

COMP130014 编译

第十一讲：指令选择

徐 辉

xuh@fudan.edu.cn



大纲

一、AArch64指令集

二、phi指令处理

三、指令选择问题

一、AArch64指令集

指令集架构

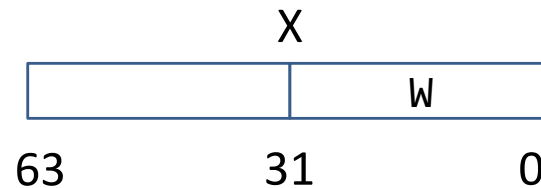
- 精简指令集（RISC）
 - 每条指令只做一件事（主要指内存访问和运算分离）
 - AArch/ARM、RISC-V
- 复杂指令集（CISC）
 - 一条指令可以包含多个底层操作（load-add-store）
 - X86（Intel IA-32）、X86-64架构（AMD64）
- 其它
 - VLIW：Very long instruction word (Intel IA-64)
 - EPIC：Explicitly parallel instruction computing

目标指令集：ARM v8a

- 32位版本：ARM Cortex-A32
- 32/64位版本：ARM Cortex-A57/A72/A73等，代表芯片
 - Apple A8/A8x/A9/A9x/A10/A10x (iPhone, iPad)
 - Apple M1/M2 (iPhone, iPad, MacBook)
 - Qualcomm Kryo: 骁龙 (Snapdragon 820)
- AArch64 vs ARM64 (Apple版本)

ARM-v8A寄存器：通用寄存器

- 通用寄存器：X0 - X30 (64-bit)
 - W0-W30：低32位



```
mov x1, 5
mov x2, 10
add x0, x1, x2
```

x1 = 5
x2 = 10
x0 = x1 + x2

```
mov w1, 5
mov w2, 10
add w0, w1, w2
```

w1 = 5
w2 = 10
w0 = w1 + w2

IR=>Assembly: 数据存取

- 栈（顶）寄存器：SP（16字节对齐）

```
%x = alloca i32  
store i32 1, i32* %x  
%x0 = load i32, i32* %x
```



```
sub sp, sp, 16  
mov w0, 1  
str w0, [sp]  
ldr w0, [sp]
```

sp = sp - 16

```
#: llc -O0 -march=arm64 -filetype=asm foo.ll -o foo.s
```

在MacOS (M1)上编译和运行

```
#: as foo.s -o foo.o  
#: ld foo.o -lSystem -syslibroot `xcrun -sdk macosx --show-sdk-path` -e _start -arch arm64
```

或

```
#: gcc test.o -e _main -arch arm64
```

在x86上运行编译和运行ARM-v8A程序

安装qemu工具链

```
#: sudo apt install qemu qemu-system qemu-user  
#: sudo apt install gcc-aarch64-linux-gnu  
#: sudo apt install qemu-user-static libc6-arm64-cross
```

交叉编译程序

```
#: aarch64-linux-gnu-gcc hello.c
```

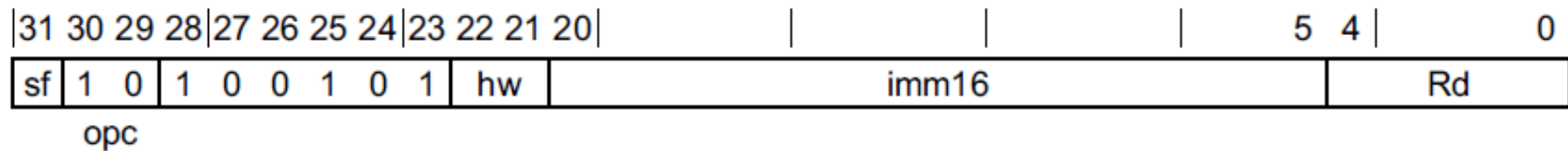
在虚拟机上运行程序

```
#: qemu-aarch64 -L /usr/aarch64-linux-gnu a.out
```


ARM-v8A指令：MOV

- MOV：任意16位立即数，或左移16/32/48位

MOV <Wd>, #<imm>



<imm> For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".

For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

```
mov x1, 65535
mov x2, 65539
mov x3, 131070
```

```
mov x8, 3
movk x8, 1, lsl 16
```

ARM-v8A指令：寻址模式

- 不支持直接寻址，间接寻址

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	[base{, #0}]	-	-
Base plus offset	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S U)XT(X W) {#imm}]
Pre-indexed	[base, #imm]!	-	-
Post-indexed	[base], #imm	[base], Xm ^a	-
Literal (PC-relative)	label	-	-

```
ldr x2, [x1]
ldr x2, [x1, 10]
ldr x2, [x1, x0]
ldr x2, [x1, 10]!
ldr x2, [x1], 10
ldr x2, [x1, x0, lsl 3]
```

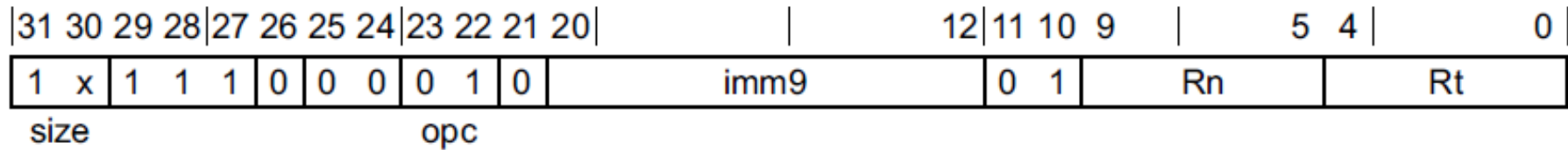
```
x2 = [x1 + 10]
x2 = [x1 + x0]
x1 = x1 + 10, x2 = [x1]
x2 = [x1], x1 = x1 + 10
x2 = [x1 + x0 * 8]
```

```
str w0, [x1]
str w0, [x1, 10]
str x2, [x1, x0]
```

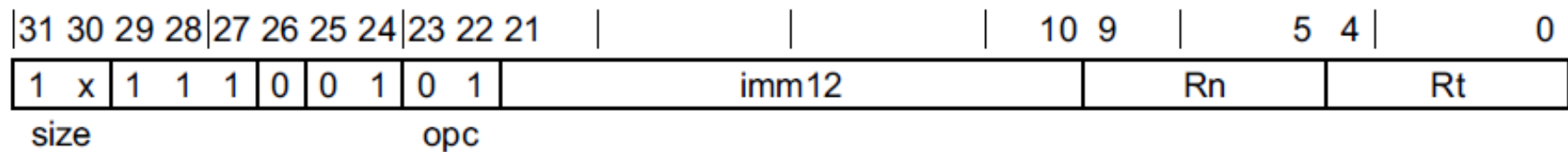
ARM-v8A指令： LDR（立即数）

```
ldr x2, [x1]
ldr x2, [x1, 10]
ldr x2, [x1, 10]!
```

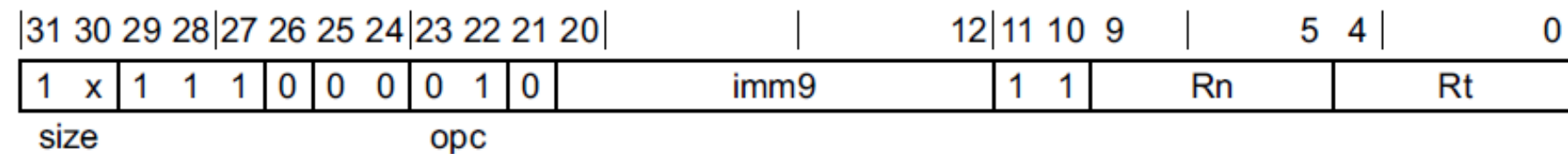
LDR <Xt>, [<Xn|SP>], #<sim>



LDR <Xt>, [<Xn|SP>{, #<pimm>}]



LDR <Xt>, [<Xn|SP>, #<sim>]!



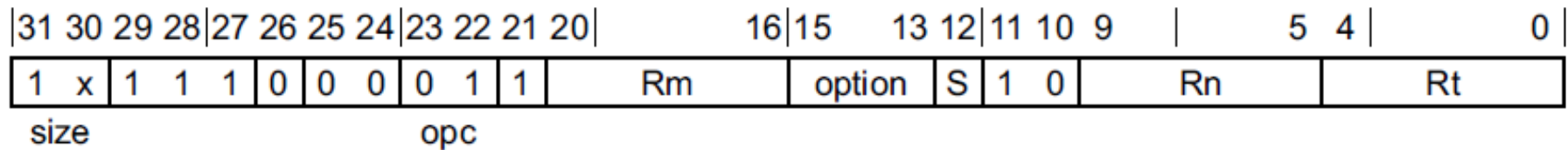
<sim> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

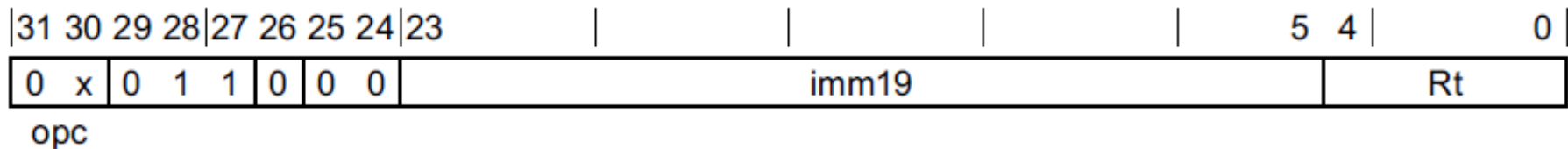
ARM-v8A指令：LDR（寄存器/标签）

LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]



```
ldr x2, [x1, x0]
ldr x2, [x1, x0, lsl 3]
```

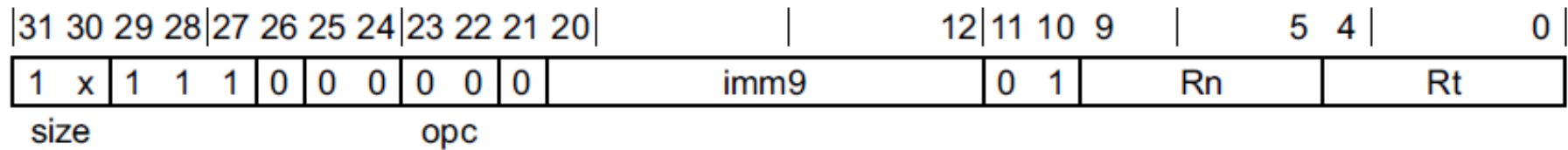
LDR <Xt>, <label>



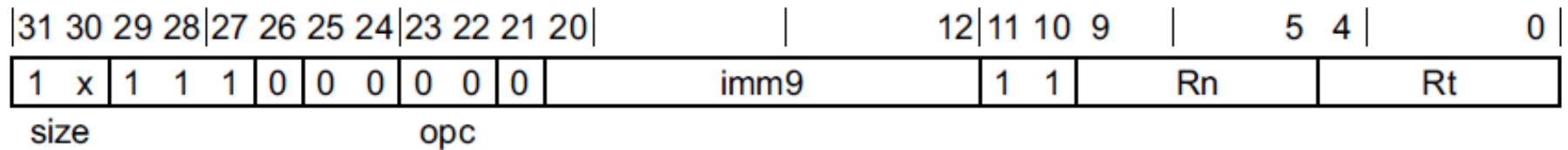
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

