1 RISC-V 介绍

1.1 什么是机器指令?

机器指令

IS 和 ISA 指令集架构 ISA instruction set architecture 也称为处理器架构

系列机, 基本指令系统结构相同的计算机? 比如说 risc-v 系列机, Arm 系列机. 说实话都是瞎几把介绍吧, 都是随便讲讲就过了.

ISA 讲的什么几把.

属性 位宽, 指定了通用寄存器的宽度. 决定了寻址范围的大小, 数据运算能力的强弱. 需要和指令编码长度分开, 指令的编码长度是越小越好的.

原则 : 1. 简单性来自于规则性 2. 越小越快 3. 加速经常性时间. 经常使用的指令尽量短. 4. 需要良好的折衷.

性能要求 :

1 完备性: 指令丰富, 功能齐全, 使用方便

2 高效性: 空间小, 速度快

3 规整性: riscv 比较规整, 但是 x86 相反.

4 兼容性: 能够向上兼容

1.2

content

- 储存器寻址 - 操作出的类型 - 控制转移类指令 - 指令格式

大端法和小端法 字地址, 字长度为 32 位 (riscv), 字地址为 addr. 小端法: 将字的低位放在 addr. 次低位放在 addr + 1. 大端法: 将字的高位放在 addr.

对齐问题

寄存器寻址　直接使用寄存器的名字立即数寻址间接访问, 访问寄存器的值, 根据寄存器的值访问寄存器

1.3

将 c 程序编译为可执行文件的过程　见 ppt 5 页

编译器 将 c 文件编译为 s 文件, 也就是汇编文件

汇编器 将 s 文件汇编为 o 文件.

链接器

加载器

汇编语言的优缺点　见 ppt8 页

1.4 risc-v 介绍

汇编指令的格式　格式为 `op dst, src1, src2`

一个操作指令 (`op`) 有三个操作数, dst 是目标, 并且有两个源操作数寄存器 (`src1, src2`). 一个指令对应一个操作. 一行最多一条指令. C 语言之中的操作会被分解为一条或者是多条指令. 汇编语言的操作对象均是寄存器.[1]

寄存器　32 个通用寄存器, $x0$ – $x31$. 注意这里仅仅涉及 RV32I 寄存器. 算术逻辑运算所操作的数据必须直接来自于寄存器. (这里和 x86 不同) 其中 $x0$ 是一个特殊的寄存器, 其值恒为 0. RV32I 指令集通用寄存器的长度为 32 位, 而 RV64I 指令集对应的长度是 64 位.

汇编是相对原始的, 汇编语言之中没有变量的概念. 直接使用寄存器操作. 直接使用寄存器能够最大化速度. 缺点在于寄存器的数量非常有限, 需要好好规划.

内存　Save Load 操作, 实现寄存器和内存之间的读写. 这里我们再次细明一下存储单元和地址. 最小单元是一个字节 8 个 bit, 若是写为十六进制, 便是两位, 比如说 `0xFF`. 可以知道, RV32 的寻址空间的大小为 2^{32}. 一个寄存器的长度为 32, 其值可以看作是一个地址的值. 于是地址便是 `0x0000` 到 `0xFFFF`. 于是总共有 2^{32} 个地址, 可以访问 2^{32} 个不同的存储单元.[2]

[1]还可是内存, 但是在 risc-v 之中不允许这种操作, 涉及内存的操作只有 `save , load` 操作

[2]需要注意的是, 这只是理论上的值

寄存器	符	说明
x0	zero	固定值为 0
x1	ra	return address
x2	sp	stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	temporary 寄存器
x8	s0/fp	save 寄存器/帧指针 (Frame Pointer)
x9	s1	save 寄存器 (Save Register)
x10-x11	a0-a1	函数参数 / 函数返回值 (Return Value)
x12-x17	a2-a7	函数参数 (Function Argument)
x18-x27	s2-s11	save 寄存器
x28-x31	t3-t6	临时寄存器

图 1: risc-v 之中寄存器的分类

寄存器分类 此乃规定, 见图 1

add 指令 格式大概是: `add rd, rs1, rs2`. 对应的是 `rd = rs1 + rs2`. 我们可以用 `add` 实现 `mv` 语句. 我们令 rs2 = x0 就行了.

进行复合计算 如果进行 `a = (b + c) - (d - e)` 的操作. 需要引入两个临时变量.

立即数 immediate 操作名后面加上了一个 i, 那么这个操作是立即数操作. 其将第二个操作数当作是立即数. 需要知道的是, 立即数会进行一个符号扩展, 扩展为一个补码表示的. e.g. `addi x1, x1, 5`

随后, 更值得注意的是, 存在 `addi`, 但是不存在 `subi`. `f = g - 10` 实际上是 `addi x3, x4, -10`

mul 指令 (multiply) mulh 取高位指令. 详情请看 ppt

div rem 指令 (divide) 详情请看 ppt.

`sw`	取字
`sd`	取双字
`sh`	取半字
`sb`	取 byte

图 2: 为命令指定大小

逻辑运算指令 (or and xor) 值得注意的是, risc-v 之中并没有取非操作. 但是我们可以使用前面的知识, 进行代替.

位移指令 值得注意的是, 没有算术左移对应的指令在一定范围内, 也就是算术左移没有发生错误的时候, 算术左移和逻辑左移是等价的. 这一点看以前的记录.

