并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室

培训三：代码生成

内容

1 ARMv7指令集架构

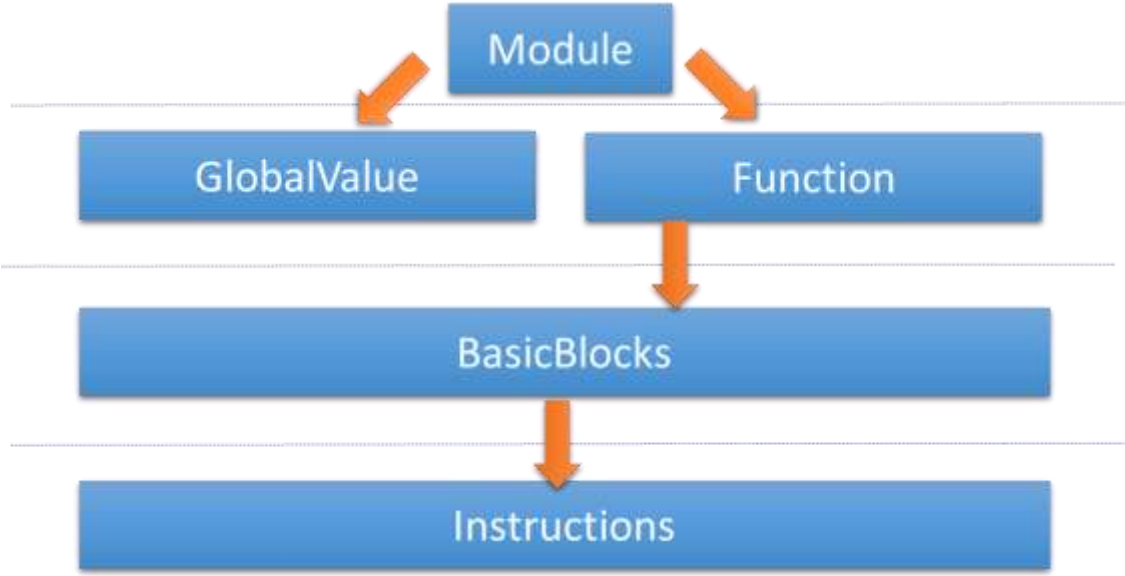
- 32位RISC架构 (load/store架构)
  - 大部分指令处理寄存器中的数据，结果写回寄存器
  - 只有load/store指令可以访问内存
- 寄存器
  - 通用寄存器: r0-r15
  - 浮点寄存器: s0-s31/d0-d15
  - 当前程序状态寄存器CPSR
- 汇编指令格式
  - <opcode>{<cond>}{s} <Rd>, <Rn> {,<op2>}
  - 参考资料
    - sysy-backend-student/doc/backend/ISA_AArch32_xml_A_profile-2023-03.pdf
    - sysy-backend-student/doc/backend/DEN0013D_cortex_a_series_PG.pdf (Section 5)

2 ARM汇编基础

- GNU汇编语法
  - label: instruction @ comment
  - 伪指令
  - 参考资料
    - sysy-backend-student/doc/backend/DEN0013D_cortex_a_series_PG.pdf (Section 4.3)
    - sysy-backend-student/doc/backend/GNU_Assembler_Directives.pdf
    - Writing ARM Assembly (Part 1) | Azeria Labs (azeria-labs.com)
- UAL(unified assembly language)汇编语法
  - GNU汇编器通过.syntax伪指令支持UAL

3 代码生成

- GlobalValue
  - 不同类型的global value放在不同的段(.data/.bss/.rodata)
  - 加载GlobalValue: 2条ldr指令分别取全局地址和取值
- Function
  - 做寄存器分配→简化的寄存器模型: 基于ld/st栈维护LocalValue的值
  - 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
  - 完成该Function头部工作(Prologue)
  - 遍历该Function的所有BasicBlock
  - 完成该Function尾部工作(Epilogue)
- BasicBlock
  - 生成该BasicBlock的Label
  - 遍历该BasicBlock的每一条IR指令
- Instruction
  - 生成IR指令对应的汇编指令(opcode, rd, rs...)



3.1 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
```

3.2 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
```

3.3 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)

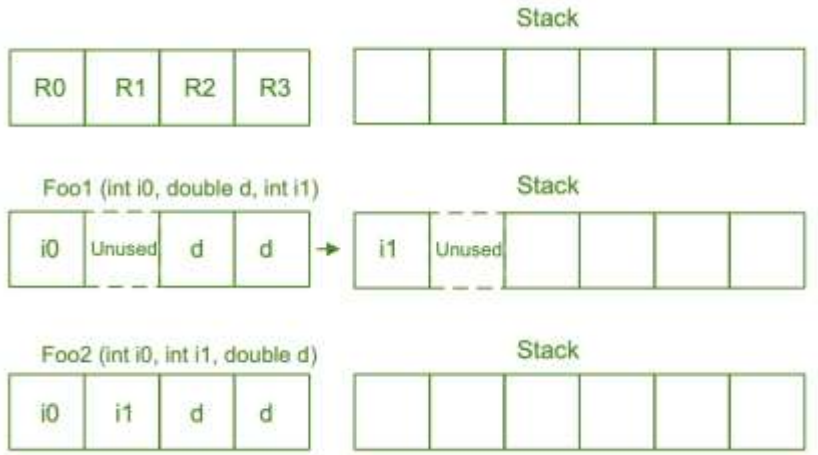
```
20  main:
21      @ Function supports interworking.
22      @ args = 0, pretend = 0, frame = 16
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      str fp, [sp, #-4]!
26      add fp, sp, #0
27      sub sp, sp, #20
28      mov r3, #1
29      str r3, [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
```

