

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. ARMv7指令集架构
3. ARM汇编基础
4. 实验步骤

1 实验介绍

■ 实验任务

- ⊕ 实现SysY编译器后端，完成代码生成功能
- ⊕ 从中间语言生成ARMv7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

■ 实验原理

- ⊕ 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

- ⊕ 能够生成SysY测试程序对应的汇编文件
- ⊕ 利用gcc汇编链接生成二进制文件，在Qemu模拟器/树莓派上运行，结果正确
- ⊕ 按分组完成实验

内容

1. 实验介绍
2. ARMv7指令集架构
3. ARM汇编基础
4. 实验步骤

2.1 ARMv7指令集架构

- 32位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

- 两个基本指令集

- ⊕ ARM指令集： 32bits
 - ⊕ Thumb指令集： 16bits / 32bits

2.1 ARMv7指令集架构

■ 指令分类

流控指令 → Branch instructions

数据处理指令 → Data-processing instructions

程序状态寄存器访问指令 → Status register access instructions

访存指令 → Load and store instructions

 Load Multiple and Store Multiple instructions

 Miscellaneous instructions

异常产生指令 → Exception-generating instructions

协处理器指令 → Coprocessor instructions

 Floating-point load and store instructions

 Floating-point register transfer instructions

浮点数据处理指令 → Floating-point data-processing instructions

2.2 常用指令

■ 数据处理指令

add r2, r0, r1	//r2<-r0+r1
sub sp, sp, #32	//sp<-sp-#32
mla r3, r0, r1, r2	//r3<-r0*r1+r2
cmp r0, r1	//比较r0和r1

■ 访存指令

ldr r2, [r1, #4]	//r2<-mem(r1+4)
str r2, [r1, #-8]	//mem(r1-8)<-r2
push {fp}	//fp压栈
pop {fp}	//fp弹栈

