Riscv 学习笔记

2024年12月21日 | 1880字 | 9分钟

目录

1. 常用指令

> **要评判一个指令集架构，不仅要看它包括了什么，而且要看它省略了什么。**

1.1. 算术运算

指令	格式	功能
`add`	`add rd, rs1, rs2`	`rd = rs1 + rs2`
`addi`	`addi rd, rs1, imm`	`rd = rs1 + imm`
`sub`	`sub rd, rs1, rs2`	`rd = rs1 - rs2`

`add`和`addi`的区别就是`add`用于将*两个寄存器*相加后存入另一个寄存器，`addi`用于将一个寄存器*加上立即数*后存入。

> **立即数**
>
> 与操作码一起放在指令代码段中的数值。直观来说就是直接写在指令后边的数。

> **为何没有`subi`指令？** >

可以用`sub rd, rs1, -imm`来实现。

1.2. 逻辑运算

指令	格式	功能
`and`	`and rd, rs1, rs2`	逻辑与
`or`	`or rd, rs1, rs2`	逻辑或
`xor`	`xor rd, rs1, rs2`	逻辑异或
`sll`	`sll rd, rs1, shamt`	逻辑左移
`srl`	`srl rd, rs1, shamt`	逻辑右移
`sra`	`sra rd, rs1, shamt`	算术右移

`srl`相较于`sra`，`srl`右移后左侧补零，而`sra`会扩展符号位。

> **为什么没有`not`指令？** >

1.3. 分支控制

指令	格式	功能
`beq`	`beq rs1, rs2, L1`	If `rs1==rs2` goto `L1`
`bne`	`bne rs1, rs2, L1`	If `rs1≠rs2` goto `L1`
`blt`	`blt rs1, rs2, L1`	If `rs1<rs2` goto `L1`
`bge`	`bge rs1, rs2, L1`	If `rs1≥rs2` goto `L1`

指令	格式	功能
`bltu`	`bltu rs1, rs2, L1`	上面两个条件的无符号版本
`bgeu`	`bgeu rs1, rs2, L1`	

Riscv的else语句通常放在if语句之前。

使用if-else语句时，判断条件是`==`，就要用`bne`；条件是`≠`，就要用`beq`：

```
IF: bne rs1, rs2, Else # ⟶ Else: rs1≠rs2
    # ↓ If: rs1==rs2
    ...
    j Then


Else:
    # code for else statement
    ...
    j Then

Then:
    # code for then statement
    ...
```

1.4. 伪指令

这些指令并不是可执行的指令，没有相应机器码，在编译时会被替换成其他指令。

以下几个伪指令就相当于是一些指令的简写，编译时会被替换为相应指令。

指令	格式	对应指令	功能
`mv`	`mv rd, rs`	`addi rs, rs, 0`	移动寄存器的值
`li`	`li rd, imm`	`addi rd, x0, imm`	加载立即数
`nop`	`nop`	`addi x0, x0, 0`	空操作

1.5. 加载与储存

有时我们需要从内存中加载数据到寄存器中，或者将寄存器中的数据保存到内存中。 我们将内存中数据的地址放在寄存器中，然后将数据读出或写入到另一个寄存器里。

指令	格式	功能
`lw`	`lw rd, imm(rs1)`	从`rs1+imm`处取一个字到`rd`
`lb`	`lb rd, imm(rs1)`	从`rs1+imm`处取一个字节到`rd`
`lbu`	`lb rd, imm(rs1)`	从`rs1+imm`处取一个字节(无符号数)到`rd`
`sw`	`lb rs1, imm(rs2)`	保存`rs1`到`rs2+imm`处
`sb`	`lb rs1, imm(rs2)`	保存`rs1`到`rs2+imm`处

例如，从栈里取一个字：

```
lw t0, 0(sp)
```

栈和寄存器`sp`的概念稍后就会讲到

这里要注意数据流动的方向：

- load: `rd`(寄存器)←`rs1+imm`(内存)
- store: `rs1`(寄存器)→`rs2+imm`(内存)

1.6. 函数调用与跳转

函数调用的六个步骤：

- 设置函数参数(`a0`到`a7`)
- 移交控制权给被调用函数(`jal`)
- 获取函数需要的本地资源
- 完成函数执行的任务
- 储存返回值并恢复寄存器的值；释放本地资源
- 移交控制权给主处理器(`ret`)