我们复习一下, 什么时候算术位移会发生错误. 我们有这样一个判断标准: 对于左移, 如果左移之后再右移, 不能回到原本的数字的话, 那么这个左移就出错了. 于是说, 对于左移, 正数丢弃了 1, 或者是负数丢弃了 0, 就会出错.

save load 指令 我们可以将寄存器之中东西塞到内存之中. 我们通过内存地址进行内存的访问.

需要注意的是, 其他指令的操作数均是寄存器之中的数, 仅有 `save load` 指令能够对内存进行操作. 这是 risc-v 的特点之一. 你可以看出 x86 里面并不是这样的. 格式如下: `memop reg, offset(bAddrReg)` 解释见 ppt. `memop` 指的是内存相关的操作 (也就是 `s` 或者 `l`), `reg` 指的是目标寄存器, 第二个操作数是 `offset(bAddrReg)`, 一其中 `offset` 是一个偏移量; `bAddrReg` 是寄存器, 将寄存器里的值当作是地址. 整体的地址便是 `offset + bAddrReg`. [3]

为传输指令指定大小, 见图 2

传输过程之中的符号扩展 我们进行 `load` 命令的时候, 要指定大小, 比如说 `w`, `d`, `h`, `b`. 这就是说, 我们要将一个长度为 32/64/16/8 的数据, 送到 32 或者是 64 位的寄存器里面. 这个时候, 计算机会对传输的数据进行 **符号扩展**. 也就是将这一小串数据, 看作是补码, 然后扩展为 32 或者 64 位的补码.

比如说一个传入了一个字节 `0x80`, 写为二进制为 `10000000`. 也就是有符号位 1, 是一个负数, 于是扩展为 32 位的补码的时候就变为 `0xFF80`.

值得注意的是, 默认读数据的时候, 写入的时候, 会将数据扩展, 看作是有符号的若是需要写入无符号数的话, 需要加上 `u`.

e.g. 传输数据的举例. 见 ppt 实际上不是很难.

[3]在 x86 里面, 内存地址的表示更为复杂, 详情请自己找

涉及掩码的数据传输

1.4.1 imm

imm 是立即数的意思.

1.4.2 add

add 指令的表示为 add rd, rst1, rst2

rd is register of destination

The command tell computer to do the computation—`rd = rst1 + rst2`. Note that rst1 rst2 are treated as signed number.

Additionally, `addi` tells computer to do `rd = rst1 + imm`, where `rst2` is replaced by an immediate number.

1.4.3 sub

`sub, rd, rst1, rst2`

is the form of the instruction, telling that `rd = rst1 - rst2`.

It is worth noting that there is no such thing as `subi`, cause `addi` can do the same thing.

1.4.4 mul

1.4.5 div rem

1.4.6 and, or, xor

Logical operations can do thing likes 掩码. To achieve this we can use `and` and `0xFFFFFFFF`. Let us assume that the length of register is 32. A number `and 0xFFFFFFFF` is the number itself. But when it `and 0x0FFFFFFF`, the number loose it first four bits.

1.4.7 shift left (right): sl(r)

`s + l/r + a/l + [i]`

is the decompostion of an instruction, where `s` for shift, `l,r` for **left** or **right**, `a,l` for **arithmetic** or **logical**. `[i]` is optional, which stands for `imm`.

About the arithmetic shift, you check 01.pdf out.

1.4.8 shamt

Indeed, for a 64-bits data store in a register. It would be of no use to shift of 64-bits, which resulting that in `slai` or other shifting command ending with `i`, only the lowest 6-bits of immediate number are useful. Other bits are abandoned. The remaining part is called shamt.

1.4.9 arithmetic shift

There is not such thing as `sla[i]`. We already know when the shift cause ailment. Exactly when the number is starting with `10` or `01` the result of `sla` is not what we want.

However, you may check that when there is no ailment, `sla` works just like `sll`. So `sla` become less needed.

1.4.10 s, l

```
s/l r, offset(AddR)
```

where `AddR` is a register. The command tells computer to load data from address `offset + AddR` to `r`, or to save the data in `r` to address `offset + AddR`.

1.4.11 address and word and byte

A **word** in risc-v has **32** bits. There arouses an interesting question: how to load 32-bit data to a 64-bit register?

No, what I am saying is that you need to care for whether the data is unsigned type or not. You need to expand a number when it is treated as a negative number.

1.4.12 slt

slt for set less than. `slt` is an instruction to compare the value of some data.

```
slt rd, rst1, rst2
```

means that `rd = rst1 and rst2`

1.4.13 beq bne blt bge

1. b for break
2. eq for equal.

3. ne for not equal.

4. lt for less then.

5. ge for greater or equal.

Use this set of command to jump which is used to achieve if-else structure. Note that for blt and bge, there exists unsigned type of commands.

1.4.14 jump 指令

jal rd, Label

称为无条件跳转. PC+4 存贮在 rd 之中. 并且 PC 赋值为 `Label`.

`Label` 是一个写在程序行首的标签, 比如说 `Exit` 或者 `Loop` 等. 程序运行的时候此标签会翻译为一个指令的地址.

The actual operation it takes is PC = PC + `offSet`, where `offSet` is translated from `Label`.

jalr rd, offSet(AddR)

是 `jalr` 的格式. 其表示, `rd` = PC + 4. 将 PC 的值赋为 `offset + AddR`.

1.4.15 la command

la rd, Label

意思是将 `Label` 所在的指令的地址传输到 `rd` 上. 其中 `Label` 表示的是当前 PC 的值和目标指令的差值, 记为 `delta`, 长度为 32 位.

1.4.16 How to write a loop

1.4.17 Basic Block

1.4.18 Function and stack

1.5 指令的表示

对于