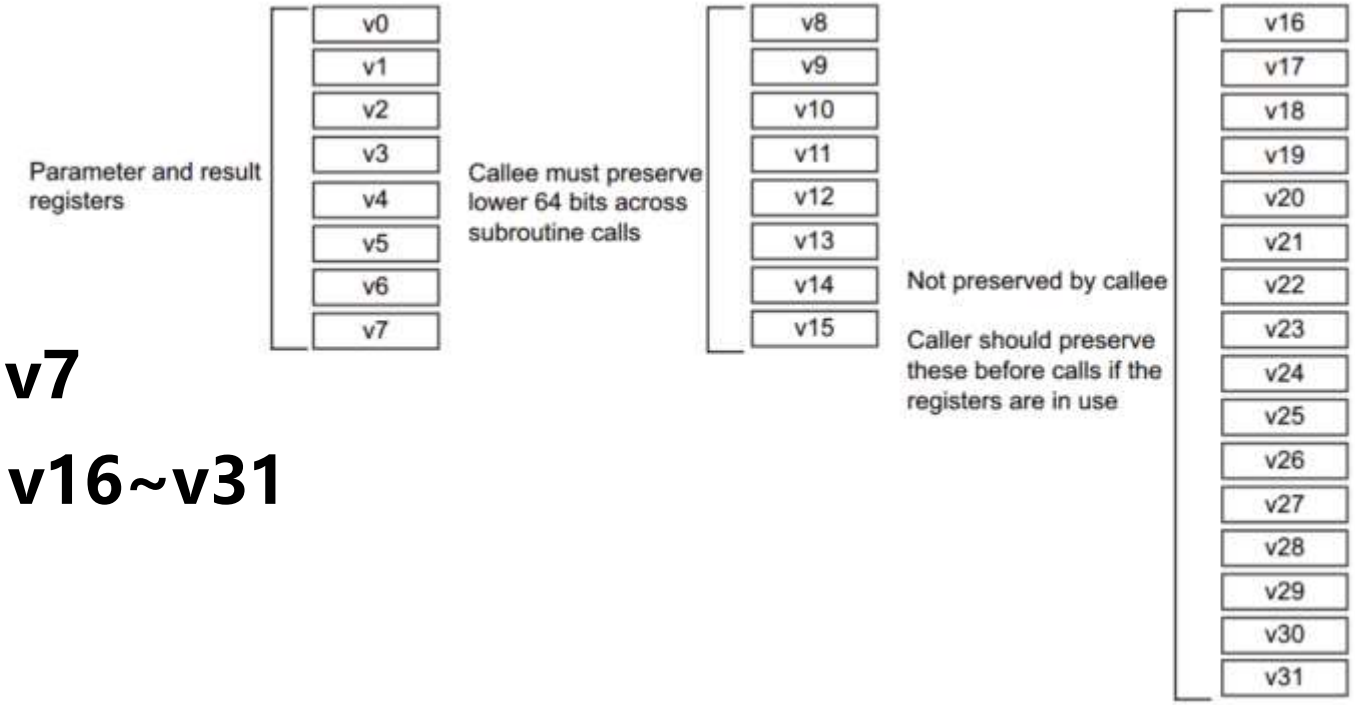
寄存器保护

- 调用者保护 (caller-saved, call-clobbered)
  - 参数和结果寄存器：x0~x7
  - 调用者保护临时寄存器：x9~x15
  - x8，x16~x17，x18也是调用者保护
- 被调用者保护 (callee-saved, call-preserved)
  - x19~x28
  - x29(fp), x30(lr)
  - sp

如Callee函数内部还有子函数调用（Callee非叶子函数），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

