第4季

RISC-V指令集

本节课主要内容

- 本章主要内容
  - RISC-V指令集介绍
  - 指令编码格式
  - RV64I指令详解
  - RV64I指令集总结
  - 指令集实验

技术手册：

1. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Version 20191213
2. The RISC-V Instruction Set Manual Volume II: Privileged Architecture

本节课主要讲解书上第3章内容

RISC-V指令集

- RISC-V采用模块化、增量化设计理念
- 最小指令集合：
  - ✓ RV32I：32位整型最小指令集合
  - ✓ RV64I：64位整型最小指令集合

表 1.1 RISC-V 扩展指令集

指令集扩展	说明
F	单精度浮点数扩展指令
D	双精度浮点数扩展指令
Q	四倍精度浮点数扩展指令
M	整型乘法和除法扩展指令
C	压缩指令
A	原子操作指令
B	位操作指令
E	表示为嵌入式设计的整型指令
H	虚拟化扩展
K	密码运算扩展
V	可伸缩矢量扩展
P	打包SIMD（Packed-SIMD）扩展
J	动态翻译语言（Dynamically Translated Languages）扩展
T	事务内存（ Transactional Memory）
N	用户态中断

例子

```
root:~# cat /proc/cpuinfo
processor     : 0
hart      : 0
isa       : rv64imafdcsu
mmu       : sv48
```

从“isa”可知该系统支持的指令扩展集为：rv64imafdcsu，

- 64位基础整型指令集I
- 乘法和除法指令集M
- 原子操作指令集A
- 单精度指令集F
- 双精度指令集D
- 压缩指令集C
- 特权模式（S）
- 用户模式（U）

指令编码

- 变长指令编码：指令编码长度不固定，空间不受限制，芯片和编译器设计复杂
- 定长指令编码：指令编码长度固定，芯片和编译器设计简单。缺点，指令编码空间固定，受限

- RISC-V CPU大部分采用32/16位固定编码。更宽的指令编码坏处多多。
  - ✓ Code size变大
  - ✓ I cache命令率会下降
- RISC-V规范支持变长指令编码

RISC-V指令编码格式

- 指令宽度：32位 （不考虑压缩指令，压缩指令宽度16位, 规范也支持64位指令，不过目前没有用）

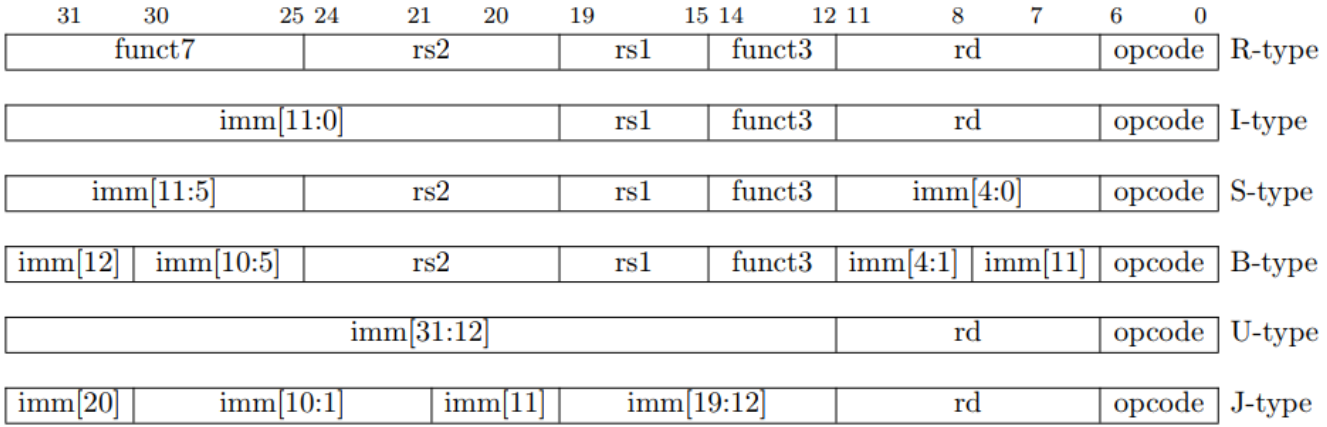
- 指令编码：让CPU知道这条指令是要做什么的？比如让CPU做加法运算

  add x3，x1，x2　//x1+x2=x3

- 指令layout的关键元素：
  - ✓ 指令分类opcode
  - ✓ 源操作数寄存器1
  - ✓ 源操作数寄存器2
  - ✓ 目标寄存器
  - ✓ 立即数

- RISC-V支持32个通用寄存器，采用5 bits就可以索引到。
- 源1+源2+目标，一共需要15 bits，剩余的17 bits可以用于 指令分类

设计一套全新的指令集，主要是构思 指令layout，让CPU和汇编器都follow这套layout
难点：设计的艺术，如何让芯片设计、编译器、OS等得到高效的平衡



RISC-V指令layout，根据长相，分成6种：

- R类型：寄存器与寄存器算术指令。
- I类型：寄存器与立即数算术指令或者加载指令。
- S类型：存储指令。
- B类型：条件跳转指令。
- U类型：长立即数操作指令。
- J类型：无条件跳转指令。

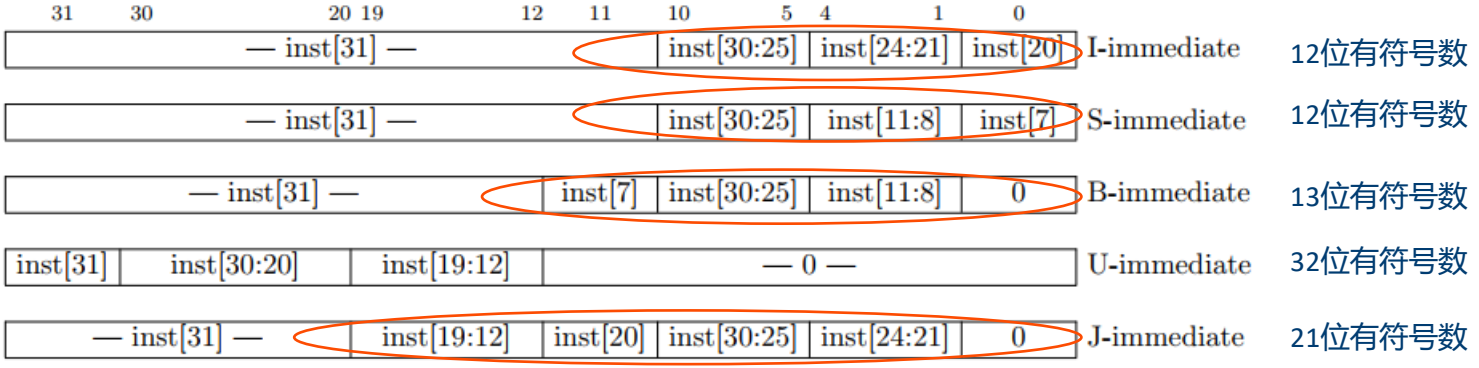
- opcode操作码字段Bit[6:0]，用于指令的分类。
- funct3/funct7功能码字段，与操作码字段结合在一起定义指令的操作功能。
- rd表示目标寄存器的编号，Bit[11:7]。
- rs1字段源操作寄存器1的编号，Bit[19:15]。
- rs2字段源操作寄存器2的编号，Bit[24:20]。
- imm表示立即数。带符号扩展的立即数。