内容

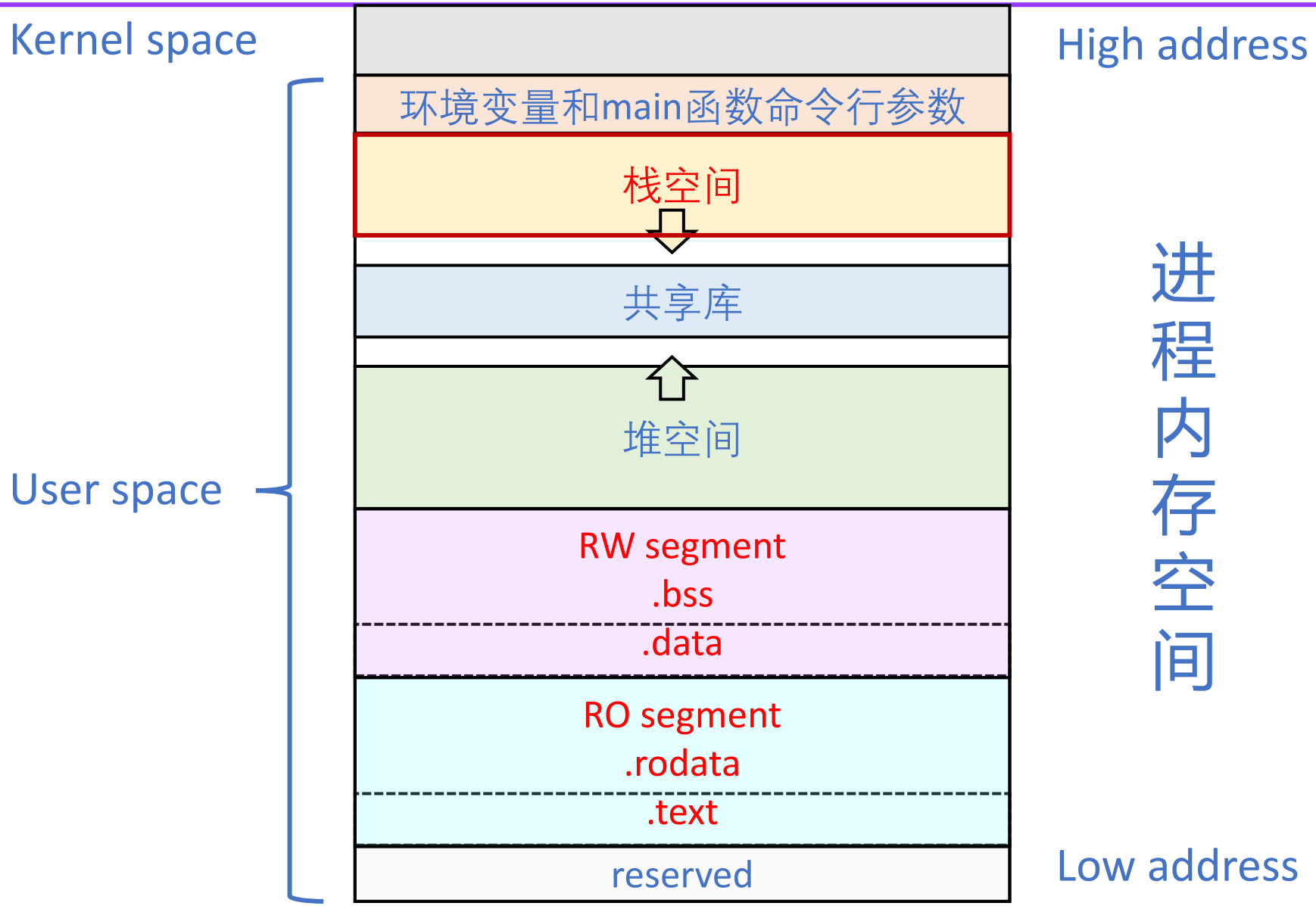
1. ARM指令集架构
2. GNU汇编基础
3. 代码生成基础
4. ARM函数调用
5. RISCV代码生成
6. Phi指令删除

4.1 函数调用约定

- 遵循AAPCS (ARM Architecture Procedure Call Standard)
  - 参数传递
    - 前4个参数通过R0-R3传递，后续参数通过栈传递
  - 返回值传递
    - char, short, int, float通过R0传递
    - double通过R0, R1传递
  - 寄存器保护
    - R0-R3: 调用者保护 (caller-saved, call-clobbered)
    - R4-R10, FP(R11), SP(R13), LR(R14): 被调用者保护 (callee-saved, call-preserved)
    - 如果Callee函数内部还有子函数调用，则需要保护LR，Callee函数返回到Caller函数时才能回到正确的返回地址