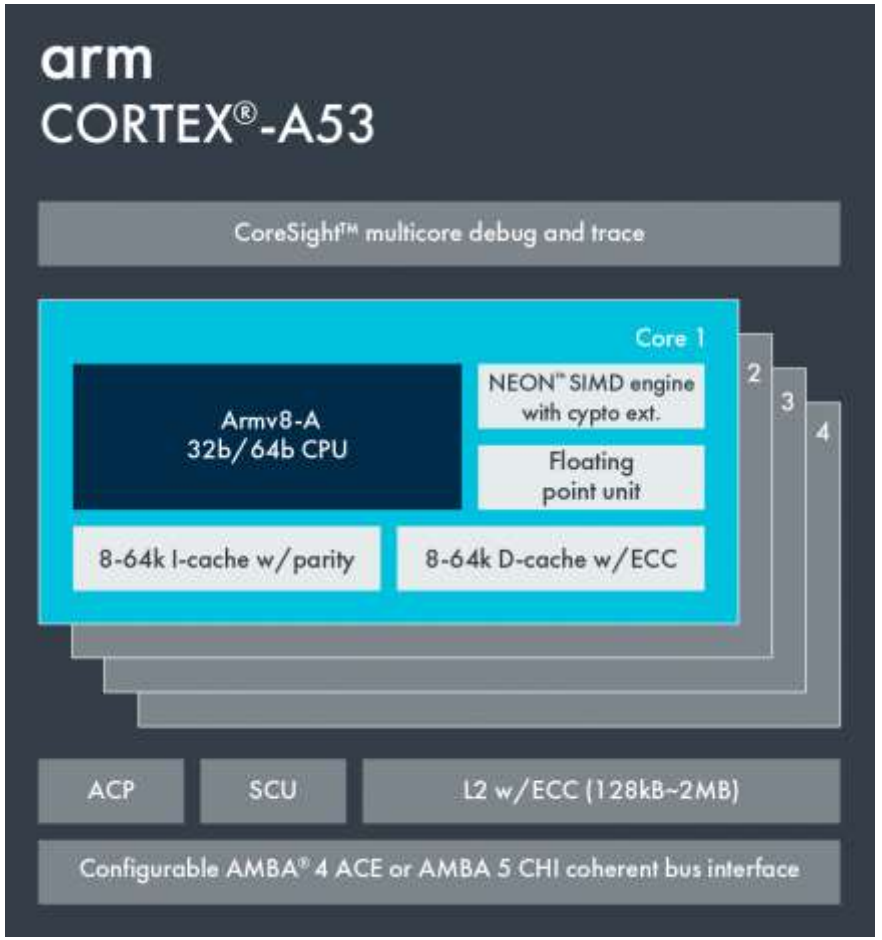
2.5 ARMv8函数调用约定

- 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
    - 参数和结果寄存器：v0~v7
    - 调用者保护临时寄存器：v16~v31
  - 被调用者保护
    - v8~v15



2025编译竞赛ARM平台

- 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

内容

3.1 汇编指令格式

格式和助记符 (mnemonic)

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

- <opcode>: 操作码
- {cond}: cond后缀，条件执行的条件域，满足条件则执行，否则不支持，部分指令支持cond后缀 (可选)
- {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)

3.1汇编指令格式

- S后缀
  - 更新条件标志位：依据指令执行的结果改变标志位
- cond后缀
  - 测试条件标志位：测试指令执行前的标志位，匹配则执行
  - 部分指令（分支、条件传送、条件比较）支持条件执行
- !后缀（访存指令相关）
  - 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

```
adds    x0, x1, x2          @根据结果更新条件标志位
bne     .Loop               @当Z=0时跳转到标号.Loop
stp     x29, x30, [sp, -16]!    @执行stp指令后，sp=sp-16
```

3.2 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

3.2 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

3.2 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

```
    .arch armv8-a
    .file   "demo-globalval.c"
    .text
    .global a
    .data
    .align  2
    .type   a, %object
    .size   a, 4
a:
    .word   1
```

3.2 GNU汇编语法

- 函数定义

  <函数名>:

  <函数体>

- 包含
  - 相关伪指令
  - Prologue
  - main body
  - Epilogue
  - 返回指令

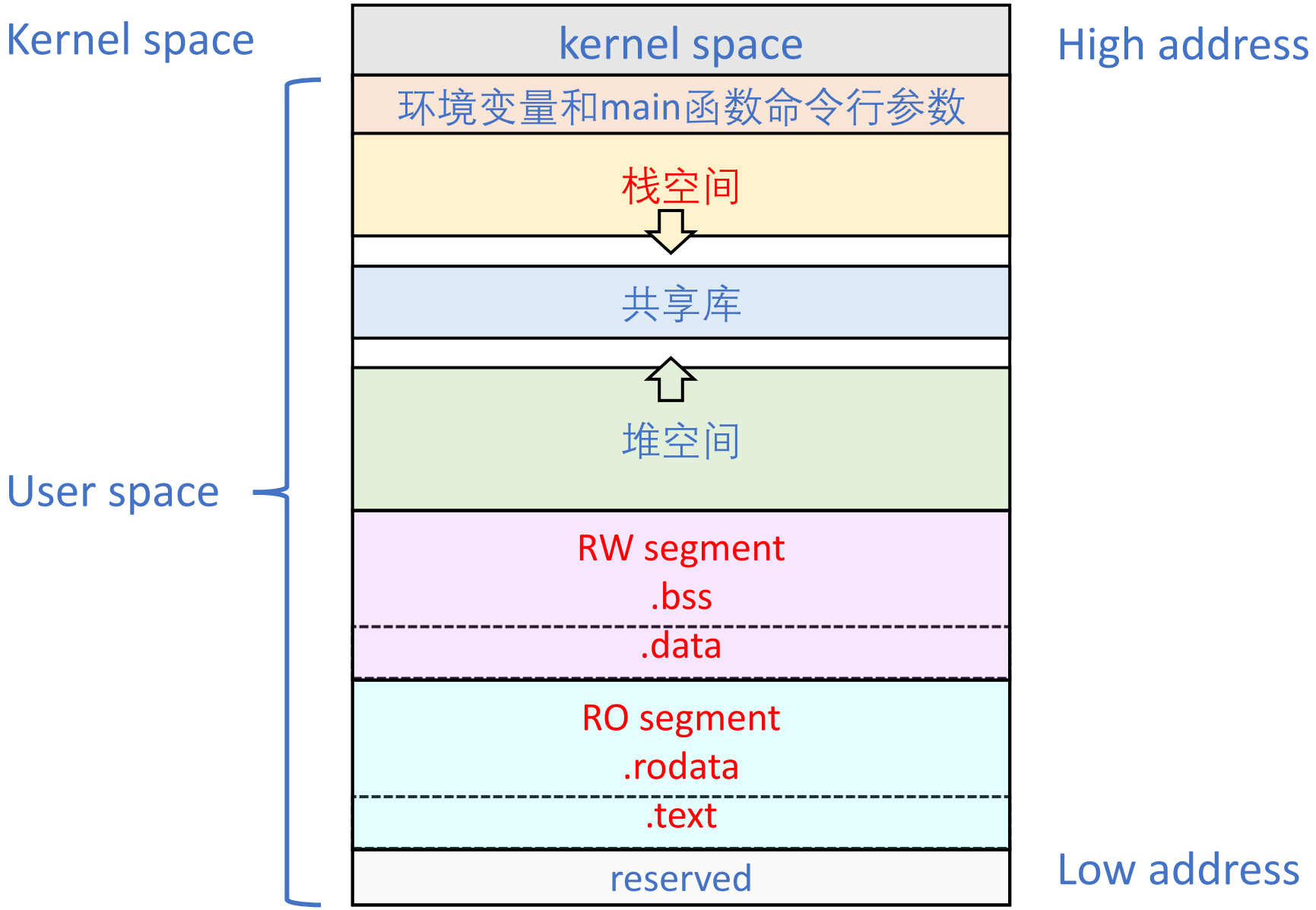
```
    .arch armv8-a
    .file   "demo-func-call.c"
    .text
    .align  2
    .global func
    .type   func, %function
func:
    nop
    ret
    .size   func, .-func
    .align  2
    .global main
    .type   main, %function
main:
    stp x29, x30, [sp, -16]!
    mov x29, sp
    bl  func
    mov w0, 0
    ldp x29, x30, [sp], 16
    ret
    .size   main, .-main
```

```
void func(){
    ;
}

int main(){
    func();
    return 0;
}
```