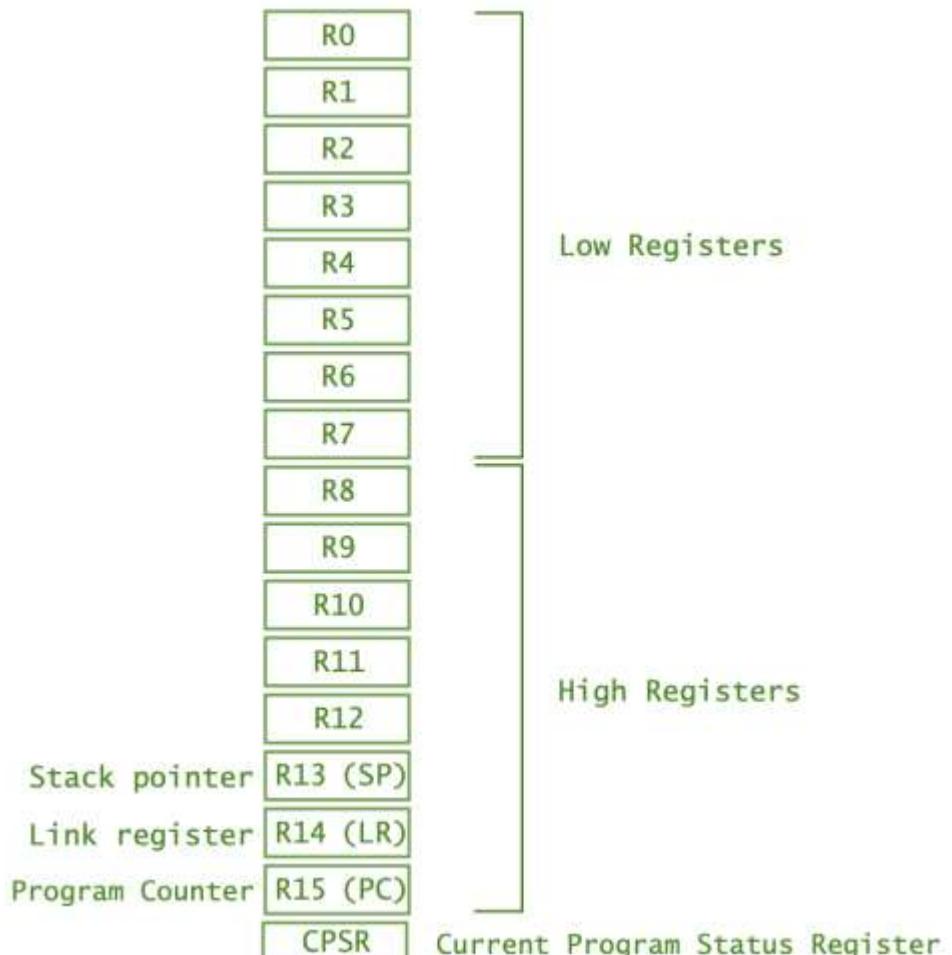
■ 流控指令

b func	//跳转到func
bl func	//跳转到func, 保存返回地址到lr
bx lr	//跳转到lr指定返回地址处

2.3 寄存器

■ 通用寄存器

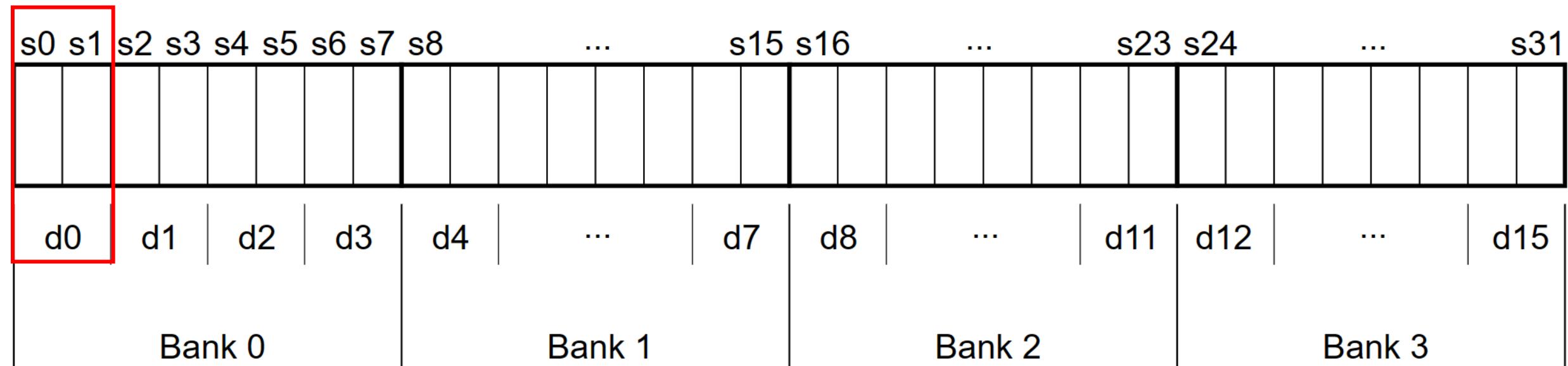
- ⊕ R0-R3: 传递函数参数和传出函数返回值
- ⊕ R4-R10: 通用寄存器
- ⊕ R11: FP (栈帧指针寄存器, 指向栈底)
- ⊕ R12: IP (内部调用暂时寄存器)
- ⊕ R13: SP (栈指针寄存器, 指向栈顶)
- ⊕ R14: LR (链接寄存器, 保存返回地址)
- ⊕ R15: PC (程序计数器)



2.3 寄存器

■ 浮点寄存器

- ⊕ 32个单精度浮点寄存器(single-precision registers): s0 to s31
- ⊕ 16个双精度浮点寄存器(double-precision registers): d0 to d15



2.3 寄存器

■ CPSR(当前程序状态寄存器)



⊕ 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
- **Q**: DSP运算指令是否发生溢出

2.4 条件执行

- 如果CPSR的N|Z|C|V位与指令条件码匹配，则执行指令，否则不执行

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	$N \neq V$
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 $N \neq V$
1110	None (AL) ^d	Always (unconditional)	Always (unconditional)	Any