- 指令编码整齐：rd, rs1，rs2都在固定位置，利于RTL实现，减少选择器

RISC-V指令编码中的立即数

立即数：采用有符号数，需要符号扩展。

- ✓ 12位的立即数0x800，它转换成10进制数，变成-2048，而不是2048。



立即数编码有点乱，不过，它是为了 RTL编码方便和实现高效，精心编排的。

- ✓ RTL通过选择器在指令编码中生成和组成立即数
- ✓ 立即数有些位域尽可能固定，这样减少RTL中使用选择器，
  - ✓ imm[10:5]基本上都来自inst[30:25]或者要么是0。2选1选择器就够了
  - ✓ imm[31]来自inst[31]

缺点：编译器稍微复杂一点，需要把立即数打散到指令编码各个位域

负数与有符号数

- 不管正数还是负数都用补码表示
- 正数的原码、反码、补码都一样
- 负数的补码 = 该负数的绝对值的原码的反码加1

- RISC-V指令编码中的立即数都是 有符号数，例如12位有符号数

例子1：用12位有符号数来表示 -2

1. 2的原码：　0000 0000 0010
2. 2的反码：　1111 1111 1101
3. 2的反码加1：　1111 1111 1110

所以，-2的补码为：　1111 1111 1110 = 0xffe

速算经验公式：0xfff – 2 + 1 = 0xffe

例子2：在12位有符号数中，0xff0的十进制是多少

1. 补码为：　1111 1111 0000
2. 除符号位外逐位取反：1000 0000 1111
3. 加1：　1000 0001 0000

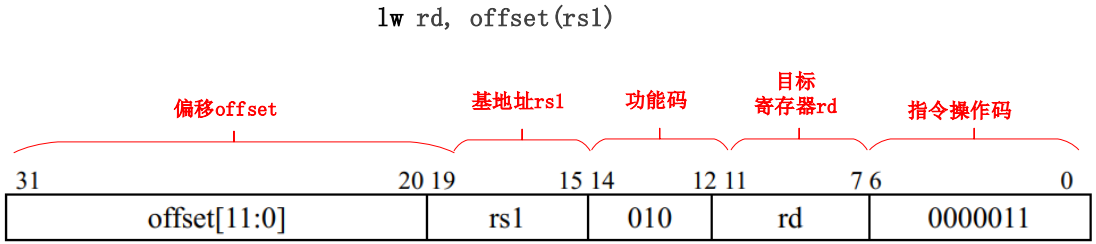
最后取原码：1 0000 = 0x10 = 16，加上符号位 = -16

速算经验公式：-(0xfff – 0xff0 + 1)　= -(0x10) = -16

常见疑问

疑问：RV64指令集支持64位宽的数据和地址寻址，为什么指令的编码宽度只有32位?

- 基于寄存器加载和存储的架构设计，所有的数据加载、存储以及处理都是在通用寄存器中完成
- 使用5位宽可以索引32个通用寄存器
- 使用32位宽比64位宽指令编码，有利于提高指令密度，提高I cache命中率



例子：lw加载指令的指令编码

指令编码非法值

- 奇葩设计：在MIPS中，全是0的指令编码 是nop指令

- RISC-V手册约定下面两种情况是非法指令
  - ✓ 低16位全为0：有可能跳转到全是0的内存区域（刚初始化的内存区域）
  - ✓ 全F（0xffffffff）：有可能跳转到没有编程的存储设备、断开的内存总线，出错的设备等
  - ✓ 方便编程者提前捕抓错误

RISC-V汇编指令书写说明

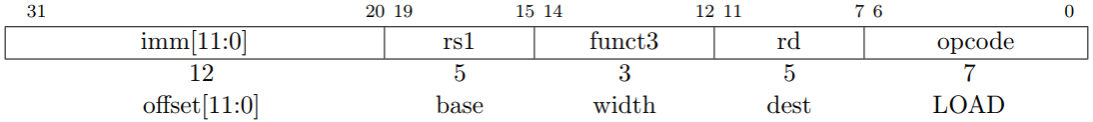
RV64指令集中常用的符号说明如下：

- rd：表示目标寄存器，可以从x0~x31通用寄存器中选择。
- rs1：表示源寄存器1，可以从x0~x31通用寄存器中选择。
- rs2: 表示源寄存器2，可以从x0~x31通用寄存器中选择。
- ()：通常用来表示寻址模式.
  - ✓ （a0）表示以a0寄存器的值为基地址进行寻址。
  - ✓ 这个前面还可以加offset，表示偏移, 例如8(a0)，表示以a0寄存器的值为基地址，然后偏移8个字节进行寻址。
- { }：表示可选项。
- imm： 表示立即数。

指令1：加载指令

加载指令格式

```
l{d|w|h|b}{u} rd, offset(rs1),
```



- {d|w|h|b}：表示加载的数据宽度。
- {u}是可选项，表示加载的数据为无符号数，即采用零扩展方式。如果没有这选项，表示加载的数据为有符号数，即采用符号扩展方式。
- rd：表示目标寄存器。
- rs1：表示源地址寄存器。
- (rs1)：表示以rs1寄存器的值为基地址进行寻址，简称rs1地址。
- offset：表示以源寄存器的值为基地址的偏移量。offset是12位有符号数，取值范围为[-2048 ~ 2047]。

例子：　lb t1, 4(t0)

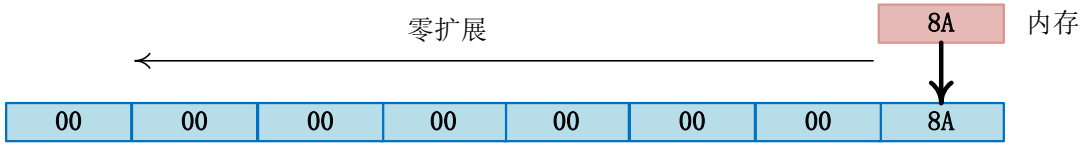
表 3.2 加载指令

加载指令	数据位宽	说明
lb rd, offset(rs1)	8	以rs1寄存器的值为基地址，在偏移offset的地址处加载一个字节数据，经过符号扩展之后写入到目标寄存器rd中
lbu rd, offset(rs1)	8	以rs1寄存器的值为基地址，在偏移offset的地址处加载一个字节数据，经过零扩展之后写入到目标寄存器rd中
lh rd, offset(rs1)	16	以rs1寄存器的值为基地址，在偏移offset的地址处加载两个字节数据，经过符号扩展之后写入到目标寄存器rd中
lhu rd, offset(rs1)	16	以rs1寄存器的值为基地址，在偏移offset的地址处加载两个字节数据，经过零扩展之后写入到目标寄存器rd中
lw rd, offset(rs1)	32	以rs1寄存器的值为基地址，在偏移offset的地址处加载四个字节数据，经过符号扩展之后写入到目标寄存器rd中
lwu rd, offset(rs1)	32	以rs1寄存器的值为基地址，在偏移offset的地址处加载四个字节数据，经过零扩展之后写入到目标寄存器rd中
ld rd, offset(rs1)	64	以rs1寄存器的值为基地址，在偏移offset的地址处加载八个字节数据，写入到目标寄存器rd中
lui rd, imm	64	先把imm立即数左移12位，然后进行符号扩展，结果写入到rd寄存器中

例子

```
1    li t0, 0x80000000
2
3    lb t1, (t0)
4    lb t1, 4(t0)
5    lbu t1, 4(t0)
6    lb t1, -4(t0)
7    ld t1, (t0)
8    ld t1, 16(t0)
```