内容

4.1 进程内存空间

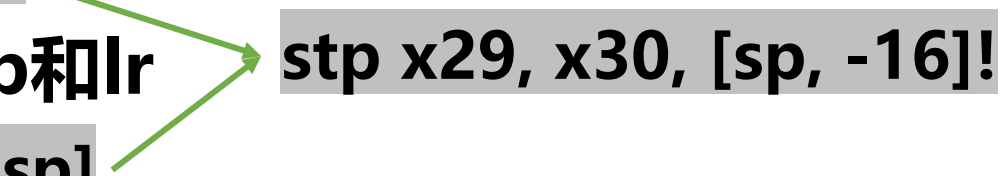

Kernel space
User space
kernel space
High address
环境变量和main函数命令行参数
栈空间
共享库
堆空间
RW segment
.bss
.data
RO segment
.rodata
.text
reserved
Low address

4.2 ARMv8函数调用栈

- ARMv8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：帧指针
  - sp：栈指针，指向栈顶 (指向最后一个入栈的数据)
  - sp做减法，栈增长；sp做加法，栈收缩
- ARMv8栈帧对齐方式
  - 分配栈桢时按16字节对齐: 栈帧大小是16字节的倍数

4.3 ARMv8函数调用

- prologue(进入callee函数后)
  - 开辟callee函数栈空间
    - sub sp, sp, #16
  - 保存caller函数fp和lr
    - stp x29, x30, [sp]
    - 保存到sp指向的栈槽，x29存入低地址，x30存入高地址
  - 保存callee sp到fp
    - mov x29, sp
    - 原因是sp在callee执行过程中可能会变化
    - 相当于fp指向保存caller函数fp的栈槽，支持堆栈回溯
  - callee是叶子函数可以不维护栈帧，只执行第1步开辟栈空间



4.3 ARMv8函数调用

- main body
  - 基于sp存储和加载栈上的变量
    - str w0, [sp, 28]
    - ldr w0, [sp, 28]
  - sp可能在callee执行中更新，也可能不变

4.3 ARMv8函数调用

- Epilogue(退出callee函数前)
  - 如果sp有更新，则恢复sp
    - mov sp, x29
  - 恢复caller函数fp和lr
    - ldp x29, x30, [sp]
    - sp指向的栈槽保存了caller fp和lr
  - 回收栈空间(sp做加法)
    - add sp, sp, #16
  - 返回caller函数
    - ret
    - 跳转到lr指定返回地址处，继续执行