默认为AL(无条件执行)

内容

1. 实验介绍
2. ARMv7指令集架构
3. ARM汇编基础
4. 实验步骤

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>}: 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)

3.1 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 R0, R1, R2	⊕当Z=1时执行指令($R1+R2 \rightarrow R0$)，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， $sp=sp-4$

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
- ⊕ **.arm, .thumb**: 表示随后的代码是32位arm指令集或16位thumb指令集
- ⊕ **.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

```

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

```

3.2 GNU汇编语法

■ 寄存器命名约定

⊕ FP: R11, SP: R13, LR: R14, PC: R15

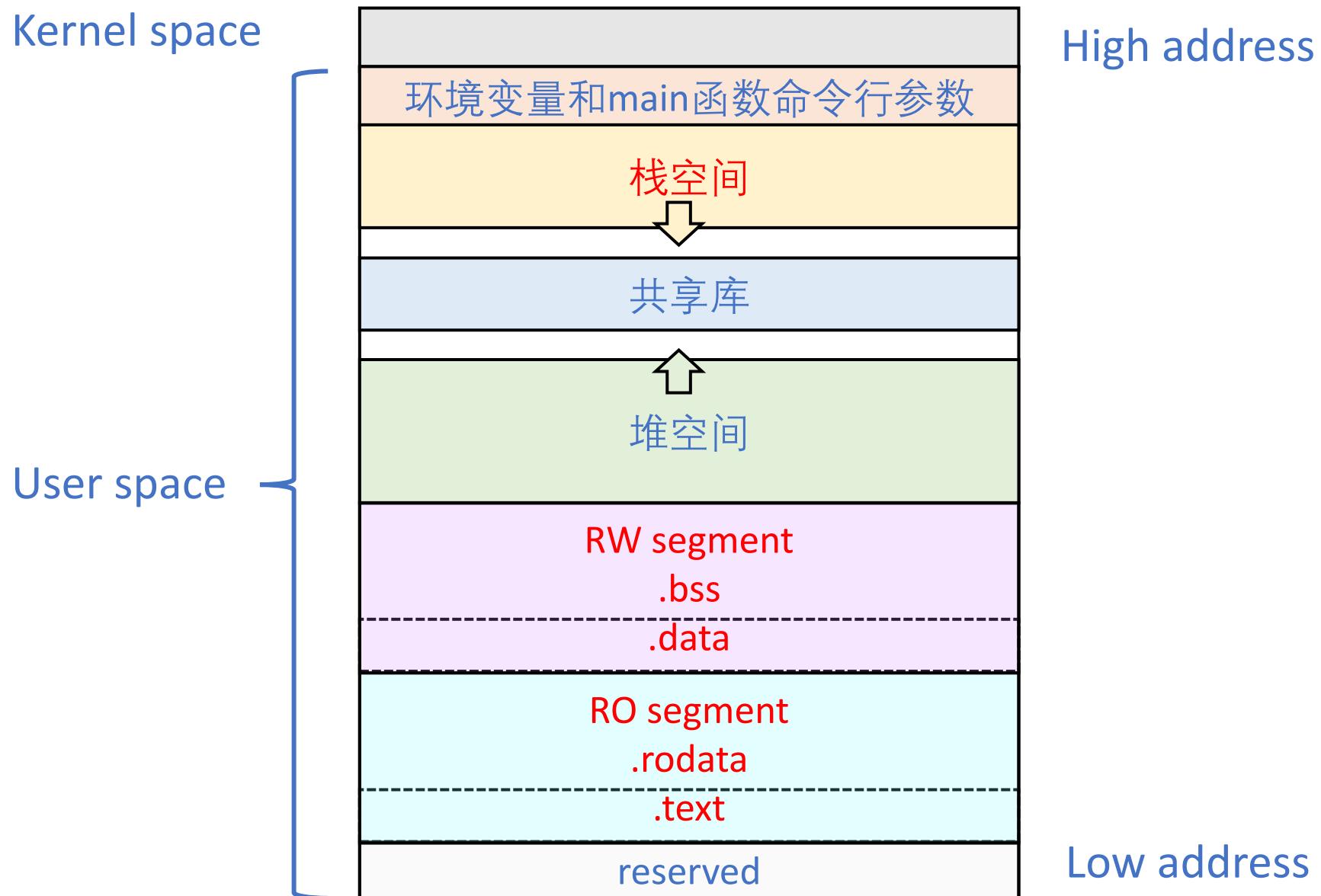
■ 函数定义

<FUNC_NAME>:

<FUNC_BODY>

```
22    .text
23    .align 2
24    .global main
25    .syntax unified
26    .arm
27    .type  main, %function
28 main:
29    @ Function supports interworking.
30    @ args = 0, pretend = 0, frame = 0
31    @ frame_needed = 1, uses_anonymous_args = 0
32    @ link register save eliminated.
33    str fp, [sp, #-4]!
34    add fp, sp, #0
35    ldr r3, .L3
36    ldr r3, [r3]
37    add r3, r3, #1
38    ldr r2, .L3
39    str r3, [r2]
40    mov r3, #0
41    mov r0, r3
42    add sp, fp, #0
43    @ sp needed
44    ldr fp, [sp], #4
45    bx lr
46 .L4:
47    .align 2
48 .L3:
49    .word  a
50    .size  main, .-main
```

3.3 进程内存空间



3.4 函数调用栈

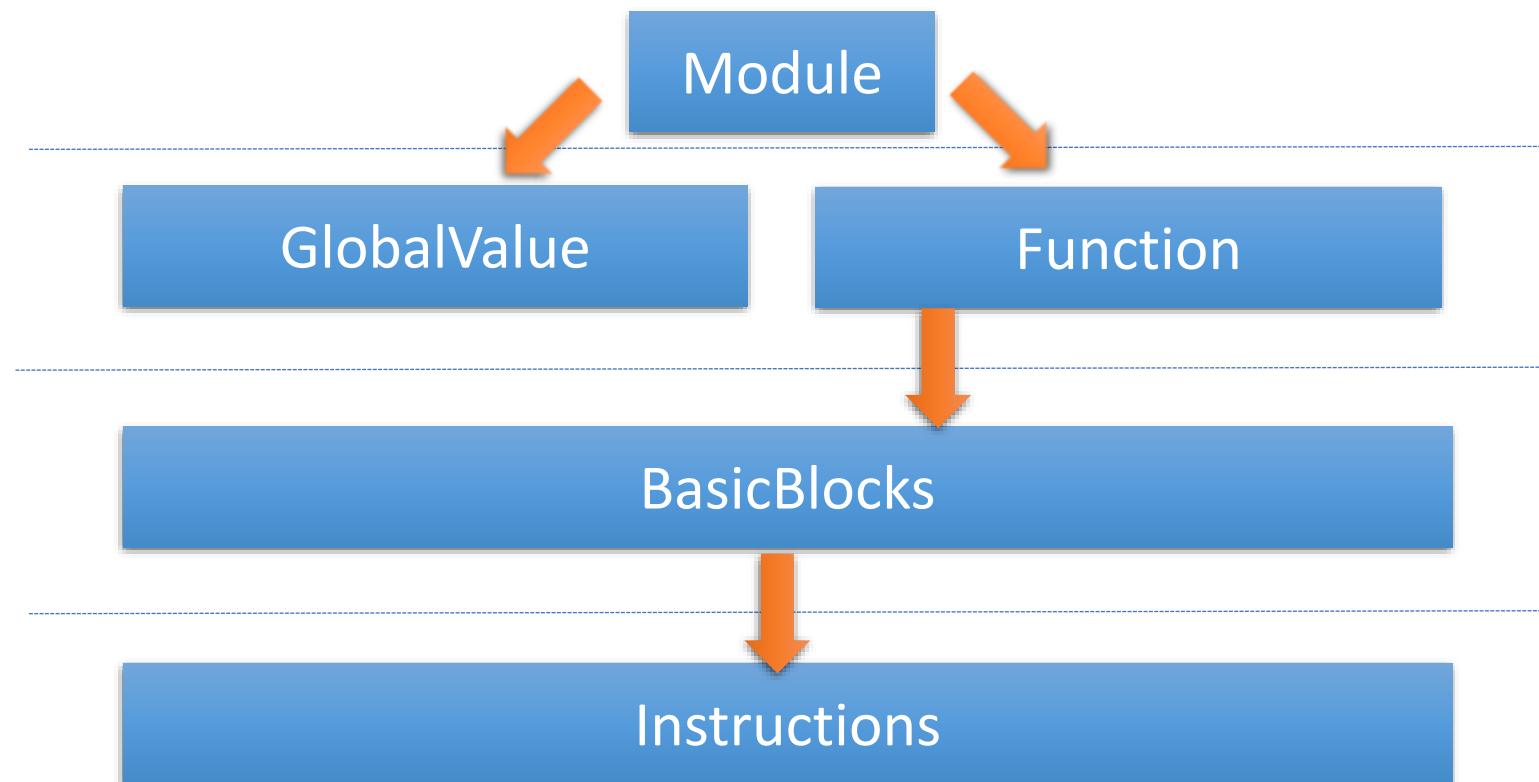
- ARM采用满减栈FD(Full Descending) , 栈由高地址向下增长
- 每个函数对应一个栈帧, 使用两个指针维护
 - ⊕ FP: 指向栈底
 - ⊕ SP: 指向栈顶 (指向最后一个入栈的数据)