符号扩展



零扩展

例子：一个错误的例子

```
lb a1, 2048(a0)
lb a1,-2049(a0)
```

编译出错

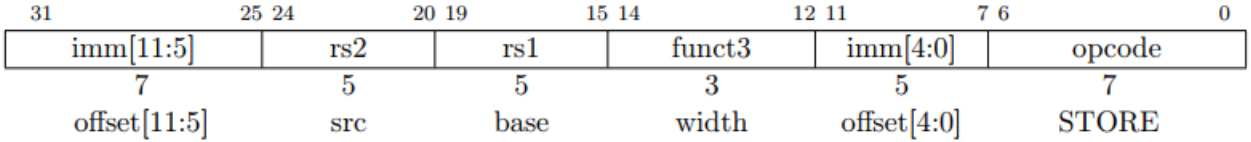
```
AS   build_src/boot_s.o
src/boot.S: Assembler messages:
src/boot.S:6: Error: illegal operands `lb a1,-2049(a0)'
src/boot.S:7: Error: illegal operands `lb a1,2048(a0)'
make: *** [Makefile:28: build_src/boot_s.o] Error 1
```

offset是12位有符号数，取值范围为[-2048 ~ 2047]。

指令2：存储指令

存储指令格式

```
s{d|w|h|b} rs2, offset(rs1),
```



- {d|w|h|b}：表示加载的数据宽度。
- rd：表示目标寄存器。
- rs1：表示源地址寄存器。
- (rs1)：表示以rs1寄存器的值为基地址进行寻址，简称rs1地址。
- rs2：源寄存器2。
- offset：表示以源寄存器的值为基地址的偏移量。offset是12位有符号数，取值范围为[-2048 ~ 2047]。

例子：　sd t1, 8(t0)

不对齐访问

RISC-V规范中没有强制 约定不对齐访问。让CPU设计者做出选择：

1. 纯硬件实现：RTL中实现不对齐访问。
2. 软件实现：遇到不对齐访问时，触发异常，由软件来处理，例如通过lb/sb指令

QEMU模拟器支持不对齐访问

小练习 1

在benos中练习：
请在BenOS里做如下练习，可以新建一个汇编文件。

1. 使用li指令把 0x80200000加载到a0寄存器。
   使用li指令把立即数16加载到a1寄存器。
2. 从0x80200000地址中读取4个字节的数据。
3. 从0x80200010地址中读取8个字节的数据。
4. 下面lui指令，最后结果是多少。

```
lui t0, 0x8034f
lui t1, 0x400
```

GDB查看内存：x

x/16xb：
- ✓ 16表示查看16个内存单元
- ✓ X表示16进制
- ✓ b表示1字节为一个内存单元

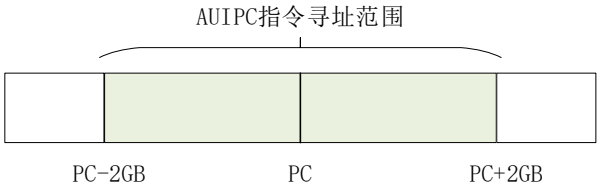
编写汇编代码，并使用 GDB来单步调试。
提示：在GDB中可以使用x命令来读取内存地址的值，然后和寄存器的值进行比较来验证是否正确。
提示：在gdb中可以使用x命令来读取内存地址的值，然后和寄存器的值进行比较，来验证是否正确
(gdb) x/16xb 0x80000

```
(gdb) x/16xb 0x80200000
0x80200000 <_start>:    0x73    0x10    0x40    0x10    0xef    0x00    0x00    0x70
0x80200008 <_start+8>:  0x97    0x00    0x00    0x00    0xe7    0x80    0xc0    0x01
```

指令3：PC相对寻址

- PC：程序计数器（Program Counter，PC）用来指示下一条指令的地址。
- RISC-V指令集提供了1条PC相对寻址的指令：AUIPC

```
auipc rd, imm
```



- 把imm立即数左移12位并符号扩展到64位后得到一个新的立即数，这个新的立即数是一个有符号的立即数，然后加上当前PC值，然后存储到rd寄存器中。
- 这条指令能寻址的范围为基于当前PC偏移±2GB

rd = PC + Signed (imm << 12)

- LUI指令：把imm立即数左移12位得到一个新的32位立即数，并且符号扩展到64位存储到rd寄存器中。

rd = Signed (imm << 12)

LUI和AUIPC指令编码长的非常像，不同的地方，LUI指令的opcode为0110111，AUIPC指令的opcode为：0010111

例子：

1. 假设当前PC值为0x80200000，分别执行如下指令，那么a5和a6寄存器的值是多少。

```
auipc   a5,0x2
lui     a6, 0x2
```

a5寄存器的值为： PC + sign_extend(0x2 << 12) = 0x80200000 + 0x2000 = 0x80202000。
a6寄存器的值为： 0x2 << 12 = 0x2000。

2.假设当前PC值为0x80200000，想加载0x80202008地址的数据，那用auipc指令怎么写？

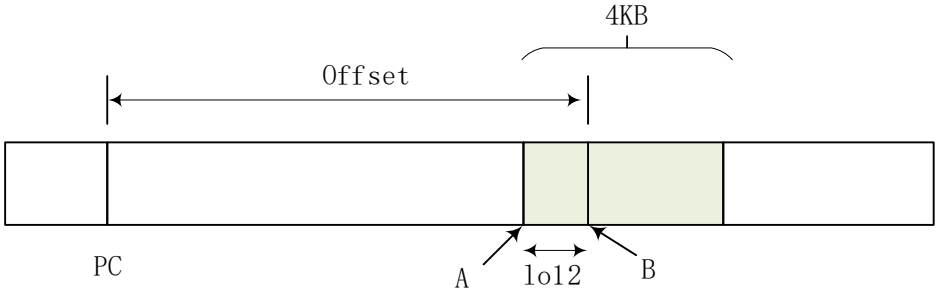
AUIPC指令通常和ADDI联合使用来实现32位PC相对寻址。AUIPC指令可以寻址与被访问地址4KB对齐的地方，即被访问地址的高20位部分。ADDI指令可以在[-2048 ~ 2047]范围内寻址，即被访问地址的低12位部分。

```
auipc a5，2
addi  a0，a5，8
ld t0，(a0)
```

```
auipc a5，2
ld t0，8(a5)
```

计算hi20和lo12

如果知道了当前PC值和目标地址B，那如何计算AUIPC和ADDI指令的参数呢?