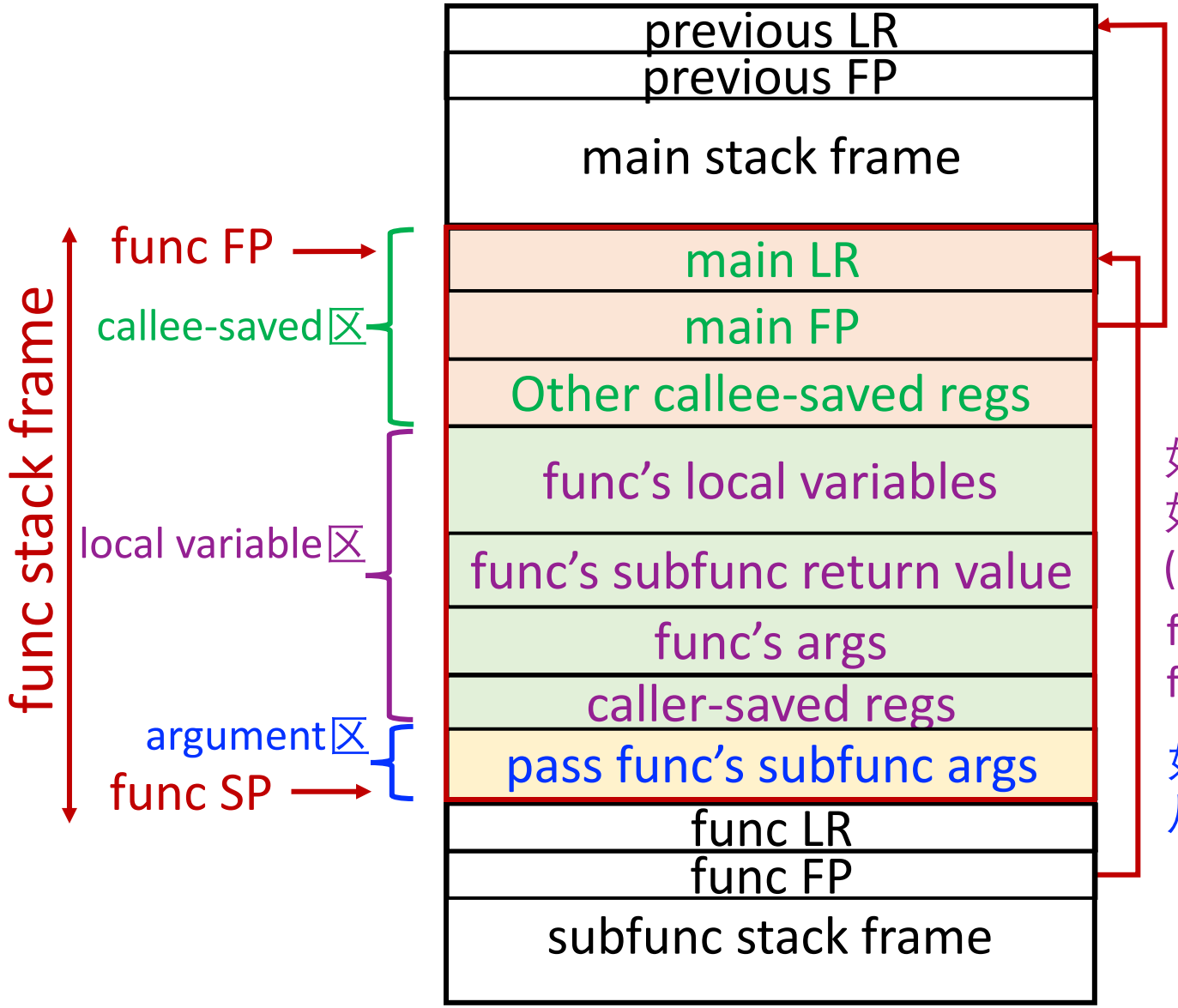
4.2 函数调用栈


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
进程内存空间

4.2 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)

4.2 函数调用栈



如func有subfunc调用，则保存main的LR

保存main的FP

如func会使用R4-R10，则保存main中该Reg的原值

如考虑寄存器分配和溢出，则保存溢出到栈的变量
如不考虑寄存器分配，则保存func所有的局部变量
(包括func调用subfunc的返回值)

func本身接收的前4个形参的值(arg_0-> arg_3)入栈
func作为caller需要保护的寄存器入栈(如需要)

如子函数参数个数大于4，需要通过栈传参
从第5个开始后面的实参逆序入栈(arg_x->arg_4)

4.3 函数调用相关指令

- PUSH和POP指令
  - 现场保护和恢复现场，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}>: 将寄存器列表按regs3, 2, 1的顺序压入栈中
  - POP <{regs1, 2, 3}>: 从栈中按regs1, 2, 3的顺序恢复寄存器列表

```
1   void func(){
2       ;
3   }
4
5   int main(){
6       func();
7       return 0;
8   }
9
```

```
20  func:
21      @ Function supports interworking.
22      @ args = 0, pretend = 0, frame = 0
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      str fp, [sp, #-4]!      @ push {fp}
26      add fp, sp, #0
27      nop
28      add sp, fp, #0
29      @ sp needed
30      ldr fp, [sp], #4        @ pop {fp}
31      bx  lr
32      .size   func, .-func
```

4.3 函数调用相关指令

- BL和BX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
  - BX LR: 跳转到LR指定返回地址处

```
38  main:
39      @ Function supports interworking.
40      @ args = 0, pretend = 0, frame = 0
41      @ frame_needed = 1, uses_anonymous_args = 0
42      push    {fp, lr}
43      add fp, sp, #4
44      bl  func
45      mov r3, #0
46      mov r0, r3
47      sub sp, fp, #4
48      @ sp needed
49      pop {fp, lr}
50      bx  lr
51      .size   main, .-main
```

```
56  @@@extract part asmcode for demo@@@
57  func:
58      str fp, [sp, #-4]!      @ push {fp}
59      add fp, sp, #0
60      nop
61      add sp, fp, #0
62      @ sp needed
63      ldr fp, [sp], #4        @ pop {fp}
64      bx  lr
```

4.3 函数调用相关指令

- B指令
  - 调用Callee函数，调用后不会再返回到Caller函数原来的执行处

```
5    __start:
6        ldr sp,=0X80200000      @set sp
7        b main                  @jump to main function
```

  - 基本块之间的跳转
    - 如if-then-else
    - 使用B指令, 或B与条件码的结合(如BEQ, BNE等)