ldp x29, x30, [sp], 16

4.4 函数调用相关指令

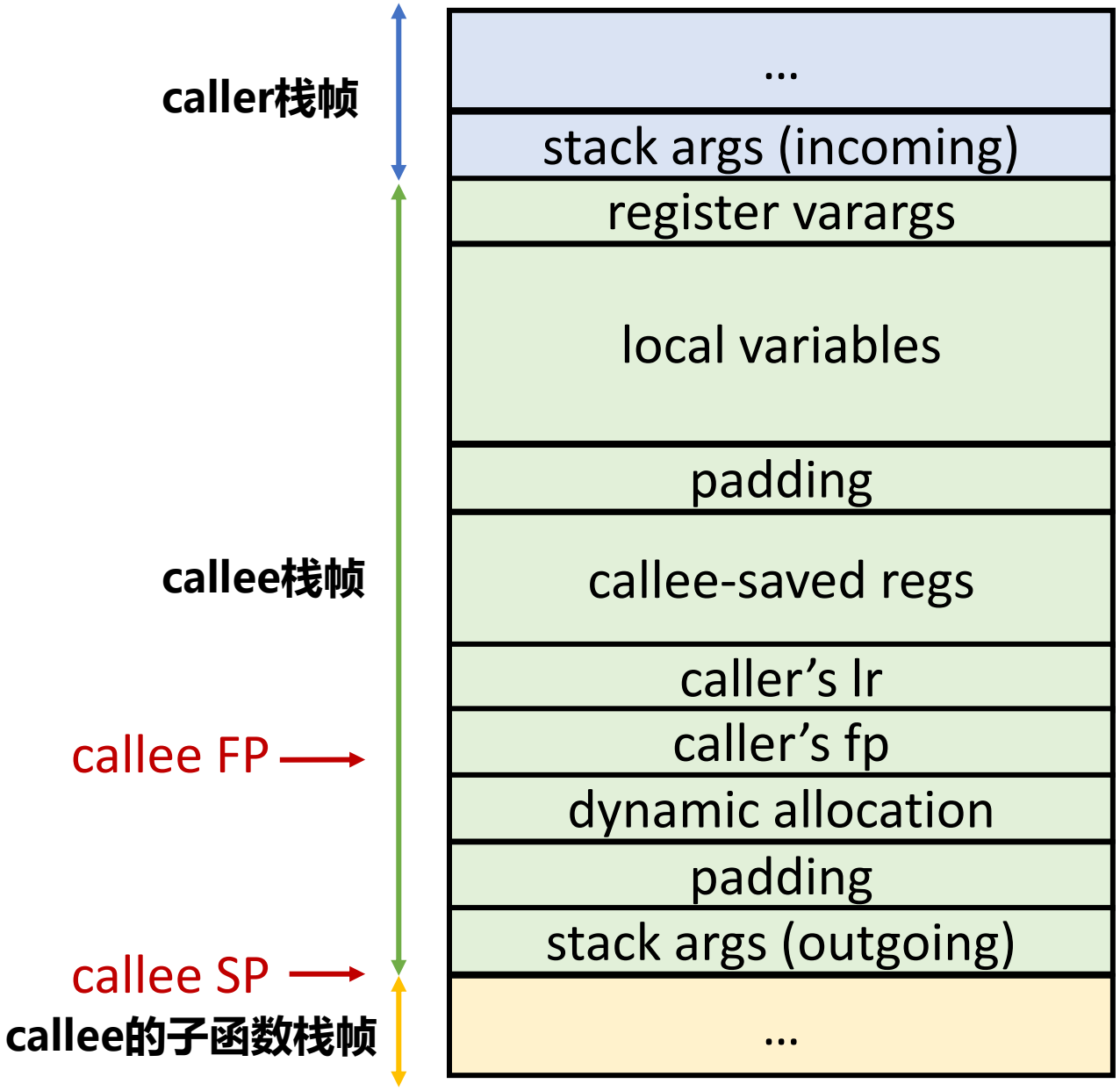
- ldp和stp指令
  - 成对加载/存储指令(一次操作2个寄存器)
  - 保护现场（压栈）
    - stp x29, x30, [sp, -16]!
    - 将x29(fp),x30(lr)存入栈上[sp-16]位置
    - x29存入低地址，x30存入高地址
    - 压栈后sp更新为sp-16
  - 恢复现场（出栈）
    - ldp x29, x30, [sp], 16
    - 从当前sp指向的栈槽读取数据，低地址数据存入x29，高地址数据存入x30
    - 弹栈后sp=sp+16

4.4 函数调用相关指令

- bl指令
  - 带返回的分支指令 bl <Label>
  - 实现函数调用
  - 跳转到标号处，并将返回地址(函数返回后下一条指令的PC值)保存到lr中
- ret指令 ret
  - 实现返回到caller函数
  - 根据lr的值设置pc寄存器，跳转到callee返回后的下一条指令继续执行
- b指令
  - 无条件跳转指令 b <Label>
  - 可用于尾递归优化

```
    .arch armv8-a
    .file   "demo-func-call.c"
    .text
    .align  2
    .global func
    .type   func, %function
func:
    nop
    ret
    .size   func, .-func
    .align  2
    .global main
    .type   main, %function
main:
    stp x29, x30, [sp, -16]!
    mov x29, sp
    bl  func
    mov w0, 0
    ldp x29, x30, [sp], 16
    ret
    .size   main, .-main
```

4.5 ARMv8函数栈布局



4.6 ARMv8参数传递

- arg0~arg7
  - caller将参数从caller栈上加载到参数寄存器w0~w7
  - callee将w0~w7存入callee栈上
- argX(X>=8)
  - caller将参数从caller栈上加载到栈顶[sp],[sp, 8],[sp, 16]...
  - callee通过[sp, 偏移]获取caller栈帧栈顶的参数
  - 通过栈传递的参数按8字节对齐（简化ldr/str）