ARM-v8A指令：STR

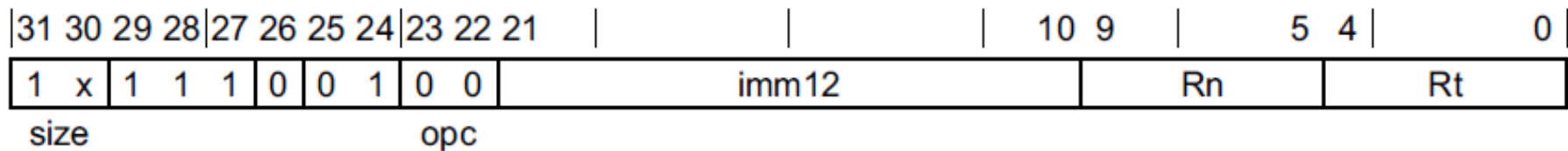
STR <Wt>, [<Xn|SP>], #<imm>



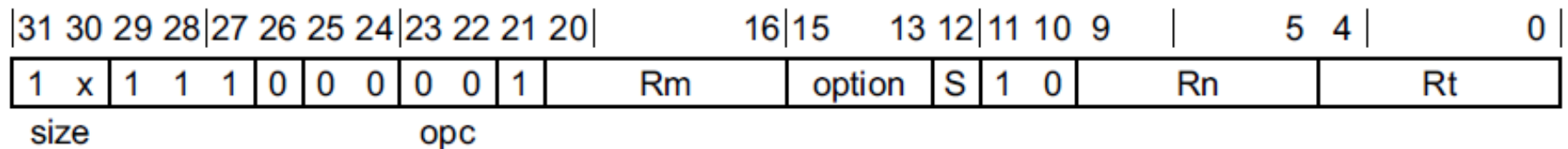
STR <Xt>, [<Xn|SP>], #<imm>!



STR <Xt>, [<Xn|SP>{, #<pimm>}]



STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]



复合类型如何翻译？

```
define i32 @main() {  
  %1 = alloca [2 x i32]  
  %2 = getelementptr [2 x i32], [2 x i32]* %1, i32 0, i32 0  
  store i32 99, i32* %2  
  %3 = load i32, i32* %2  
  ret i32 %3  
}
```

```
%mystruct = type { i32, i32 }  
define i32 @main() {  
  %1 = alloca %mystruct  
  %2 = getelementptr %mystruct, %mystruct* %1, i32 0, i32 0  
  store i32 1, i32* %2  
  ret i32 0  
}
```

IR=>Assembly: 算术运算

```
%r1 = add i32 %0, %1  
%r2 = sub i32 %r1, 2  
%r3 = mul i32 %r2, %1  
%r4 = sdiv i32 %r3, %1
```

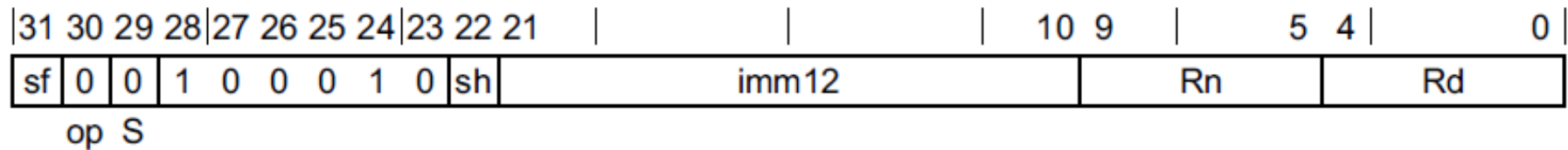


```
add    w8, w0, w1  
sub     w8, w8, 2  
mul     w8, w8, w1  
sdiv    w0, w8, w1
```

ARM-v8A指令： 算数运算： ADD

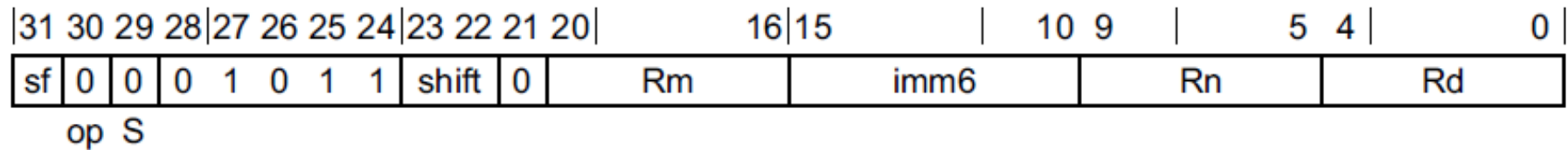
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}

sh == 1, imm左移12位



ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```
add x0, x1, 4095
add x0, x1, 4097
add x0, x1, 20480
```

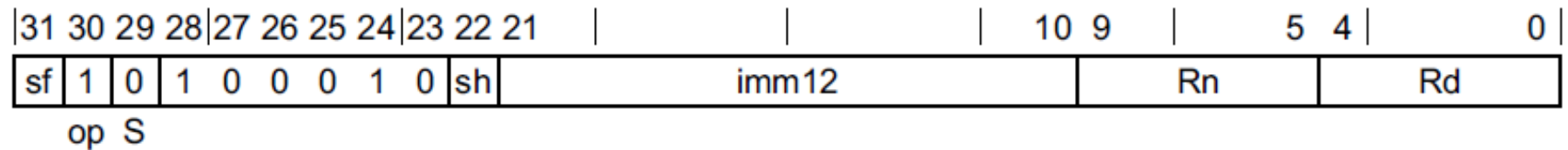


```
add x0, x1, x2
```

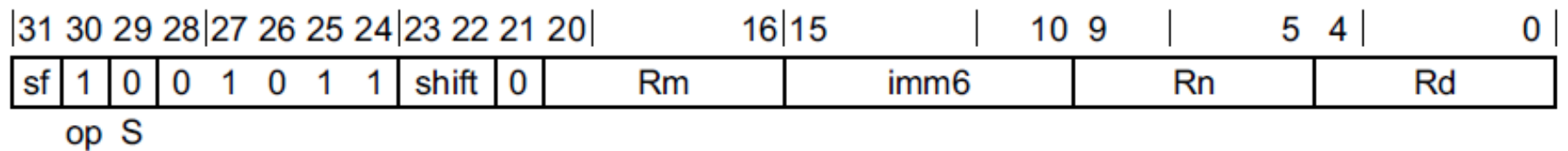
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

ARM-v8A指令： 算数运算： SUB

SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}



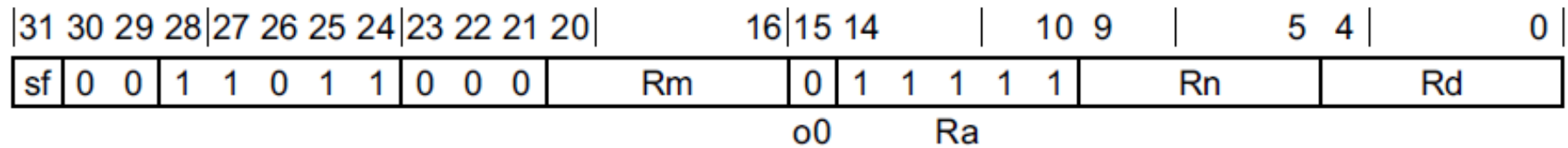
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}



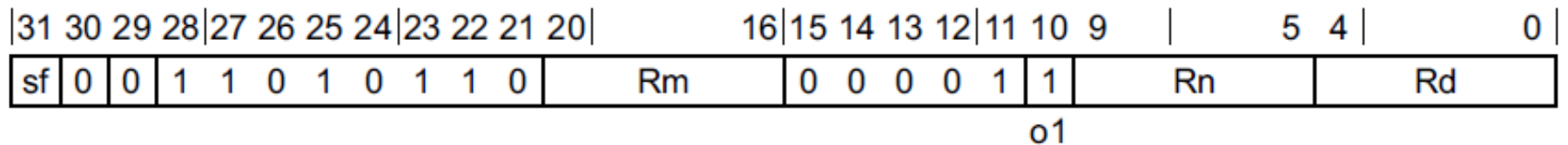
ARM-v8A指令： 算数运算： MUL/SDIV

MUL <Xd>, <Xn>, <Xm>

不支持立即数

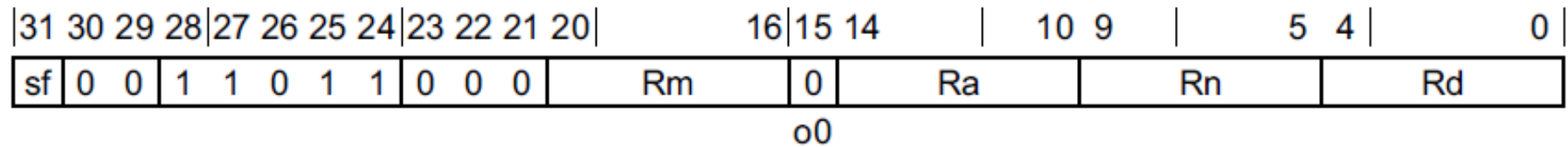


SDIV <Xd>, <Xn>, <Xm>

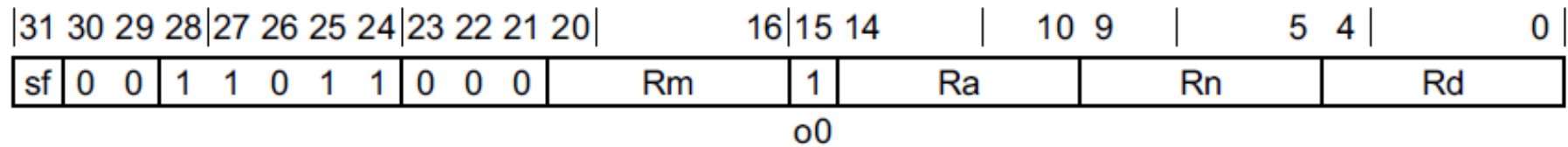


ARM-v8A指令：复合算数运算

MADD <Xd>, <Xn>, <Xm>, <Xa>



MSUB <Xd>, <Xn>, <Xm>, <Xa>



```
madd x0, x1, x2, x3
```

```
msub x0, x1, x2, x3
```

$x0 = x1 * x2 + x3$

$x0 = x1 * x2 - x3$

IR=>Assembly: 比较运算和结果获取

```
%r1 = icmp sgt i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



```
cmp  w0, w1  
cset w0, gt
```

条件

```
%r1 = icmp sge i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



```
cmp  w0, w1  
cset w0, ge
```

```
%r1 = icmp eq i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



```
cmp  w0, w1  
cset w0, eq
```

```
%r1 = icmp ne i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