- B：目标地址
- offset：为地址B与当前PC值的偏移量
- A：是地址B与4KB对齐的地方
- lo12：地址A与地址B的偏移量，
  有符号数的12位数值，取值范围为[-2048 ~ 2047]。

对后面理解链接重定位非常有帮助

```
hi20 = (offset >> 12) + offset[11]
lo12 = offset & 0xfff
```

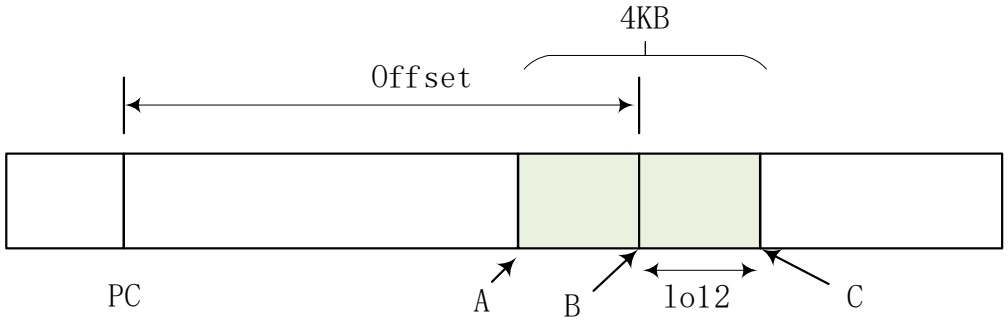
- hi20表示地址的高20位部分，用在AUIPC指令的imm操作数中。
- lo12表示地址的低12位部分，用于ADDI指令的imm操作数中。
- 计算hi20时需要加上offset[11]，是为了抵消低12位的有符号数的影响。
- lo12是一个12位有符号数，取值范围为[-2048 ~ 2047]。

```
auipc a0, hi20
addi a1, a0, lo12
```

例子

假设PC值为0x80200000，地址B为0x80201800。地址B正好在4KB的正中间。地址B离地址A的偏移量为2048，而离地址C的偏移量为-2048。

请计算hi20和lo12，完成auipc和addi指令。



根据公式：

hi20 = (0x1800 >> 12) + offset[11] = 2
lo12 = 0x800

auipc a0, 2

addi a1, a0, **-2048**

lo12为12位有符号数，所以12位有符号数0x800用十进制数来表示为-2048

如果写成如下，编译器则报错：

addi a1, a0, 0x800

```
AS   build_src/boot_s.o
src/boot.S: Assembler messages:
src/boot.S:6: Error: illegal operands `addi a1,a0,0x800'
make: *** [Makefile:28: build_src/boot_s.o] Error 1
```

因为汇编器把字符“0x800”当成64位数值来解析的，即2048，它已经超过了ADDI指令立即数的取值范围

与PC相关的加载和存储伪指令

- 因为编写汇编代码时不知道当前PC值是多少。计算hi20和lo12的过程通常留给链接器在重定位时完成。
- RISC-V定义了几条常用的伪指令，这些伪指令是基于AUIPC指令的

表 3.4 与 PC 相关的加载和存储伪指令

伪指令	指令组合	说明
la rd, symbol (非PIC)	auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]	加载符号的绝对地址 其中delta = symbol − pc
la rd, symbol (PIC)	auipc rd, delta[31 : 12] + delta[11] l{w|d} rd, rd, delta[11:0]	加载符号的绝对地址 其中 delta = GOT[symbol] − pc
lla rd, symbol	auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]	加载符号的本地地址 其中delta = symbol − pc
l{b|h|w|d} rd, symbol	auipc rd, delta[31 : 12] + delta[11] l{b|h|w|d} rd, delta[11:0](rd)	加载符号的内容
s{b|h|w|d} rd, symbol, rt	auipc rt, delta[31 : 12] + delta[11] s{b|h|w|d} rd, delta[11:0](rt)	存储内容到符号中 其中rt为临时寄存器
li rd, imm	根据情况扩展为多条指令	加载立即数imm到rd寄存器中

例子：观察li伪指令

```
<asm.S>
.global asm_test
asm_test:
     li t0, 0xffffffff80200000
     ret
```

在QEMU+RISC-V平台上编译。

```
# gcc main.c asm.S -O2 -g -o test
```

在QEMU+RISC-V平台上编译。

```
root# objdump -d test

0000000000005fc <asm_test>:
 5fc:72e1                lui t0,0xffff8
 5fe:4012829b            addiw   t0,t0,1025
 602:02d6                slli    t0,t0,0x15
 604:8082                ret
```

li伪指令由lui、addiw和slli三条指令组成

小练习2：PC相对地址寻址

1. 实验目的

熟悉PC相对地址寻址的加载和存储指令。

2. 实验要求

在BenOS里做如下练习。
在汇编文件中输入如下代码：

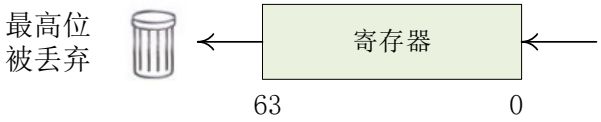
```
#define MY_OFFSET -2048

auipc t0, 1
addi t0, t0，  MY_OFFSET
ld t1, MY_OFFSET(t0)
```

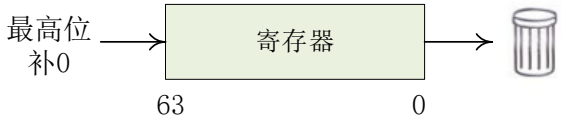
请问t0和t1寄存器的值分别是多少？
请使用GDB单步调试并观察寄存器的变化。

指令4：移位指令

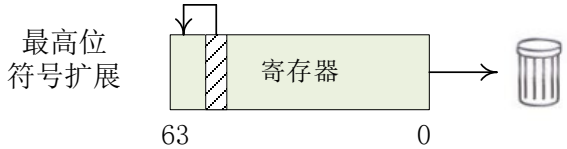
- sll：逻辑左移（Shift Left Logical）指令，最高位会被丢弃，最低位补0
- srl：逻辑右移（Shift Right Logical）指令，最高位补0，最低位会被丢弃。
- sra：算术右移（Shift Right Arithmetic）指令，最低位会被丢弃，最高位会按照符号进行扩展



（a）sll逻辑左移



（b）srl逻辑右移



（c）sra算术右移

sll	逻辑左移	sll rd, rs1, rs2	rd = rs1 << rs2[5:0] 其中 rs2 只取最低 6 位数值。
srl	逻辑右移	srl rd, rs1, rs2	rd = rs1 >> rs2[5:0] 其中 rs2 只取最低 6 位数值。
sra	算术右移	sra rd, rs1, rs2	tmp = rs1 >> rs2[5:0] rd = Signed_with_rs1(tmp) 其中 rs2 只取最低 6 位数值，另外结果需要根据 rs1 的符号位来做符号扩展。

注意地方：
1．RISC-V指令集里没有单独设置一个算术左移的指令，因为sll逻辑左移指令会把最高位丢弃。
2．逻辑右移和算术右移的区别在于是否考虑符号问题。

例如，源操作数二进制数：1010101010
逻辑右移一位：[0]101010101　（最高一位，永远补0）
算术右移一位：[1]101010101　（算术右移，需要按照**源操作数**进行符号扩展）

3.RV64指令集中，SLL、SRL以及SRA指令中只使用rs2寄存器中低6位数据做移位操作。

移位指令的扩展 – 带立即数的移位指令

slli	立即数逻辑左移	slli rd, rs1, shamt	rd = rs1 << shamt 其中 shamt 为 6 位无符号数
srli	立即数逻辑右移	srli rd, rs1, shamt	rd = rs1 >> shamt 其中 shamt 为 6 位无符号数
srai	立即数算术右移	srai rd, rs1, shamt	tmp = rs1 >> shamt rd = Signed_with_rs1(tmp) 其中 shamt 为 6 位无符号数，另外结果需要根据 rs1 的符号位来做符号扩展。