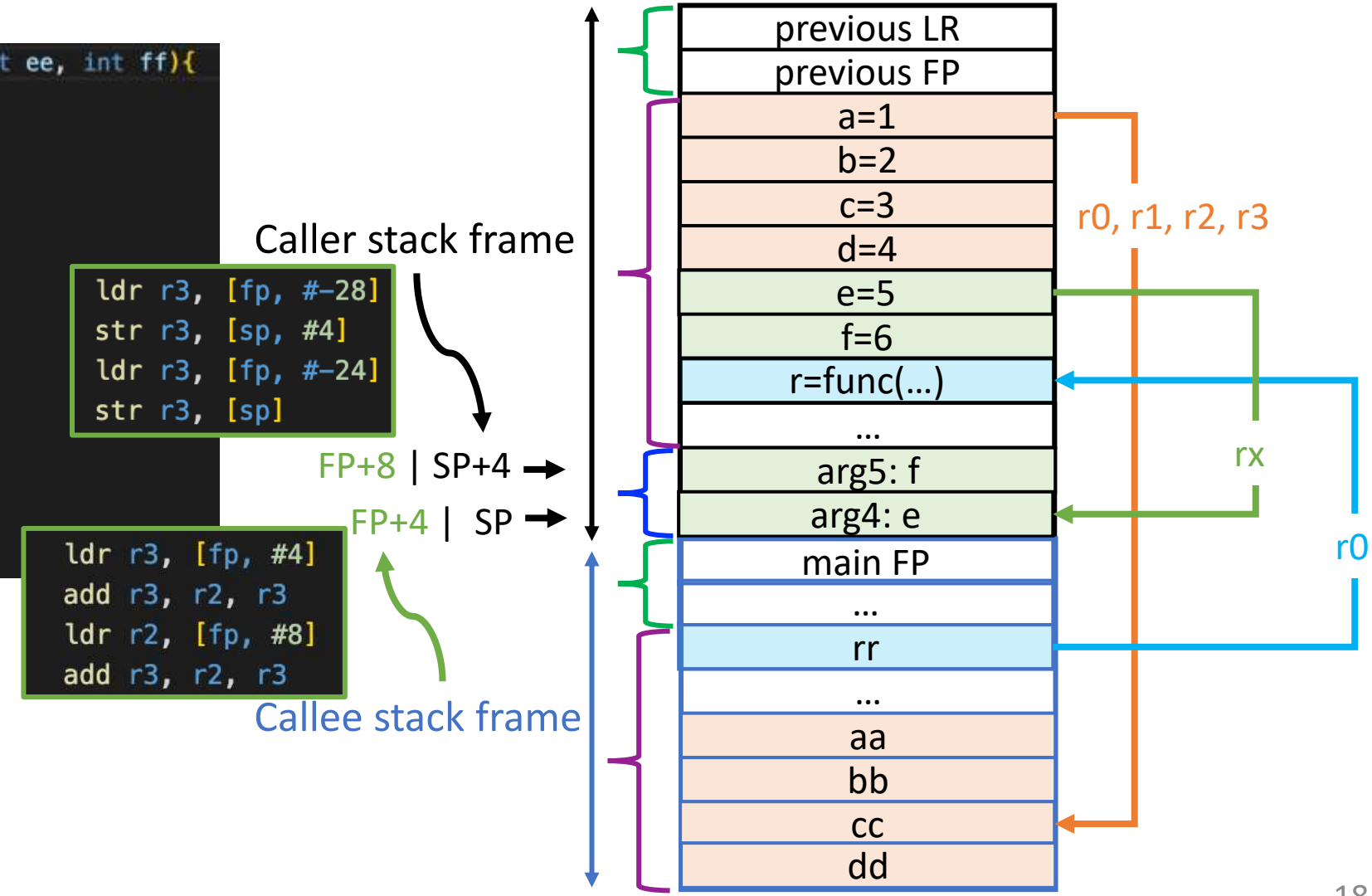
4.4 函数调用参数传递

```c
int func(int aa, int bb, int cc, int dd, int ee, int ff){
    int rr = aa+bb+cc+dd+ee+ff;
    return rr;
}

int main(){
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    int e = 5;
    int f = 6;
    int r = func(a, b, c, d, e, f);
    return r;
}
```

```
ldr r3, [fp, #-28]
str r3, [sp, #4]
ldr r3, [fp, #-24]
str r3, [sp]
```

```
ldr r3, [fp, #4]
add r3, r2, r3
ldr r2, [fp, #8]
add r3, r2, r3
```