指令	格式	功能	对应指令
`j`	`j label`	跳转	`jal x0, label`
`jr`	`jr rs1`	跳转到寄存器	
`jal`	`jal rd, label`	跳转并链接	
`jalr`	`jalr rd, rs1, offset`	跳转并链接(带偏移)	
`ret`	`ret`	返回	`jr ra`

`jal` 相当于`addi ra, zero, imm; j label`，省去每次都手动设置返回地址的麻烦。 但实际上j才是伪指令，这里只是说作用上的的等价。

Tip

```
1008 addi ra, zero, 1016
1012 sum
```

等价于：

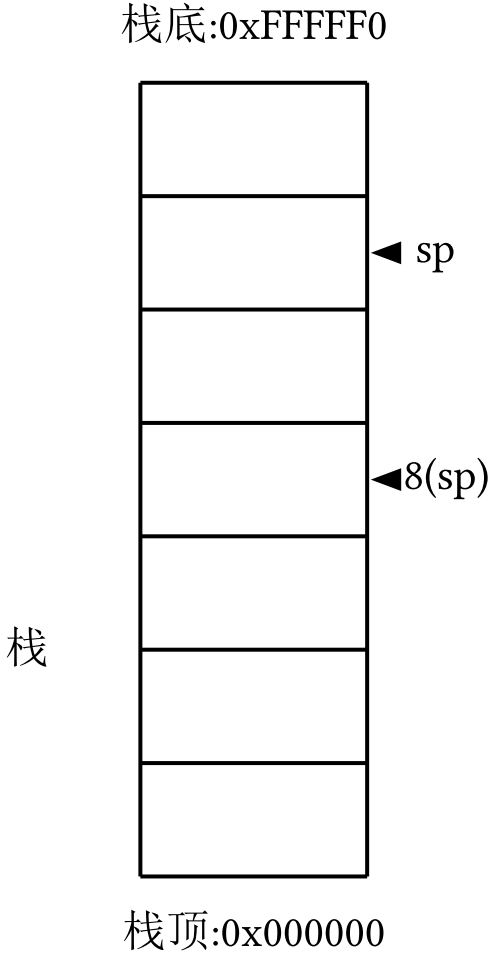
```
1008 jal sum
```

1.6.1. 栈

Tip

寄存器	功能	别名
`x0`	0	`zero`
`x1`	返回地址	`ra`
`x2`	栈指针	`sp`

栈是一种先进后出的数据结构，用来保存函数调用时的临时变量。我们使用`sp`寄存器来指向栈顶，`sp`自增或自减来移动栈顶指针。要注意的是，栈是一块连续的内存空间，*由高地址向低地址增长*，故增长栈的时候，栈指针`sp`要向低地址移动，反之增加。



例子：

```
int Leaf(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

我们把`Leaf`的四个参数分别放在`a0`，`a1`，`a2`，`a3`中，把`f`放在`s0`中。为了进行计算，我们可能还需要一个临时变量`s1`。 所以我们还需要2*4=8个字节的栈空间用来保存`s0`和`s1`的原始值，以便返回后恢复。

> **RISC-V中没有类似`push`、`pop`的指令，而是直接通过`addi`和`lw`等来实现栈操作。**

```
Leaf:

  # Step 1: prologue
  addi sp, sp, -8 # 增加8个字节的栈空间
  sw s1, 4(sp) # 保存临时变量s1
```

```
    sw s0, 0(sp) # 保存临时变量s0

    # Step 2
    add s0, a0, a1 # 计算f = (g + h)
    add s1, a2, a3 # 计算s = (i + j)
    sub s0, s0, s1 # 计算f = (g + h) - (i + j)