移位指令的扩展 – 32位移位指令（仅限RV64）

sllw	逻辑左移（低 32 位版本）	sllw rd, rs1, rs2	tmp = rs1[31:0] << rs2[4:0] tmp = tmp[31:0] rd = signed (tmp) 其中 rs2 只取最低 5 位数值。
srlw	逻辑右移（低 32 位版本）	srlw rd, rs1, rs2	tmp = rs1[31:0] >> rs2[4:0] rd = signed (tmp) 其中 rs2 只取最低 5 位数值。
sraw	算术右移（低 32 位版本）	sraw rd, rs1, rs2	tmp = rs1[31:0] >> rs2[4:0] rd = Signed_with_rs1(tmp) 其中 rs2 只取最低 5 位数值，另外结果需要以 rs1 的 Bit[31] 为符号位来做符号扩展。
slliw	立即数逻辑左移（低 32 位版本）	slliw rd, rs1, shamt	tmp = rs1[31:0] << shamt tmp = tmp[31:0] rd = signed (tmp) 其中 shamt 为 6 位无符号数
srliw	立即数逻辑右移（低 32 位版本）	srliw rd, rs1, shamt	tmp = rs1[31:0] >> shamt tmp = tmp[31:0] rd = signed (tmp) 其中 shamt 为 6 位无符号数
sraiw	立即数算术右移（低 32 位版本）	sraiw rd, rs1, shamt	tmp = rs1[31:0] >> shamt tmp = tmp[31:0] rd = Signed_with_rs1(tmp) 其中 shamt 为 6 位无符号数，另外结果需要以 rs1 的 Bit[31] 为符号位来做符号扩展。

小练习3：移位指令

实验要求

在BenOS里做如下练习。
在汇编文件中输入如下代码：

```
shift_test:
        li t0, 0x8000008a00000000
        srai a1, t0, 1
        srli t1, t0, 1

        li t0, 0x128000008a
        sraiw a2, t0, 1
        srliw t1, t0, 1
        slliw a3, t0, 1

        li t0, 0x124000008a
        sraiw a2, t0, 1
        srliw t1, t0, 1
        slliw a4, t0, 1
        ret
```

请使用GDB单步调试并观察寄存器的变化。

```
li t0, 0x8000008a00000000
srai a1, t0, 1
srli t1, t0, 1
```

srai是立即数算术右移指令，把0x8000008a00000000右移一位并且根据源二进制数的最高位需要进行有符号扩展，最后结果为0xc000004500000000。
srli是立即数逻辑右移指令，把0x8000008a00000000右移一位并且最高位补0，最后结果为0x4000004500000000。

```
1    li t0, 0x128000008a
2    sraiw a2, t0, 1
3    srliw a3, t0, 1
4
5    li t0, 0x124000008a
6    sraiw a4, t0, 1
```

第2行，截取t0寄存器低32位的值（0x8000008a）作为新的源操作数，然后右移一位等于0x40000045，根据新的源二进制数的最高位需要进行有符号扩展，最后结果为0xfffffffc0000045。
第3行，截取t0寄存器低32位的值（0x8000008a）作为新的源操作数，然后右移一位并且进行有符号零扩展，最后结果为0x40000045。
第6行，截取t0寄存器低32位的值（0x4000008a）作为新的源操作数，然后右移一位等于0x20000045，根据新的源二进制数的最高位需要进行有符号扩展，最后结果为0x20000045。

指令5：位操作指令

- RV64I指令集提供与（and）、或（or）以及异或（xor）三种位操作指令

and	与操作	and rd, rs1, rs2	rd = rs1 & rs2
xor	异或操作	xor rd, rs1, rs2	rd = rs1 ^ rs2
or	或操作	or rd, rs1, rs2	rd = rs1 | rs2

- 立即数位操作指令

xori	立即数异或操作	xori rd, rs1, imm	rd = rs1 ^ imm 其中 imm 为 12 位有符号数
ori	立即数或操作	ori rd, rs1, imm	rd = rs1 | imm 其中 imm 为 12 位有符号数
andi	立即数与操作	andi rd, rs1, imm	rd = rs1 & imm 其中 imm 为 12 位有符号数

异或操作的妙用

- 异或的3个小特点

(1) 0^0=0,0^1=1， 0异或任何数 = 任何数
(2) 1^0=1,1^1=0， 1异或任何数 = 任何数取反
(3) 任何数异或自己 = 把自己置0

- 异或的几个小妙用

（1）使某些特定的位翻转

例如想把10100**00**1的第1位和第2位翻转，则可以将该数与00000110进行按位异或运算。

10100001^00000110 = 10100111

（2）交换两个数.

例如交换两个整数a=10100001，b=00000110的值，可通过下列语句实现：

```
a = a^b;        //a=10100111
b = b^a;        //b=10100001
a = a^b;        //a=00000110
```

（3）在汇编里让变量设置为0，例如x1寄存器

```
xor x1, x1
```

（4）判断两个是否相等

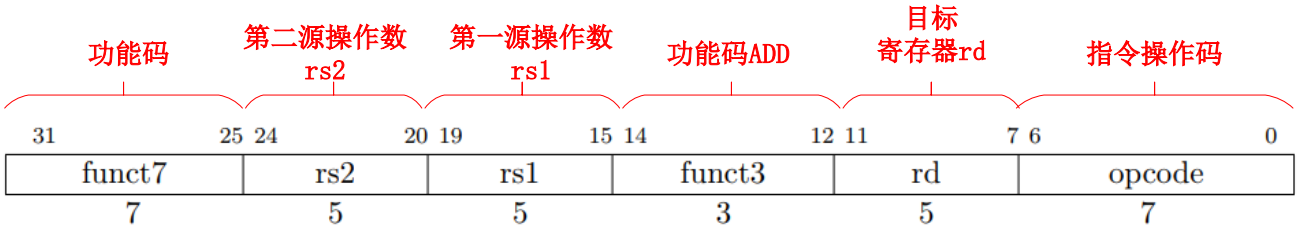
```
return ((a ^ b) == 0)
```

指令6：算术运算指令

- RV64I指令集只提供最基础的加法和减法指令

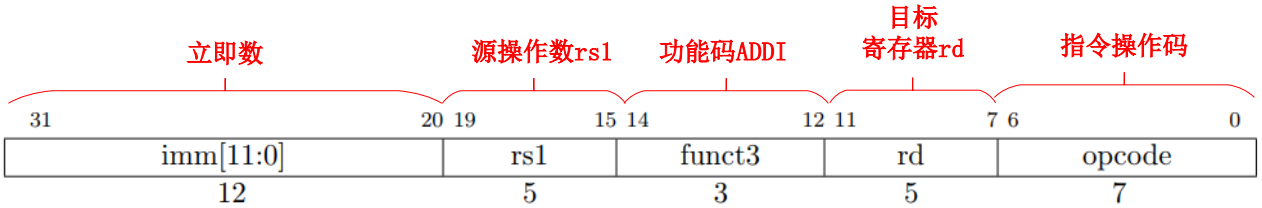
add	加法指令	add rd, rs1, rs2	rd = rs1 + rs2
sub	减法指令	sub rd, rs1, rs2	rd = rs1 - rs2
addi	立即数加法	addi rd, rs1, imm	rd = rs1 + imm 其中 imm 为 12 位有符号数
addiw	立即数加法（低 32 位版本）	addi rd, rs1, imm	tmp = rs1[31:0] + imm tmp = tmp[31:0] rd = signed (tmp) 其中 imm 为 12 位有符号数
addw	加法（低 32 位版本）	addw rd, rs1, rs2	tmp = rs1[31:0] + rs2[31:0] tmp = tmp[31:0] rd = signed (tmp)
subw	减法（低 32 位版本）	subw rd, rs1, rs2	tmp = rs1[31:0] - rs2[31:0] tmp = tmp[31:0] rd = signed (tmp)

add rd, rs1, rs2



R类型

addi rd, rs1, imm



I类型

注意：imm立即数位12位有符号数

例子

下面两条指令哪一条是非法指令

```
addi a1, t0, 0x800
addi a1, t0, 0xfffffffffffff800
```

第一条指令为非法指令：

src/asm_test.S: Assembler messages:
src/asm_test.S:12: Error: illegal operands `addi a1,t0,0x800'