4.5 函数调用的代码生成

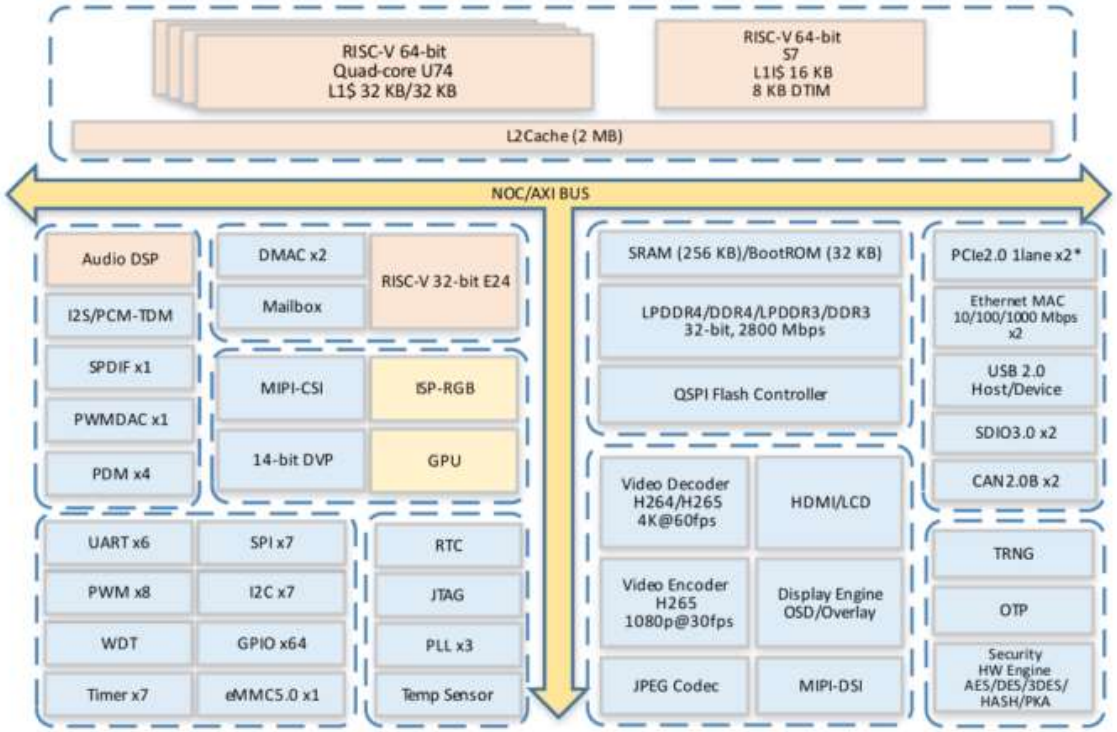
- 为CallInst指令生成代码
  - 如果被调用函数有参数，生成通过R0~R3进行传参的相关指令
    - 如果参数个数大于4，还需生成通过栈传参的相关指令 (参数压入栈顶，利用SP+偏移)
  - 生成跳转指令 (BL <FunctionLabel>)
  - 如果被调用函数有返回值，在CallInst后，生成通过R0获得返回值并存入栈中的相关指令

内容

1. ARM指令集架构
2. GNU汇编基础
3. 代码生成基础
4. 函数调用
5. RISCV代码生成
6. Phi指令删除

6.1 VisionFive2

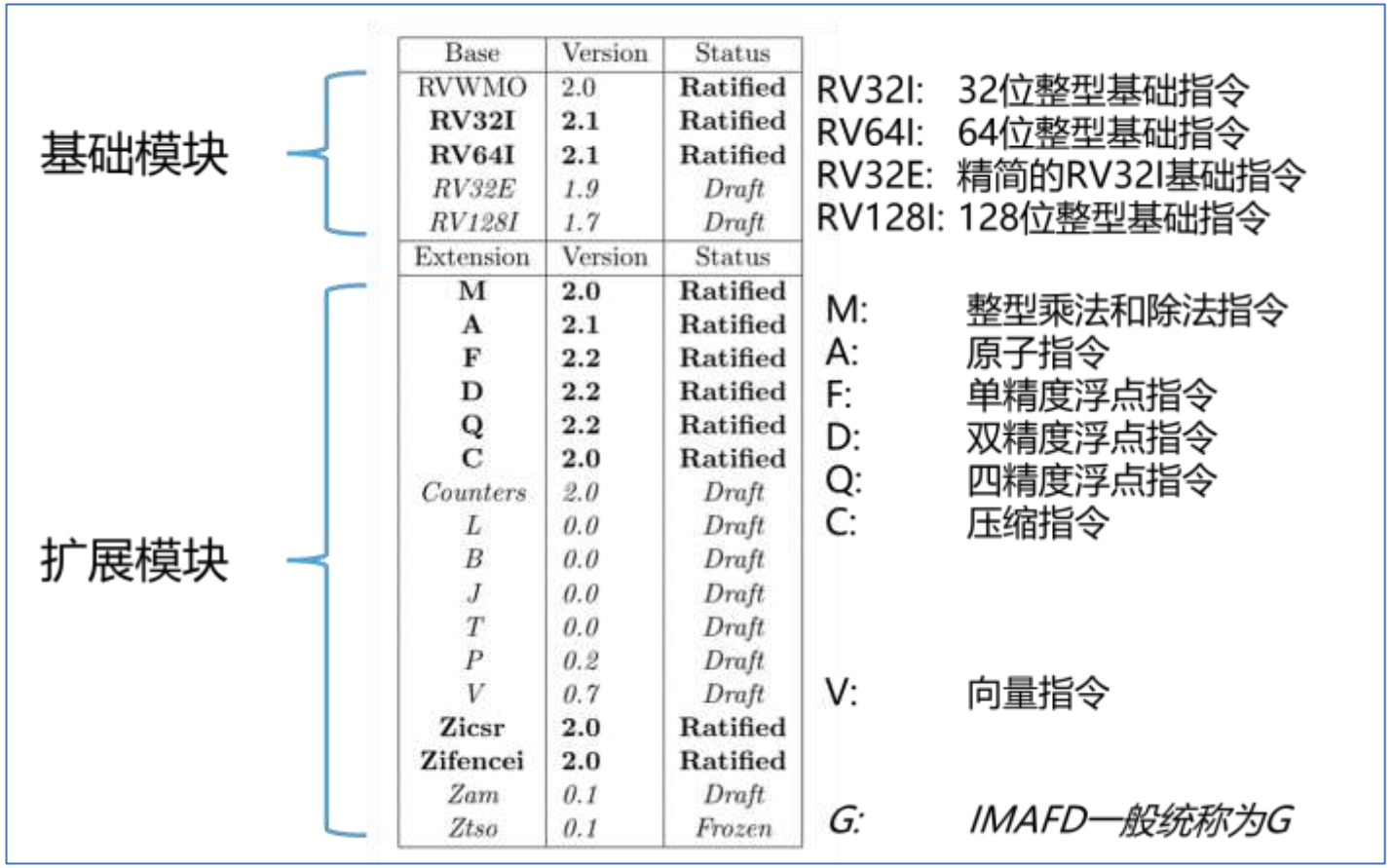
- RV64GC ISA SoC @1.5GHz
- Quadcore U74, L1D 32KB, L1I 32KB
  - Dual issue, in-order 8 stage Harvard Pipeline
- L2 2MB, 2/4/8GB LPDDR4