```
cmp  w0, w1  
cset w0, ne
```

```
%r1 = icmp sle i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



```
cmp  w0, w1  
cset w0, le
```

```
%r1 = icmp lt i32 %0, %1  
%r2 = zext i1 %r1 to i32
```



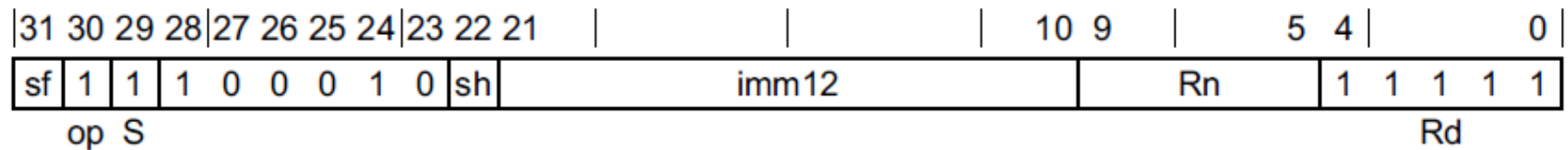
```
cmp  w0, w1  
cset w0, lt
```

ARM-v8A指令：比较运算：CMP

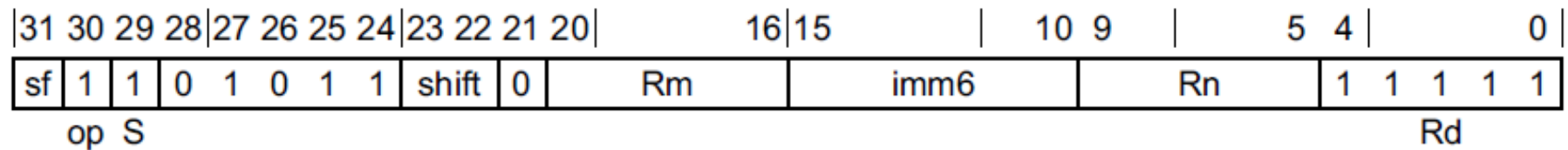
- 基于PSR（NZCV）寄存器实现

CMP <Xn|SP>, #<imm>{, <shift>}

SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}



CMP <Wn>, <Wm>{, <shift> #<amount>}



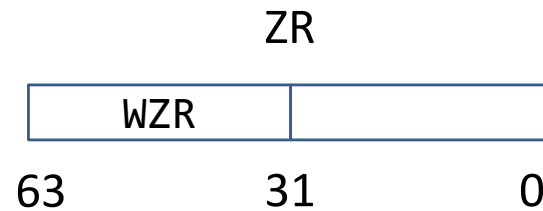
ARM-v8A指令：读取NZCV

- 基于减法实现，更新CPSR寄存器：
 - N（31位）：符号标志位；如果负，则N=1
 - Z（30位）：0标志位；如果0，则Z=1
 - C（29位）：进位标志位；
 - 无符号数：加法进位，或减法不借位，则C=1
 - V（28位）：溢出标志位；有符号运算溢出，则V=1

Result	N	Z	C	V
Greater than	0	0	1	0
Less than	1	0	0	0
Equal	0	1	1	0

```
mrs x0, nzcv
```

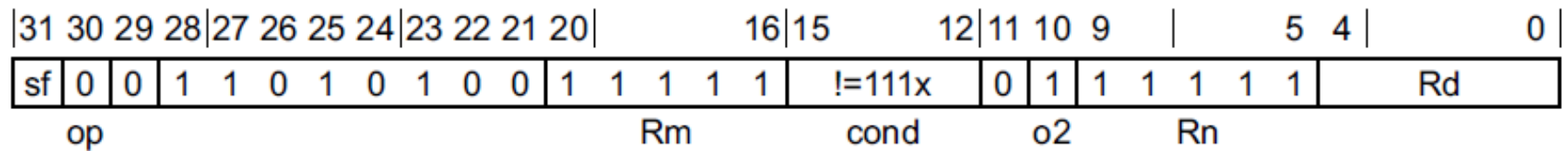
ARM-v8A寄存器：零寄存器



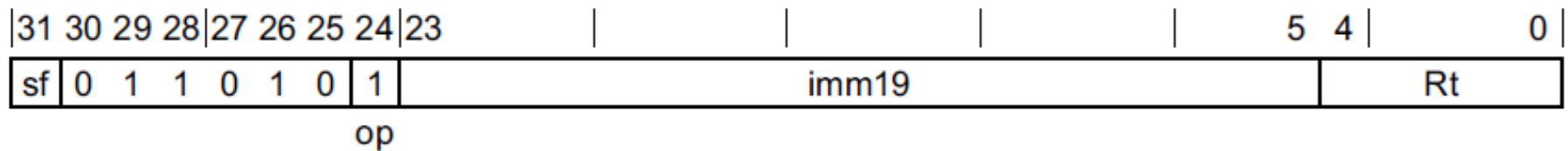
```
mov x1, xzr    x1 = 0
mov w1, wzr     w2 = 0
```

ARM v8a指令：条件指令（举例）

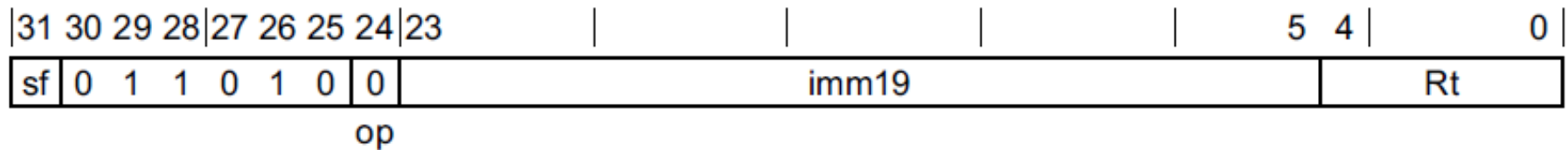
CSET <Xd>, <cond>



CBNZ <Xt>, <label>



CBZ <Xt>, <label>



IR=>Assembly: 跳转语句

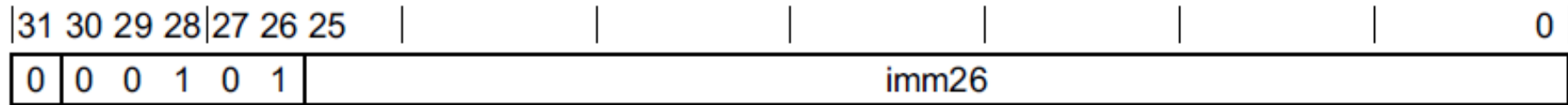
```
bb0:  
    %r1 = icmp sgt i32 %0, %1  
    br i1 %r1, label %bb1, label %bb2  
bb1:  
    br label %bb3  
bb2:  
    br label %bb3  
bb3:
```



```
cmp     w0, w1  
b.le    .LBB0_2  
b       .LBB0_3  
.LBB0_2:  
.LBB0_3:
```

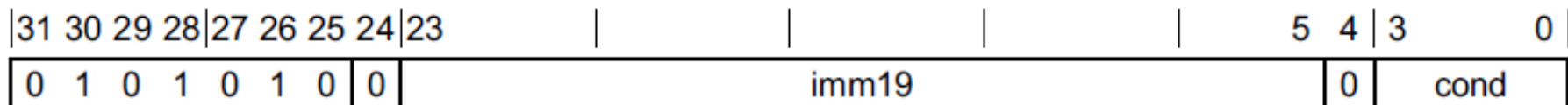
ARM-v8A指令：跳转指令

B <label>



op

B.<cond> <label>



Mnemonic	Instruction	Branch offset range from the PC
B.cond	Branch conditionally	±1MB
BC.cond	Branch Consistent conditionally	±1MB
CBNZ	Compare and branch if nonzero	±1MB
CBZ	Compare and branch if zero	±1MB
TBNZ	Test bit and branch if nonzero	±32KB
TBZ	Test bit and branch if zero	±32KB

```
add w0, w1, w2
cbz zero_set
...
zero_set:
```

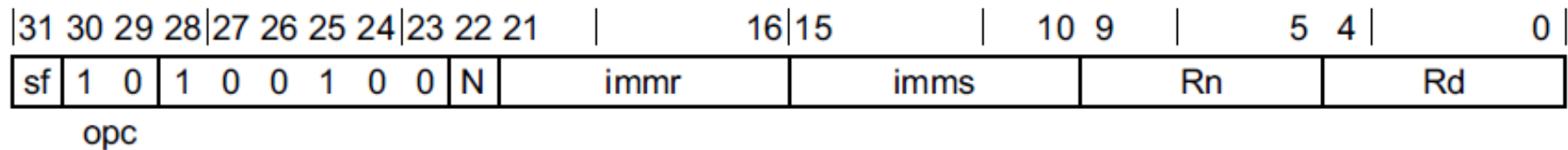
IR=>Assembly: 异或运算（逻辑NOT）

```
%r2 = xor i1 %r1, 0
```

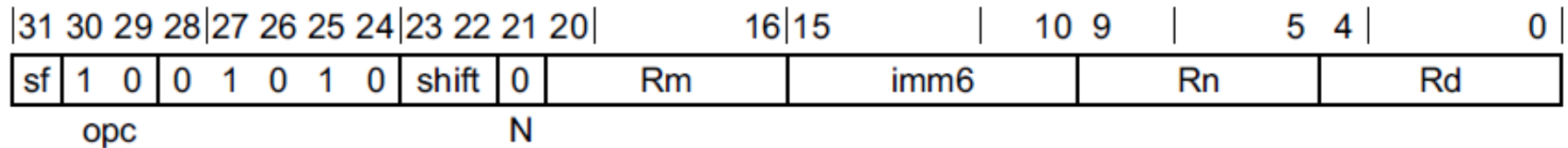


```
eor w0, w0, 1
```

EOR <Xd|SP>, <Xn>, #<imm>



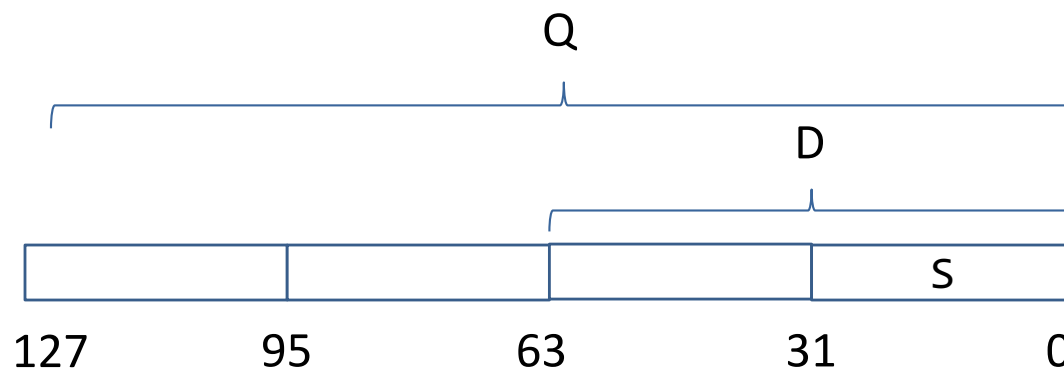
EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}



