
第三章 程序的转换与指令系统

程序转换概述

LA32指令系统

LA64指令系统

程序的转换与指令系统

- 主要教学目标

- 了解高级语言程序转换为目标代码的过程
- 了解汇编语言与机器语言之间的关系
- 了解指令集体系结构的基本内容和指令系统风格
- 掌握LA32/LA64的指令格式、操作数类型、寻址方式等
- 掌握LA32/LA64常用指令类型和常用指令功能

- 主要教学内容

- 程序转换：机器指令和汇编指令、ISA、机器代码的生成过程
- LA32/LA64指令系统概述
- LA32/LA64 常用指令类型和常用指令功能

程序的转换与指令系统

- 分以下五个部分介绍

- **第一讲：程序转换概述**

- 机器指令和汇编指令
 - 机器级程序员感觉到的属性和功能特性
 - 高级语言程序转换为机器代码的过程

- **第二讲：LA32/LA64指令系统**


- 机器指令格式和数据类型
 - 寄存器组织和寻址方式

- **第三讲：LA32/LA64基础整数指令**

- 整数运算类指令和移位指令
 - 普通访存指令
 - 程序执行流控制指令

- **第四讲：LA32/LA64基础浮点指令**

- 浮点普通访存指令
 - 浮点运算类指令
 - 浮点转换指令和传送指令



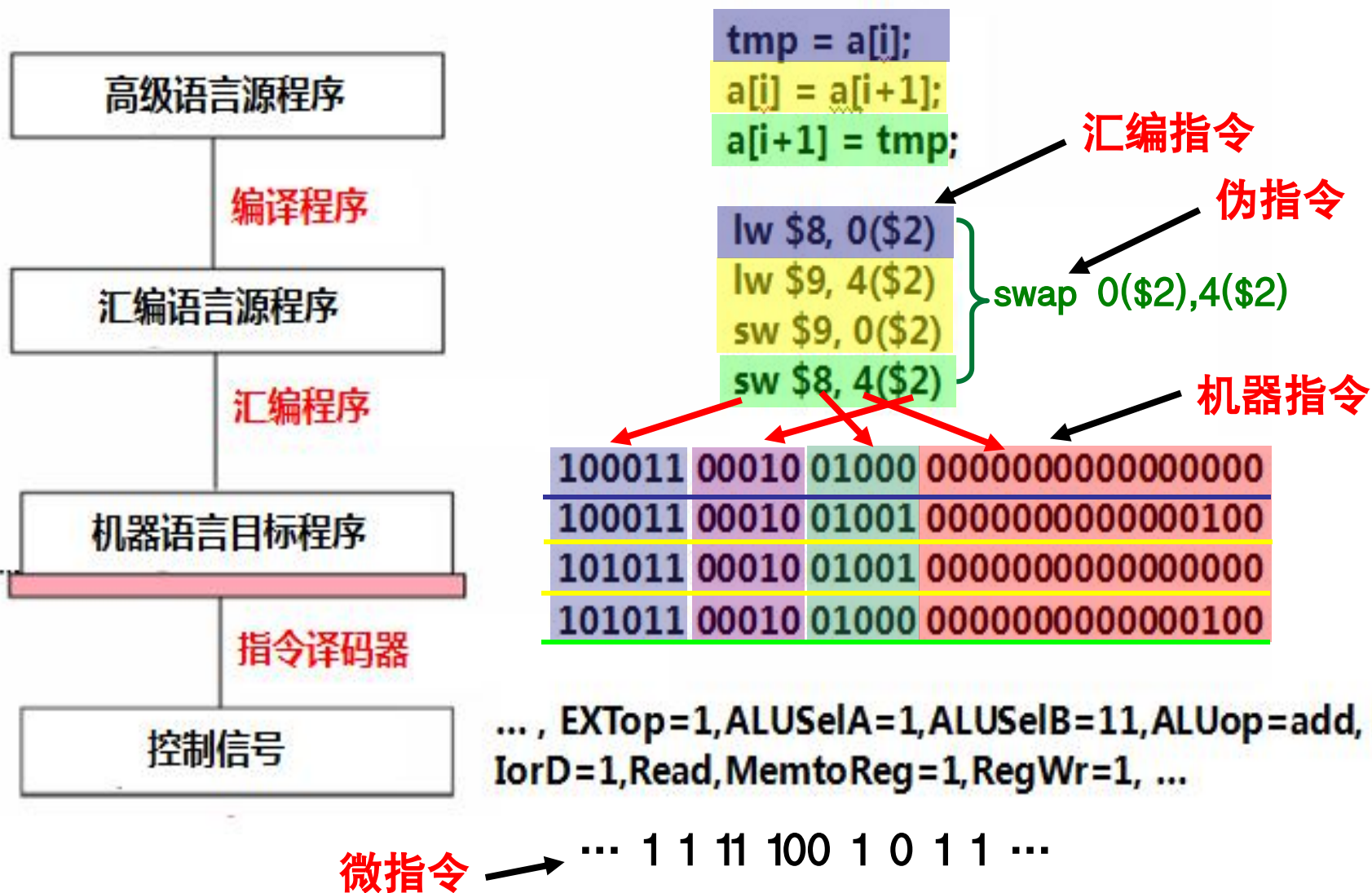
围绕LA32/LA64指令系统，解释具体的指令功能

“指令”的概念

- 有**微指令**、**机器指令**、**汇编指令**、**伪（宏）指令**等指令相关概念
- 微指令是微程序级命令，属于硬件范畴
- **机器指令**介于二者之间，处于硬件和软件的交界面
 - 本章中提及的指令都指机器指令
- **汇编指令**是机器指令的汇编表示形式，即符号表示
- 机器指令和汇编指令一一对应，它们都与具体机器结构有关，都属于**机器级指令**
- **伪指令**是由若干汇编指令组成的序列，属于软件范畴，如：
LA指令 “or \$r4, \$r12, \$r0” 可表示为伪指令 “move \$r4, \$r12”
RV32I中有60条伪指令，其中，指令 “addi a4,a5,0” 可表示为伪指令 “mv a4,a5”

不同层次语言之间的等价转换

计算机软件
计算机硬件



回顾：程序和指令执行过程举例

假设模型机M中8位指令，格式有两种：R型、M型

R—寄存器（Register） M—存储器（Memory）

格式	4位	2位	2位	功能说明
R型	op	rt	rs	$R[rt] \leftarrow R[rt] \text{ op } R[rs]$ 或 $R[rt] \leftarrow R[rs]$
M型	op	addr		$R[0] \leftarrow M[addr]$ 或 $M[addr] \leftarrow R[0]$

rs和rt为通用寄存器编号； addr为主存单元地址

R型： op=0000，寄存器间传送（mov）； op=0001，加（add）

M型： op=1110，取数（load）； op=1111，存数（store）

问题：指令 1110 0111的功能是什么？

答：因为op=1110，故是M型load指令，功能为：

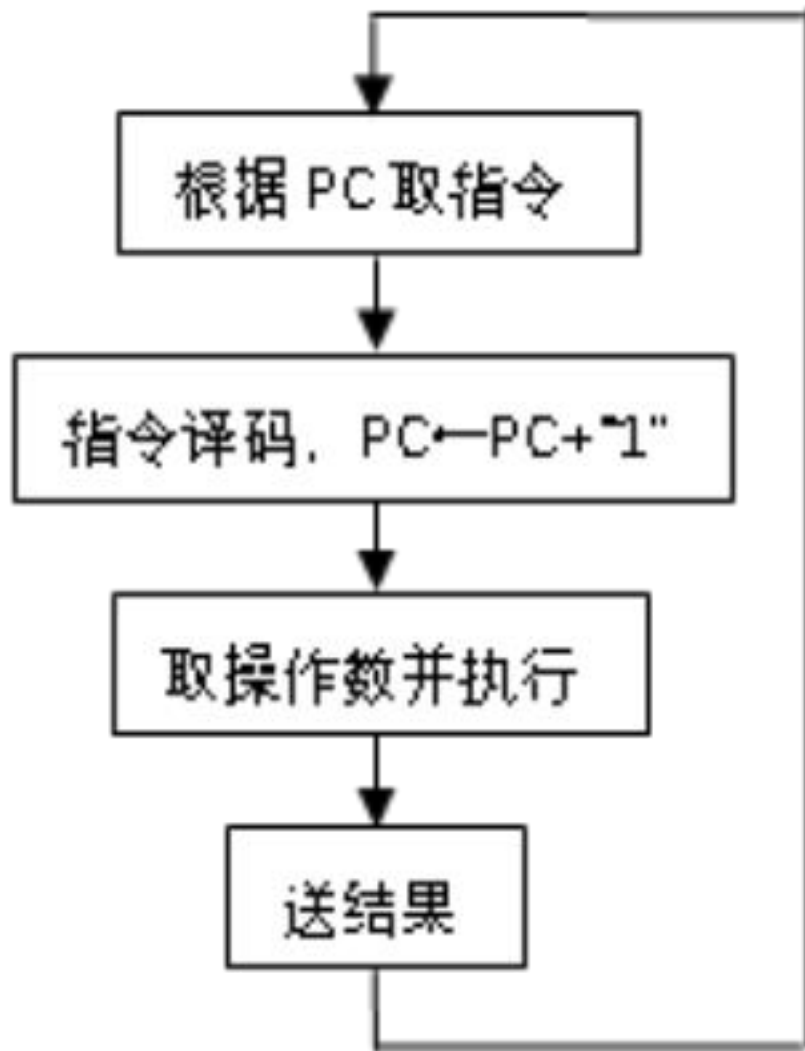
$R[0] \leftarrow M[0111]$ ，即：将主存地址0111（7号单元）中的8位数据装入到0号寄存器中。

回顾：程序和指令执行过程举例

若在M上实现“ $z=x+y$ ”，x和y分别存放在主存5和6号单元中，结果z存放在7号单元中，则程序在主存单元中的初始内容为：

主存地址	主存单元内容	内容说明（li表示第i条指令）	指令的符号表示
0	1110 0110	I1: $R[0] \leftarrow M[6]$; op=1110: 取数操作	load r0, 6#
1	0000 0100	I2: $R[1] \leftarrow R[0]$; op=0000: 传送操作	mov r1, r0
2	1110 0101	I3: $R[0] \leftarrow M[5]$; op=1110: 取数操作	load r0, 5#
3	0001 0001	I4: $R[0] \leftarrow R[0] + R[1]$; op=0001: 加操作	add r0, r1
4	1111 0111	I5: $M[7] \leftarrow R[0]$; op=1111: 存数操作	store 7#, r0
5	0001 0000	操作数x, 值为16	程序执行过程 及其结果是什么 ?
6	0010 0001	操作数y, 值为33	
7	0000 0000	结果z, 初始值为0	

回顾：程序和指令执行过程举例



程序执行过程

指令I1 (PC=0) 的执行过程

	I1: 1110 0110
取指令	$IR \leftarrow M[0000]$
指令译码	op=1110, 取数
PC增量	$PC \leftarrow 0000 + 1$
取数并执行	$MDR \leftarrow M[0110]$
送结果	$R[0] \leftarrow MDR$
执行结果	$R[0] = 33$

随后执行PC=1中的指令I2

机器级指令

- 机器指令和汇编指令一一对应，都是机器级指令
- 机器指令是一个0/1序列，由若干**字段**组成
- 如龙芯架构（LA32/LA64）中指令：

补码1111 1110 0100的真值为多少？
-28

st.w ↵	si12 ↵	rj ↵	rd ↵
00 1010 0110 ↵	1111 1110 0100 ↵	10110 ↵	01100 ↵

操作码
器编号

立即数(位移量)

寄存

- 汇编指令是机器指令的符号表示（可能有不同的格式）

st.w \$r12, \$r22, -28(0xfe4)

其中，st.w、\$r12、\$r22等都是助记符

指令的功能为： $M[R[r22]-28, \text{WORD}] \leftarrow R[r12]$

R: 寄存器内容
M: 存储单元内容

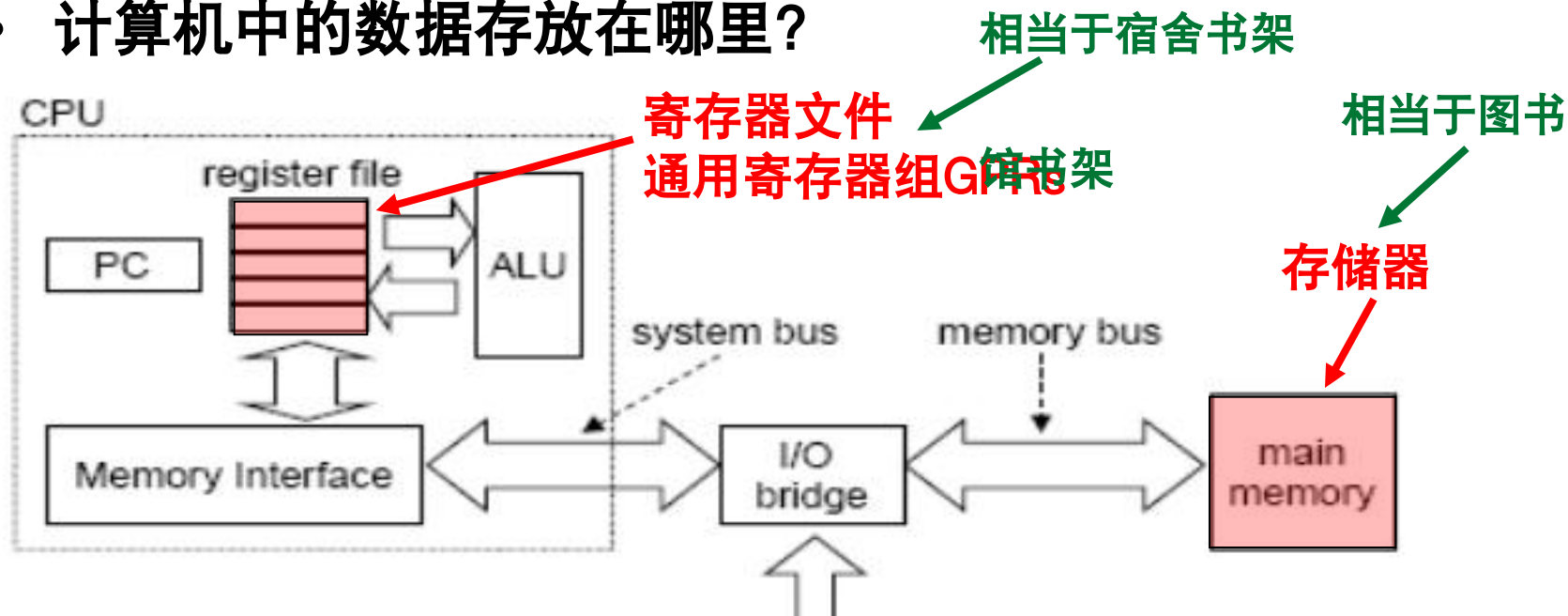
寄存器传送语言 RTL (Register Transfer Language)

回顾：指令集体系结构ISA

- ISA (Instruction Set Architecture) 位于软件和硬件之间
- 硬件的功能通过ISA提供出来
- 软件通过ISA规定的“指令”使用硬件
- ISA规定了：
 - 可执行的指令的集合，包括指令格式、操作种类以及每种操作对应的操作数的相应规定；
 - 指令可以接受的操作数的类型；
 - 操作数所能存放的寄存器组的结构，包括每个寄存器的名称、编号、长度和用途；
 - 操作数所能存放的存储空间的大小和编址方式；
 - 操作数在存储空间存放时按照大端还是小端方式存放；
 - 指令获取操作数的方式，即寻址方式；
 - 指令执行过程的控制方式，包括程序计数器、条件码定义等。