    # Step 3: epilogue
    jal func
    lw s0, 0(sp) # 恢复s0
    lw s1, 4(sp) # 恢复s1
    addi sp, sp, 8 # 释放栈空间
    ret
```

~~*prologue*[?] 是施法前摇，而 *epilogue*[?] 是后摇。~~

1.6.2. 嵌套函数的返回

设想这样的场景：函数A调用函数B，记录下A的栈指针；B又调用C，记录下B的栈指针。但问题是，我们只有一个栈指针寄存器`sp`，怎么保存函数调用过程中的两个栈指针呢？

寄存器	ABI名称	功能	Saver
`x0`	`$zero`	硬编码为0	-
`x1`	`ra`	返回地址	Caller
`x2`	`sp`	栈指针	Callee
`x3`	`gp`	全局指针	-
`x4`	`tp`	线程指针	-
`x5`	`t0`	临时寄存器	Caller
`x6-7`	`t1-2`	临时寄存器	Caller
`x8`	`s0/fp`	保存寄存器/帧指针	Callee
`x9`	`s1`	保存寄存器	Callee
`x10-11`	`a0-1`	参数寄存器、返回值寄存器	Caller
`x12-17`	`a2-7`	参数寄存器	Caller
`x18-27`	`s2-11`	保存寄存器	Callee

寄存器	ABI名称	功能	Saver
`x28-31`	`t3-6`	临时寄存器	Caller

Preserved(callee-saved)	NonPreserved(caller-saved)
Saved registers:`s0-s11`	Return address: `ra`
Stack pointer:`sp`	Argument registers:`a0-a7`
	Return values:`a0-a1`
	Temporary registers:`t0-t6`

Caller-saved指的是，在调用函数之前，调用者需要自己保存的寄存器，比如为了传递参数，需要覆盖参数寄存器`a0-a7`，这些寄存器的原始值就需要调用者保存，即放到栈上，以便在被调用函数返回时恢复。（实际上恢不恢复由具体情况决定，总之是调用者在管理）

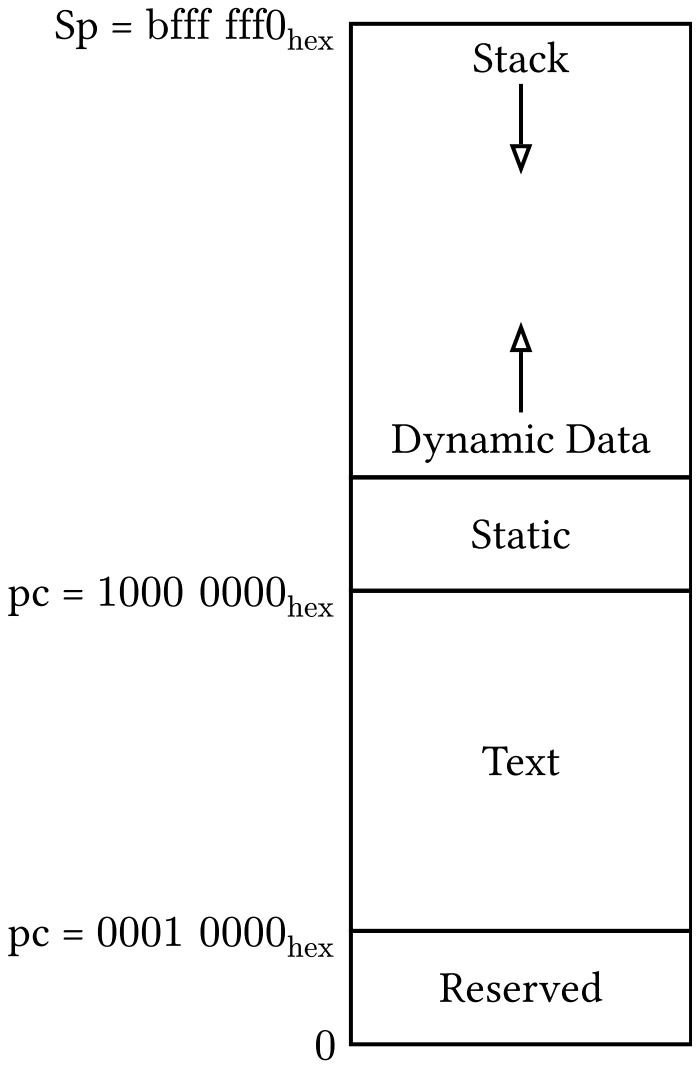
Callee-saved指的是，在调用函数时，Callee需要保存的寄存器，比如`s0-s11`，被调用函数需要提前保存原始值并在返回时恢复。

区分二者最简单的方法就是看Callee退出时需不需要恢复其值，如果需要，就属于Callee-saved；否则就属于Caller-saved。

Caller保存寄存器	Callee保存寄存器
函数调用前后可能改变	函数调用前后不变
会被Callee污染	需要Callee维护，防止污染 “谁污染，谁治理”

1.7. 程序的位置

一个简单的程序在riscv上运行时在内存里的位置如图：



其中：

- `Reserved`：保留空间，系统调用、io等就在这里
- `Text`：代码段，由`pc`指向当前代码的位置
- `Static`：静态数据(即全局变量)
- `Dynamic Data`：动态数据，`malloc`分配的空间就在这里
- `Stack`：栈。

2. 小结

有了以上知识，我们就可以用汇编来实现几乎所有的程序了，其他的无非是一些系统调用和“让我们生活更加轻松的东西”。

芜湖！