Mnemonic	Instruction
AND	Bitwise AND
ANDS	Bitwise AND and set flags
EOR	Bitwise exclusive OR
ORR	Bitwise inclusive OR
TST	Test bits

向量寄存器和SIMD指令

- 向量寄存器：Q0-Q31



```
VLDR S0, [X0]  
VLDR S1, [X1]  
VADD.F32 S2, S0, S1
```

浮点数运算

```
VLDR Q0, [X0]  
VLDR Q1, [X1]  
VADD.I32 Q2, Q0, Q1
```

向量运算

IR=>Assembly: 函数

```
@g = global i32 10

define i32 @foo(i32 %0) {
    %x = alloca i32
    store i32 %0, i32* %x
    %g0 = load i32, i32* @g
    ret i32 %g0
}

define i32 @main() {
    %r0 = call i32 @foo(i32 1)
    ret i32 %r0;
}
```



```
foo:
    sub     sp, sp, 16
    adrp    x8, g
    add     x8, x8, :lo12:g
    str     w0, [sp, 12]
    ldr     w0, [x8]
    add     sp, sp, 16
    ret

main:
    str     x30, [sp, -16]!
    mov     w0, #1
    bl     foo
    ldr     x30, [sp], 16
    ret

g:
    .word   10
```

1 word = 4 byte

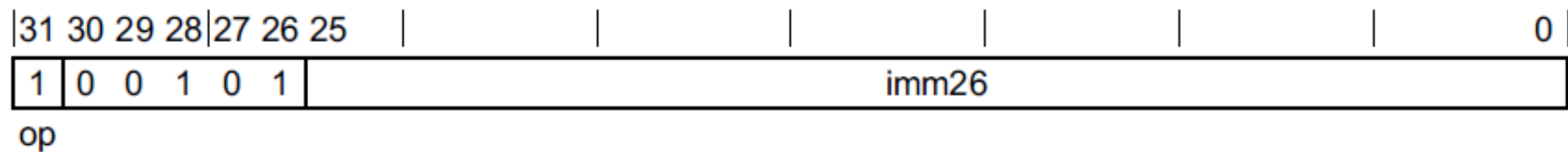
调用规约

- 参数传递: X0-X7
- 返回值: X0-X1
- Caller-saved Registers: X9-X15 （临时寄存器）
- Callee-saved Registers: X19-X28
- X29：一般用于栈帧基指针
- X30：一般用于返回地址
- SP：栈顶指针

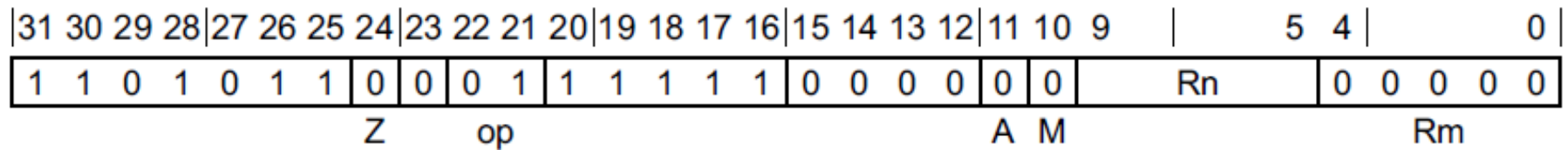
ARM-v8A指令：函数调用

- 跳转并将X30设置为PC+4

BL <label>



BLR <Xn>



IR=>Assembly: 取址（全局变量）

Mnemonic	Instruction
ADRP	Compute address of 4KB page at a PC-relative offset
ADR	Compute address of label at a PC-relative offset.

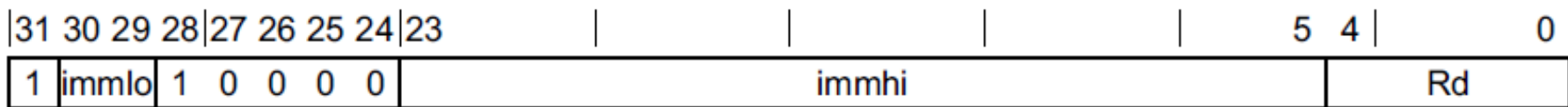
4KB对齐：末尾12位为0

```
adrp    x8, g
add     x8, x8, :lo12:g
```

或：

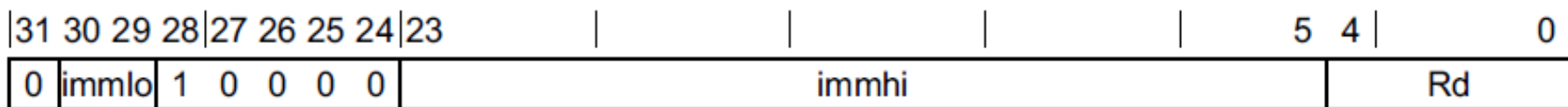
```
adrp    x8, g@PAGE
add     x8, x8, g@PAGEOFF
```

ADRP <Xd>, <label>



op

ADR <Xd>, <label>



op

二、phi指令处理

phi指令的引入

```
if(a==0) {  
    a = a + b;  
}  
let r = a + c;
```

```
bb1:  
    %r1 = icmp eq i32 %a1, 0  
    br i1 %r1, label %bb2, label %bb3  
bb2:  
    %a2 = add i32 %a1, %b1  
    br label %bb3  
bb3:  
    %a3 = phi i1 [%a1, %bb1], [%a2, %bb2]  
    %r1 = add i32 %a3, %b1
```

消除phi指令方式一：还原store-load

```
bb1:
    %a = alloca i32
    %r1 = icmp eq i32 %a1, 0
    store i32 %a1, i32* %a
    br i1 %r1, label %bb2, label %bb3
bb2:
    %a2 = add i32 %a1, %b1
    store i32 %a2, i32* %a
    br label %bb3
bb3:
    %a3 = load i32, i32* %a
    %r1 = add i32 %a3, %b1
```

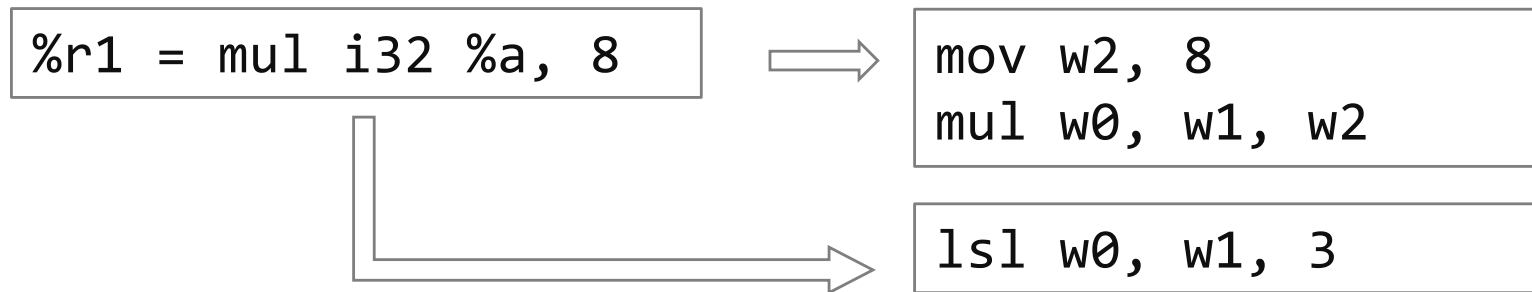
消除phi指令方式二：使用伪指令

```
bb1:
    %r1 = icmp eq i32 %a1, 0
    %a3 = %a1
    br i1 %r1, label %bb2, label %bb3
bb2:
    %a2 = add i32 %a1, %b1
    %a3 = %a2
    br label %bb3
bb3:
    %a3 = phi i1 [%a1, %bb1], [%a2, %bb2]
    %r1 = add i32 %a3, %b1
```