回顾：计算机中数据的存储

- 计算机中的数据存放在哪里？



指令中需给出的信息：

操作性质（操作码）

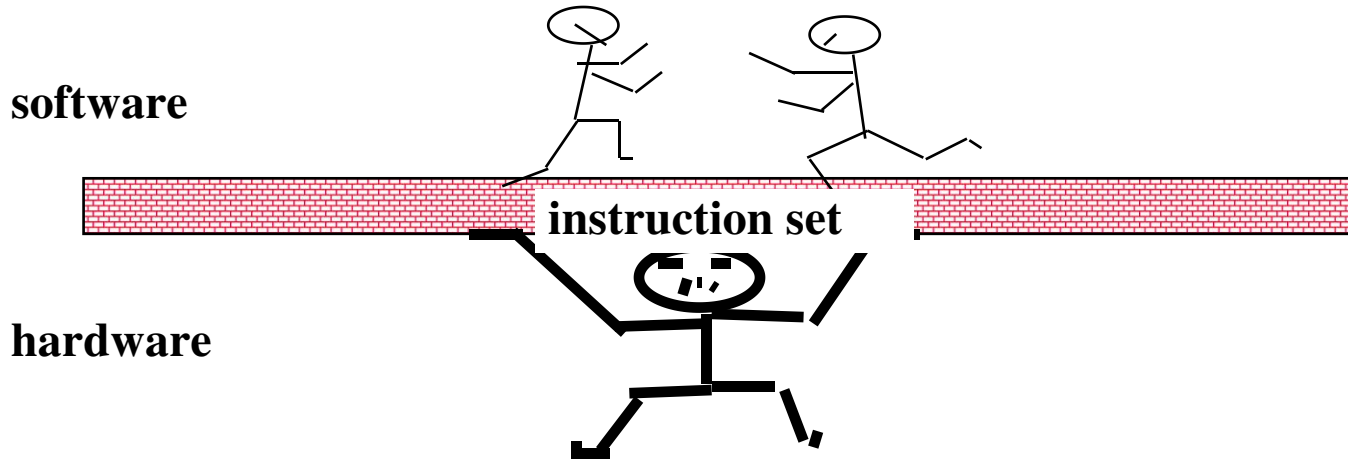
源操作数1 或/和 源操作数2 （立即数、寄存器编号、存储地址）

目的操作数地址 （寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

指令系统概述

- 指令系统处在软/硬件交界面，同时被硬件设计者和系统程序员看到
- 硬件设计者角度：指令系统为CPU提供功能需求，要求易于硬件设计
- 系统程序员角度：通过指令系统来使用硬件，要求易于编写编译器
- 指令系统设计的好坏还决定了：计算机的性能和成本



回顾：冯·诺依曼结构机器对指令规定：

- u 用二进制表示，和数据一起存放在主存中
- u 由两部分组成：操作码和操作数（或其地址码）
 - **Operation Code**: defines the operation type
 - **Operands**: indicate operation source and destination

一条指令须包含的信息

问题：一条指令必须**明显**或**隐含**包含的信息有哪些？

操作码：指定操作类型

(操作码长度：固定 / 可变)

源操作数参照：一个或多个源操作数所在的地址

(操作数来源：主(虚)存/寄存器/I/O端口/指令本身)

结果值参照：产生的结果存放何处（目的操作数）

(结果地址：主(虚)存/寄存器/I/O端口)

下一条指令地址：下条指令存放何处

(下条指令地址：主(虚)存)

(正常情况隐含在PC中，改变顺序时由指令给出)

指令设计风格 -- 按操作数位置指定风格来分

Accumulator: (earliest machines) 累加器型

特点: 其中一个操作数（源操作数1）和目的操作数总在累加器中

1 address	add A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$
1(+x) address	add x A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A + x]$

Stack: (e.g. HP calculator, Java virtual machines) 栈型

特点: 总是将栈顶两个操作数进行运算，指令无需指定操作数地址

0 address	add	$\text{tos} \leftarrow \text{tos} + \text{next}$
-----------	-----	--

General Purpose Register: (e.g. IA-32, Motorola 68xxx) 通用寄存器型

特点: 操作数可以是寄存器或存储器数据（即A、B和C可以是寄存器或存储单元）

2 address	add A B	$\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$
3 address	add A B C	$\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

Load/Store: (e.g. SPARC, MIPS, RISC-V) 装入/存储型

特点: 运算操作数只能是寄存器数据，只有load/store能访问存储器

3 address	add Ra Rb Rc	$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$
	load Ra Rb	$\text{Ra} \leftarrow \text{mem}[\text{Rb}]$
	store Ra Rb	$\text{mem}[\text{Rb}] \leftarrow \text{Ra}$

指令风格比较

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

。 Code sequence for $C = A + B$ for four classes of instruction sets:

Stack	Accumulator	Register (register- memory)	Register (load - store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

指令条数较少

复杂表达式时，累加器型风格指令条数变多，因为所有运算都要用累加器，使得程序中多出许多移入 / 移出累加器的指令！

想象一下 “ $C=a*x+b*y+x*y$ ” 用累加器型风格实现的情况！

75年开始，寄存器型占主导地位，原因：

- 寄存器速度快，使用大量通用寄存器可减少访存操作
- 表达式编译时与顺序无关（相对于Stack）

（Java Virtual Machine 采用Stack型）

Examples of Register Usage

每条典型ALU指令中的存储器地址个数

	每条典型ALU指令中的最多操作数个数	Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC, RISC-V
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

In VAX(CISC): **ADDL (R9), (R10), (R11)** 一条指令!
; mem[R9] ← mem[R10] + mem[R11]

In MIPS(RISC):

哪种CPI小?

MIPS!

lw R1, (R10)	: R1 ← mem[R10]	} 四条指令!
lw R2, (R11)	: R2 ← mem[R11]	
add R3, R1, R2	: R3 ← R1+R2	
sw R3, (R9)	: mem[R9] ← R3	

Examples of Register Usage

每条典型ALU指令中的存储器地址个数

	每条典型ALU指令中的最多操作数个数	Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC, RISC-V
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

In VAX(CISC): **ADDL (R9), (R10), (R11)** 一条指令!
; mem[R9] ← mem[R10] + mem[R11]

In MIPS(RISC):

哪种CPI小?

MIPS!

lw R1, (R10)	: R1 ← mem[R10]	} 四条指令!
lw R2, (R11)	: R2 ← mem[R11]	
add R3, R1, R2	: R3 ← R1+R2	
sw R3, (R9)	: mem[R9] ← R3	

指令设计风格 – 按指令格式的复杂度来分

按指令格式的复杂度来分，有两种类型计算机：

复杂指令集计算机CISC (Complex Instruction Set Computer)

精简指令集计算机RISC (Reduce Instruction Set Computer)

早期CISC设计风格的主要特点

(1) 指令系统复杂

变长操作码 / 变长指令字 / 指令多 / 寻址方式多 / 指令格式多

(2) 指令周期长

绝大多数指令需要多个时钟周期才能完成

(3) 各种指令都能访问存储器

除了专门的存储器读写指令外，运算指令也能访问存储器

(4) 采用微程序控制

(5) 有专用寄存器

(6) 难以进行编译优化来生成高效目标代码

例如，VAX-11/780小型机

16种寻址方式； 9种数据格式； 303条指令；

一条指令包括1~2个字节的操作码和下续N个操作数说明符。

一个说明符的长度达1~10个字节。

复杂指令集计算机CISC

u CISC的缺陷

- 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。

u 1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC (Reduce Instruction Set Computer)。

u 对CISC进行测试，发现一个事实：

- 在程序中各种指令出现的频率悬殊很大，最常使用的是一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%。

u 1982年美国加州伯克利大学的RISC-I，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- Simple instructions dominate instruction frequency

(简单指令占主要部分, 使用频率高!)

[BACK](#)

RISC设计风格的主要特点

(1) 简化的指令系统

指令少 / 寻址方式少 / 指令格式少 / 指令长度一致

(2) 以RR方式工作

除Load/Store指令可访问存储器外，其余指令都只访问寄存器。

(3) 指令周期短

以流水线方式工作，因而除Load/Store指令外，其他简单指令都只需一个或一个不到的时钟周期就可完成。

(4) 采用大量通用寄存器，以减少访存次数

(5) 采用组合逻辑电路控制，不用或少用微程序控制

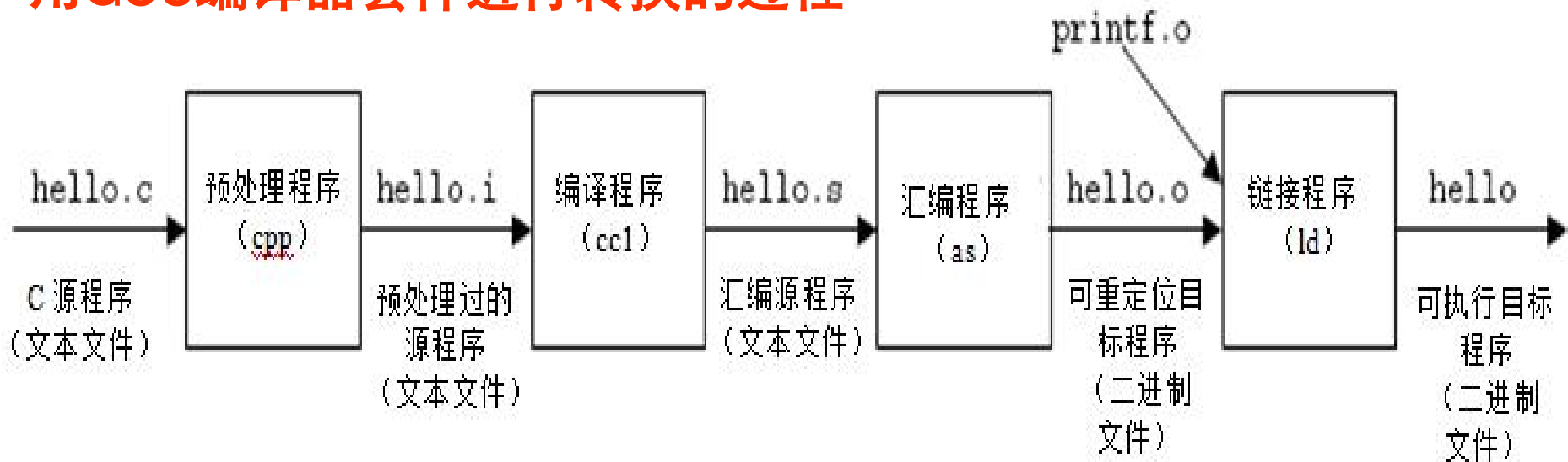
(6) 采用优化的编译系统，力求有效地支持高级语言程序

MIPS、RISC(RISC-I到RISC-V) 系列架构是典型的RISC处理器，82年以来新的指令集大多采用RISC体系结构

x86因为“兼容”的需要，保留了CISC的风格，同时也借鉴了RISC思想

高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



预处理: 在高级语言源程序中插入所有用`#include`命令指定的文件和用`#define`声明指定的宏。

编译: 将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编: 由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

链接: 由链接器将多个可重定位的机器语言目标文件以及库例程（如`printf()`库函数）链接起来，生成最终的可执行目标文件。

GCC使用举例

```
1 // test.c
2 int add(int i, int j )
3 {
4     int x = i +j;
5     return x;
6 }
```

两个源程序文件test1.c和test2.c, 最终生成可执行文件为test

`gcc -O1 test1.c test2.c -o test`

选项-O1表示一级优化, -O2为二级优化, 选项-o指出输出文件名

`gcc -E test.c -o test.i`
`gcc -S test.i -o test.s`
`gcc -S test.c -o test.s`

文件中以“.”开头的行都属于汇编指示符

```
1. .file "test.c"
2. .text
3. .Ltext0:
4. .align 2
5. .globl add
6. .type add, @function
7. add:
8. .LFB0 = .
9. addi.w $r3, $r3, -48
10. .LCFI0 = .
11. st.w $r22, $r3, 44
12. .LCFI1 = .
13. addi.w $r22, $r3, 48
14. .LCFI2 = .
15. st.w $r4, $r22, -36
16. st.w $r5, $r22, -40
```

```
17. ld.w $r13, $r22, -36
18. ld.w $r12, $r22, -40
19. add.w $r12, $r13, $r1
20. st.w $r12, $r22, -20
21. ld.w $r12, $r22, -20
22. or $r4, $r12, $r0
23. ld.w $r22, $r3, 44
24. .LCFI3 = .
25. addi.w $r3, $r3, 48
26. .LCFI4 = .
27. jr $r1
28. .LFE0:
29. .size add, .-add
30. .section .debug_frame,"",@progbits
```

GCC使用举例

test.s文件中以“.”开头的行都属于汇编指示符

常见的汇编指示符含义如下：

.file: 给出对应的源程序文件名。

.text: 指示代码节（.text）从此处开始。

.globl add: 声明add是一个全局符号。

.type add, @function: 声明add是一个函数。

.data: 指示已初始化数据节（.data）从此处开始。

.bss: 指示未初始化数据节（.bss）从此处开始。

.section .rodata: 指示只读数据节（.rodata）从此处开始。

.align 2: 指示代码从此处开始按2²=4字节对齐。

.balign 4: 指示数据从此处开始按4字节对齐。

.string "Hello, %s!\n": 定义以null结尾的字符串"Hello, %s!\n"。

GCC使用举例

LA32中编译得到的 test.s部分内容

“gcc -c test.s -o test.o” 将test.s汇编为test.o
“objdump -d test.o” test.o 反汇编结果如下:

add:

```
addi.w $r3, $r3, -48
st.w    $r22, $r3, 44
addi.w  $r22, $r3, 48
st.w    $r4, $r22, -36
st.w    $r5, $r22, -40
ld.w    $r13, $r22, -36
ld.w    $r12, $r22, -40
add.w   $r12, $r13, $r1
st.w    $r12, $r22, -20
ld.w    $r12, $r22, -20
or      $r4, $r12, $r0
ld.w    $r22, $r3, 44
addi.w  $r3, $r3, 48
```

jr \$r1

00000000 <add>:

0:	02bf4063
4:	2980b076
8:	0280c076
c:	29bf72c4
10:	29bf62c5
14:	28bf72cd
18:	28bf62cc
1c:	001031ac
20:	29bfb2cc
24:	28bfb2cc
28:	00150184
2c:	2880b076
30:	0280c063
34:	4c000020

位移量 机器指令

addi.w	\$r3, \$r3, -48(0xfd0)
st.w	\$r22, \$r3, 44(0x2c)
addi.w	\$r22, \$r3, 48(0x30)
st.w	\$r4, \$r22, -36(0xfdc)
st.w	\$r5, \$r22, -40(0xfd8)
ld.w	\$r13, \$r22, -36(0xfdc)
ld.w	\$r12, \$r22, -40(0xfd8)
add.w	\$r12, \$r13, \$r12
st.w	\$r12, \$r22, -20(0xfec)
ld.w	\$r12, \$r22, -20(0xfec)
move	\$r4, \$r12
ld.w	\$r22, \$r3, 44(0x2c)
addi.w	\$r3, \$r3, 48(0x30)
jirl	\$r0, \$r1, 0

汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异: 1) 后者立即数多了16进制表示; 2) 两者都可能出现伪指令

两种目标文件

test.o: 可重定位目标文件, test: 可执行目标文件

“objdump -d test.o” 结果

00000000 <add>:

0:	02bf4063	addi.w	\$r3, \$r3, -48(0xfd0)
4:	2980b076	st.w	\$r22, \$r3, 44(0x2c)
8:	0280c076	addi.w	\$r22, \$r3, 48(0x30)
c:	29bf72c4	st.w	\$r4, \$r22, -36(0xfdc)
10:	29bf62c5	st.w	\$r5, \$r22, -40(0xfd8)
14:	28bf72cd	ld.w	\$r13, \$r22, -36(0xfdc)
18:	28bf62cc	ld.w	\$r12, \$r22, -40(0xfd8)
1c:	001031ac	add.w	\$r12, \$r13, \$r12
20:	29bfb2cc	st.w	\$r12, \$r22, -20(0xfec)
24:	28bfb2cc	ld.w	\$r12, \$r22, -20(0xfec)
28:	00150184	move	\$r4, \$r12
2c:	2880b076	ld.w	\$r22, \$r3, 44(0x2c)
30:	0280c063	addi.w	\$r3, \$r3, 48(0x30)
34:	4c000020	jirl	\$r0, \$r1, 0

“objdump -d test” 结果

000106b0 <add>:

106b0:	02bf4063	addi.w	...
106b4:	2980b076	st.w	...
106b8:	0280c076	addi.w	...
106bc:	29bf72c4	st.w	...
106c0:	29bf62c5	st.w	...
106c4:	28bf72cd	ld.w	...
106c8:	28bf62cc	ld.w	...
106cc:	001031ac	add.w	...
106d0:	29bfb2cc	st.w	...
106d4:	28bfb2cc	ld.w	...
106d8:	00150184	move	...
106dc:	2880b076	ld.w	...
106e0:	0280c063	addi.w	...
106e4:	4c000020	jirl	...