在GNU AS汇编中，0x800被看作是一个数值为2048的无符号数，而不是12位宽的带符号扩展的立即数。如果想表示“-2048”立即数，我们需要使用0xfffffffffffff800

例子

```
1    li t0, 0x140200000
2    li t1, 0x40000000
3
4    addi a1, t0, 0x80
5    addiw a2, t0, 0x80
6
7    add a3, t0, t1
8    addw a4, t0, t1
```

第4行，a1寄存器的值为0x140200080。
第5行，先把t0寄存器的值截取低32位得到0x40200000，然后与立即数0x80相加，最后做符号扩展并存入到a2寄存器中，a2寄存器的值为0x40200080。

第7行，a3寄存器的值为0x180200000。
第8行，先把t0寄存器的值截取低32位得到0x40200000，把t1寄存器的值截取低32位得到0x40000000，然后相加，结果为0x80200000，最后做符号扩展并写入到a4寄存器中，a4寄存器的值为0xffffffff80200000。

小练习4：算术运算指令

实验要求

在BenOS里做如下练习。
在汇编文件中输入如下代码：

```
51 add_sub_test:
52         addi a1, t0, 0x800
53         addi a1, t0, 0xfffffffffffff800
54
55         li t0, 0x140200000
56         li t1, 0x40000000
57
58         addi a1, t0, 0x700
59         addi a1, t0, 0xfffffffffffff800
60         addiw a2, t0, 0x80
61
62         add a3, t0, t1
63         addw a4, t0, t1
64
65         li t0, 0x180200000
66         li t1, 0x200000
67
68         sub a0, t0, t1
69         subw a1, t0, t1
70         ret
```

请指出下面那条指令是错误的。
请使用GDB单步调试并观察寄存器的变化。

指令7：比较指令

- RV64I指令集提供4条比较指令

slt	小于比较指令	slt rd, rs1, rs2	rd = (rs1 < rs2) ? 1 : 0 其中 rs1 和 rs2 为有符号数
sltu	小于比较指令	sltu rd, rs1, rs2	rd = (rs1 < rs2) ? 1 : 0 其中 rs1 和 rs2 为无符号数
slti	小于比较指令	sltu rd, rs1, rs2	rd = (rs1 < imm) ? 1 : 0 其中 imm 为 12 位有符号数, rs1 为有符号数
sltiu	立即数小于比较	sltiu rd, rs1, imm	rd = (rs1 < imm) ? 1 : 0 其中 imm 为 12 位有符号数, rs1 为无符号数

- RV64I指令集提供比较伪指令

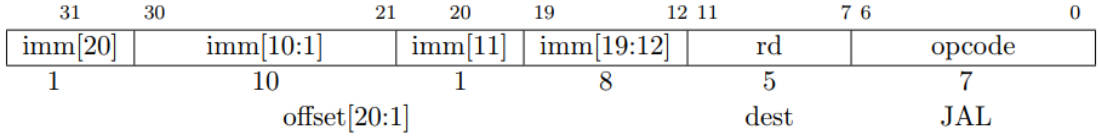
伪指令	伪指令格式	说明
sltz	sltz rd, rs1	小于0则置位指令。 如果rs1的值小于0，向rd寄存器写入1，否则写入0。
snez	snez rd, rs2	不等于则置位指令。 如果rs2寄存器的值不等于0，向rd寄存器写入1，否则写入0。
seqz	seqz rd, rs1	等于0则置位指令。 如果rs1寄存器的值等于0，向rd寄存器写入1，否则写入0。
sgtz	sgtz rd, rs1	大于0则置位指令。 如果rs1的值大于0，向rd寄存器写入1，否则写入0。

指令8：无条件跳转指令

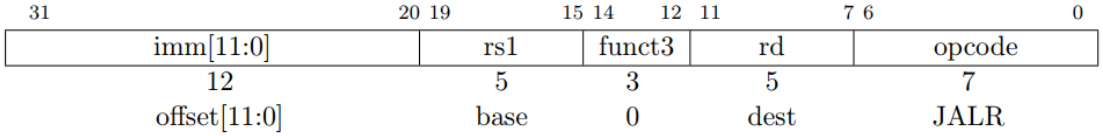
- RV64I指令集提供2条无条件跳转指令：

jal	跳转并链接指令	jal rd, label	PC = PC + offset rd = PC + 4 其中 offset 为 21 位有符号数（offset 为 2 字节对齐，最低位为 0），它的数值为 label 地址与当前 PC 值的偏移量。
jalr	使用寄存器跳转并链接指令	jalr rd, offset(rs1)	PC = rs1 + offset rd = PC + 4 其中 offset 为 12 位有符号数

- JAL（jump and link）指令使用J类型的指令编码，其中操作数offset[20:1]由指令编码的Bit[31:12]构成，它默认是2的倍数，因此它的跳转范围大约为当前PC值偏移±1 MB范围
- 把返回地址（即PC + 4）存储到rd寄存器中
- 如果把返回地址存储到ra寄存器中，则可以实现函数返回



- JALR（jump and link register）指令使用I类型指令编码，要跳转的地址由rs1寄存器和offset操作数组成，其中offset是一个12位的有符号立即数。



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

- RV64I指令集提供无条件跳转伪指令。

伪指令	指令组合	说明
j label	jal x0, offset	跳转到lable标签处，不带返回地址
jal label	jal ra, offset	跳转到label标签处，返回地址存储在ra寄存器中。
jr rs	jalr x0, 0(rs)	跳转到rs寄存器地址处，不带返回地址
jalr rs	jalr ra, 0(rs)	跳转到rs寄存器地址处，返回地址存储在ra寄存器中。
ret	jalr x0, 0(ra)	从ra寄存器中获取返回地址，并返回。常用于子函数返回。
call func	auipc ra, offset[31:12]+ offset[11] jalr ra, offset[11:0](ra)	调用子函数func，返回地址保存到ra寄存器中。
tail func	auipc x6, offset[31:12]+ offset[11] jalr x0, offset[11:0](x6)	调用子函数func，不保存返回地址。

例子

假设执行如下各条指令时的当前PC值为0x80200000，下面指令哪些是非法指令。

```
1    jal a0, 0x800fffff
2    jal a0, 0x80300000
```

汇编器会打印如下如下错误信息

```
AS   build_src/boot_s.o
build_src/boot_s.o: in function `.L0 ':
/home/rlk/rlk/riscv_trainning/benos/src/boot.S:10:(.text.boot+0x0):
relocation truncated to fit: R_RISCV_JAL against `*UND*'
```