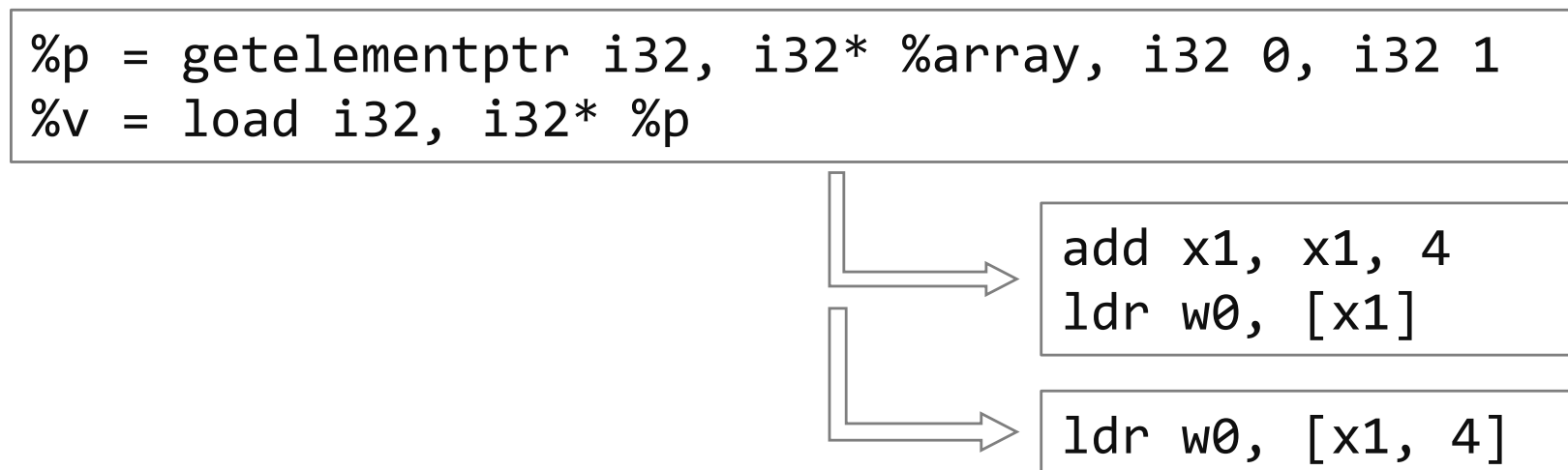
三、指令选择问题

IR指令存在多种ASM翻译方式

一条IR指令，多种ASM翻译方式



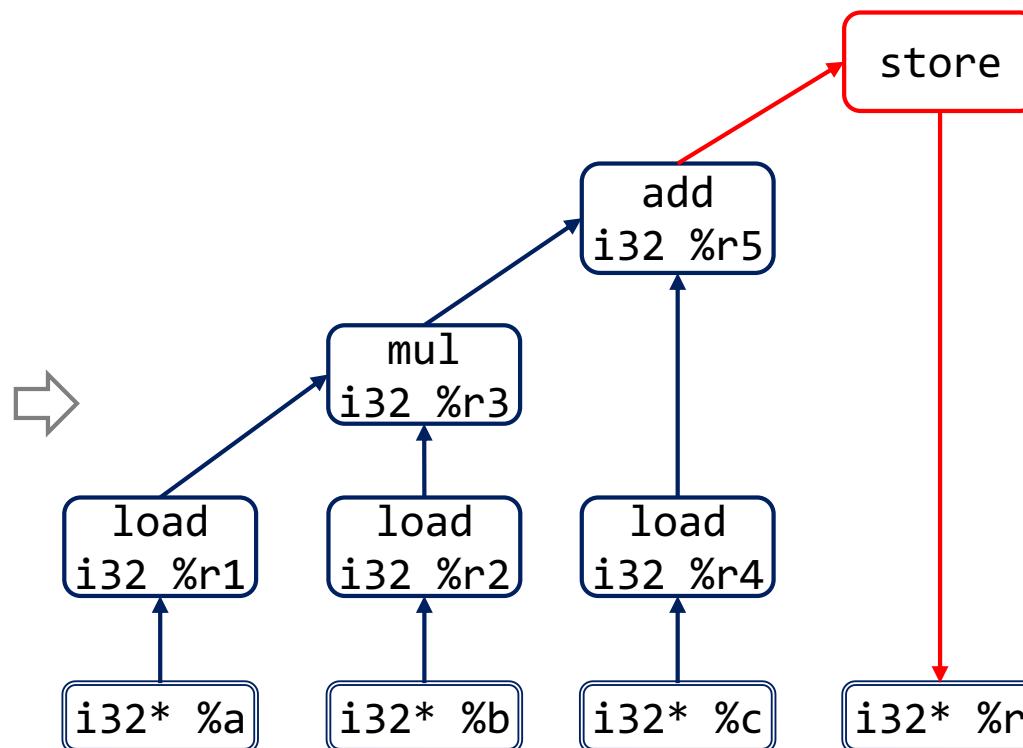
IR指令组合，多种ASM翻译方式



数据流分析

```
r = a * b + c;
```

```
%r1 = load i32 %a;  
%r2 = load i32 %b;  
%r3 = mul i32 %r1, %r2;  
%r4 = load i32 %c;  
%r5 = add i32 %r3, %r4;  
store i32 %r5, %r;
```



指令数据流图

翻译汇编指令满足拓扑顺序即可

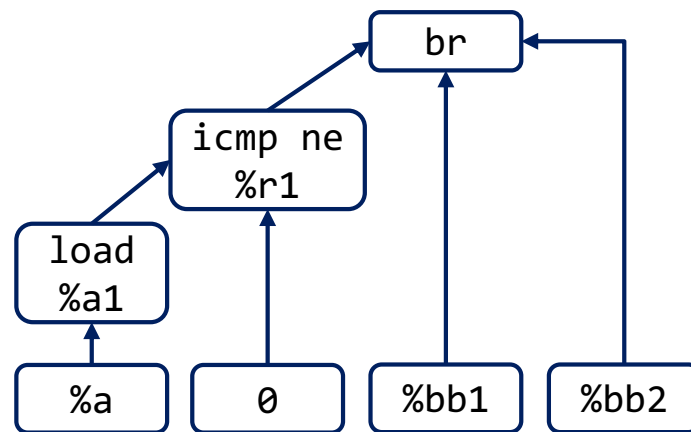
单个基本块的数据流图：可能有环

```
if(a==0) {  
    a = a + b;  
}  
let r = a + c;
```

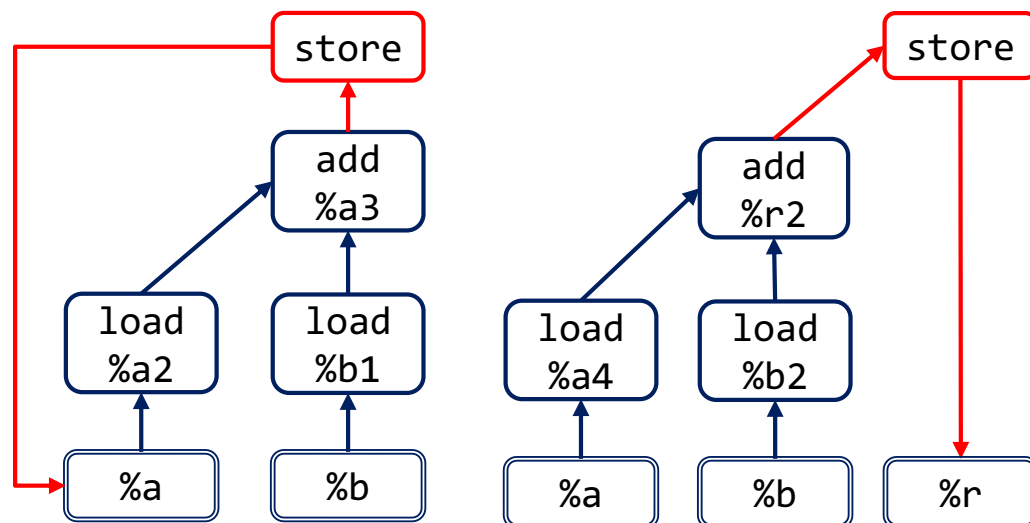
```
bb1:  
    %a1 = load i32, i32* %a  
    %r1 = icmp eq i32 %a1, 0  
    br i1 %r1, label %bb2,  
        label %bb3
```

```
bb2:  
    %a2 = load i32, i32* %a  
    %b1 = load i32, i32* %b  
    %a3 = add i32 %a2, %b1  
    store i32 %a3, i32* %a  
    br label %bb3
```

```
bb3:  
    %a4 = load i32, i32* %a  
    %b2 = load i32, i32* %b  
    %r2 = add i32 %a4, %b2  
    store i32 %r2, i32* %r
```



%bb1 指令数据流图



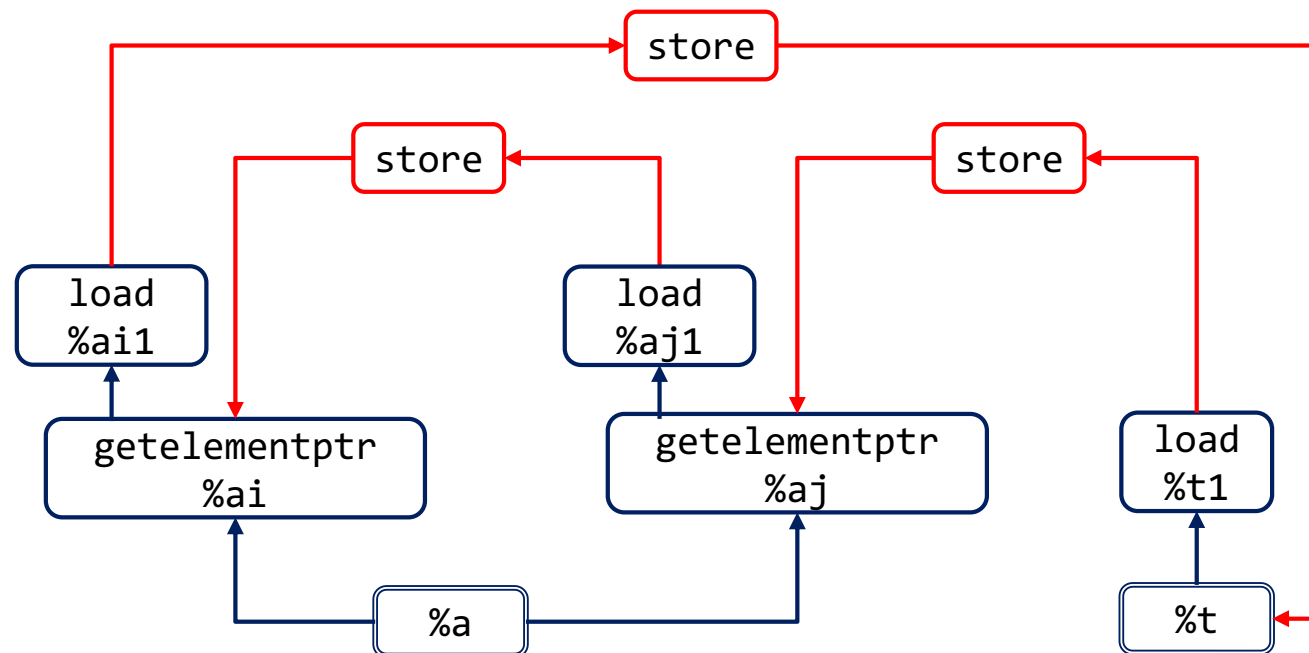
%bb2 指令数据流图

%bb3 指令数据流图

数组的情况：多种load-store顺序

```
a[i] = a[j]  
a[j] = t  
t = a[i]
```

```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i  
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j  
%aj1 = load i32, i32* %aj  
store i32 %aj1, i32* %ai  
%t1 = load i32, i32* %t  
store i32 %t1, i32* %aj  
%ai1 = load i32, i32* %ai  
store i32 %ai1, i32* %t
```

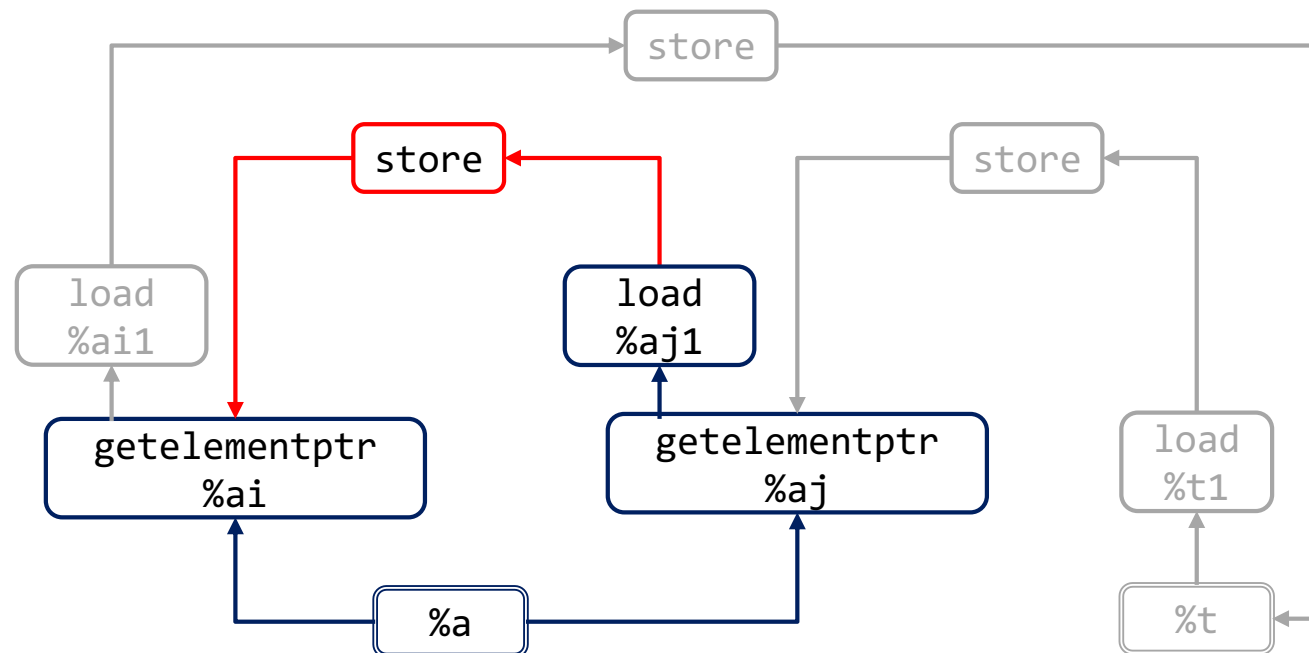


指令数据流图

数组的情况：内存同步问题

```
a[i] = a[j]
a[j] = t
t = a[i]
```

```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j
%aj1 = load i32, i32* %aj
store i32 %aj1, i32* %ai
%t1 = load i32, i32* %t
store i32 %t1, i32* %aj
%ai1 = load i32, i32* %ai
store i32 %ai1, i32* %t
```