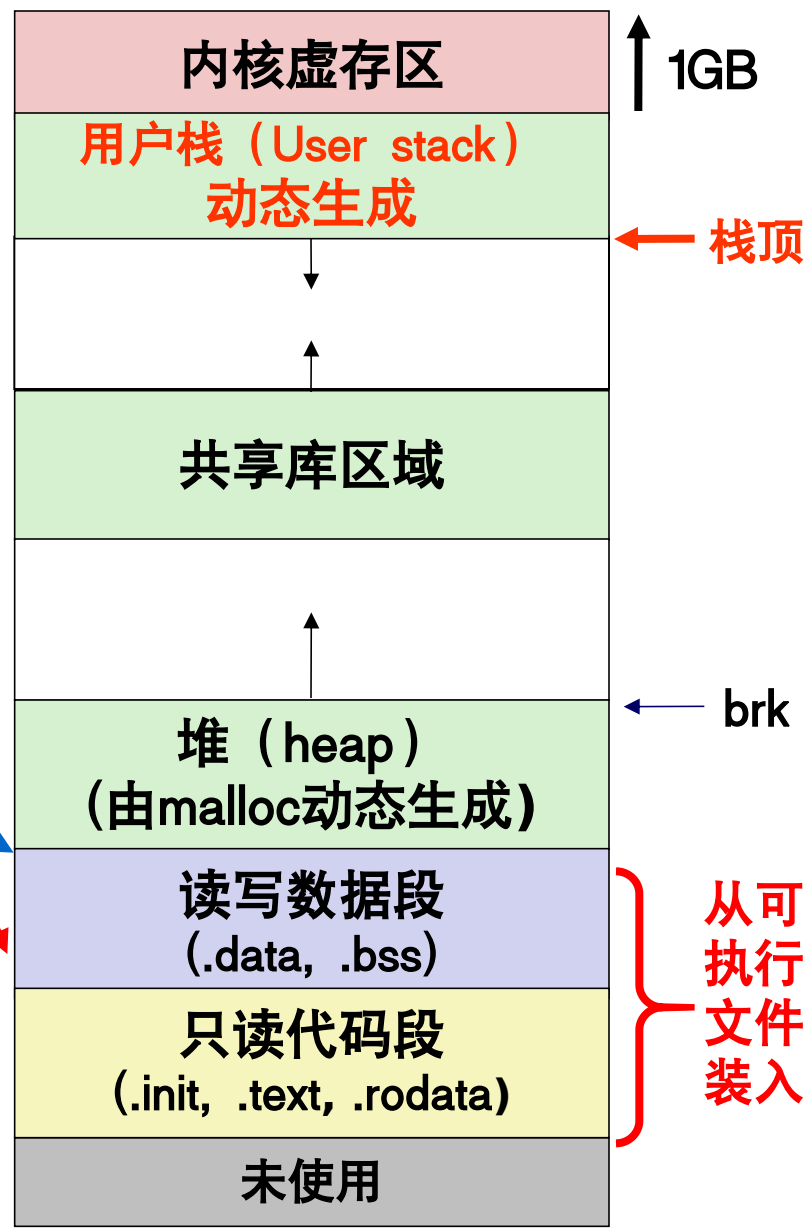
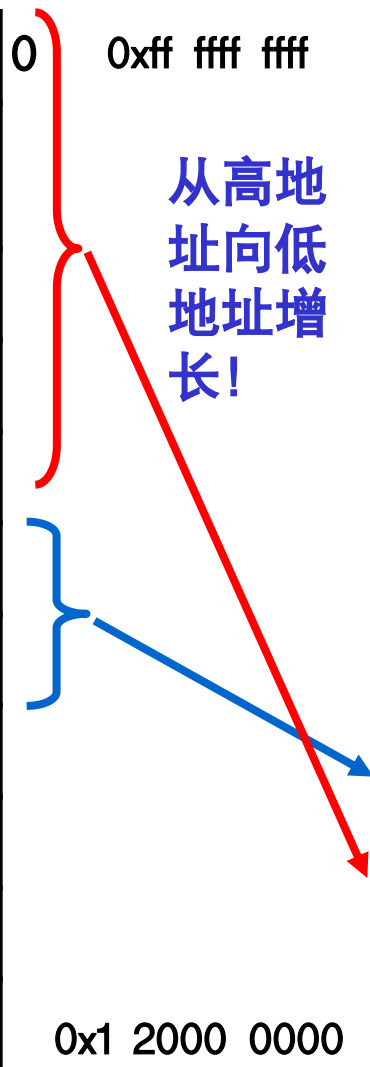
test.o中的代码从地址0开始, test中的代码从000106b0开始!

LA32中只读代码段起始地址为0x10000

可执行文件的存储器映像

程序(段)头表描述如何映射

ELF 头
程序 (段) 头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节



LA32中只读代码段起始地址为0x10000

程序的转换与指令系统

- 分以下五个部分介绍

- 第一讲：程序转换概述

- 机器指令和汇编指令
 - 机器级程序员感觉到的属性和功能特性
 - 高级语言程序转换为机器代码的过程

- 第二讲：LA32/LA64指令系统


- 机器指令格式和数据类型
 - 寄存器组织和寻址方式

- 第三讲：LA32/LA64基础整数指令

- 整数运算类指令和移位指令
 - 普通访存指令
 - 程序执行流控制指令

- 第四讲：LA32/LA64基础浮点指令

- 浮点普通访存指令
 - 浮点运算类指令
 - 浮点转换指令和传送指令



围绕LA32/LA64指令系统，解释具体的指令功能

一条指令中应该有几个地址码字段？

零地址指令

- (1) 无需操作数 如：空操作 / 停机等
- (2) 所需操作数为默认的 如：堆栈 / 累加器等

形式：

OP

一地址指令

其地址既是操作数的地址，也是结果的地址

- (1) 单目运算：如：取反 / 取负等
- (2) 双目运算：另一操作数为默认的 如：累加器等

形式：

OP	A1
----	----

二地址指令（最常用）

分别存放双目运算中两个操作数，并将其中一个地址作为结果的地址。

形式：

OP	A1	A2
----	----	----

三地址指令（RISC风格）

分别作为双目运算中两个源操作数的地址和一个结果的地址。

形式：

OP	A1	A2	A3
----	----	----	----

多地址指令

用于成批数据处理的指令，如：向量 / 矩阵等运算的SIMD指令。

从指令执行周期看指令设计涉及的问题



指令执行的每一步都可能发生异常或中断，因此，指令集系统架构（ISA）还需要考虑异常和中断机制

指令格式的设计

指令格式的选择应遵循的几条基本原则

- 应尽量短
- 要有足够的操作码位数
- 指令编码必须有唯一的解释，否则是不合法的指令
- 指令字长应是字节的整数倍
- 合理地选择地址字段的个数
- 指令尽量规整

一般通过对操作码进行不同的编码来定义不同的含义，操作码相同时，再由功能码定义不同的含义！

与指令集设计相关的重要方面

- 操作码的全部组成：操作码个数 / 种类 / 复杂度
LD/ST/INC/BRN 四种指令已足够编制任何可计算程序，但程序会很长
- 数据类型：对哪几种数据类型完成操作
- 指令格式：指令长度 / 地址码个数 / 各字段长度
- 通用寄存器：个数 / 功能 / 长度
- 寻址方式：操作数地址的指定方式
- 下条指令的地址如何确定：顺序，PC+1；条件转移；无条件转移；……
- 异常和中断机制，包括存储保护方式等

操作数类型和存储方式

操作数是指令处理的对象，与高级语言数据类型对应，基本类型有哪些？

地址（指针）

被看成无符号整数，用来参加运算以确定主(虚)存地址

数值数据

定点数(整数)：一般用二进制补码表示

浮点数(实数)：大多数机器采用IEEE754标准

十进制数：用NBCD码表示，压缩/非压缩（汇编程序设计时用）

位、位串、字符和字符串

用来表示文本、声音和图像等

- 4 bits is a nibble（一个十六进制数字）
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word

逻辑(布尔)数据

按位操作（0-假 / 1-真）

操作数存放在寄存器
或内存单元中，也可以立即数的方式直接出现在指令中

IA-32 & RISC-V Data Type

- IA-32
 - 基本类型:
 - 字节、字(16位)、双字(32位)、四字(64位)
 - 整数:
 - 16位、32位、64位三种2-补码表示的整数
 - 18位压缩8421 BCD码表示的十进制整数
 - 无符号整数（8、16或32位）
 - 近指针：32位段内偏移（有效地址）
 - 浮点数：IEEE 754（80位扩展精度浮点数寄存器）
- RISC-V
 - 基本类型:
 - 字节、半字(16位)、字(32位)、四字(64位)
 - 整数：16位、32位、64位三种2-补码表示的整数
 - 无符号整数：16位、32位、64位整数
 - 浮点数：IEEE 754（32位/64位浮点数寄存器）

LoongArch指令系统概述

- LoongArch架构分32位（LA32）和64位（LA64）两个版本
 - LA64架构应用级向下二进制兼容LA32架构。所谓“应用级向下二进制兼容”，一方面指采用LA32架构的应用软件的二进制可直接运行在兼容LA64架构的机器上并获得相同的运行结果，另一方面指仅限于应用软件兼容，架构规范并不保证在兼容LA32架构的机器上运行的系统软件（如操作系统内核）的二进制直接在兼容LA64架构的机器上运行时总是获得相同的运行结果
 - LoongArch有限借鉴了MIPS架构，进而推翻了大量MIPS设计，且不受其授权限制。龙芯将把LoongArch免费开放，并把部分处理器IP核源码开源，组建自主指令系统联盟。新流片的龙芯CPU均支持LoongArch，不再支持MIPS，但出于兼容性，可以通过LAT二进制翻译引擎100%翻译执行MIPS程序，藉此兼容MIPS生态，同时重点发扬LoongArch原生生态

LoongArch机器指令格式

RISC架构， 32位定长指令字， 以下列举几种典型指令格式

opcode为操作码字段，位数不等；ra、rj、rk、rd为寄存器编号，IN/IN[n:m]是立即数域，其中N表示位数，n:m表示N位中从n到m的连续位号。

[illegible]

LoongArch ABI规定的LA64中数据类型

C 语言数据类型规格

Table 3. LP64 数据模型（对应基础 ABI 类型：lp64d lp64f lp64s）

标量类型	大小（字节）	对齐（字节）
bool / _Bool	1	1
unsigned char / char	1	1
unsigned short / short	2	2
unsigned int / int	4	4
unsigned long / long	8	8
unsigned long long / long long	8	8
指针类型	8	8
float	4	4
double	8	8
long double	16	16

long double型浮点运算由编译器用软件模拟实现

LoongArch ABI规定的LA32中数据类型

Table 4. ILP32 数据模型（对应基础 ABI 类型：`ilp32d ilp32f ilp32s`）

标量类型	大小（字节）	对齐（字节）
<code>bool / _Bool</code>	1	1
<code>unsigned char / char</code>	1	1
<code>unsigned short / short</code>	2	2
<code>unsigned int / int</code>	4	4
<code>unsigned long / long</code>	4	4
<code>unsigned long long / long long</code>	8	8
指针类型	4	4
<code>float</code>	4	4
<code>double</code>	8	8
<code>long double</code>	16	16

long double型浮点运算由编译器用软件模拟实现

对于任何基础 ABI 类型，`char` 默认是有符号类型。

LoongArch 汇编指令助记符

- u 所有128位向量指令名以V开头；所有256位向量指令名以XV开头。所有非向量浮点数指令名以F开头；所有128位向量浮点指令名以VF开头；所有256位向量浮点指令名以XVF开头。
- u 如“ADD.D rd, rj, rk”中，指令名ADD.D表示一条整数加法指令。
“FADD.S fd, fj, fk”中，指令名以F开头，故是一条浮点加法指令。
- u 指令名后缀为.B、.H、.W、.D、.BU、.HU、.WU、.DU分别表示该指令操作数是带符号字节、带符号半字、带符号字、带符号双字、无符号字节、无符号半字、无符号字、无符号双字。
- u 当操作数是带符号还是无符号整数不影响运算结果时，指令名中的后缀均不带U，此时并不限制操作对象只能是带符号整数。
- u 操作对象是浮点数类型时，指令名后缀.H、.S、.D、.W、.L、.WU、.LU分别表示该指令操作数是半精度浮点数、单精度浮点数、双精度浮点数、带符号字、带符号双字、无符号字、无符号双字
- u 向量指令中，后缀.V表示将整个向量数据作为一个整体操作。
- u 并不是所有指令都用“.XX”形式后缀指示指令操作对象。当指令操作数位宽由系统字长决定时，不加指令名后缀，如SLT和SLTU指令。

LoongArch 汇编指令助记符

- u 若源操作数位宽与目的操作数位宽不同，则有两个后缀，前为目的操作数后缀，后为源操作数后缀。
- u 如 “**MULW.D.WU** rd, rj, rk” 中，.D对应目的操作数rd，.WU对应源操作数rj和rk，表明两个无符号字相乘，得到的双字结果写入rd中。
- u 若操作数情况更复杂，则从左往右依次列出目的操作数和每个源操作数的情况，其次序与其后操作数顺序一致。
- u 如 “**CRC.W.B.W** rd, rj, rk” 中，.W对应rd，.B对应rj，第二个.W对应rk，表明该CRC校验操作是将rj中的字节消息与rk 中32位原校验码，经某种处理生成新的32位校验码写入到rd中。
- u 大部分指令需区分操作数类型。如：
 - ü “FADD.S fd, fj, fk” 操作数都为float型
 - ü “FADD.D fd, fj, fk” 操作数都为double型
 - ü “MULW.D.W rd, rj, rk” 源操作数为int型，目的操作数为long long型
 - ü “MULW.D.WU rd, rj, rk” 源操作数为unsigned int型，目的操作数为long long类型

LoongArch定点寄存器组织

- u 不考虑I/O指令，指令中操作数有立即数、寄存器操作数和存储器操作数
 - ü 立即数在指令中，无需指定其存放位置
 - ü 寄存器操作数需要指定操作数所在寄存器的编号
 - ü 存储器操作数需要指定操作数所在存储单元的地址
- u 基础整数指令涉及的通用寄存器
 - ü 通用寄存器GR有32个，汇编指令中记为\$r0~\$r31，其中，0号寄存器r0内容恒为0。寄存器编号为00000B~11111B
 - ü 汇编指令中GR位宽记为GRLEN。LA32架构下GRLEN=32，LA64架构下GRLEN=64
- u 程序计数器（PC）
 - ü PC没有编号，不能在指令中直接指定或修改，PC位宽等于GRLEN
 - ü 顺序执行时，每执行一条指令，PC加4；执行到跳转和过程调用等非顺序执行指令时，PC的内容修改为跳转目标地址

LoongArch浮点寄存器组织

- u 浮点数指令涉及的寄存器主要有**浮点寄存器 (FR)**、**条件标志寄存器 (CFR)**、**浮点控制状态寄存器 (FCSR)**
- u 浮点寄存器 (FR)
 - ü 浮点寄存器FR有32个，汇编指令中记为\$f0~\$f31，其编号为00000B~11111B
 - ü LA32和LA64中，FR位宽皆为64，单精度浮点数运算或整数字时，操作数位宽为32，使用FR中低32位，此时，FR[63:32]可以是任意值
- u 条件标志寄存器 (CFR)
 - ü CFR有8个，汇编指令中记为fcc0~fcc7
 - ü CFR位宽为1，若浮点比较指令的比较结果为真，则置1；否则置0
 - ü 浮点分支指令根据CFR的取值进行跳转
- u 浮点控制状态寄存器 (FCSR)
 - ü 浮点指令支持IEEE 754-2008 所定义的五种浮点异常：
 - ü 不精确Inexact (**I**)、下溢Underflow (**U**)、上溢Overflow (**O**)、除零Division by Zero (**Z**)、非法操作Invalid Operation (**V**)

LoongArch浮点控制状态寄存器FCSR

FCSR有4个，汇编指令中记为fcsr0~fcsr3，fcsr1是fcsr0中Enables的别名，fcsr2是fcsr0中Cause和Flags字段的别名，fcsr3是fcsr0中RM字段的别名。fcsr1~fcsr3中各字段的位置与fcsr0中的一致

位 ↴	字段 ↴	读写 ↴	描述 ↴
4:0 ↴	Enables	RW ↴	5 种浮点异常对应的陷入使能位。位 4:0 分别对应 V、Z、O、U、I。 ↴
7:5 ↴	0 ↴	R0 ↴	保留字段。只读，返回 0，不允许软件改变其值。 ↴
9:8 ↴	RM ↴	RW ↴	舍入模式控制字段。含义如下： ↴ 0: RNE，对应 IEEE 754-2008 中的就近舍入到偶数； ↴ 1: RZ，对应 IEEE 754-2008 中的向零方向舍入； ↴ 2: RP，对应 IEEE 754-2008 中的向 $+\infty$ 方向舍入； ↴ 3: RM，对应 IEEE 754-2008 中的向 $-\infty$ 方向舍入。 ↴
15:10 ↴	0 ↴	R0 ↴	保留字段。只读，返回 0，不允许软件改变其值。 ↴
20:16 ↴	Flags ↴	RW ↴	自该字段 <u>对应位被软件清空</u> 后所产生但未陷入的浮点异常情况。 ↴ 位 20:16 分别对应 V、Z、O、U、I。 ↴
23:21 ↴	0 ↴	R0 ↴	保留字段。只读，返回 0，不允许软件改变其值。 ↴
28:24 ↴	Cause ↴	RW ↴	最近一次浮点操作所产生的浮点异常。位 28:24 分别对应 V、Z、O、U、I。
31:29 ↴	0 ↴	R0 ↴	保留字段。只读，返回 0，不允许软件改变其值。 ↴

Addressing Modes (寻址方式)

- 什么是“寻址方式”？

指令或操作数地址的指定方式。即：根据地址找到指令或操作数的方法。

- 地址码编码由操作数的寻址方式决定

- 地址码编码原则：

指令地址码尽量短 $\xrightarrow{\text{为什么?}}$ 目标代码短，省空间
操作数存放位置灵活，空间应尽量大 \rightarrow 利于编译器优化产生高效代码
地址计算过程尽量简单 \longrightarrow 指令执行快

- u 指令的寻址-----简单

正常：PC增值

跳转（jump / branch / call / return）：同操作数的寻址

- u 操作数的寻址-----复杂（想象一下高级语言程序中操作数情况多复杂）

操作数来源：寄存器 / 外设端口 / 主(虚)存 / 栈顶

操作数结构：位 / 字节 / 半字 / 字 / 双字 / 一维表 / 二维表 / ...