StarFive JH7110

6.2 RV64GC指令集架构

- RV64GC
  - 地址和数据大小均为64位 (32位也是有效数据类型)，指令编码为32位
  - G = IMAFD (整型基础，整型乘除，原子，单、双精度浮点指令)
  - C = 压缩指令



Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	**Version**	**Status**
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

6.2 RV64GC指令集架构

- 通用寄存器
  - 32个64位寄存器: x0-x31
  - x0寄存器读取时总为0，写入无效
- 浮点寄存器
  - 32个64位浮点寄存器: f0-f31
  - 单精度占寄存器的低32位
- PC
  - 独立于通用寄存器
- CSR寄存器
  - 理论最多4096个
  - 通过zicsr指令集操作
  - FCSR寄存器: 浮点控制状态寄存器

x0 / zero	硬连线为 0
x1 / ra	返回地址
x2 / sp	栈指针（Stack pointer）
x3 / gp	全局指针（Global pointer）
x4 / tp	线程指针（Thread pointer）
x5 / t0	临时寄存器
x6 / t1	临时寄存器
x7 / t2	临时寄存器
x8 / s0 / fp	保存寄存器，帧指针（Frame pointer）
x9 / s1	保存寄存器
x10 / a0	函数参数，返回值
x11 / a1	函数参数，返回值
x12 / a2	函数参数
x13 / a3	函数参数
x14 / a4	函数参数
x15 / a5	函数参数
x16 / a6	函数参数
x17 / a7	函数参数
x18 / s2	保存寄存器
x19 / s3	保存寄存器
x20 / s4	保存寄存器
x21 / s5	保存寄存器
x22 / s6	保存寄存器
x23 / s7	保存寄存器
x24 / s8	保存寄存器
x25 / s9	保存寄存器
x26 / s10	保存寄存器
x27 / s11	保存寄存器
x28 / t3	临时寄存器
x29 / t4	临时寄存器
x30 / t5	临时寄存器
x31 / t6	临时寄存器

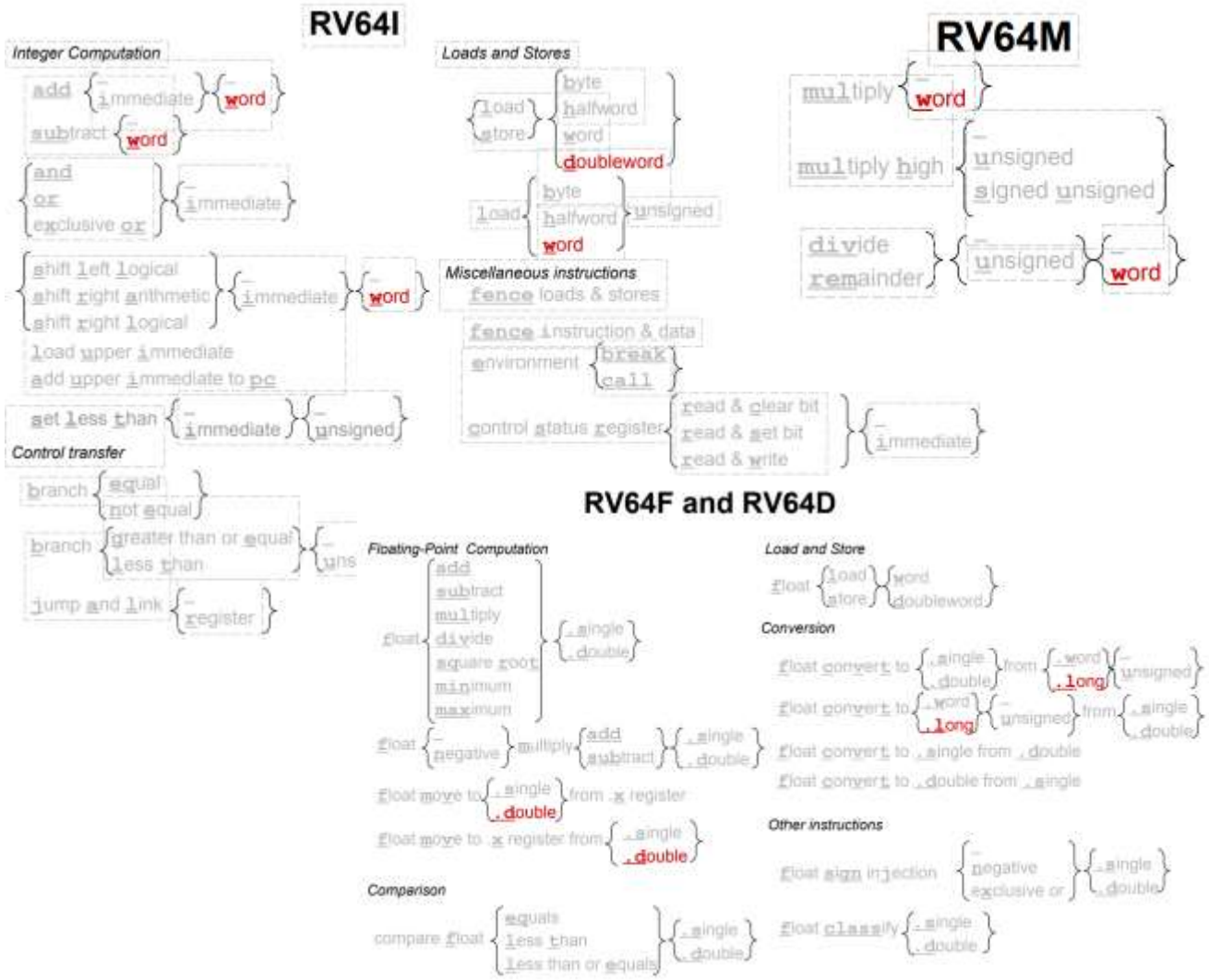
32

pc

32

6.2 RISCV64GC指令集架构

- 指令格式
  - R型: 寄存器-寄存器操作
  - I型: 短立即数指令和load操作
  - S型: store操作
  - B型: 条件跳转
  - U型: 长立即数指令
  - J型: 无条件跳转
- 汇编指令
  - 操作码, 目的寄存器rd,
    源寄存器rs1(, rs2, 立即数imm),
    .d双字，.w字