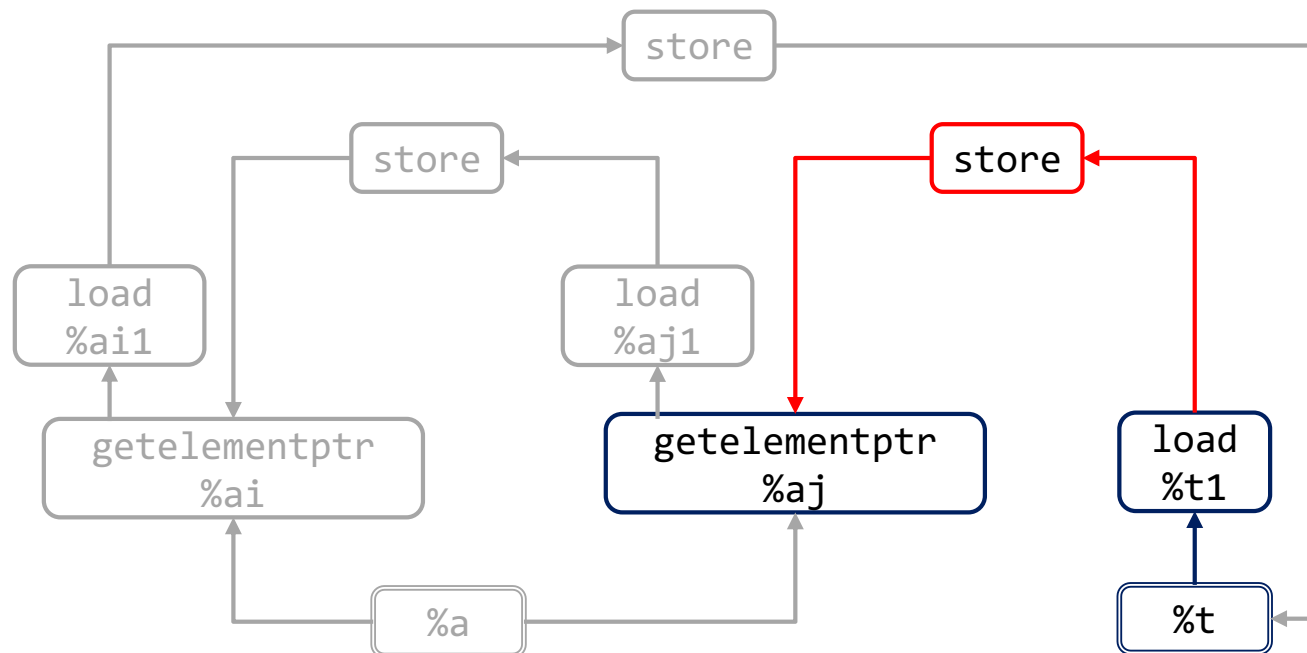


指令数据流图

数组的情况：内存同步问题

```
a[i] = a[j]
a[j] = t
t = a[i]
```

```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j
%aj1 = load i32, i32* %aj
store i32 %aj1, i32* %ai
%t1 = load i32, i32* %t
store i32 %t1, i32* %aj
%ai1 = load i32, i32* %ai
store i32 %ai1, i32* %t
```

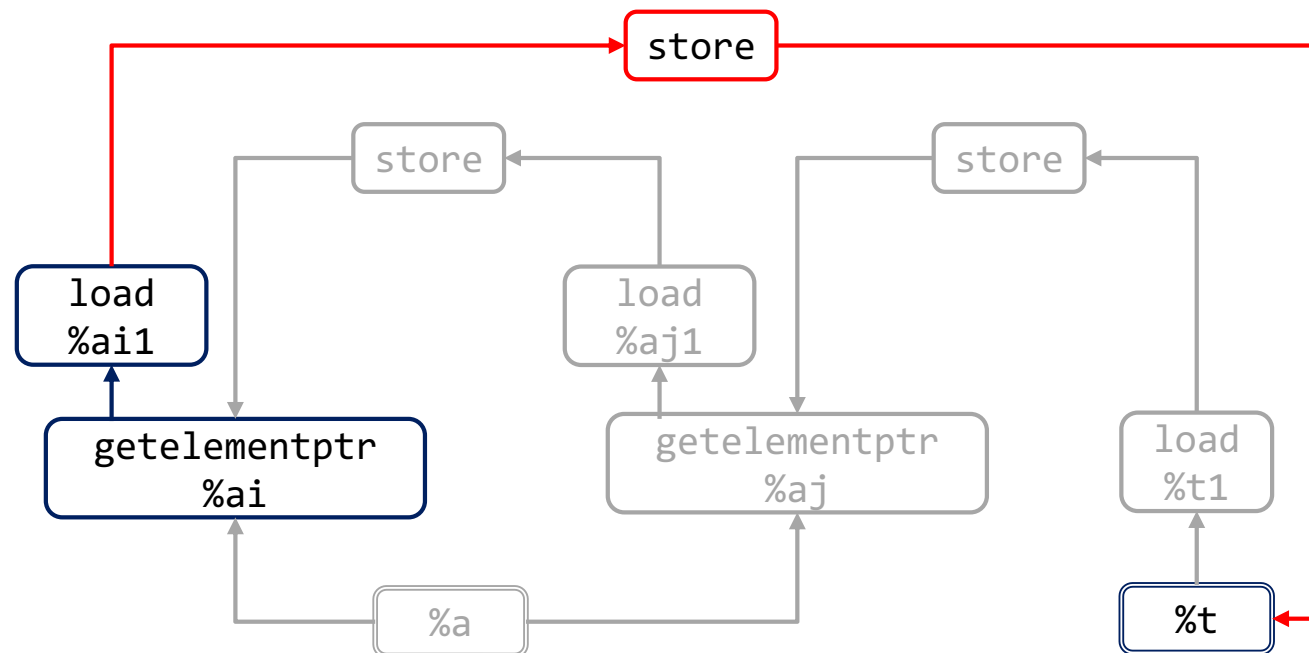


指令数据流图

数组的情况：内存同步问题

```
a[i] = a[j]  
a[j] = t  
t = a[i]
```

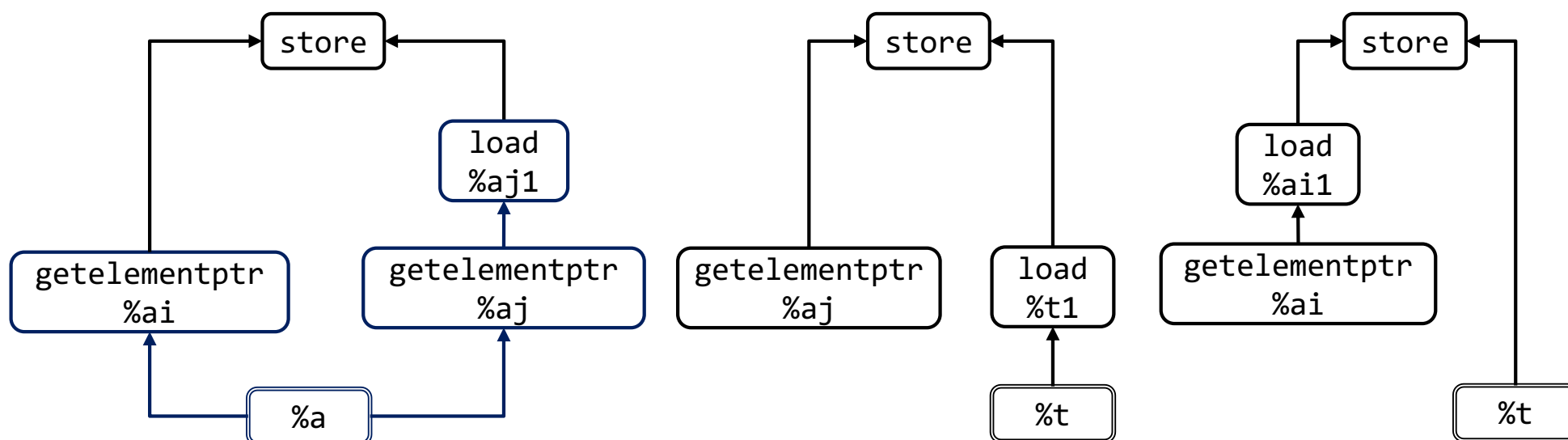
```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i  
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j  
%aj1 = load i32, i32* %aj  
store i32 %aj1, i32* %ai  
%t1 = load i32, i32* %t  
store i32 %t1, i32* %aj  
%ai1 = load i32, i32* %ai  
store i32 %ai1, i32* %t
```