通常寻址方式指“操作数的寻址方式”

Addressing Modes (寻址方式)

- 寻址方式的确定

- (1) 没有专门的寻址方式位 (由操作码确定寻址方式)

- 如: MIPS指令, 一条指令中最多仅有一个主(虚)存地址, 且仅有一到两种寻址方式, Load/store型机器指令属于这种情况。

- (2) 有专门的寻址方式位

- 如: X86指令, 一条指令中有多个操作数, 且寻址方式各不相同, 需要各自说明寻址方式, 因此每个操作数有专门的寻址方式位。

- 有效地址的含义

- 操作数所在存储单元的地址 (可能是逻辑地址或物理地址), 可通过指令的寻址方式和地址码计算得到

- 基本寻址方式

- 立即 / 直接 / 间接 / 寄存器(直接) / 寄存器间接 / 偏移 / 栈

- 基本寻址方式的算法及优缺点

基本寻址方式的算法和优缺点

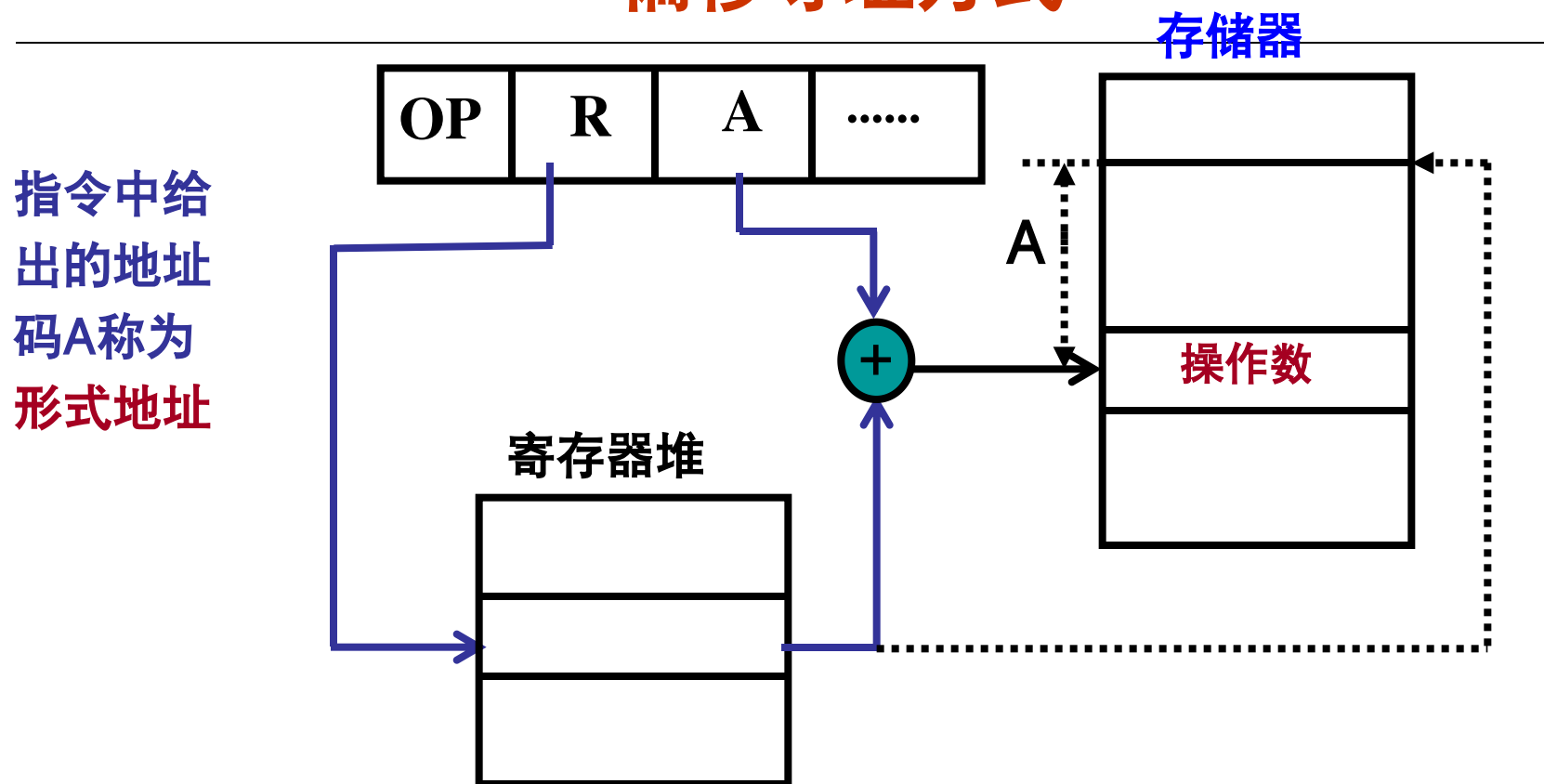
假设：A=地址字段值，R=寄存器编号，
EA=有效地址，(X)=X中的内容



方式	算法	主要优点	
主要缺点			
立即数 有限	操作数=A	指令执行速度快	操作数幅值
直接 有限	EA=A	有效地址计算简单	地址范围
间接 多次存储器访问	EA=(A)	有效地址范围大	
寄存器(直接) 范围有限	操作数=(R)	指令执行快，指令短	地址范
寄存器间接 存储器访问	EA=(R)	地址范围大	额外
问题：以上各种寻址方式下，操作数在寄存器中还是在存储器中？			
偏移 需要计算有效地址	EA=A+(R)	灵活	复
栈	EA=栈顶	指令短	应

问题：以上各种寻址方式下，操作数在寄存器中还是在存储器中？

偏移寻址方式



偏移寻址: $EA = A + (R)$ R可以明显给出, 也可以隐含给出

R可以为PC、基址寄存器B、

变址寄存器 I

- 相对寻址: $EA = A + (PC)$ 相对于当前指令处偏移量为A的单元
- 基址寻址: $EA = A + (B)$ 相对于基址(B)处偏移量为A的单元
- 变址寻址: $EA = A + (I)$ 相对于首址A处偏移量为(I)的单元

变址寻址实现

- 自动变址

指令中的地址码A给定数组首址，变址器I每次自动加/减数组元素的长度x。

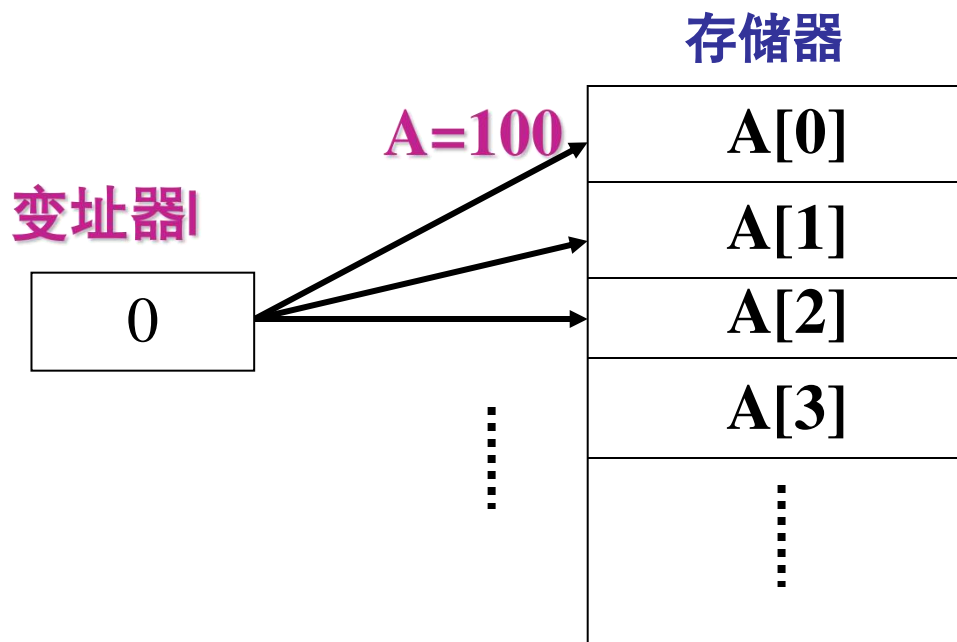
$$EA = A + (I)$$

$$I = (I) \pm x$$

例如，X86中的串操作指令

- 对于“for (i=0; i<N; i++) ...”，即地址从低→高变化：加
- 对于“for (i=N-1; i>=0; i--) ...”，即地址从高→低变化：减
- 可提供对线性表的方便访问

假定一维数组A从100号单元开始



假设按字节编址，则：

每个元素为一个字节时， $I = (I) \pm 1$

每个元素为4个字节时， $I = (I) \pm 4$

一般RISC机器不提供自动变址寻址，并将变址和基址寻址统一成一种偏移寻址方式

LoongArch的寻址方式

- LoongArch为RISC，寻址方式简单，仅有4种基本方式：

- 立即寻址、寄存器寻址、基址寻址、相对寻址

- 寻址方式举例

- `addi.w $r12, $r13, -28(0xfe4)`

采用2RI12-型指令格式，功能为： $R[r12] \leftarrow R[r13] + (-28)$

其中，源操作数 $R[r13]$ 和目的操作数 $R[r12]$ 为**寄存器寻址**，寄存器编号01101和01100分别由rj和rd字段给出

另一源操作数(-28)为**立即寻址**，立即数0xfe4由l12字段给出

- `st.w $r12, $r22, -28(0xfe4)`

采用2RI12-型指令格式，功能为： $M[R[r22]-28, \text{WORD}] \leftarrow R[r12]$

目的操作数为**基址寻址**，立即数字段指出偏移量，源操作数为**寄存器寻址**

- `bge $r13, $r12, 24(0x18)`

采用2RI16-型指令格式，功能为：若大于等于，则 $PC = PC + \text{偏移量}$ （**相对寻址**），第25:10位为立即数字段 $\text{offs16} = 0000\ 0000\ 0000\ 0110B$ ，偏移量= $\text{SignExtend}(\{\text{offs16}, 2'b0\}, \text{GRLEN}) = 0x18$

Instruction Format(指令格式)

u 操作码的编码有两种方式

- Fixed Length Opcodes (定长操作码法)
- Expanding Opcodes (扩展操作码编法)

u instructions size

- 代码长度更重要时：采用变长指令字、变长操作码
- 性能更重要时：采用定长指令字、定长操作码

为什么？

变长指令字和变长操作码使机器代码更紧凑；定长指令字和定长操作码便于快速访问和译码。学了CPU设计就更明白了。

问题：是否可以有定长指令字、变长操作码？定长操作码、变长指令字呢？

指令长度是否可变与操作码长度是否可变没有绝对关系，但通常是“定长操作码不一定是定长指令字”、“变长操作码一般是变长指令字”。

定长操作码编码 Fixed Length Opcodes

基本思想

指令的操作码部分采用固定长度的编码

如：假设操作码固定为6位，则系统最多可表示64种指令

特点

译码方便，但有信息冗余

举例

IBM360/370采用：

8位定长操作码，最多可有256条指令

只提供了183条指令，有73种编码为冗余信息

机器字长32位，按字节编址

有16个32位通用寄存器，基址器B和变址器X可用其中任意一个

问题：通用寄存器编号有几位？ B和X的编号占几位？ 都是4位！

扩展（变长）操作码编码

基本思想

将操作码的编码长度分成几种固定长的格式。被大多数指令集采用。

LoongArch是典型的扩展操作码机器。

种类

等长扩展法：4-8-12； 3-6-9； …… / 不等长扩展法

举例说明如何扩展

设某指令系统指令字是16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？

解：操作码按短到长进行扩展编码

二地址指令：(0000 ~ 1110)

一地址指令：11110 (00000 ~ 11111); 11111 (00000 ~ 00001)

零地址指令：11111 (00010 ~ 11111) (000000 ~ 111111)

故零地址指令最多有 $30 \times 2^6 = 15 \times 2^7$ 种

扩展（变长）操作码编码

LoongArch是典型的扩展操作码机器

	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0		
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
2R- 型	opcode																				rj				rd							
3R- 型	opcode												rk				rj				rd											
4R- 型	opcode								ra				rk				rj				rd											
3RI2- 型	opcode										I2		rk				rj				rd											
1RI20- 型	opcode				I20																								rd			
2RI8- 型	opcode								I8								rj				rd											
2RI12- 型	opcode						I12										rj				rd											
2RI14- 型	opcode					I14													rj				rd									
2RI16- 型	opcode				I16																rj				rd							
1RI21- 型	opcode				I21[15:0]																rj				I21[20:16]							
I26- 型	opcode				I26[15:0]																I26[25:16]											

程序的转换与指令系统

- 分以下五个部分介绍

- 第一讲：程序转换概述

- 机器指令和汇编指令
 - 机器级程序员感觉到的属性和功能特性
 - 高级语言程序转换为机器代码的过程

- 第二讲：LA32/LA64指令系统


- 机器指令格式和数据类型
 - 寄存器组织和寻址方式

- 第三讲：LA32/LA64基础整数指令

- 整数运算类指令和移位指令
 - 普通访存指令
 - 程序执行流控制指令

- 第四讲：LA32/LA64基础浮点指令

- 浮点普通访存指令
 - 浮点运算类指令
 - 浮点转换指令和传送指令



围绕LA32/LA64指令系统，解释具体的指令功能

LA32/LA64基础整数指令概述

64位系统既要能处理64位数据，还要能处理4字节的int型数据和2字节的short型数据等，因此其中包含了32位架构LA32的所有指令，即LA32是LA64的一个子集，该子集所含指令列表如下：

算术运算指令	ADD.W, SUB.W, ADDI.W, ALSL.W, ALSL.WU, LU12I.W, SLT, SLTU, SLTI, SLTUI; PCADDI, PCADDU12I, PCALAU12I; AND, OR, NOR, XOR, ANDN, ORN, ANDI, ORI, XORI; MUL.W, MULH.W, MULH.WU, DIV.W, MOD.W, DIV.WU, MOD.WU
移位运算指令	SLL.W, SRL.W, SRA.W, ROTR.W, SLLI.W, SRLI.W, SRAI.W, ROTRI.W
位操作指令	EXT.W.B, EXT.W.H, CLO.W, CLZ.W, CTO.W, CTZ.W, BYTEPICK.W; REVB.2H, BITREV.4B, BITREV.W, BSTRINS.W, BSTRPICK.W, MASKEQZ, MASKNEZ
跳转指令	BEQ, BNE, BLT, BGE, BLTU, BGEU, BEQZ, BNEZ, B, BL, JIRL
普通访存指令	LD.B, LD.H, LD.W, LD.BU, LD.HU, ST.B, ST.H, ST.W, PRELD
原子访存指令	LL.W, SC.W
栅障指令	DBAR, IBAR
其他杂项指令	SYSCALL, BREAK, RDTIMEL.W, RDTIMEH.W, CPUCFG

LA32/LA64整数运算类指令

- LoongArch中整数运算类指令包括**算术运算、逻辑运算、比较运算等**，执行这些指令时**不对溢出进行处理**，也**不会生成标志位**
- 加减运算指令