内容

1. 实验介绍
2. ARMv7指令集架构
3. ARM汇编基础
4. 实验步骤

4.1 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



4.2 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

■ 构建GlobalValueTable符号表

- ⊕ 记录每个GlobalValue被哪些Instruction使用
- ⊕ 在Function内加载GlobalValue需要2条ldr指令
 - 第1条ldr：取GlobalValue的GlobalLabel(全局地址)
 - 第2条ldr：取GlobalValue的值

```

49  @@@extract part asmcode for demo@@@
50  @@
51  .global a
52  .data
53  .align 2
54  .type a, %object
55  .size a, 4
56 a:
57  .word 4660
58  @@
59 main:
60  @@
61  ldr r3, .L3
62  ldr r3, [r3]
63
64 .L3:
65  .word a

```

4.3 为所有Function生成代码

■ 遍历所有Function, 在每个Function内部

- ⊕ 做寄存器分配(本实验不考虑)
- ⊕ 为该Function使用的每个GlobalValue建立LocalLabel
- ⊕ 检查该Function是否有子函数调用, 获取参数个数(本实验不考虑)
- ⊕ 完成该Function头部工作
- ⊕ 遍历该Function的所有BasicBlock
- ⊕ 完成该Function尾部工作

4.3.1 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，然后所占用的寄存器立即释放

4.3.2 Function头部工作

■ Function头部工作

- ⊕ 在.text段生成该Function的 FunctionLabel以及.type等汇编伪指令
- ⊕ 进入该Function后，保存上一级Function 的FP
- ⊕ 通过上一级Function的SP设置该 Function的FP
- ⊕ 更新该Function的SP (即开辟栈空间)

```
20    main:  
21        @ Function supports interworking.  
22        @ args = 0, pretend = 0, frame = 16  
23        @ frame_needed = 1, uses_anonymous_args = 0  
24        @ link register save eliminated.  
25        str fp, [sp, #-4]!  
26        add fp, sp, #0  
27        sub sp, sp, #20  
28        mov r3, #1  
29        str r3, [fp, #-8]  
30        ldr r3, .L3  
31        str r3, [fp, #-12]  
32        mov r3, #2  
33        str r3, [fp, #-16]  
34        ldr r3, .L3+4  
35        str r3, [fp, #-20]  
36        ldr r3, [fp, #-8]  
37        mov r0, r3  
38        add sp, fp, #0  
39        @ sp needed  
40        ldr fp, [sp], #4  
41        bx lr
```