指令数据流图

指令选择图：有向无环图

- 翻译汇编指令满足拓扑顺序即可
- 除了store外，函数调用也应当作为同步原语



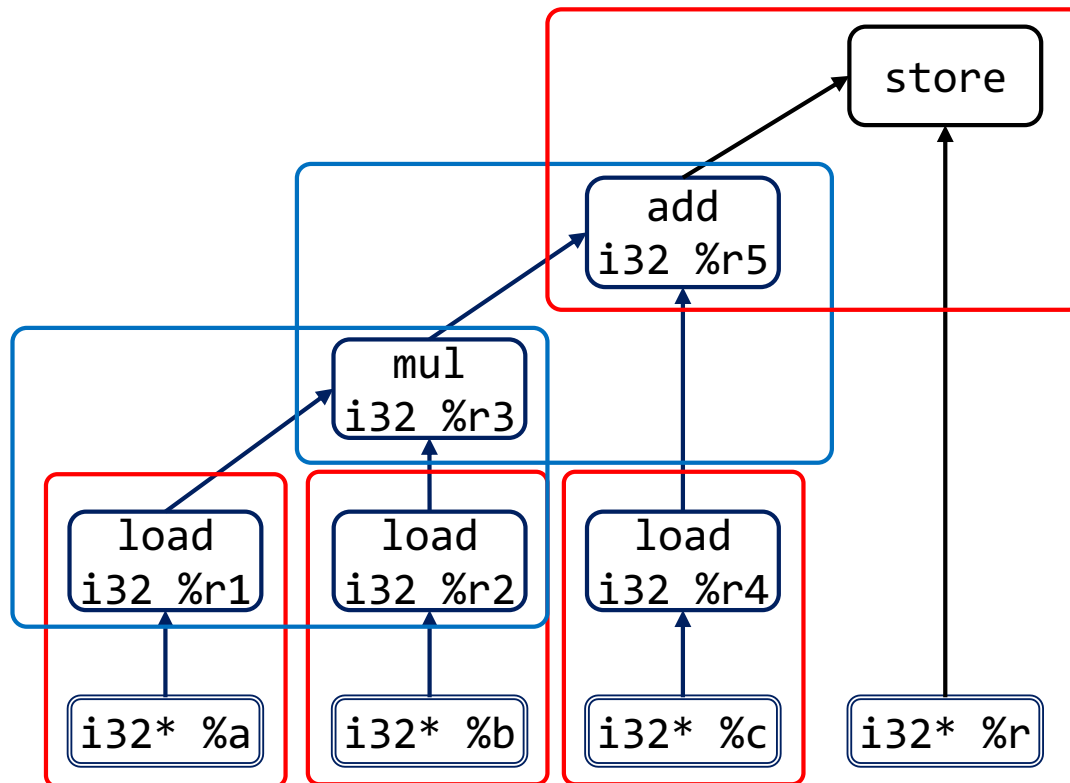
指令选择问题=>铺树问题

- 输入Selection DAG，如何铺树（图）使得最终的汇编代码：
 - 体积小（指令数少）
 - 运算快

指令	开销
LDR	4
STR	1
ADD/ADD	1
SUB/SUBS	1
MUL	3
MADD/MSUB	3
SDIV	4-20
MOV	1
ADR/ADRP	1
B/BL/RET	1
CBZ/TBZ...	1

指令运行开销假设
(Arm Cortex-A72)

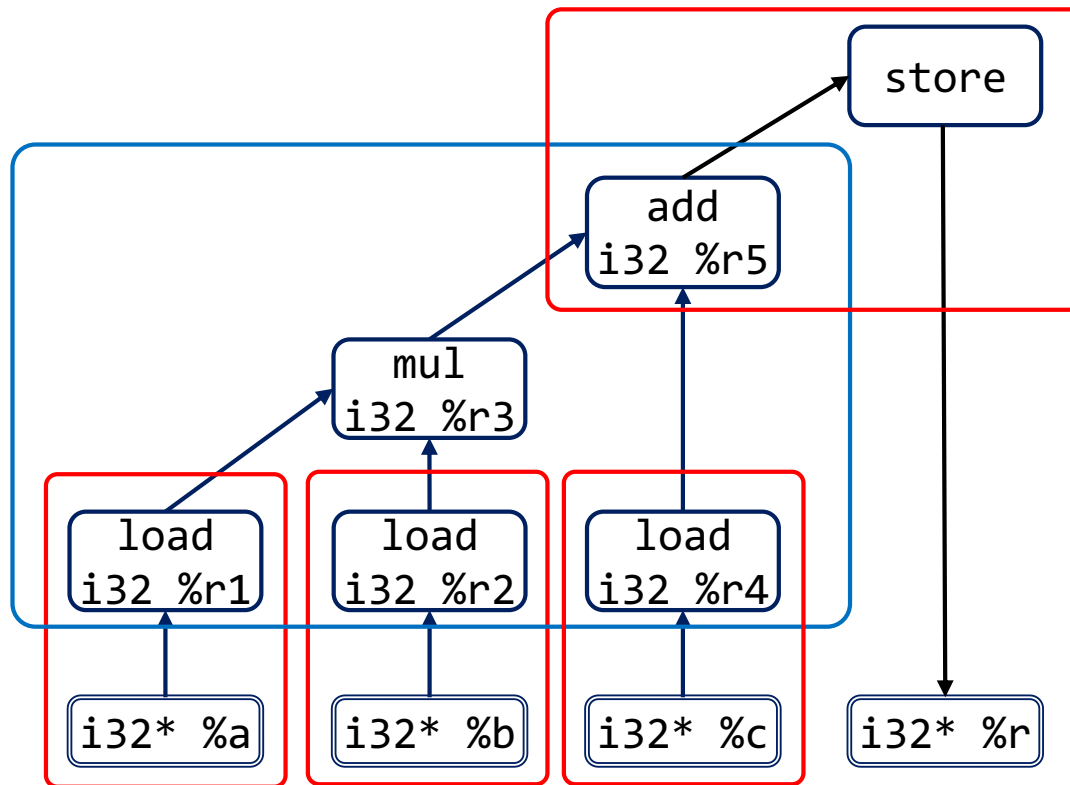
指令选择问题=>铺树问题



```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
ldr w3, [sp, .c]
mul w3, w1, w2
add w5, w3, w4
str w5, [sp, .r]
```

方式一

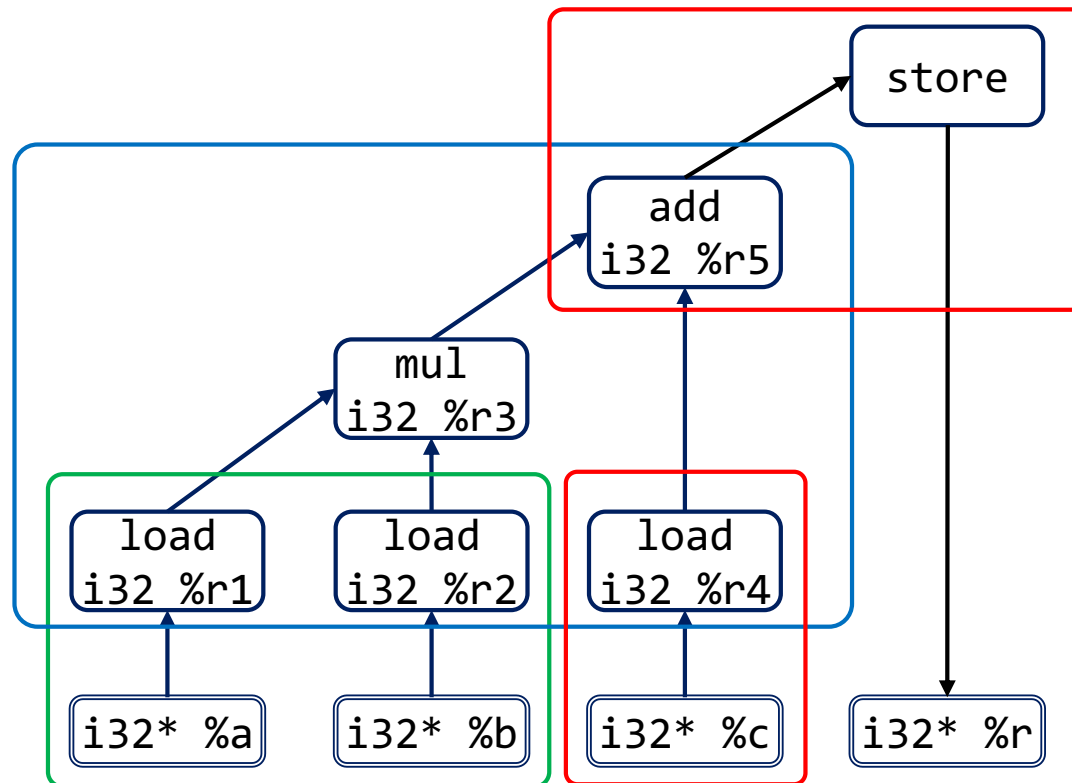
指令选择问题=>铺树问题



```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
ldr w4, [sp, .c]
madd w5, w1, w2, w4
store w5, [sp, .r]
```

方式二

指令选择问题=>铺树问题

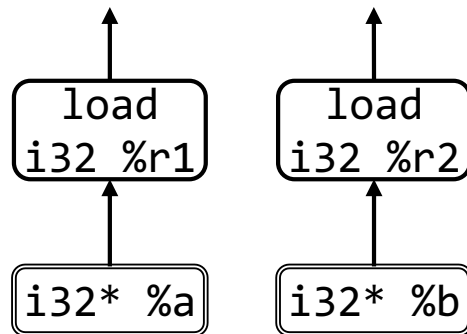


```
ldp w1, %r2, [sp, .a]
ldr w3, [sp, .c]
madd w5, w1, w2, r4
store w5, [sp, .r]
```

方式三

load + load/store + store

- 假设a和b的地址连续

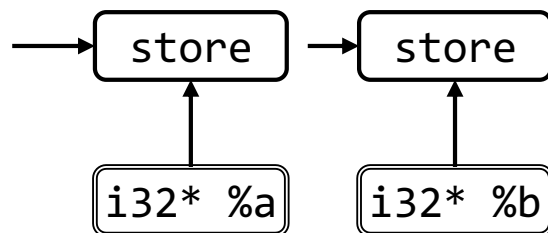


```
ldr w1, [sp, .a]  
ldr w2, [sp, .b]
```

开销: 8

```
ldp w1, w2, [sp, .a]
```

开销: 4



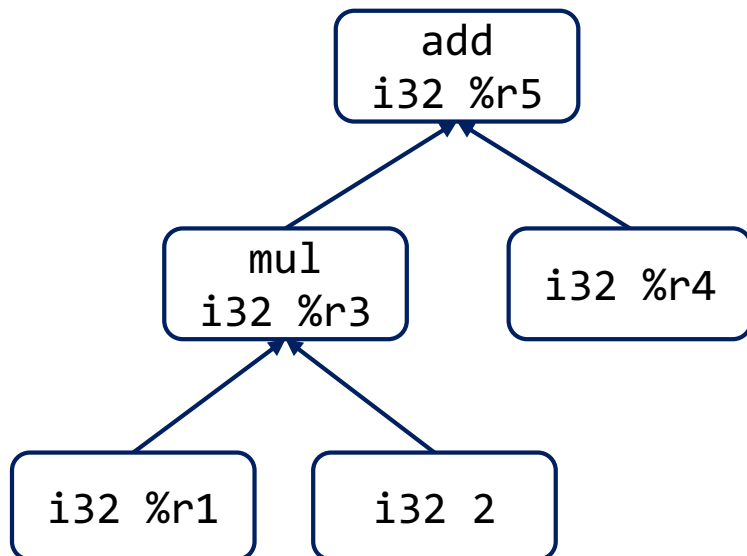
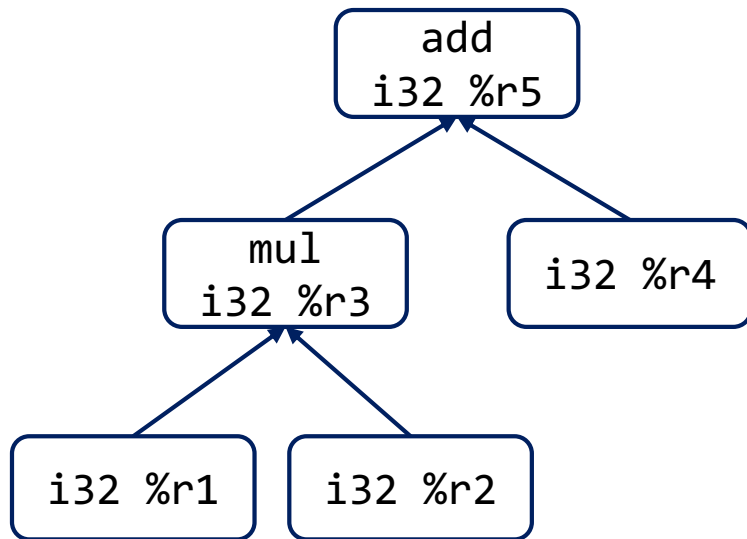
```
str w1, [sp, .a]  
str w2, [sp, .b]
```