6.3 RISCV汇编语法

- 汇编器指示符 (assembler directives)
  - .text, .data, .bss, .section, .align n, .balign n, .word, .float…
- RISCV伪指令
  - 常规指令的特例
  - 包括依赖于零寄存器和不依赖于零寄存器的伪指令

依赖x0

伪指令	基础指令	含义
nop	addi x0, x0, 0	空操作
neg rd, rs	sub rd, x0, rs	取负
negw rd, rs	subw rd, x0, rs	取负字
snez rd, rs	sltu rd, x0, rs	不等于零时置位
sltz rd, rs	slt rd, rs, x0	小于零时置位
sgtz rd, rs	slt rd, x0, rs	大于零时置位
beqz rs, offset	beq rs, x0, offset	等于零时分支
bnez rs, offset	bne rs, x0, offset	不等于零时分支
blez rs, offset	bge x0, rs, offset	小于等于零时分支
bgez rs, offset	bge rs, x0, offset	大于等于零时分支
bltz rs, offset	blt rs, x0, offset	小于零时分支
bgtz rs, offset	blt x0, rs, offset	大于零时分支
j offset	jal x0, offset	跳转
jr rs	jalr x0, rs, 0	寄存器跳转
ret	jalr x0, x1, 0	从子过程返回

不依赖x0

伪指令	基础指令	含义
lla rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	装入局部地址
la rd, symbol	*PIC*: auipc rd, GOT[symbol][31:12] l{w|d} rd, rd, GOT[symbol][11:0] 非 *PIC*: 与 lla rd, symbol 相同	装入地址
l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	读全局符号
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	写全局符号
fl{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w|d} rd, symbol[11:0](rt)	读全局浮点符号
fs{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w|d} rd, symbol[11:0](rt)	写全局浮点符号
li rd, immediate	多种指令序列	装入立即数
mv rd, rs	addi rd, rs, 0	复制寄存器
not rd, rs	xori rd, rs, -1	取反
sext.w rd, rs	addiw rd, rs, 0	符号扩展字
seqz rd, rs	sltiu rd, rs, 1	等于零时置位

6.4 函数调用约定

- 参数传递
  - a0-a7
- 返回值传递
  - a0-a1
- 寄存器保护
  - 被调用者保护寄存器(callee-saved)
    - 栈指针sp
    - 帧指针fp (s0)
    - 保存寄存器s1-s11
    - 浮点保存寄存器fs0-fs11
  - 调用者保护寄存器(caller-saved)
    - 返回地址ra
    - 函数参数/返回值寄存器a0-a7
    - 浮点函数参数/返回值寄存器fa0-fa7
    - 临时寄存器t0-t6
    - 浮点临时寄存器ft0-ft11

寄存器	ABI 名称	描述	调用前后是否一致?
x0	zero	硬连线为 0	—
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	—
x4	tp	线程指针	—
x5	t0	临时寄存器/备用链接寄存器	否
x6–7	t1–2	临时寄存器	否
x8	s0/fp	保存寄存器/帧指针	是
x9	s1	保存寄存器	是
x10–11	a0–1	函数参数/返回值	否
x12–17	a2–7	函数参数	否
x18–27	s2–11	保存寄存器	是
x28–31	t3–6	临时寄存器	否
f0–7	ft0–7	浮点临时寄存器	否
f8–9	fs0–1	浮点保存寄存器	是
f10–11	fa0–1	浮点参数/返回值	否
f12–17	fa2–7	浮点参数	否
f18–27	fs2–11	浮点保存寄存器	是
f28–31	ft8–11	浮点临时寄存器	否

6.4 函数调用约定

- 函数调用栈
  - 采用满减栈FD(Full Descending)，栈由高地址向下增长
    FP指向栈底，SP指向栈顶
- 函数调用相关指令

伪指令	等价指令	功能	示例
jal offset	jal x1, offset　//x1=ra	跳转到offset指定位置，返回地址保存在ra	jal foo
jalr rs	jalr x1, 0(rs)	跳转到rs指定位置，返回地址保存在ra	jalr s1
j offset	jal x0, offset　//x0=0	跳转到offset指定位置，不保存返回地址	j loop
jr rs	jalr x0, 0(rs)	跳转到rs指定位置，不保存返回地址	jr s1
call offset	auipc x1, offsetHi jalr x1, offsetLo(x1)	调用远距离子过程，保存返回地址	call foo
tail offset	auipc x6, offsetHi jalr x0, offsetLo(x6)	尾调用远距离子过程，不保存返回地址	tail foo
ret	jalr x0, 0(x1)　//x1=ra	从callee返回，返回地址从ra读取	ret

6.4 函数调用约定

- Function头部工作(Prologue)
  - 开辟栈空间，设置SP
  - 保存ra到栈上 (如果本Function内部还有子函数调用)
  - 保存callee-saved寄存器到栈上(如果本Function用到)
    - 保存寄存器s1-s11,
    - 浮点保存寄存器fs0-fs11
- Function尾部工作(Epilogue)
  - 恢复callee-saved寄存器
  - 恢复ra，获得返回地址
  - 释放栈空间
  - 通过ret指令返回调用点