两条指令都超过了JAL指令的跳转范围。以PC值为0x80200000，JAL指令的跳转范围为[0x80100000 ~ 0x802ffffe]。

指令9：条件跳转指令

- RV64I指令集提供多条条件跳转指令：

指令	指令格式	说明
beq	beq rs1, rs2, label	如果rs1和rs2寄存器的值相等，则跳转到label标签处
bne	bne rs1, rs2, label	如果rs1和rs2寄存器的值不相等，则跳转到label标签处
blt	blt rs1, rs2, label	如果rs1寄存器的值小于rs2，则跳转到label标签处
bltu	bltu rs1, rs2, label	与blt指令类似，只不过rs1和rs2的值为无符号数
bgt	bgt rs1, rs2, label	如果rs1寄存器的值大于rs2，则跳转到label标签处
bgtu	bgtu rs1, rs2, label	与bgt指令类似，只不过rs1和rs2的值为无符号数
bge	bge rs1, rs2, label	如果rs1寄存器的值大于等于rs2，则跳转到label标签处
bgeu	bgeu rs1, rs2, label	与bge指令类似，只不过rs1和rs2的值为无符号数

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

跳转范围：

- offset表示label标签地址基于当前PC地址的偏移量。
- offset是13 位有符号立即数，其中offset[12:1]由指令编码的Bit[31:25]以及Bit[11:7]共同构成，offset[0]为0，它默认是2的倍数，因此它最大寻址范围是[-4KB ~ 4KB]
- 跳转到当前PC地址±4KB的范围

例子

```
main:
        addi x1, x0, 33
        addi x2, x0, 44
        bne x1, x2, .L1
.L0:
        li     a5,-1
.L1:
        mv     a0,a5
        ret
```

```
$ riscv64-linux-gnu-objdump -d my_asm

Disassembly of section .text:

0000000000000000 <main>:
   0:   02100093            li      ra,33
   4:   02c00113            li      sp,44
   8:   00209463            bne     ra,sp,10 <.L1>
   c:   fff00793            li      a5,-1

0000000000000010 <.L1>:
  10:   00078513            mv      a0,a5
  14:   00008067            ret
```

指令编码

条件跳转伪指令

伪指令	指令组合	判断条件
beqz rs, label	beq rs, x0, label	rs == 0
bnez rs, label	bne rs, x0, label	rs != 0
blez rs, label	bge x0, rs, label	rs <= 0
bgez rs, label	bge rs, x0, label	rs >= 0
bltz rs, label	blt rs, x0, label	rs < 0
bgtz rs, label	blt x0, rs, label	rs > 0
bgt rs, rt, label	blt rt, rs, label	rs > rt
ble rs, rt, label	bge rt, rs, label	rs <= rt
bgtu rs, rt, label	bltu rt, rs, label	rs > rt （无符号数比较）
bleu rs, rt, label	bleu rs, rt, label	rs <= rt （无符号数比较）

小练习5：条件跳转指令1

实验要求

在BenOS里做实验。编写一个汇编函数实现如下功能，下面是该函数的C语言伪代码。

```
unsigned long compare_and_return(unsigned long a, unsigned long b)
{
    if (a >= b)
        return 0;
    else
        return 0xffffffffffffffff;
}
```

小练习6：条件跳转指令2

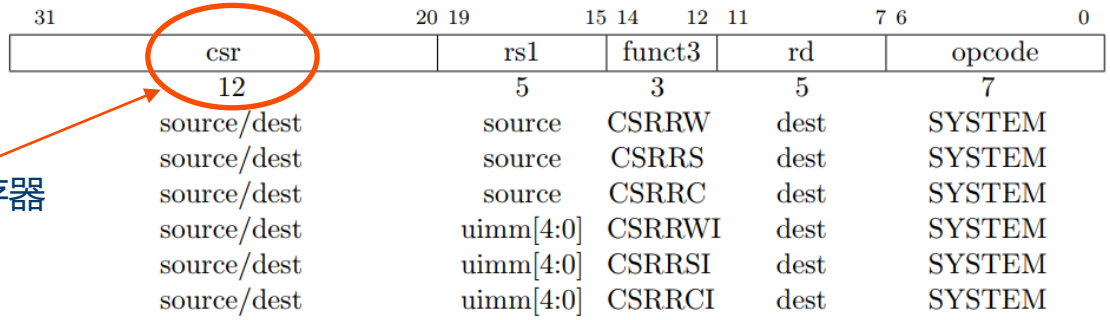
实验要求

在BenOS里做实验。请使用条件选择指令来实现如下C语言函数。

```
unsigned long sel_test(unsigned long a, unsigned long b)
{
    if (a == 0)
        return b+2;
    else
        return b-1;
}
```

指令9：CSR指令

- RV64I指令集提供一组访问CSR寄存器的指令



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

csr字段用来索引CSR寄存器

CSR 指令	指令格式	说明	
csrrw	csrrw rd，csr，rs1	原子地交换CSR和rs1寄存器的值。 读取在CSR的旧值，将其零扩展到64位，然后写入rd寄存器中，与此同时，rs1寄存器的旧值将被写入CSR中。	rd = csr csr = rs1
csrrs	csrrs rd, csr, rs1	原子地读CSR寄存器的值并且设置CSR寄存器中相应的位。 指令读取CSR寄存器的旧值，将其零扩展到64位，然后写入rd寄存器中。与此同时，把rs1寄存器的值做作为掩码，设置CSR寄存器相应的位。	rd = csr csr |= rs1
csrrc	csrrc rd, csr, rs1	原子地读CSR寄存器的值并且清CSR寄存器中相应的位。 指令读取CSR寄存器的旧值，将其零扩展到64位，然后写入rd寄存器中。与此同时，把rs1寄存器的值做作为掩码，清CSR寄存器相应的位。	rd = csr csr &=~ rs1
csrrwi	csrrwi rd，csr，uimm	作用与csrrw指令类似，区别在于使用5位无符号立即数替代rs1。	
csrrsi	csrrsi rd, csr, uimm	作用与csrrs指令类似，区别在于使用5位无符号立即数替代rs1。	
csrrci	csrrci rd, csr, uimm	作用与csrrc指令类似，区别在于使用5位无符号立即数替代rs1。	

例子

```
1    csrrw t0, sscratch, tp
2    csrrw tp, sscratch, tp
3    csrrs t0, sstatus, t1
```

第1行，交换tp和sscratch寄存器的值，即把sscratch寄存器的旧值读取t0寄存器中，与此同时，把tp寄存器的旧值写入到sscratch寄存器中。
用C语言伪代码来表示：

t0 = sscratch， sscratch = tp

第2行，把sscratch寄存器的旧值先读取tp寄存器，与此同时，把tp寄存器的旧值写入到sscratch寄存器。
用C语言伪代码来表示：

tp = sscratch， sscratch = tp

第3行，把sstatus寄存器的旧值写入到t0寄存器中，与此同时，以t1寄存器的值为掩码，设置sstatus寄存器相应的位。
用C语言伪代码来表示：

t0 = sstatus， sstatus |= t1

➢ RV64I指令集基于CSR的伪指令

伪指令	指令组合	说明
csrr rd, csr	csrrs rd, csr, x0	读取CSR寄存器的值
csrw csr, rs	csrrw x0, csr, rs	写CSR寄存器的值
csrs csr, rs	csrrs x0, csr, rs	设置CSR寄存器的字段（csr |= rs）
csrc csr, rs	csrrc x0, csr, rs	清CSR寄存器的字段（csr &=~ rs）
csrwi csr, imm	csrrwi x0, csr, imm	把立即数imm写入CSR寄存器中
csrsi csr, imm	csrrsi x0, csr, imm	设置CSR寄存器的字段（csr |= imm）
csrci csr, imm	csrrci x0, csr, imm	清CSR寄存器的字段（csr &=~ imm）