指令 ↵		功能 ↵
add.w ↵	rd, rj, rk	$tmp = R[rj][31:0] + R[rk][31:0]$ ↵ $R[rd] \leftarrow \text{SignExtend}(tmp[31:0], GRLEN)$
sub.w ↵	rd, rj, rk	$tmp = R[rj][31:0] - R[rk][31:0]$ ↵ $R[rd] \leftarrow \text{SignExtend}(tmp[31:0], GRLEN)$
add.d ↵	rd, rj, rk	$R[rd] \leftarrow R[rj][63:0] + R[rk][63:0]$ ↵
sub.d ↵	rd, rj, rk	$R[rd] \leftarrow R[rj][63:0] - R[rk][63:0]$ ↵

指令 ↵		功能 ↵
addi.w ↵	rd, rj, si12	$tmp = R[rj][31:0] + \text{SignExtend}(si12, 32)$ ↵ $R[rd] \leftarrow \text{SignExtend}(tmp[31:0], GRLEN)$ ↵
addi.d ↵	rd, rj, si12	$R[rd] \leftarrow R[rj][63:0] + \text{SignExtend}(si12, 64)$ ↵
addu16i.d ↵	rd, rj, si16	$R[rd] \leftarrow R[rj][63:0] + \text{SignExtend}(\{si16, 16'b0\}, 64)$

定点加减运算指令举例

- 已知R[r15][31:0]为0x8080 0350，在LA32和LA64中执行指令“addi.w \$r13,\$r15,-16(0xff0)”后，r13中的内容分别是什么？

解：0xff0符号扩展得0xffff fff0， $R[r15][31:0] + \text{SignExtend}(0xff0, 32) = 0x8080\ 0350 + 0xffff\ fff0 = 0x8080\ 0340$ 。

LA32中，该指令执行后，r13中的内容为0x8080 0340

LA64中，通用寄存器r13有64位，求和的结果符号扩展32位后写入r13，其内容为0xffff ffff 8080 0340。

- LA32中，若R[r12]=0xffff fff0，R[r13]=0xffff fffa，则指令“sub.w \$r14,\$r12,\$r13”执行后，r12、r13、r14中的内容各是什么？

解：“sub.w \$r14,\$r12,\$r13”功能是 $R[r14] \leftarrow R[r12] - R[r13]$ 。sub指令执行后，差存放在r14中，LA32中sub.w指令不生成标志信息。

在补码加减运算器中做减法，Sub=1，加法器的Y'端为反相器输出（各位取反）， $FFFF\ FFF0H - FFFF\ FFFAH = FFFF\ FFF0H + 0000\ 0005H + 1 = FFFF\ FFF6H$ ，即R[r14]= 0xffff fff6。r12和r13中内容不变

定点加减运算指令举例

- 对于高21位（高53位）为全0或全1的常数，可利用ADDI.W指令进行符号扩展的特性完成立即数的装载

指令 ↵		功能 ↵
addi.w ↵	rd, rj, si12	$tmp = R[rj][31:0] + \text{SignExtend}(si12, 32)$ ↵ $R[rd] \leftarrow \text{SignExtend}(tmp[31:0], GRLEN)$ ↵
addi.d ↵	rd, rj, si12	$R[rd] \leftarrow R[rj][63:0] + \text{SignExtend}(si12, 64)$ ↵
addu16i.d ↵	rd, rj, si16	$R[rd] \leftarrow R[rj][63:0] + \text{SignExtend}(\{si16, 16'b0\}, 64)$

- 假定变量x分配在寄存器r12中，分别给出C语言语句“long x=-500;”对应的LA32和LA64机器级代码。

解：-500的64位机器数为0xffff ffff ffff ffff ffff ffff fe0c，x在LA64中占64位（高53位为全1），在LA32中占32位（高21位为全1），故在LA32和LA64中都可用指令ADDI.W将低12位1110 0000 1100写入r12。C语言语句“int x=-500;”在LA32和LA64下所用指令相同。

```
02b8300c  addi.w $r12,$r0,-500(0xe0c)
```

LA32/LA64整数运算类指令

按位逻辑运算指令

指令		功能 ↵	指令		功能 ↵
and	rd, rj, rk	$R[rd] \leftarrow R[rj] \& R[rk]$	andn ↵	rd, rj, rk	$R[rd] \leftarrow R[rj] \& (\sim R[rk])$
or ↵	rd, rj, rk	$R[rd] \leftarrow R[rj] R[rk]$ ↵	nor ↵	rd, rj, rk	$R[rd] \leftarrow \sim(R[rj] R[rk])$ ↵
xor	rd, rj, rk	$R[rd] \leftarrow R[rj] \wedge R[rk]$ ↵	orn ↵	rd, rj, rk	$R[rd] \leftarrow R[rj] (\sim R[rk])$ ↵

指令AND、OR、XOR和NOR分别将rj与rk内容按位逻辑与、或、异或、或非，结果写入rd。指令ANDN将rk内容按位取反后再与rj内容按位逻辑与，结果写入rd。指令ORN将rk内容按位取反后再与rj内容按位逻辑或，结果写入rd

指令		功能 ↵
andi	rd, rj, ui12	$R[rd] \leftarrow R[rj] \& \text{ZeroExtend}(ui12, GRLEN)$
ori ↵	rd, rj, ui12	$R[rd] \leftarrow R[rj] \text{ZeroExtend}(ui12, GRLEN)$ ↵
xori	rd, rj, ui12	$R[rd] \leftarrow R[rj] \wedge \text{ZeroExtend}(ui12, GRLEN)$ ↵

指令ANDI、ORI和XORI分别将rj内容与立即数si12零扩展后的数据按位逻辑与、或、异或，结果写入通用寄存器rd

LA32/LA64整数运算类指令

- 立即数按位逻辑运算指令

指令		功能 ↗
andi	rd, rj, ui12	$R[rd] \leftarrow R[rj] \& \text{ZeroExtend}(ui12, GRLEN)$
ori ↗	rd, rj, ui12	$R[rd] \leftarrow R[rj] \text{ZeroExtend}(ui12, GRLEN) \swarrow$
xori	rd, rj, ui12	$R[rd] \leftarrow R[rj] \wedge \text{ZeroExtend}(ui12, GRLEN) \swarrow$

- 假定变量x分配在寄存器r12中，给出C语句“int x=2048;”对应的LA32机器级代码。

2048的32位机器数为0000 0000 0000 0000 0000 1000 0000 0000B，其中高20位全为0，低12位中高位为1，通过与r0中的0按位或，可将低12位1000 0000 0000写入r12。C语句“int x=2048;”对应的LA32机器指令和汇编指令如下（指令码中加粗位为立即数字段）：

0000001110 **1000 0000 0000** 00000 01100 ori \$r12,\$r0,0x800

LA32/LA64整数运算类指令

- 立即数装载指令

指令 ↵		功能
lu12i.w	rd, si20 ↵	$R[rd] \leftarrow \text{SignExtend}(\{si20, 12'b0\}, GRLEN)$ ↵
lu32i.d	rd, si20 ↵	$R[rd] \leftarrow \{\text{SignExtend}(si20, 32), GR[rd][31:0]\}$
lu52i.d	rd, rj, si12	$R[rd] \leftarrow \{si12, GR[rj][51:0]\}$ ↵

LU12I.W、LU32I.D、LU52I.D中的si20、si12分别表示20位和12位立即数，这些指令与ORI指令配合使用，用于将超过12位的立即数装载到通用寄存器中

- 假定变量x分配在寄存器r12中，给出C语句“int x=-8191;”对应的LA32机器级代码。

-8191的32位机器数为1111 1111 1111 1111 1110 0000 0000 0001B，高20位中有0也有1，因此，需将-8191分解为两个立即数，先将其高20位装入r12高20位（低12位清0），再将r12内容与低12位零扩展结果0000 0000 0001按位与。C语句“int x=-8191;”对应的LA32机器指令和汇编指令如下

```
0001010 1111 1111 1111 1111 1110 01100 lu12i.w $r12,-2(0xfffffe)
0000001110 0000 0000 0001 01100 01100 ori    $r12,$r12,0x1
```

LA32/LA64整数运算类指令

- 小于比较指令

指令 ↵		功能 ↵
slt ↵	rd, rj, rk ↵	$R[rd] \leftarrow (\text{signed}(R[rj]) < \text{signed}(R[rk])) ? 1 : 0$ ↵
sltu ↵	rd, rj, rk ↵	$R[rd] \leftarrow (\text{unsigned}(R[rj]) < \text{unsigned}(R[rk])) ? 1 : 0$ ↵
slti ↵	rd, rj, si12	tmp = SignExtend(si12, GRLEN) ↵ $R[rd] \leftarrow (\text{signed}(R[rj]) < \text{signed}(tmp)) ? 1 : 0$ ↵
sltui	rd, rj, si12	tmp = SignExtend(si12, GRLEN) ↵ $R[rd] \leftarrow (\text{unsigned}(R[rj]) < \text{unsigned}(tmp)) ? 1 : 0$ ↵

- 若x、y和z都是long long型，x的高、低32位分别存放在r15、r14中；y的高、低32位分别存放在r17、r16中；z的高、低32位分别存放在r13、r12中。写出C语句“z=x+y;”对应的LA32机器级代码

可通过SLTU指令将低32位的进位加入高32位中

```
000000000000100000 10000 01110 01100  add. w  $r12,$r14,$r16
000000000000100101 01110 01100 10010  sltu    $r18,$r12,$r14
000000000000100000 10001 01111 01101  add. w  $r13,$r15,$r17
000000000000100000 01101 10010 01101  add. w  $r13,$r18,$r13
```


LA32/LA64整数运算类指令

- PC增量指令

指令		功能
pcaddi ↵	rd, si20	$R[rd] \leftarrow PC + \text{SignExtend}(\{si20, 2'b0\}, GRLEN)$ ↵
pcaddu12i	rd, si20	$R[rd] \leftarrow PC + \text{SignExtend}(\{si20, 12'b0\}, GRLEN)$
pcaddu18i	rd, si20	$R[rd] \leftarrow PC + \text{SignExtend}(\{si20, 18'b0\}, GRLEN)$
pcalau12i ↵	rd, si20	$tmp = PC + \text{SignExtend}(\{si20, 12'b0\}, GRLEN)$ ↵ $R[rd] \leftarrow \{tmp[GRLEN-1:12], 12'b0\}$ ↵

- 可方便计算相对于当前指令的跳转目的地址或某个常数所在的地址。假设在si20低位添加n个0，则增量值为 $si20 * 2^n$ ， 2^n 为比例因子。PCALAU12i会将跳转目的地址低12位置0，即地址按4K字节对齐。
- 给出LA32中C语句“float f=5.0;”对应的部分机器级代码

```
1068c: 1c000cac    pcaddu12i $r12, 101(0x65)
10690: 02a1318c    addi.w    $r12, $r12, -1972(0x84c)
```

PCADDU12i执行后，r12中为0x0001 068c+0x0006 5000=0x0007 568c; ADDI.W执行后，r12中为0x0007 568c+0xffff f84c=0x0007 4ed8，即5.0存放在地址为0x0007 4ed8。5.0表示为0x40a0 0000，LoongArch是小端方式，故在0x0007 4ed8中存放的是0x00，0x0007 4eda中存放的是0xa0。

LA32/LA64整数运算类指令

- 乘运算指令
MUL.W和MUL.D将低n位乘积写入目的寄存器，包括无符号和带符号乘
MULH.W和MULH.WU分别用于带符号和无符号32位乘运算的高32位乘积
MULH.D和MULH.DU，分别用于带符号和无符号64位乘运算的高64位乘积
MULW.D.W和MULW.D.WU分别用于带符号和无符号32位乘运算的全部64位乘

指令		功能 ↵
mul.w ↵	rd, rj, rk ↵	product = signed(R[rj][31:0]) * signed(R[rk][31:0]) ↵ R[rd] ← SignExtend(product[31:0], GRLEN) ↵
mulh.w ↵	rd, rj, rk ↵	product = signed(R[rj][31:0]) * signed(R[rk][31:0]) ↵ R[rd] ← SignExtend(product[63:32], GRLEN) ↵
mulh.wu ↵	rd, rj, rk ↵	product = unsigned(R[rj][31:0]) * unsigned(R[rk][31:0]) ↵ R[rd] ← SignExtend(product[63:32], GRLEN) ↵
mul.d ↵	rd, rj, rk ↵	product = signed(R[rj][63:0]) * signed(R[rk][63:0]) ↵ R[rd] ← product[63:0] ↵
mulh.d ↵	rd, rj, rk ↵	product = signed(R[rj][63:0]) * signed(R[rk][63:0]) ↵ R[rd] ← product[127:64] ↵
mulh.du ↵	rd, rj, rk ↵	product = unsigned(R[rj][63:0]) * unsigned(R[rk][63:0]) ↵ R[rd] ← product[127:64] ↵
mulw.d.w ↵	rd, rj, rk ↵	product = signed(R[rj][31:0]) * signed(R[rk][31:0]) ↵ R[rd] ← product[63:0] ↵
mulw.d.wu ↵	rd, rj, rk ↵	product = unsigned(R[rj][31:0]) * unsigned(R[rk][31:0]) ↵ R[rd] ← product[63:0] ↵

LA32/LA64整数运算类指令

除运算指令

DIV.W、MOD.W、
DIV.D、MOD.D进行
除运算时，操作数均
视作带符号整数

DIV.WU、DIV.DU
MOD.WU、MOD.DU
进行除运算时，操作
数均视作无符号数

LoongArch规定：若
商溢出或除数为0，
CPU不触发任何异常
，除数为0时结果可
为任意值，编译器检
查除数是否为0，若
是，则通过执行陷阱
指令给出出错信息并
终止程序执行

指令		功能
div.w ↵	rd, rj, rk	$\text{quotient} = \text{signed}(R[rj][31:0]) / \text{signed}(R[rk][31:0]) \quad \hookleftarrow$ $R[rd] \leftarrow \text{SignExtend}(\text{quotient}[31:0], \text{GRLEN}) \quad \hookleftarrow$
mod.w ↵	rd, rj, rk	$\text{remainder} = \text{signed}(R[rj][31:0]) \% \text{signed}(R[rk][31:0]) \quad \hookleftarrow$ $R[rd] \leftarrow \text{SignExtend}(\text{remainder}[31:0], \text{GRLEN}) \quad \hookleftarrow$
div.wu ↵	rd, rj, rk	$\text{quotient} = \text{unsigned}(R[rj][31:0]) / \text{unsigned}(R[rk][31:0]) \quad \hookleftarrow$ $R[rd] \leftarrow \text{SignExtend}(\text{quotient}[31:0], \text{GRLEN}) \quad \hookleftarrow$
mod.wu ↵	rd, rj, rk	$\text{remainder} = \text{unsigned}(R[rj][31:0]) \% \text{unsigned}(R[rk][31:0])$ $R[rd] \leftarrow \text{SignExtend}(\text{remainder}[31:0], \text{GRLEN}) \quad \hookleftarrow$
div.d ↵	rd, rj, rk	$R[rd] \leftarrow \text{signed}(R[rj][63:0]) / \text{signed}(R[rk][63:0]) \quad \hookleftarrow$
mod.d ↵	rd, rj, rk	$R[rd] \leftarrow \text{signed}(R[rj][63:0]) \% \text{signed}(R[rk][63:0]) \quad \hookleftarrow$
div.du ↵	rd, rj, rk	$R[rd] \leftarrow \text{unsigned}(R[rj][63:0]) / \text{unsigned}(R[rk][63:0]) \quad \hookleftarrow$
mod.du ↵	rd, rj, rk	$R[rd] \leftarrow \text{unsigned}(R[rj][63:0]) \% \text{unsigned}(R[rk][63:0]) \quad \hookleftarrow$

LoongArch中除法溢出和除0处理

在LA32+Linux系统中所运行程序P中主要包含以下C语言代码段:

```
int a = 0x80000000;  
int b,c;  
scanf("%d",&b);  
c = a/b;  
printf("%d\n", c);
```

没有相关编译选项时, 执行结果如下:

b=-1

-2147483648

b=0

Floating point exception(core dumped)

编译器将上述赋值语句“c=a/b;”转换为机器级代码:

```
ld.w    $r13, $r22, -28(0xffe4) #R[r13]←b;
```

```
ld.w    $r14, $r22, -20(0xfec)   #R[r14]←a;
```

```
div.w    $r12, $r14, $r13        #R[r12]←a/b;
```

```
bne     $r13, $r0, .L0          #若R[r13]! = 0则转.L0
```

```
break    0x7                    #触发断点异常
```

```
.L0
```

```
st.w    $r12, $r22, -24(0xfe8)   #c←a/b;
```