开销: 2

```
stp w1, w2, [sp, .a]
```

开销: 1

mul + add



```
mul w3, w1, w2
add w5, w3, w4
```

开销: 4

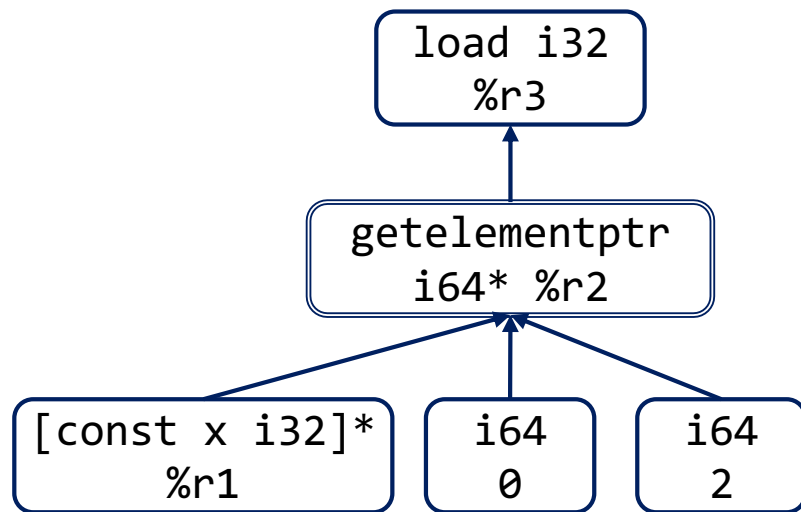
```
madd w5, w1, w2, w4
```

开销: 3

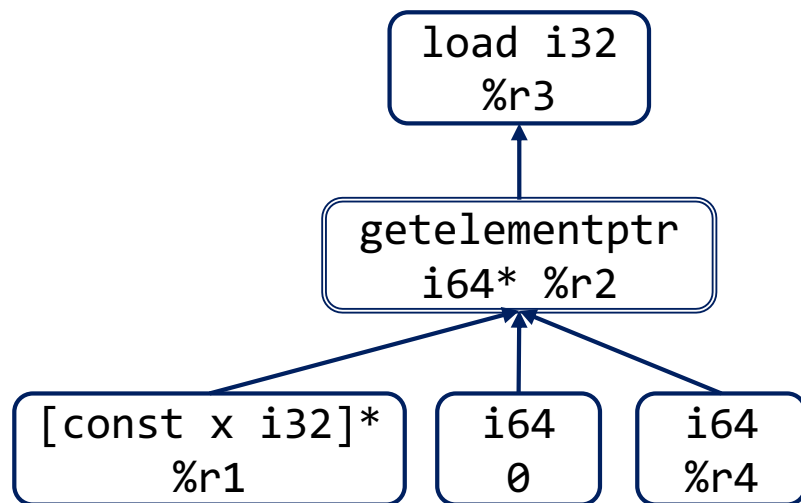
```
mov w0, 2
mul w3, w1, w0
add w5, w3, w4
```

开销: 5

load + getelementptr: 数组



常量索引



变量索引

```
add .w2, w1, 16
ldr .w3, [w2]      开销: 5
```

```
ldr %w3, [w1, 16]  开销: 4
```

```
mov w0, 8
mul w2, w4, w0
add w2, w1, w2
ldr w3, [w2]      开销: 9
```

```
mov %w0, 8
mul %w2, w4, w0
ldr %w3, [w1, w2]  开销: 8
```

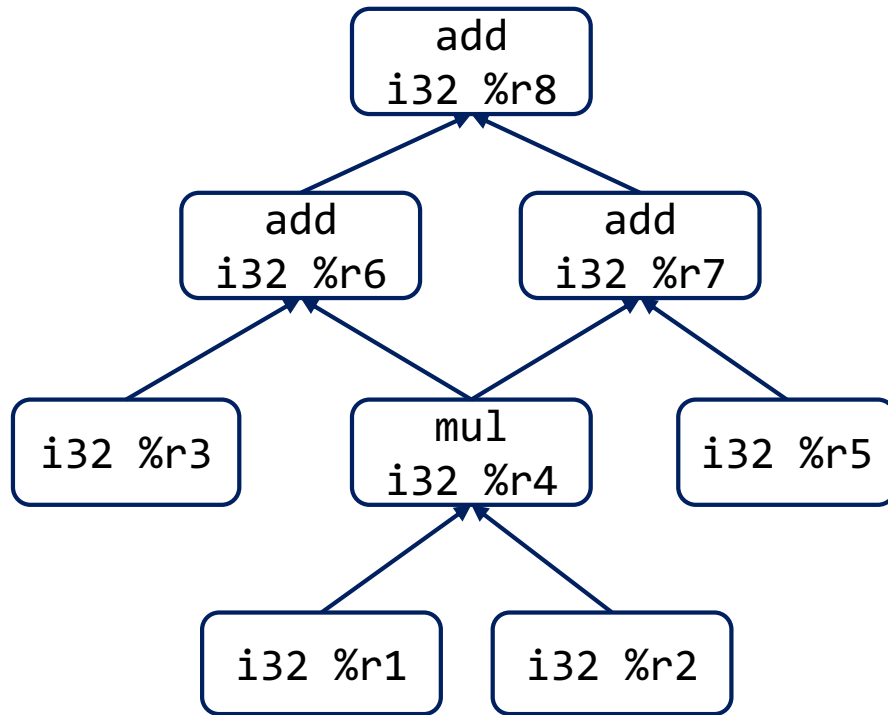
```
ldr %w3, [w1, w4, lsl 3]
```

开销: 4

铺树问题解法

- 贪心算法：Maximal Munch
 - 每次选择覆盖节点最多、开销最低的规则
 - 拓扑排序：生成汇编指令
 - 局部最优
- 动态规划
 - 从树根开始，递归搜索每个节点的最优方案

贪心法不一定能得到最优解: mul + add



```
mul w4, w1, w2
add w6, w3, w4
add w7, w4, w5
add w8, w6, w7
```

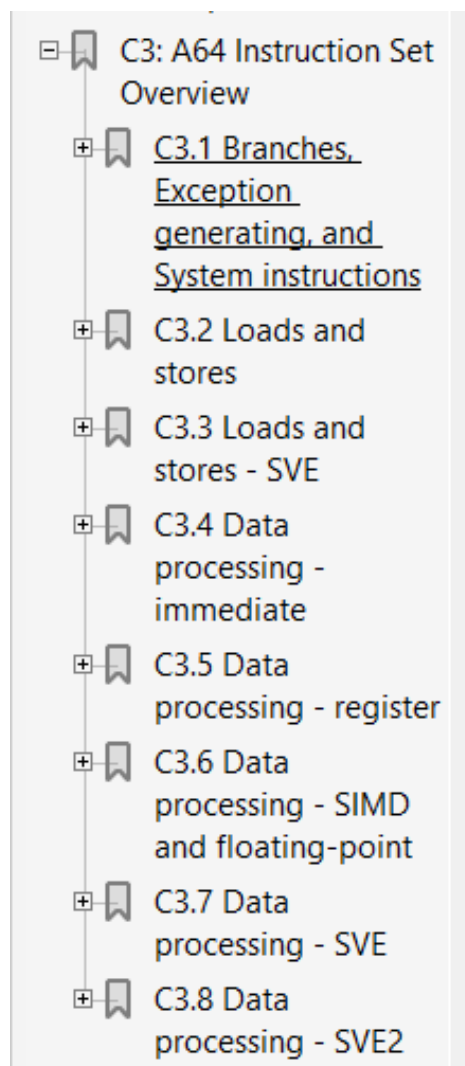
开销: 6

```
madd w6, w1, w2, w3
madd w7, w1, w2, w5
add w8, w6, w7
```

开销: 7

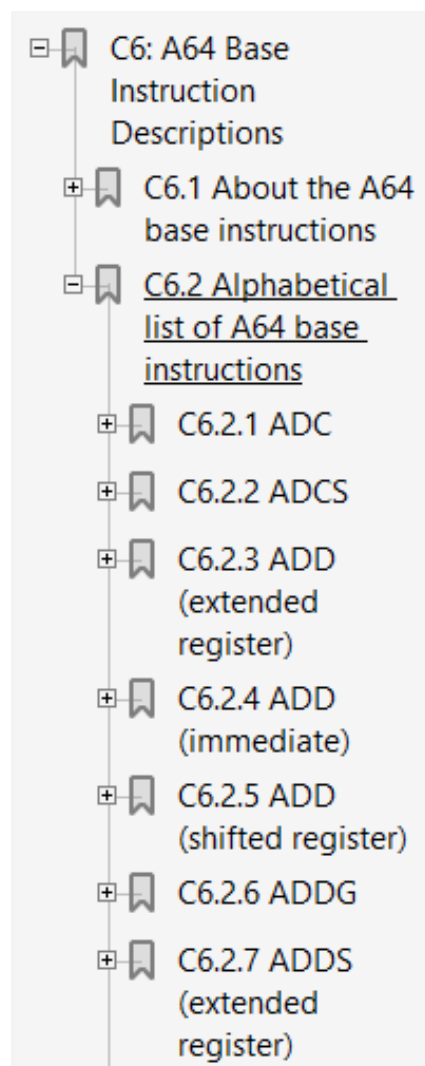
参考资料

- Arm® Architecture Reference Manual for A-profile architecture
- 在线模拟器： <http://163.238.35.161/~zhangs/arm64simulator/>



C3: A64 Instruction Set Overview

- + C3.1 Branches, Exception generating, and System instructions
- + C3.2 Loads and stores
- + C3.3 Loads and stores - SVE
- + C3.4 Data processing - immediate
- + C3.5 Data processing - register
- + C3.6 Data processing - SIMD and floating-point
- + C3.7 Data processing - SVE
- + C3.8 Data processing - SVE2



C6: A64 Base Instruction Descriptions

- + C6.1 About the A64 base instructions
- + C6.2 Alphabetical list of A64 base instructions
 - + C6.2.1 ADC
 - + C6.2.2 ADCS
 - + C6.2.3 ADD (extended register)
 - + C6.2.4 ADD (immediate)
 - + C6.2.5 ADD (shifted register)
 - + C6.2.6 ADDG
 - + C6.2.7 ADDS (extended register)