小练习7：memcpy函数实现

实验要求

在BenOS里做实验。

实现一个小的memcpy汇编函数，从0x8020000地址拷贝32字节到0x80210000地址处，并使用GDB来比较数据是否拷贝正确。

小练习8：memset函数实现

实验要求

在BenOS里做实验。

1. memset()函数的C语言原型如下。

```
void *memset(void *s, int c, size_t count)
{
        char *xs = s;

        while (count--)
                *xs++ = c;
        return s;
}
```

假设内存地址s是16字节对齐，count也是16字节对齐。请使用RISC-V汇编指令来实现这个函数。

2. 假设内存地址s以及count不是16字节对齐，请继续优化memset()函数，例如memset(0x80210004, 0x55, 102)

总结1：寻址范围

- RISC-V支持长距离寻址和短距离寻址。
  - ✓ 长距离寻址：通过AUIPC可以实现基于当前PC偏移±2GB范围的寻址，这种寻址方式叫做PC相对寻址，不过AUIPC指令只能寻址到4KB对齐的地方。
  - ✓ 短距离寻址：有些指令可以实现基于基地址短距离寻址，即寻址范围在[-2048 ~ 2047]，这个正好是4KB大小内部的寻址范围。例如ADDI指令、加载和存储指令等。
  - ✓ 长距离寻址和短距离寻址结合可以实现基于当前PC偏移±2GB范围的任意地址的寻址。

- RISC-V支持长距离跳转和短距离跳转。
  - ✓ 长跳转模式：通过AUIPC与JALR指令的结合可以实现基于当前PC偏移±2GB的范围跳转。
  - ✓ 短跳转模式：JAL指令可以实现基于当前PC偏移±1 MB范围内跳转。

总结2：立即数范围和是否需要符号扩展

- RV64I指令集中的立即数大部分都是有符号数，需要特别注意其取值范围
  - ✓ LUI和AUIPC的立即数是20位有符号数（左移12位得到32位有符号数）
  - ✓ JAL指令的offset是21位有符号数
  - ✓ JALR指令的offset是12位有符号数
  - ✓ Load和store指令的offset是12位有符号数
  - ✓ addi，xori等指令这种立即数版本的指令，它的imm是12位有符号数
  - ✓ B指令中的offset是13位有符号数

- 需要特别注意最终结果是否需要符号扩展
  - ✓ 不带u的load
  - ✓ 低32位版本的指令，例如addw等
  - ✓ 算术右移指令

陷阱：为啥ret之后就进入死循环？

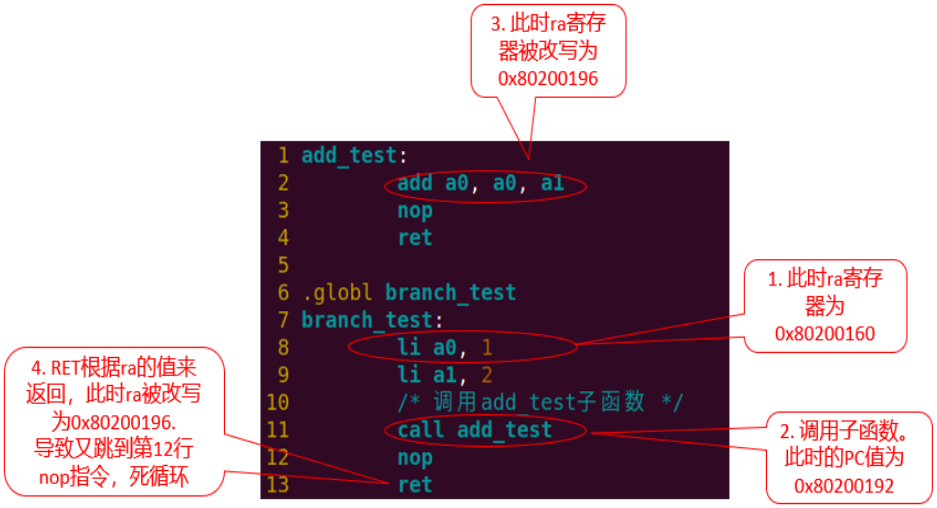
汇编文件 asm.S 如下。

<asm.S>

```
1     add_test:
2         add a0, a0, a1
3         nop
4         ret
5
6     .globl branch_test
7     branch_test:
8         li a0, 1
9         li a1, 2
10        /* 调用add_test子函数 */
11        call add_test
12        nop
13        ret
```

asm_test()函数如下。

```
1     void asm_test(void)
2     {
3         /* 调用汇编函数 */
4         branch_test();
5     }
```




```
0000000080200154 <asm_test>:
    80200154:1141                addi    sp,sp,-16
    80200156:e406                sd  ra,8(sp)
    80200158:e022                sd  s0,0(sp)
    8020015a:0800                addi    s0,sp,16
    8020015c:032000ef            jal ra,8020018e <branch_test>
    80200160:0001                nop
    80200162:60a2                ld  ra,8(sp)
    80200164:6402                ld  s0,0(sp)
    80200166:0141                addi    sp,sp,16
    80200168:8082                ret

0000000080200188 <add_test>:
    80200188:952e                add a0,a0,a1
    8020018a:0001                nop
    8020018c:8082                ret

000000008020018e <branch_test>:
    8020018e:4505                li  a0,1
    80200190:4589                li  a1,2
    80200192:ff7ff0ef            jal ra,80200188 <add_test>
=> 80200196:0001                nop
    80200198:8082                ret
```

解决办法是：在遇到嵌套调用函数时需要在父函数里把ra寄存器保存到一个临时寄存器。当父函数ret返回之前先从临时寄存器中恢复ra寄存器的值，再执行ret返回

```
1  add_test:
2          add a0, a0, a1
3          nop
4          ret
5
6  .globl branch_test
7  branch_test:
8          /*把返回地址ra保存到临时寄存器里
9          ¦ 以免调用子函数时被破坏*/
10         mv x18, ra
11         li a0, 1
12         li a1, 2
13         /* 调用add_test子函数 */
14         call add_test
15         nop
16         /*
17         ¦  恢复返回地址
18         ¦*/
19         mv ra, x18
20         ret
```

把返回地址保存到临时寄存器

```
.globl branch_test
branch_test:
        /*把返回地址ra寄存器保存到栈里*/
        addi    sp,sp,-8
        sd      ra,(sp)

        li a0, 1
        li a1, 2
        /* 调用add_test子函数 */
        call add_test
        nop

        /* 从栈中恢复ra返回地址*/
        ld      ra,(sp)
        addi    sp,sp,8
        ret
```

把返回地址保到栈里

小练习9：子函数跳转

实验要求

在BenOS里做刚才branch_test的案例。分析其进入死循环的过程

小练习10：在汇编中实现串口打印功能

在实际项目开发中，如果没有硬件仿真器例如JLINK仿真器，那么可以在汇编代码中利用下面几个常见的调试技巧。

利用LED灯实现一个跑马灯

串口打印

请在BenOS上用汇编代码实现串口的打印功能，并打印“Booting at asm”。

总结：RISC-V指令集设计

- 重新设计的指令集：简洁，干净，没有历史包袱，与微架构设计解耦
- 指令集开源，采用BSD开源协议
- 可扩展性：允许用户自定义添加指令（可变长指令编码、预留了足够扩展空间）
- 模块化设计：最小整数指令集集合 + 模块化扩展
- 基金会负责规范review和发布