1) 未检测除法溢出

2) LA32中规定: div.w指令在除数为0时正常执行, 结果可为任意值

3) 软件检测整除0, 并在检测到除数为0时, 触发断点(break)异常

4) break指令陷入内核后, 最终执行do_bp()函数

5) 这里编译器可优化一下, 先检测后做除法

异常举例—除法溢出和除0

在LA32+Linux系统中所运行程序P中主要包含以下C语言代码段:

```
int a = 0x80000000;  
int b,c;  
scanf("%d",&b);  
c = a/b;  
printf("%d\n", c);
```

有-fsanitize=undefined和-fsanitize=undefined-trap-on-error编译选项时, 执行结果如下:

b=-1

Trace/breakpoint trap(core dumped)

b=0

Trace/breakpoint trap(core dumped)

编译器将上述赋值语句“c=a/b;”转换为机器级代码段:

```
.....  
srli.w $r13,$r13,0x18  
beq $r13,$r0, .L1  
break 0x0 #发生除法溢出  
.L1  
div.w $r13,$r16,$r12  
bne $r12,$r0, .L0  
break 0x7 #发生除数为0  
.L0  
st.w $r13,$r22,-24(0xfe8)  
.....
```

- 1) 软件检测除法溢出, 并在检测到除法溢出时, 触发断点(break)异常
- 2) 软件检测整除0, 并在检测到除数为0时, 触发断点(break)异常
- 3) 将除法溢出看成未定义行为, 而整除0

在没有上述编译选项时, 也会检测

LA32/LA64整数运算类指令

• 基址加比例变址指令

ALSL.W将rj中31:0位逻辑左移sa2+1位后加rk中31:0位，符号扩展后写入rd

ALSL.WU将rj中31:0位逻辑左移sa2+1位后加rk中31:0位，零扩展后写入rd

在LA32中，上述两条指令的结果没有区别。

ALSL.D将rj中63:0位逻辑左移sa2+1位后加rk中63:0位，所得结果写入rd

sa2是指令中一个两位的立即数字段，故最大为3。

指令	功能
alsl.w rd, rj, rk, sa2	$\text{tmp} = (\text{R}[\text{rj}][31:0] \ll (\text{sa2} + 1)) + \text{R}[\text{rk}][31:0]$ $\text{R}[\text{rd}] \leftarrow \text{SignExtend}(\text{tmp}[31:0], \text{GRLEN})$
alsl.wu rd, rj, rk, sa2	$\text{tmp} = (\text{R}[\text{rj}][31:0] \ll (\text{sa2} + 1)) + \text{R}[\text{rk}][31:0]$ $\text{R}[\text{rd}] \leftarrow \text{ZeroExtend}(\text{tmp}[31:0], \text{GRLEN})$
alsl.d rd, rj, rk, sa2	$\text{R}[\text{rd}] \leftarrow (\text{R}[\text{rj}][63:0] \ll (\text{sa2} + 1)) + \text{R}[\text{rk}][63:0]$

sa2+1的取值为1、2、3、4，故比例因子分别为2、4、8、16

C语言中对数组、结构、联合等数据元素的访问，对应机器级代码中需有“**基址加比例变址**”等寻址方式。比例变址时，变址值等于变址寄存器内容乘以比例系数S（也称**比例因子**），S的含义为操作数的字节数，取值可以是1、2、4或8等

汇编指令中立即数是sa2+1，即实际移位值，而非指令机器中立即数。如汇编指令“alsl.w \$r14,\$r13,\$r12,0x2”的指令机器码为000000000000010 01 01100 01101 01110，其中sa2为01（红字），而汇编指令中填入的是0x2。

LA32/LA64移位指令

- 基础整数指令集提供了16条移位指令，有算术移位、逻辑移位或循环移位
- 移位位数可以是寄存器rk中低m位或立即数所确定的值

右边是移位位数由rk中低m位确定的移位指令

SLL.W、SLL.D是逻辑左移，高位移出，低位补0
逻辑左移与算术左移的结果一致，因此SLL.W和SLL.D也适用于算术左移
SRL.W、SRL.D是逻辑右移，低位丢弃，高位补0
SRA.W、SRA.D是算术右移，低位丢弃，高位补符号
ROTR.W、ROTR.D是循环右移，每右移一次，最低位移到最高位。

指令		功能
sll.w	rd, rj, rk	$tmp = SLL(R[rj][31:0], R[rk][4:0])$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
srl.w	rd, rj, rk	$tmp = SRL(R[rj][31:0], R[rk][4:0])$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
sra.w	rd, rj, rk	$tmp = SRA(R[rj][31:0], R[rk][4:0])$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
rotr.w	rd, rj, rk	$tmp = ROTR(R[rj][31:0], R[rk][4:0])$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
sll.d	rd, rj, rk	$R[rd] \leftarrow SLL(R[rj][63:0], R[rk][5:0])$
srl.d	rd, rj, rk	$R[rd] \leftarrow SRL(R[rj][63:0], R[rk][5:0])$
sra.d	rd, rj, rk	$R[rd] \leftarrow SRA(R[rj][63:0], R[rk][5:0])$
rotr.d	rd, rj, rk	$R[rd] \leftarrow ROTR(R[rj][63:0], R[rk][5:0])$

LA32/LA64移位指令

- 基础整数指令集提供了16条移位指令，有算术移位、逻辑移位或循环移位
- 移位位数可以是寄存器rk中低m位或立即数所确定的值

右边是移位位数由立即数确定的移位指令

SLLI.W、SLLI.D是逻辑左移指令

SRLI.W、SRLI.D是逻辑右移指令

SRAI.W、SRAI.D是算术右移指令

ROTRI.W、ROTRI.D是循环右移指令

ROTRI.W、ROTRI.D是循环右移指令

ROTRI.W、ROTRI.D是循环右移指令

ROTRI.W、ROTRI.D是循环右移指令

ROTRI.W、ROTRI.D是循环右移指令

指令		功能
slli.w ↺	rd, rj, ui5	$tmp = SLL(R[rj][31:0], ui5) \leftarrow$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
srli.w ↺	rd, rj, ui5	$tmp = SRL(R[rj][31:0], ui5) \leftarrow$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
srai.w ↺	rd, rj, ui5	$tmp = SRA(R[rj][31:0], ui5) \leftarrow$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
rotri.w ↺	rd, rj, ui5	$tmp = ROTR(R[rj][31:0], ui5) \leftarrow$ $R[rd] \leftarrow SignExtend(tmp[31:0], GRLEN)$
slli.d ↺	rd, rj, ui6	$R[rd] \leftarrow SLL(R[rj][63:0], ui6) \leftarrow$
srli.d ↺	rd, rj, ui6	$R[rd] \leftarrow SRL(R[rj][63:0], ui6) \leftarrow$
srai.d ↺	rd, rj, ui6	$R[rd] \leftarrow SRA(R[rj][63:0], ui6) \leftarrow$
rotri.d ↺	rd, rj, ui6	$R[rd] \leftarrow ROTR(R[rj][63:0], ui6) \leftarrow$

LA32/LA64移位指令举例

- 在LA64系统中，假设执行下列C语言代码段前long型变量x、y和n已被编译器分配在寄存器r12、r13和r14中，并且执行完该段代码后变量t1、t2和t3的运算结果分别存放在r12、r13和r14中，给出下列C代码段对应的汇编指令序列

```
long t1=x*80;
```

```
long t2=t1&y;
```

```
long t3=t2>>n;
```

解: slli.d \$r15, \$r12, 0x2#x*4

add.d \$r12, \$r15, \$r12 #x*4+x=5*x

slli.d \$r12, \$r12, 0x4#5*x*16=80*x

and \$r13, \$r12, \$r13 #t1&y

sra.d \$r14, \$r13, \$r14 #t2>>n

- 普通访存指令用于实现通用寄存器和存储单元之间传送

1. 基址加立即数访存指令 偏移量由立即数si12符号扩展得到

LD.{B/H/W}从存储单元取出字节/半字/字，符号扩展后送入rd，**LD.D**从存储单元取出双字直接送入rd
LD.{BU/HU/WU}从存储单元取出字节/半字/字，零扩展后送入rd

ST.{B/H/W/D}分别将rd中7:0/15:0/31:0/63:0位写入存储单元中

指令		功能
ld.b	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) byte = M[addr, BYTE] R[rd] ← SignExtend(byte, GRLEN)
ld.h	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) halfword = M[addr, HALFWORD] R[rd] ← SignExtend(halfword, GRLEN)
ld.w	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) word = M[addr, WORD] R[rd] ← SignExtend(word, GRLEN)
ld.d	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) R[rd] ← M[addr, DOUBLEWORD]
ld.bu	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) byte = M[addr, BYTE] R[rd] ← ZeroExtend(byte, GRLEN)
ld.hu	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) halfword = M[addr, HALFWORD] R[rd] ← ZeroExtend(halfword, GRLEN)
ld.wu	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) word = M[addr, WORD] R[rd] ← ZeroExtend(word, GRLEN)
st.b	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) M[addr, BYTE] ← R[rd][7:0]
st.h	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) M[addr, HALFWORD] ← R[rd][15:0]
st.w	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) M[addr, WORD] ← R[rd][31:0]
st.d	rd, rj, si12	addr = R[rj] + SignExtend(si12, GRLEN) M[addr, DOUBLEWORD] ← R[rd][63:0]

2. 基址加寄存器内容访存指令

偏移量由通用寄存器的内容给出

LDX.{B/H/W}从存储单元取出字节/半字/字后，送入rd

LDX.D从存储单元取出双字后送入rd

LDX.{BU/HU/WU}从存储单元取出字节/半字/字，经零扩展后送入rd

STX.{B/H/W/D}将rd中7:0/15:0/31:0/63:0位数据写入存储单元中

ldx.b ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $byte = M[addr, BYTE] \leftarrow$ $R[rd] \leftarrow SignExtend(byte, GRLEN) \leftarrow$
ldx.h ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $halfword = M[addr, HALFWORD] \leftarrow$ $R[rd] \leftarrow SignExtend(halfword, GRLEN) \leftarrow$
ldx.w ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $word = M[addr, WORD] \leftarrow$ $R[rd] \leftarrow SignExtend(word, GRLEN) \leftarrow$
ldx.d ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $R[rd] \leftarrow M[addr, DOUBLEWORD] \leftarrow$
ldx.bu ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $byte = M[addr, BYTE] \leftarrow$ $R[rd] \leftarrow ZeroExtend(byte, GRLEN) \leftarrow$
ldx.hu ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $halfword = M[addr, HALFWORD] \leftarrow$ $R[rd] \leftarrow ZeroExtend(halfword, GRLEN) \leftarrow$
ldx.wu ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $word = M[addr, WORD] \leftarrow$ $R[rd] \leftarrow ZeroExtend(word, GRLEN) \leftarrow$
stx.b ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $M[addr, BYTE] \leftarrow R[rd][7:0] \leftarrow$
stx.h ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $M[addr, HALFWORD] \leftarrow R[rd][15:0] \leftarrow$
stx.w ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $M[addr, WORD] \leftarrow R[rd][31:0] \leftarrow$
stx.d ↵	rd, rj, rk	$addr = R[rj] + R[rk] \leftarrow$ $M[addr, DOUBLEWORD] \leftarrow R[rd][63:0] \leftarrow$

LA32/LA64普通访存指令

3. 基址加4倍立即数访存
偏移量由立即数si14乘4（左移2位）后再符号扩展得到

LDPTR.W从存储单元取出一个字，经符号扩展后送入rd

LDPTR.D从存储单元取出双字后直接送入rd

STPTR.{W/D}将rd中31:0/63:0位直接写入存储单元

指令	功能
ldptr.w rd, rj, si14	$\text{addr} = R[rj] + \text{SignExtend}(\{si14, 2'b0\}, \text{GRLEN})$ $\text{word} = M[\text{addr}, \text{WORD}] \leftarrow$ $R[rd] \leftarrow \text{SignExtend}(\text{word}, \text{GRLEN}) \leftarrow$
ldptr.d rd, rj, si14	$\text{addr} = R[rj] + \text{SignExtend}(\{si14, 2'b0\}, \text{GRLEN})$ $R[rd] \leftarrow M[\text{addr}, \text{DOUBLEWORD}] \leftarrow$
stptr.w rd, rj, si14	$\text{addr} = R[rj] + \text{SignExtend}(\{si14, 2'b0\}, \text{GRLEN})$ $M[\text{addr}, \text{WORD}] \leftarrow R[rd][31:0] \leftarrow$
stptr.d rd, rj, si14	$\text{addr} = R[rj] + \text{SignExtend}(\{si14, 2'b0\}, \text{GRLEN})$ $M[\text{addr}, \text{DOUBLEWORD}] \leftarrow R[rd][63:0] \leftarrow$

LDPTR.W和LD.W功能相近。前者采用“基址+比例位移”，后者采用“基址+位移”方式，LDPTR.W的立即数位数更多且偏移量是4的倍数，可得到更大的位移空间

只要其访存地址自然对齐，都不会触发非对齐异常。非自然对齐时，若硬件实现支持非对齐访存，且当前运算环境配置为允许非对齐访存，则不会触发非对齐异常，否则将触发非对齐异常。

LA32/LA64程序执行流程控制指令

- 程序执行流控制指令包括3条**无条件跳转指令**和8条**条件跳转指令**
- 有直接跳转和间接跳转两种方式
 - ü **直接跳转**指跳转目标地址由立即数作为偏移量而计算得到
 - ü **间接跳转**指跳转目标地址间接存储在某寄存器或存储单元中
- 跳转目标地址的计算方法有两种
 - ü 通过PC加偏移量得到，因偏移量是带符号整数，故跳转目标地址为PC内容增加或减少某一个数值，即采用相对寻址方式得到，可以看成是以当前PC内容为基准往前或往后跳转，称为**相对跳转**
 - ü 直接将指令中设置的目标地址送入PC，称为**绝对跳转**
- 3条无条件跳转指令：B/BL/JIRL

指令**B**实现无条件跳转

指令**BL**用作过程调用，
r1中存放返回地址

指令**JIRL**可实现过程调用的返回，也可实现绝对或相对跳转

b offs26 ↵	$PC = PC + \text{SignExtend}(\{\text{offs26}, 2'b0\}, \text{GRLEN})$ ↵
bl offs26 ↵	$R[r1] = PC + 4$ ↵ $PC = PC + \text{SignExtend}(\{\text{offs26}, 2'b0\}, \text{GRLEN})$ ↵
jirl rd, rj, offs16	$R[rd] = PC + 4$ ↵ $PC = R[rj] + \text{SignExtend}(\{\text{offs16}, 2'b0\}, \text{GRLEN})$

LA32/LA64程序执行流程控制指令

- 8条条件跳转指令

ü 以标志位作为跳转依据，也称分支指令。若满足条件，则跳转到目标地址处执行；否则执行下条指令。采用相对寻址方式进行直接跳转

指令		功能
beq ↵	rj, rd, offs16	if $R[rj] == R[rd]$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
bne ↵	rj, rd, offs16	if $R[rj] != R[rd]$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
blt ↵	rj, rd, offs16	if $\text{signed}(R[rj]) < \text{signed}(R[rd])$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
bge ↵	rj, rd, offs16	if $\text{signed}(R[rj]) \geq \text{signed}(R[rd])$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
bltu ↵	rj, rd, offs16	if $\text{unsigned}(R[rj]) < \text{unsigned}(R[rd])$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
bgeu ↵	rj, rd, offs16	if $\text{unsigned}(R[rj]) \geq \text{unsigned}(R[rd])$ $PC = PC + \text{SignExtend}(\{offs16, 2'b0\}, GRLEN)$ ↵
beqz ↵	rj, offs21 ↵	if $R[rj] == 0$ $PC = PC + \text{SignExtend}(\{offs21, 2'b0\}, GRLEN)$ ↵
bnez ↵	rj, offs21 ↵	if $R[rj] != 0$ $PC = PC + \text{SignExtend}(\{offs21, 2'b0\}, GRLEN)$ ↵

通过减运算实现比较，减运算在补码加减运算器中生成标志位，根据标志位判定两数大小，从而确定跳转到何处执行指令。

LA32/LA64程序执行流程控制指令

- 8条条件跳转指令

ü 若执行A减B，则可得进/借位标志CF、符号标志SF、溢出标志OF和零标志ZF。标志位的含义以及A和B的大小判断规则如下：

LoongArch中，有专门的电路实现减运算并产生标志位，可按标志位判断跳转条件是否满足，但这些标志位并不保存在某个特定的寄存器中。

不同于Intel x86架构，在x86架构中，某些运算指令会生成标志位，并保存在专门的标志寄存器中。

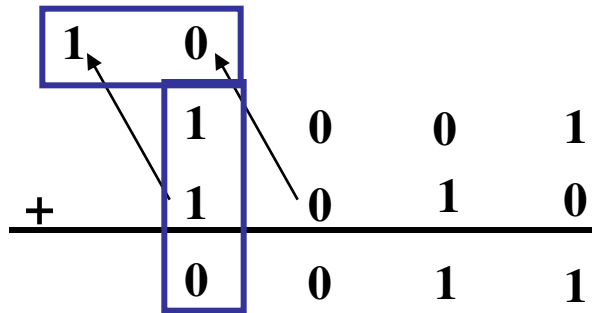
CF=0 且 ZF=0	无符号整数 $A > B$
CF=0	无符号整数 $A \geq B$
CF=1	无符号整数 $A < B$
CF=1 或 ZF=1	无符号整数 $A \leq B$
SF=OF 且 ZF=0	带符号整数 $A > B$
SF=OF	带符号整数 $A \geq B$
SF \neq OF	带符号整数 $A < B$
SF \neq OF 或 ZF=1	带符号整数 $A \leq B$