相关资料

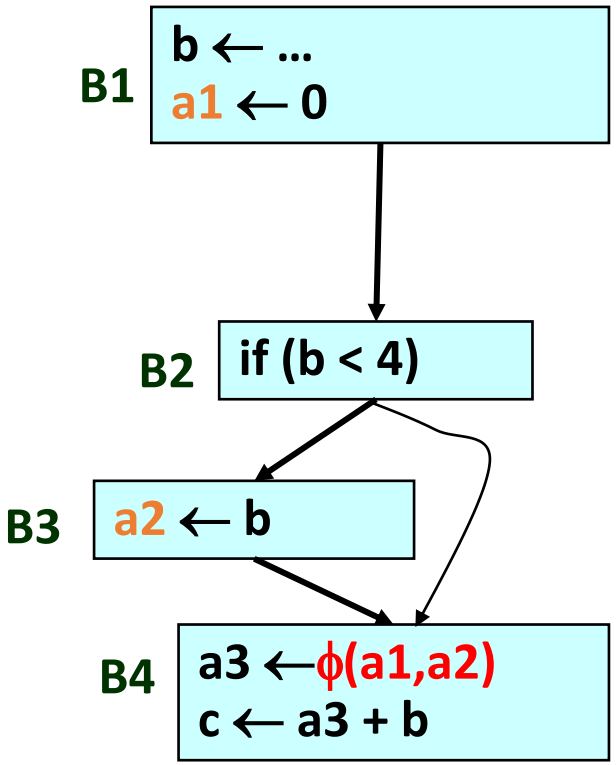
- RISCV交叉编译工具链
  - https://github.com/riscv-collab/riscv-gnu-toolchain
- RISC-V 开放架构设计之道（The RISC-V Reader中文译本）
  - sysy-backend-student/doc/backend/RISC-V-Reader-Chinese-v1.pdf (2023.11)
- RISCV指令集手册
  - sysy-backend-student/doc/backend/riscv-spec-20191213.pdf

内容

1. ARM指令集架构
2. GNU汇编基础
3. 代码生成基础
4. ARM函数调用
6. RISCV代码生成
5. Phi指令删除

5.1 SSA IR

- 编译中端，插入Phi函数，将stack形式的IR转换为SSA形式的IR

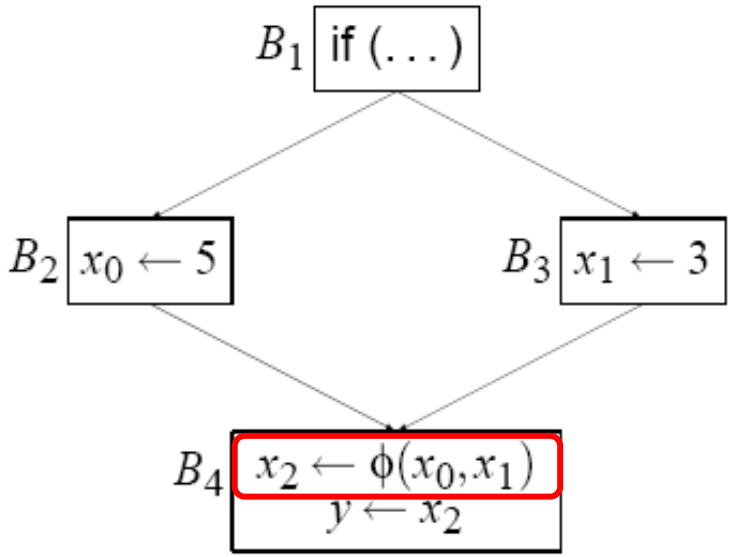


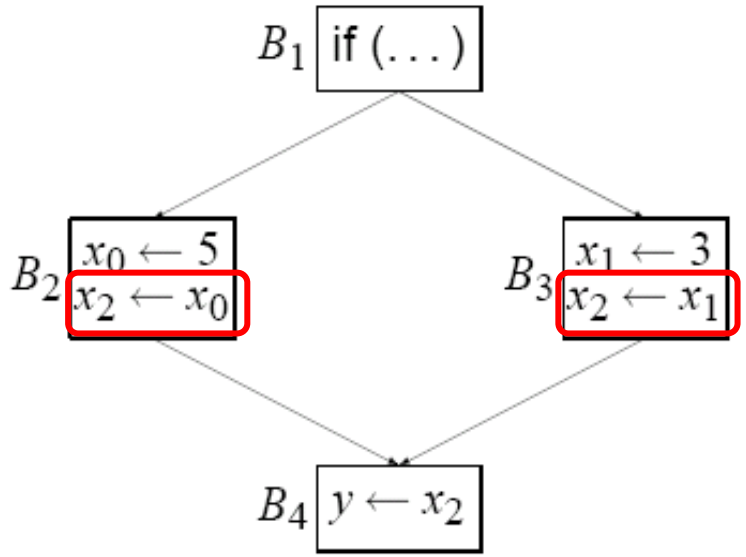
$$\phi(a1,a2) = \begin{cases} a1 \text{ if arriving at B4 from B2} \\ a2 \text{ if arriving at B4 from B3} \end{cases}$$

针对Phi函数，如何做代码生成?

5.2 删除Phi函数

- 编译后端，删除Phi函数，转回Non-SSA形式
  - 删除Phi函数
  - Phi的前驱均只有汇合结点一个后继: 在前驱的定值后，插入copy或move指令





5.2 删除Phi函数

- 编译后端，删除Phi函数，转回非SSA形式
  - 删除Phi函数
  - 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