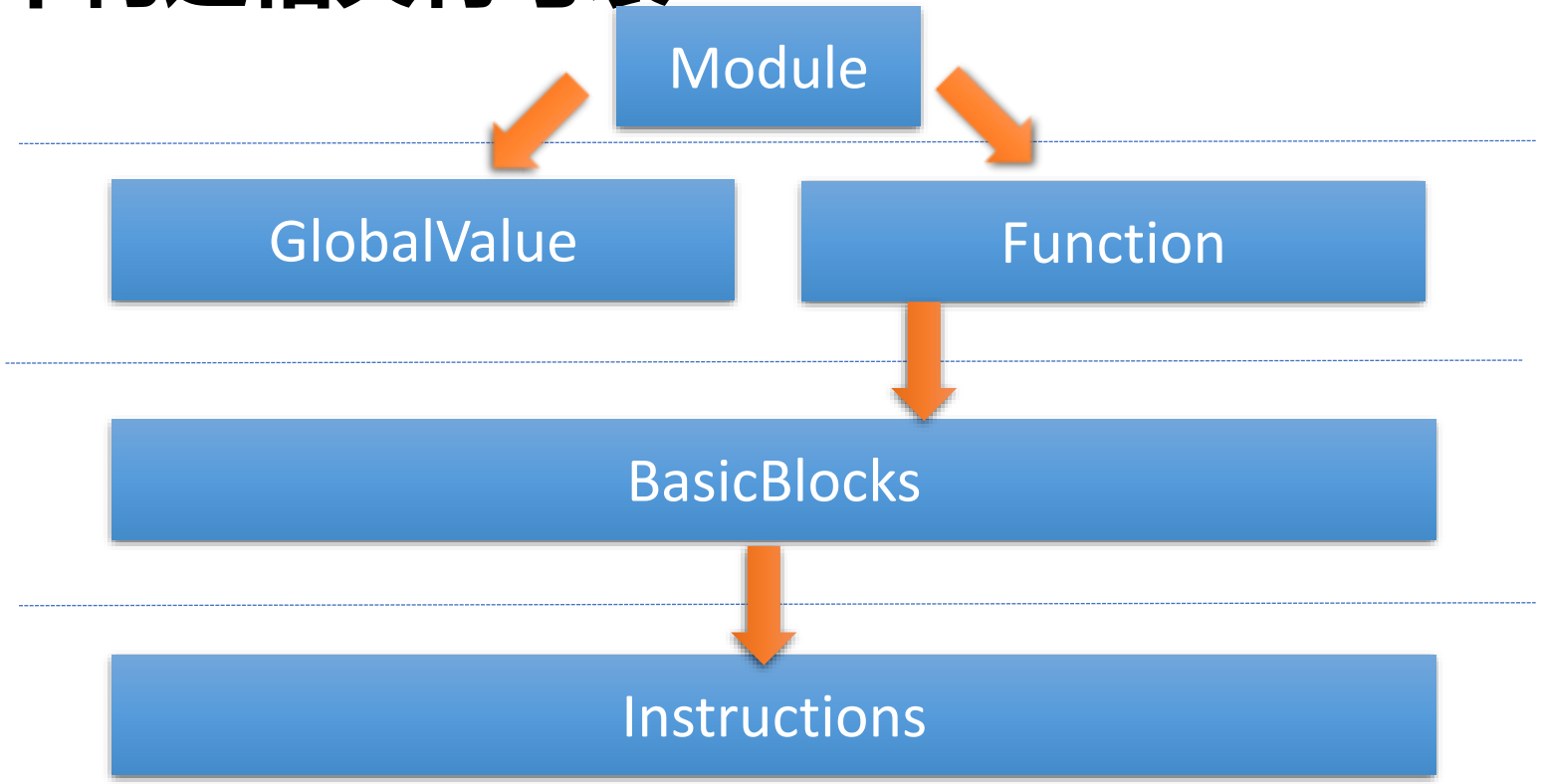
```
caller:
        sub sp, sp, #80
        ...
        ldr    w0, [sp, 40]  @ 从栈上取参数
        str    w0, [sp, 8]   @ 加载到栈顶+8
        ldr    w0, [sp, 44]  @ 从栈上取参数
        str    w0, [sp]      @ 加载到栈顶
```

```
callee:
        sub sp, sp, #48
        ...
        ldr w0, [sp, 48]  @ 访问caller栈帧
        add w0, w1, w0
        ldr w1, [sp, 56]  @ 访问caller栈帧
        add w0, w1, w0
```