4.3.3 开辟函数栈空间

开辟栈空间

- 通过SP做减法开辟栈空间

SUB SP, SP, #N

- 需要要考虑开辟多大栈空间

访问栈

- 函数FP不变，因此后续栈访问操作可以基于FP执行

```

20  main:
21      @ Function supports interworking.
22      @ args = 0, pretend = 0, frame = 16
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      str fp, [sp, #-4]!
26      add fp, sp, #0
27      sub sp, sp, #20
28      mov r3, #1
29      str r3, [fp, #-8]
30      ldr r3, .L3
31      str r3, [fp, #-12]
32      mov r3, #2
33      str r3, [fp, #-16]
34      ldr r3, .L3+4
35      str r3, [fp, #-20]
36      ldr r3, [fp, #-8]
37      mov r0, r3
38      add sp, fp, #0
39      @ sp needed
40      ldr fp, [sp], #4
41      bx lr

```

4.3.4 Function尾部工作

Function尾部工作

- ⊕ 恢复上一级Function的FP和SP
- ⊕ 返回上一级Function (生成bx lr指令)

```
20  main:  
21      @ Function supports interworking.  
22      @ args = 0, pretend = 0, frame = 16  
23      @ frame_needed = 1, uses_anonymous_args = 0  
24      @ link register save eliminated.  
25      str fp, [sp, #-4]!  
26      add fp, sp, #0  
27      sub sp, sp, #20  
28      mov r3, #1  
29      str r3, [fp, #-8]  
30      ldr r3, .L3  
31      str r3, [fp, #-12]  
32      mov r3, #2  
33      str r3, [fp, #-16]  
34      ldr r3, .L3+4  
35      str r3, [fp, #-20]  
36      ldr r3, [fp, #-8]  
37      mov r0, r3  
38      add sp, fp, #0  
39      @ sp needed  
40      ldr fp, [sp], #4  
41      bx lr
```

4.4 为所有BasicBlock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)

4.5 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

■ AllocInst

- ⊕ 根据指令的Operand分配栈空间，将基于FP的偏移记录在StackTable 符号表中

■ LoadInst

- ⊕ 以FP为基址，根据符号表中的偏移，生成1条ldr指令

■ StoreInst

- ⊕ 以FP为基址，根据符号表中的偏移，生成1条str指令

4.5 为每一条指令生成代码

■ 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 ($B <Cond> <ThenLabel>$)，之后需要生成无条件跳转指令跳转到Else Block ($B <ThenLabel>$)
- ⊕ 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R0的相关指令

4.6 初始化局部变量和常数

- ARMv7指令为32位, 不能够编码任意32位常数

- 常数在指令可表示立即数范围内

- ⊕ 通过mov指令加载常数 [\(方法一\)](#)
 - ⊕ `mov r3, #10`

- 常数超出可表示立即数范围

- ⊕ 通过movw, movt指令加载常数 [\(方法二\)](#)
 - ⊕ `movw r3, #:lower16:label` //加载低16位
 - `movt r3, #:upper16:label` //加载高16位

- 无论是否超出立即数取值范围

- ⊕ 在text段构建常量池放置常数, 使用时从常量池加载, 需要1次ldr指令 [\(方法三\)](#)
 - ⊕ `ldr <LocalLabel>` //在LocalLabel处存放了常量, <LocalLabel>需在ldr的寻址范围内 (4KB)

```

49  @@@extract part asmcode for demo@@@
50  main:
51  @...
52  @use mov #imm
53  mov r3, #1
54  @use ldr LocalLabel
55  ldr r3, .L3
56  mov r0, r3
57  @...
58  .L3:
59  .word 4660

```

4.7 测试

■ 生成汇编文件

```
➤ sysyc test.sy > test.s
```

■ 生成二进制并测试

⊕ @笔记本上安装模拟器 (课程)

```
➤ arm-none-linux-gnueabihf-gcc test.s -o test.out
➤ qemu-arm ./test.out
➤ echo $?
```

⊕ @树莓派 (竞赛)

```
➤ gcc test.s -o test.out
➤ ./test.out
```