LoongArch汇编指令中的偏移量是指令机器码中的立即数经相应左移操作（如offs26<<2、offs16<<2、offs21<<2）后得到的值，即汇编指令中给出的偏移量是以字节为单位的偏移值。

回顾：标志信息是干什么的？

Ex1: $-7 - 6 = -7 + (-6) = +3$

$$9 - 6 = 3$$

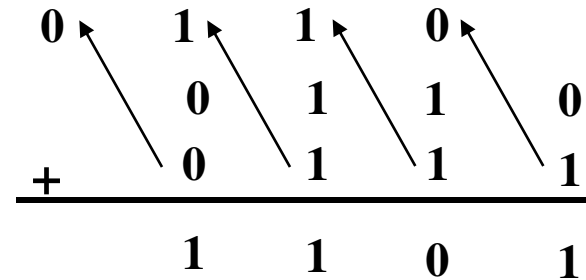


OF=1、ZF=0

SF=0、借位CF=0

$$6 - (-7) = 6 + 7 = -3$$

$$6 - 9 = 13$$



OF=1、ZF=0

SF=1、借位CF=1

做减法以比较大小，规则：

Unsigned: CF=0且ZF=0时，大于；CF=1时，小于


Signed: OF=SF且ZF=0时，大于；OF≠SF时，小于

若相等，则CF=0，OF=SF，故CF=1、OF ≠ SF时，一定小于

LA32/LA64跳转指令举例

```
int sum(int a[ ], unsigned len)
{
    int i, s = 0;
    for (i = 0; i <= len-1; i++)
        s += a[i];
    return s;
}
```

```
sum:
    ...
.L3:
    ...
    ld.w    $r12,$r22,-40(0xfd8)
    addi.w  $r13,$r12,-1(0xff)
    ld.w    $r12,$r22,-20(0xfec)
    bgeu    $r13,$r12,-56(0x3ffc8)
    ...
```



当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。 Why?

i 和 len-1分别存放在哪个寄存器中？ \$r12？ \$r13？

i 在\$r12中， len-1在\$r13中

\$r12 : 0000 ... 0000

\$r13 : 1111 ... 1111

bgeu 指令的执行结果是什么？

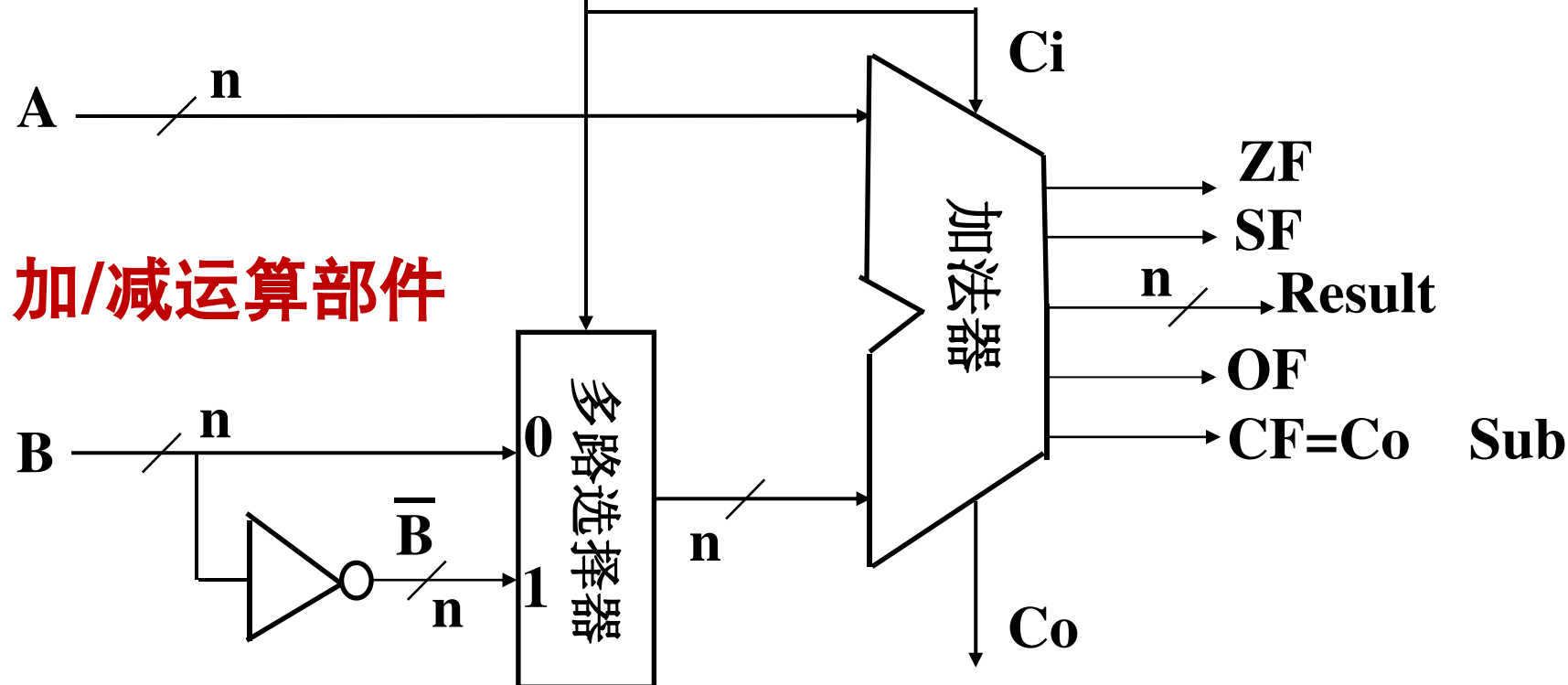
bgeu \$r13,\$r12,-56执行结果

当Sub为1时，做减法
当Sub为0时，做加法

Sub

已知\$r13中为 len-1=FFFF FFFFH

\$r12中为 i=0000 0000H



“bgeu \$r13,\$r12,-56” : A=FFFF FFFFH , B=0000 0000H , Sub=1
， 因此 CF=0, ZF=0, OF=0, SF=1

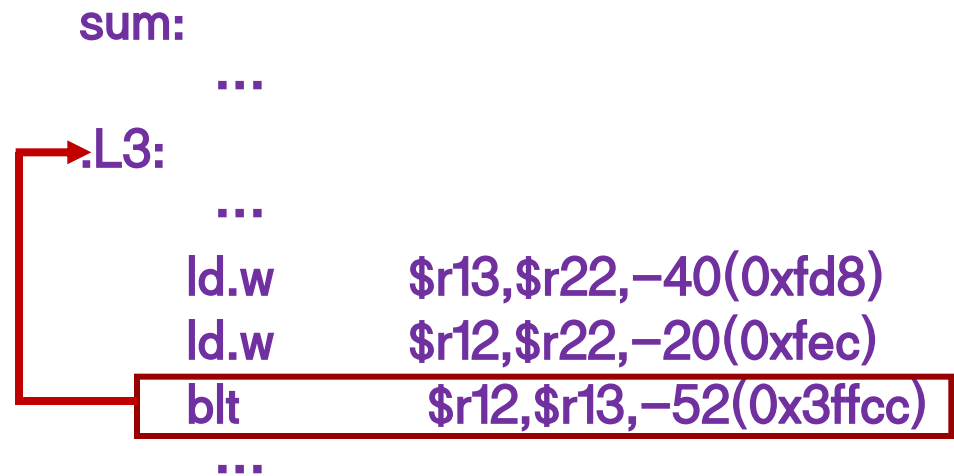
bgeu \$r13,\$r12,-56执行结果

指令	转移条件	说明
bgu label	CF=0 AND ZF=0	无符号数A>B
bgeu label	CF=0 OR ZF=1	无符号数A≥B
bltu label	CF=1 AND ZF=0	无符号数A<B
blteu label	CF=1 OR ZF=1	无符号数A≤B
bg label	SF=OF AND ZF=0	有符号数A>B
bge label	SF=OF OR ZF=1	有符号数A≥B
blt label	SF≠OF AND ZF=0	有符号数A<B
blte label	SF≠OF OR ZF=1	有符号数A≤B

“bgeu \$r13,\$r12,-56” 执行结果是 CF=0, ZF=0, OF=0, SF=1 , 说明满足条件, 应转到.L3执行! 显然, 对于每个 i 都满足条件, 因为任何无符号数都比32个1小, 因此循环体被不断执行, 最终导致数组访问越界而发生存储器访问异常。

LA32/LA64跳转指令举例

```
int sum(int a[ ], int len)
{
    int i, s = 0;
    for (i = 0; i <= len-1; i++)
        s += a[i];
    return s;
}
```



正确的做法是将参数len声明为int型。 Why?

当参数len为0时，返回值应该是0，执行结果确实是0！

i 和 len 分别存放在哪个寄存器中？ \$r12？ \$r13？

i 在\$r12中， len 在\$r13中

\$r12 : 0000 ... 0000

\$r13 : 0000 ... 0000

blt 指令的执行结果是什么？

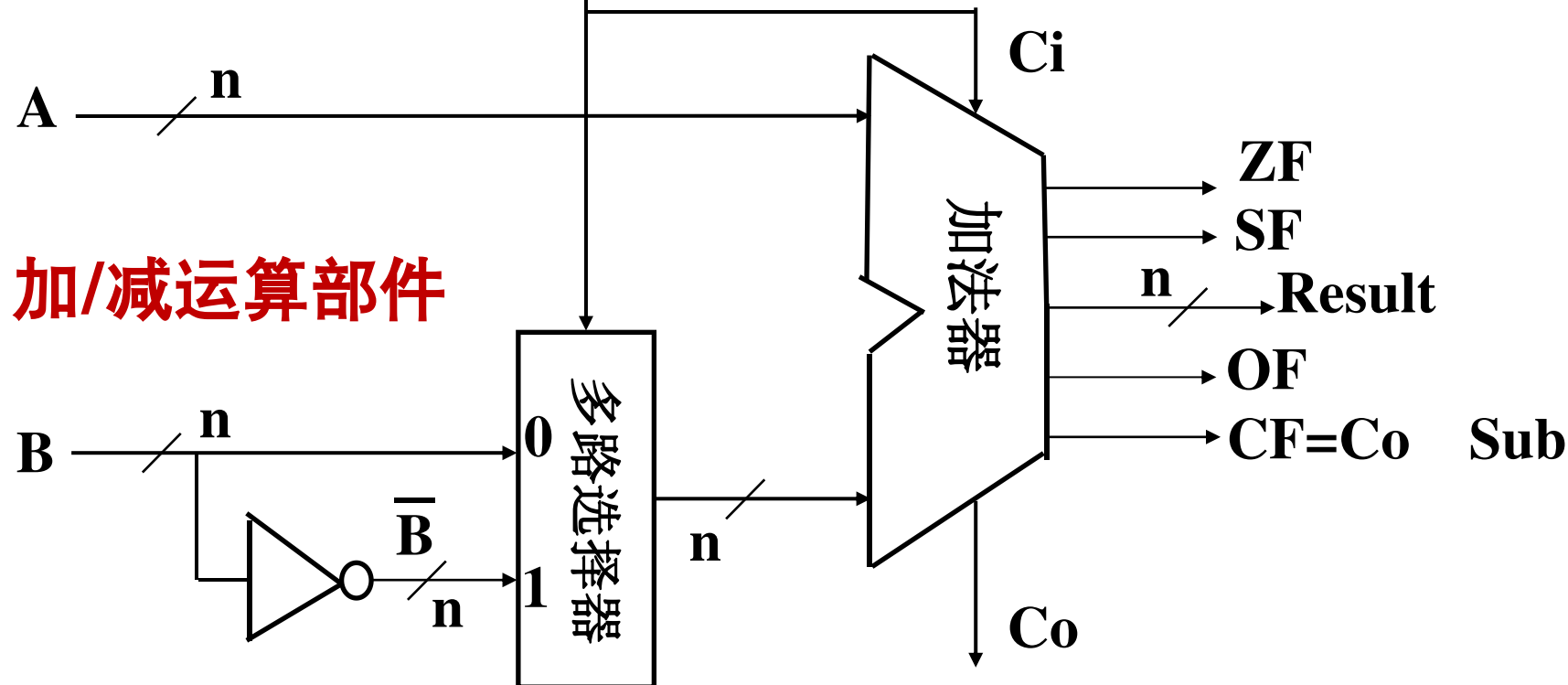
bgeu \$r13,\$r12,-56执行结果

当Sub为1时，做减法
当Sub为0时，做加法

Sub

已知\$r13中为 len=0000 0000H

\$r12中为 i=0000 0000H



“blt \$r12,\$r13,-52” : A=0000 0000H , B=0000 0000H , Sub=1,
因此 CF=0, ZF=1, OF=0, SF=0

blt \$r12,\$r13,-52执行结果

指令	转移条件	说明
bgu label	CF=0 AND ZF=0	无符号数 $A > B$
bgeu label	CF=0 OR ZF=1	无符号数 $A \geq B$
bltu label	CF=1 AND ZF=0	无符号数 $A < B$
blteu label	CF=1 OR ZF=1	无符号数 $A \leq B$
bg label	SF=OF AND ZF=0	有符号数 $A > B$
bge label	SF=OF OR ZF=1	有符号数 $A \geq B$
blt label	SF \neq OF AND ZF=0	有符号数 $A < B$
blte label	SF \neq OF OR ZF=1	有符号数 $A \leq B$

“blt \$r12,\$r13,-52” 执行结果是 CF=0, ZF=1, OF=0, SF=0 , 说明不满足条件, 应跳出循环! 程序执行结束, 返回值为0。

程序的转换与指令系统

- 分以下五个部分介绍

- 第一讲：程序转换概述

- 机器指令和汇编指令
 - 机器级程序员感觉到的属性和功能特性
 - 高级语言程序转换为机器代码的过程

- 第二讲：LA32/LA64指令系统


- 机器指令格式和数据类型
 - 寄存器组织和寻址方式

- 第三讲：LA32/LA64基础整数指令

- 整数运算类指令和移位指令
 - 普通访存指令
 - 程序执行流控制指令

- 第四讲：LA32/LA64基础浮点指令

- 浮点普通访存指令
 - 浮点运算类指令
 - 浮点转换指令和传送指令



围绕LA32/LA64指令系统，解释具体的指令功能

LA32/LA64基础浮点指令

- 基础浮点指令集提供了浮点**运算**类指令、浮点**比较**指令、浮点**转换**指令、浮点**传送**指令、浮点**分支**指令、浮点**访存**指令等常用类型
- 浮点普通访存指令

fld.s	fd, rj, si12	$\text{addr} = \text{R}[\text{rj}] + \text{SignExtend}(\text{si12}, \text{GRLEN})$ $\text{R}[\text{fd}][31:0] \leftarrow \text{M}[\text{addr}, \text{WORD}]$
fld.d	fd, rj, si12	$\text{addr} = \text{R}[\text{rj}] + \text{SignExtend}(\text{si12}, \text{GRLEN})$ $\text{R}[\text{fd}] \leftarrow \text{M}[\text{addr}, \text{DOUBLEWORD}]$
fst.s	fd, rj, si12	$\text{addr} = \text{R}[\text{rj}] + \text{SignExtend}(\text{si12}, \text{GRLEN})$ $\text{M}[\text{addr}, \text{WORD}] \leftarrow \text{R}[\text{fd}][31:0]$
fst.d	fd, rj, si12	$\text{addr} = \text{R}[\text{rj}] + \text{SignExtend}(\text{si12}, \text{GRLEN})$ $\text{M}[\text{addr}, \text{DOUBLEWORD}] \leftarrow \text{R}[\text{fd}]$

访存地址计算与LD和ST指令相同，将rj的内容与si12符号扩展后相加得到
FLD.S从指定存储单元取出32位送入fd的低32位。若浮点寄存器位宽为64位，则fd的高32位不确定

