

## 1/4: 第二章习题:

1. CISC (复杂指令集计算机): **优势** ① 由于指令集复杂, 故可以完成复杂的任务, 进行复杂数据处理 (图形、视频处理等); ② 使编译器在编写程序时更易进行代码优化, 提高代码执行效率; ③ 寻址方式较为灵活; ④ 以微程序控制器为核心, 指令和数据存储器共享同一个物理存储空间, 性能强大。
- 劣势**: ① 指令集过于复杂, 导致芯片设计的复杂性, 不利于 VLSI (超大规模集成电路) 的实现, 测试验证成本高; ② 指令集复杂性让处理器执行指令时能耗增加, 导致性能降低, 且不利于使用先进结构提升性能; ③ 容易产生错误和漏洞, 使系统的稳定性和安全性降低。
- RISC (精简指令集计算机): **优势**: ① 指令集结构简单, 使处理器设计简便且易于实现; ② 执行指令时时间和功耗较低, 性能提高, 执行效率高; ③ 更加稳定且安全, 降低出错可能。
- 劣势**: ① 指令量少, 无法处理一些如浮点数的复杂数据操作与任务; ② 寻址方式不够灵活, 编译器优化效果相对 CISC 较低 (对编译器要求高)。
2. RISC-V 基本指令集: RV32I / RV64I / RV128I: 分别为 32 位 / 64 位 / 128 位 整数指令集
- 该指令集中包括: 加载、存储、算术、逻辑、分支、访存和跳转等基本指令 (RV32E: 32 位嵌入式指令集)
- RISC-V 标准扩展指令集: ① RV32M / RV64M: 包含乘除法和取模求余指令, 实现整数乘除法运算; ② RV32F / RV64F: 单精度浮点指令, 实现浮点运算; ③ RV32D / RV64D: 双精度浮点指令, 实现浮点运算; ④ RV32A / RV64A: 原子指令集扩展, 包含原子读、修改和写指令, 可实现各种同步操作; ⑤ RV32C / RV64C: 压缩指令, 其单指令长度为 16 位, 主要用于改善程序的大小。
4. (1) RV32I 中的 add 指令操作数为 0110011, RV64I 中的 addw 指令操作数为 0111011, 指令操作数不同。
- RV32I 中的 add 指令操作数为 0110011, RV64I 中的 add 指令操作数为 0110011, 指令操作数相同。
- 分析: ① RV32I 中的 add 指令和 RV64I 指令虽然均为 32 位整数加法指令, 但在 RV64I 中实际是将两个寄存器操作数相加后结果截断为 32 位, 并写入寄存器, 忽略并丢弃溢出的结果, 故在加法后需多进行一步截断操作, 故操作数不同。
- ② RV32I 和 RV64I 中的 add 指令是具有继承关系的, 其执行的操作均为操作数的相加, 操作相同, 故操作数相同。
- (2) addw 和 addiw 指令目标寄存器中存放的 32 位运算结果需要进行额外的符号扩展才能用于后续 64 位的计算, 因为虽然结果中有符号扩展, 但 64 位中在运算时仍需保持符号不变, 故需将结果符号扩展至 64 位, 进行后续计算。



5. RISC-V的I标准指令集中的HINT指令空间: HINT指令空间可被用定义一些轻量级的指令, 这些指令主要用于向微架构传递一些处理器性能的提示信息。且如同NOP指令, 该类指令除推动PC及任何可用性计数器外, 并不会改变体系结构中可见的状态, 在大多数情况下可被忽略。HINT指令空间故主要用于指令流水线的流程控制和性能优化, 改善流水线性能, 优化程序执行顺序, 提高程序的吞吐量。

6.  $\text{div } a_2, a_0, a_1$  是进行  $a_0 \div a_1$  的除法运算, 并将商存入  $a_2$  寄存器中, 故  $a_2$  寄存器中值为 3。

$\text{rem } a_3, a_0, a_1$  是进行  $a_0$  除以  $a_1$  后的取余操作, 并将余数存入  $a_3$  寄存器中, 故  $a_3$  寄存器中值为 1。

RISC-V的M标准指令集中:  $\text{div}$  指令进行有符号整数除法运算,  $\text{divu}$  进行无符号整数的除法运算。

若被除数与除数符号相同, 结果为正; 若被除数与除数符号相反, 结果为负 (\* 当出现  $\pm 0$  时,  $\text{DIVU}$  结果为  $2^{XLEN}-1$ ,  $\text{DIV}$  结果为  $-1$ )

$\text{rem}$  指令进行有符号整数的除法后取余运算,  $\text{remu}$  进行无符号整数的除法后取余运算。

$\text{rem}$  指令运算后的结果符号与被除数的符号相同 (\* 当出现  $\pm 0$  时,  $\text{REM}$  和  $\text{REMU}$  的结果均为  $\pm 0$ )。

11. (1)  $\text{jial ra, 0x88}$ : 使用偏移量寻址: 将下一条指令的地址 ( $\text{PC}+4$ ) 存入  $\text{rd}$  寄存器, 并将  $\text{PC}$  地址加上  $0x88$  进行跳转。

(2)  $\text{jialr x0, ra, 0}$ : 使用立即数寻址: 在  $x_0$  中存入  $\text{PC}+4$  的值, 并将目标地址寄存器  $\text{ra}$  加上立即数  $0$ , 进行跳转。

(3)  $\text{addi a_0, a_1, 4}$ : 使用立即数寻址: 将  $a_1$  寄存器与立即数  $4$  相加, 并将结果存储在寄存器  $a_0$  中。

(4)  $\text{mul a_0, a_1, a_2}$ : 使用寄存器直接寻址: 将  $a_1$  寄存器和  $a_2$  寄存器中存储值相乘的结果存储在寄存器  $a_0$  中。

(5)  $\text{ld a_4, 16(sp)}$  使用偏移量寻址: 将栈指针  $\text{sp}$  指向的地址加上立即数  $16$  后地址处数据读取并存储在寄存器  $a_4$  中。