内容

5.1 实验思路

- 从IR Module开始，自顶向下遍历
- 将IR逐条翻译成对应的汇编代码（宏扩展的指令选择方法）
- 遍历过程中构建相关符号表



5.2 为所有Function生成代码

COMP CCRG Open MP

- 遍历所有Function，在每个Function内部
  - 做寄存器分配(本实验考虑简化的寄存器模型)
  - 为该Function使用的每个GlobalValue生成代码
  - 检查该Function是否有子函数调用，获取参数个数
  - 生成本函数的FunctionLabel以及相关伪指令
  - 生成该Function的prologue
  - 遍历该Function的所有BasicBlock
  - 生成该Function的epilogue

5.3 为所有GlobalValue生成代码

- 遍历所有GlobalValue，不同类型的GlobalValue放在不同的段
  - 已初始化全局变量：.data
  - 未初始化全局变量：.bss
  - const全局变量：.rodata

```
int a = 0x1234;

int main(void){

    return a;
}
```

```
    .text
    .global a
    .data
    .align  2
    .type   a, %object
    .size   a, 4
a:
    .word   4660
    .text
    .align  2
    .global main
    .type   main, %function
main:
    adrp    x0, a
    add x0, x0, :lo12:a
    ldr w0, [x0]
    ret
    .size   main, .-main
```

- 构建GlobalValueTable符号表
  - 例如，记录每个GlobalValue被哪些Instruction使用
- 在Function内使用GlobalValue
  - adrp：计算GlobalValue的高位地址（页基地址）
  - add：加上页内偏移得到完整64位绝对地址
  - ldr：取GlobalValue的值

5.4 简化的寄存器模型

- 所有LocalValue的最新副本保存在栈上
- 维护StackTable符号表，记录每个LocalValue在栈上的位置
- LocalValue的使用和定值
  - 每次使用前，从栈上加载到寄存器
  - 每次定值后，从寄存器存储回栈上，然后释放所占用的寄存器

5.5 为所有BasicBloack生成代码

- 遍历一个Function内所有BasicBlock，在每个BasicBlock内部
  - 生成该BasicBlock的Label
  - 遍历该BasicBlock里的每一条IR，生成对应汇编代码
  - 采用宏扩展的指令选择方法 (one-by-one translation)

5.6 为每一条指令生成代码

- UnaryInst和BinaryInst
  - 获得指令的Operand和指令类型，生成相应指令
- AllocaInst
  - 根据指令的Operand分配栈空间，将基于SP的偏移记录在StackTable符号表中
- LoadInst
  - 以SP为基址，根据符号表中的偏移，生成1条ldr指令
- StoreInst
  - 以SP为基址，根据符号表中的偏移，生成1条str指令

5.6 为每一条指令生成代码

COMP CCRG Open MP

- UncondBrInst
  - 获得跳转目标BasicBlock的Label，生成一条无条件跳转指令(B <BBLabel>)
- CondBrInst
  - 该指令一定位于CMP指令之后，因此首先获得条件码
  - 生成对应的条件跳转指令跳转到Then Block (B<Cond> <ThenLabel>), 还需要生成无条件跳转指令跳转到Else Block (B <ElseLabel>)
  - SSA：通过内存读写(ldr/str)来解决
- ReturnInst
  - 当本函数有返回值时，生成将返回值保存到x0/w0的相关指令

5.6 为每一条指令生成代码

- CallInst
  - 如果被调用函数有参数，生成通过w0~w7进行传参的相关指令
  - 如果参数个数大于8，还需生成通过栈传参的相关指令
    - Caller将参数压入栈顶 (Callee通过SP+偏移获取)
  - 生成跳转指令 (BL <FunctionLabel>)
  - 如果被调用函数有返回值，在CallInst后，生成通过w0获取返回值并存入栈中的相关指令

5.7 初始化局部变量和常数

- 常数在指令可表示立即数范围内
  - 直接使用
    - mov w3, 1
    - add w3, w3, #8
- 常数超出可表示立即数范围（大立即数）
  - 通过mov或movz + movk指令加载常数

```
@ x0=0x123456789ABCDEF0

mov     x0, 57072   @ 57072 = 0xdef0
movk    x0, 0x9abc, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
```

文档和工具

- ARM指令集手册、Programmer's guide，AAPCS64
- GCC交叉编译工具链
  - https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads
- Qemu模拟器