LA32/LA64基础浮点指令

- 浮点普通访存指令

fldx.s	fd, rj, rk	$addr = R[rj] + R[rk]$ $R[fd][31:0] \leftarrow M[addr, WORD]$
fldx.d	fd, rj, rk	$addr = R[rj] + R[rk]$ $R[fd] \leftarrow M[addr, DOUBLEWORD]$
fstx.s	fd, rj, rk	$addr = R[rj] + R[rk]$ $M[addr, WORD] \leftarrow R[fd][31:0]$
fstx.d	fd, rj, rk	$addr = R[rj] + R[rk]$ $M[addr, DOUBLEWORD] \leftarrow R[fd]$

访存地址计算与LDX和STX相同，将rj的内容与rk的内容相加得到
FLDX.S从指定存储单元取出32位送入fd的低32位。若浮点寄存器位
宽为64位，则fd的高32位不确定

LA32/LA64基础浮点指令

• 浮点加、减、乘、除运算类指令

fadd.s	fd, fj, fk	$R[fd][31:0] \leftarrow R[fj][31:0] + R[fk][31:0]$
fadd.d	fd, fj, fk	$R[fd] \leftarrow R[fj] + R[fk]$
fsub.s	fd, fj, fk	$R[fd][31:0] \leftarrow R[fj][31:0] - R[fk][31:0]$
fsub.d	fd, fj, fk	$R[fd] \leftarrow R[fj] - R[fk]$
fmul.s	fd, fj, fk	$R[fd][31:0] \leftarrow R[fj][31:0] * R[fk][31:0]$
fmul.d	fd, fj, fk	$R[fd] \leftarrow R[fj] * R[fk]$
fdiv.s	fd, fj, fk	$R[fd][31:0] \leftarrow R[fj][31:0] / R[fk][31:0]$
fdiv.d	fd, fj, fk	$R[fd] \leftarrow R[fj] / R[fk]$

已知float型变量x1、x2、x3被分配在地址为R[r22]-20、R[r22]-24、R[r22]-28的存储单元中，给出C语句“x3=x1+x2;”的LoongArch汇编代码

```
fld.s    $f0, $r22, -20(0xfec)    #读取x1
fld.s    $f1, $r22, -24(0xfe8)    #读取x2
fadd.s   $f0, $f1, $f0             #计算x1+x2
fst.s    $f0, $r22, -28(0xfe4)    #写入x3
```


LA32/LA64基础浮点指令

- 浮点最大值/最小值指令

fmax.s ↵	fd, fj, fk	$R[fd][31:0] \leftarrow \max(R[fj][31:0], R[fk][31:0])$ ↵
fmax.d ↵	fd, fj, fk	$R[fd] \leftarrow \max(R[fj], R[fk])$ ↵
fmin.s ↵	fd, fj, fk	$R[fd][31:0] \leftarrow \min(R[fj][31:0], R[fk][31:0])$ ↵
fmin.d ↵	fd, fj, fk	$R[fd] \leftarrow \min(R[fj], R[fk])$ ↵
fmaxa.s	fd, fj, fk	$R[fd][31:0] \leftarrow \maxMag(R[fj][31:0], R[fk][31:0])$
fmaxa.d	fd, fj, fk	$R[fd] \leftarrow \maxMag(R[fj], R[fk])$ ↵
fmina.s	fd, fj, fk	$R[fd][31:0] \leftarrow \minMag(R[fj][31:0], R[fk][31:0])$
fmina.d	fd, fj, fk	$R[fd] \leftarrow \minMag(R[fj], R[fk])$ ↵

选择浮点寄存器 fj 中的单精度/双精度浮点数与浮点寄存器 fk 中的单精度/双精度浮点数中的较大者或较小者、绝对值较大者或绝对值较小者送入浮点寄存器fd。

LA32/LA64基础浮点指令

- 浮点绝对值/相反数指令

指令		功能
fabs.s	fd, fj	$R[fd][31:0] \leftarrow \text{abs}(R[fj][31:0])$
fabs.d	fd, fj	$R[fd] \leftarrow \text{abs}(R[fj])$
fneg.s	fd, fj	$R[fd][31:0] \leftarrow \text{negate}(R[fj][31:0])$
fneg.d	fd, fj	$R[fd] \leftarrow \text{negate}(R[fj])$

将浮点寄存器 fj 中的单精度/双精度浮点数的绝对值或相反数（取负）送入浮点寄存器fd。

取绝对值时将其符号位置0，其它部分不变；取相反数时将符号位取反，其它部分不变。

LA32/LA64基础浮点指令

- 单精度与双精度浮点数之间的转换指令

fcvt.s.d	fd, fj	$R[fd][31:0] \leftarrow \text{FP32_convertFormat}(R[fj], \text{FP64})$
fcvt.d.s	fd, fj	$R[fd] \leftarrow \text{FP64_convertFormat}(R[fj][31:0], \text{FP32})$

FCVT.S.D将 fj 中的双精度浮点数转换为单精度浮点数送入fd

FCVT.D.S将 fj 中的单精度浮点数转换为双精度浮点数送入fd

已知double型变量d存于\$f0中，以下情况下，执行“float f=d;”语句对应的指令“fcvt.s.d \$f0, \$f0”后，\$f0的内容发生什么变化？

当 $d=1e10=2\ 540B\ E400H=1.0010\ 1010\ 0000\ 0101\ 1111\ 001 \times 2^{33}$ 时，\$f0中原内容为4202 A05F 2000 0000H，转换为单精度后，\$f0中低32位内容为5015 02F9H，转换前后保留了24位有效数字，故 $f=d$ 。

当 $d=1e11=17\ 4876\ E800H=1.0111\ 0100\ 1000\ 0111\ 0110\ 1110\ 1 \times 2^{36}$ 时，\$f0中原来内容为4237 4876 E800 0000H，转换为单精度后，\$f0中低32位内容为51BA 43B7H，转换后丢弃了最后两个有效数字01，故 $f < d$ 。

LA32/LA64基础浮点指令

- 单精度与双精度浮点数之间的转换指令

fcvt.s.d	fd, fj	$R[fd][31:0] \leftarrow \text{FP32_convertFormat}(R[fj], \text{FP64})$
fcvt.d.s	fd, fj	$R[fd] \leftarrow \text{FP64_convertFormat}(R[fj][31:0], \text{FP32})$

FCVT.S.D将 fj 中的双精度浮点数转换为单精度浮点数送入fd

FCVT.D.S将 fj 中的单精度浮点数转换为双精度浮点数送入fd

已知double型变量d存于\$f0中，以下情况下，执行“float f=d;”语句对应的指令“fcvt.s.d \$f0, \$f0”后，\$f0的内容发生什么变化？

当 $d=1e40=1D\ 6329\ F1C3\ 5CA5\ 0000\ 0000\ 0000\ 0000\ 0000H=1.1101\ 0110\ 0011\ 0010\ 1001\ 1111\ 0001\ 1100\ 0011\ 0101\ 1100\ 1010\ 0101 \times 2^{132}$ 时，\$f0中原内容为485D 6329 F1C3 5CA5H，因为1e40的二进制位数为133，而float型浮点数最多只能表示128位，所以转换为单精度后， $f=+\infty$ ，即发生浮点溢出，\$f0中低32位内容为7F80 0000H。

LA32/LA64基础浮点指令

- 浮点数与整型数据之间的转换指令（进行等值转换）

S、D表示浮点数后缀，W、L表示整数后缀，SINT32、SINT64分别表示32位、64位带符号整数；浮点数向整数转换时，存在小数部分的舍入和有效数位丢失等问题，需考虑舍入根据FCSR寄存器中舍入模式字段RM，有就近舍入到偶数(RNE)、向零舍入(RZ)、向正无穷舍入(RP)、向负无穷舍入(RM)4种。如指令ftintrz.w.s表示float型数转换为int型数时向0方向舍入。

ffint.s.w	fd, fj	$R[fd][31:0] \leftarrow \text{FP32_convertFromInt}(R[fj][31:0], \text{SINT32})$ ↺
ffint.s.l ↺	fd, fj	$R[fd][31:0] \leftarrow \text{FP32_convertFromInt}(R[fj], \text{SINT64})$ ↺
ffint.d.w	fd, fj	$R[fd] \leftarrow \text{FP64_convertFromInt}(R[fj][31:0], \text{SINT32})$ ↺
ffint.d.l ↺	fd, fj	$R[fd] \leftarrow \text{FP64_convertFromInt}(R[fj], \text{SINT64})$ ↺
ftint.w.s ↺	fd, fj	$R[fd][31:0] \leftarrow \text{FP32convertToSint32}(R[fj][31:0], \text{FCSR.RM})$ ↺
ftint.l.s ↺	fd, fj	$R[fd] \leftarrow \text{FP32convertToSint64}(R[fj][31:0], \text{FCSR.RM})$ ↺
ftint.w.d	fd, fj	$R[fd][31:0] \leftarrow \text{FP64convertToSint32}(R[fj], \text{FCSR.RM})$ ↺
ftint.l.d ↺	fd, fj	$R[fd] \leftarrow \text{FP64convertToSint64}(R[fj], \text{FCSR.RM})$ ↺

LA32/LA64基础浮点指令

- 浮点寄存器之间传送指令

FMOV.{S/D}将fj内容送入fd

fmov.s	fd, fj	$R[fd][31:0] \leftarrow R[fj][31:0]$
fmov.d	fd, fj	$R[fd] \leftarrow R[fj]$

- 浮点寄存器与通用寄存器之间传送指令（不进行等值转换！）

MOVGR2FR.W将通用寄存器rj低32位送入浮点寄存器fd低32位。

MOVGR2FRH.W将rj的低32位送入fd的高32位，而fd的低32位不变。

MOVFR2GR.S/MOVFRH2GR.S将浮点寄存器fj的低32位/高32位符号扩展后写入通用寄存器rd。

movgr2fr.w	fd, rj	$R[fd][31:0] \leftarrow R[rj][31:0]$
movgr2frh.w	fd, rj	$R[fd][63:32] \leftarrow R[rj][31:0], R[fd][31:0] \leftarrow R[fd][31:0]$
movgr2fr.d	fd, rj	$R[fd] \leftarrow R[rj]$
movfr2gr.s	rd, fj	$R[rd] \leftarrow \text{SignExtend}(R[fj][31:0], \text{GRLEN})$
movfrh2gr.s	rd, fj	$R[rd] \leftarrow \text{SignExtend}(R[fj][63:32], \text{GRLEN})$
movfr2gr.d	rd, fj	$R[rd] \leftarrow R[fj]$

LA32/LA64基础浮点指令举例

已知float型变量f和int型变量i分别在地址为R[r22]-20和R[r22]-28的存储单元中，实现C语句“i=f;”的**LA64架构**指令代码如下：

```
1 12000080c: 2b3fb2c0 fld.s          $f0, $r22, -20(0xfec)
2 120000810: 011a8400 ftintrz.w.s       $f0, $f0
3 120000814: 0114b40c movfr2gr.s       $r12, $f0
4 120000818: 29bf92cc st.w           $r12, $r22, -28(0xfe4)
```

以下情况下，执行第3条指令后，r12中的内容是什么？

f 的值	f 的机器数	第 2 条指令前 f0[31:0]	第 2 条指令后 f0[31:0]	第 3 条指令后 r12	i 的值
5.9999	0x40bf ff2e	0x40bf ff2e	0x0000 0005	0x0000 0000 0000 0005	5
-5.9999	0xc0bf ff2e	0xc0bf ff2e	0xffff fffb	0xffff ffff ffff fffb	-5
2147483648	0x4f00 0000	0x4f00 0000	0x7fff ffff	0x0000 0000 7fff ffff	2147483647

“ftintrz.w.s \$f0, \$f0”执行后，保存在目的寄存器f0中的是带符号整数，且采用向零方向舍入方式。当f=±5.9999时，直接丢弃小数部分，因此 i 的值为±5；当f=2 147 483 648时，超出 i 表示最大数，该指令将可表示最大数2 147 483 647送入目的寄存器。指令“movfr2gr.s \$r12, \$f0”执行时，将f0[31:0]的内容符号扩展后送入r12。

LA32/LA64指令综合举例

以下是关于函数调用传递参数时进行类型转换的一个C语言程序：

```
1      #include <stdio.h>
2      int funct(int r) {
3          return 2*3.14*r;
4      }
5
6      int main() {
7          float x = funct(5.6);
8          printf("%f\n", x);
9          return 0;
10     }
```

先将上述程序在LA64架构上进行编译、汇编生成可重定位目标文件，然后对可重定位目标文件进行反汇编，根据反汇编结果分析该程序执行过程中进行了哪些类型转换，并分析main函数的两条bl指令的功能，给出其中立即数字段和被调用过程首地址之间的关系。

LA32/LA64指令综合举例

解：可重定位目标文件反汇编部分结果如下（省略了部分指令并加了注释）：

```
00000001200007c4 <funct>: ↵
```

```
    #include <stdio.h> ↵
```

```
    int funct(int r) { ↵
```

```
        ..... ↵
```

```
    return 2*3.14*r; ↵
```

```
7  1200007dc: 28bfb2cc  ld.w      $r12, $r22, -20(0xfec) ↵
```

```
8  1200007e0: 0114a580  movgr2fr.w $f0, $r12    #R[f0]←r ↵
```

```
9  1200007e4: 011d2001  ffint.d.w  $f1, $f0 ↵
```

```
        ..... ↵
```

```
12 1200007f0: 2b800180  fld.d      $f0, $r12, 0    #R[f0]←2*3.14 ↵
```

```
13 1200007f4: 01050020  fmul.d     $f0, $f1, $f0  #R[f0]←2*3.14*r ↵
```

```
14 1200007f8: 011a8800  ftintrz.w.d $f0, $f0 ↵
```

```
15 1200007fc: 0114b40c  movfr2gr.s $r12, $f0 ↵
```

```
16 120000800: 00150184  move      $a0, $r12 ↵
```

```
    } ↵
```

整数r送入浮点寄存器f0，再
转换为浮点数存入f1

将f0中浮点数转换为整数，
再存入通用寄存器r12中

LA32/LA64指令综合举例

0000000120000810 <main>:

.....

int main() {

.....

float f = funct(5.6);

将浮点常数5.6转换成了整型常数5

```
24 120000820: 02801404 addi.w    $a0, $zero, 5(0x5)
25 120000824: 57ffa3ff bl        -96(0xfffffa0) #1200007c4 <funct>
26 120000828: 0015008c move      $r12, $a0      #R[r12]←funct(5.6)
27 12000082c: 0114a980 movgr2fr.d $f0, $r12
28 120000830: 011d1000 ffrint.s.w $f0, $f0
29 120000834: 2b7fb2c0 fst.s     $f0, $r22, -20(0xfec)
```

printf("%f\n", f);

将funct返回的整型数转换为float型数存入f0

```
30 120000838: 2b3fb2c0 fld.s     $f0, $r22, -20(0xfec)
31 12000083c: 01192400 fcvt.d.s   $f0, $f0
32 120000840: 0114b805 movfr2gr.d $a1, $f0
```

.....

将f0中的float型数 f 转换为double型，然后送入参数寄存器

```
35 12000084c: 54608800 bl        24712(0x6088) #1200068d4 <_IO_printf>
```

.....

LA32/LA64指令综合举例

```
0000000120000810 <n
```

```
.....
```

```
int main() {
```

```
.....
```

```
float f = func
```

```
24 120000820: 02801404 addi.w $a0, $zero, 5(0x5)
25 120000824: 57ffa3ff bl -96(0xfffffa0) #1200007c4 <funct>
```

```
26 120000828: 0015008c move $r12, $a0 #R[r12]←funct(5.6)
```

```
27 12000082c: 0114a980 movgr2fr.d $f0, $r12
```

```
28 120000830: 011d1000 ffint.s.w $f0, $f0
```

```
29 120000834: 2b7fb2c0 fst.s $f0, $r22, -20(0xfec)
```

```
printf("%f\n",f);
```

```
30 120000838: 2b3fb2c0 fld.s $f0, $r22, -20(0xfec)
```

```
31 12000083c: 01192400 fcvtd.s $f0, $f0
```

```
32 120000840: 0114b805 mov
```

```
.....
```

```
35 12000084c: 54608800 bl 24712(0x6088) #1200068d4 <_IO_printf>
```

```
.....
```

实现对funct()的调用, 57ffa3ffH=0101 0111 1111 1111 1010 0011 1111 1111B, 由l26-型指令格式可知, 立即数offs26=11 1111 1111 11 1111 1111 1010 0000B=0xfff ffa0, 因此, 该bl指令的跳转目标地址为 0x1 2000 0824+0xfff ffa0=0x1 2000 07c4

该bl指令的跳转目标地址为 0x1 2000 084c+0x0 6088=0x1 2000 68d4