

## Homework 2

PB22051080 王珏


2024 年 3 月 17 日

# 目录

<b>1 关于 RISC-V 指令</b>	<b>1</b>
1.1 回顾指令类型 . . . . .	1
1.2 具体指令的分析 . . . . .	2
<b>2 读懂 RISC-V 程序代码</b>	<b>2</b>
2.1 例题 . . . . .	2
2.1.1 $x6$ 寄存器初始化为 10, $x5$ 的最终值是? . . . . .	2
2.1.2 编写等效 C 代码 . . . . .	3
2.1.3 理解循环次数 . . . . .	3
2.1.4 替换指令后再写等效 C 代码 . . . . .	3
<b>3 计算机双字地址存储——大端对齐和小端对齐</b>	<b>4</b>
<b>4 当各种指令的时钟周期数不再看作相等时, 求 CPI</b>	<b>5</b>
4.1 平均 CPI . . . . .	5
4.2 性能优化后 . . . . .	5
4.3 再次优化 . . . . .	6
<b>5 Thinking</b>	<b>6</b>
5.1 CPU 的 ISA 要定义哪些内容? . . . . .	6
5.2 <code>main()</code> 与 <code>swap()</code> 状态保存异同 . . . . .	7
5.3 过程调用时程序的内存数据是否需要保存? . . . . .	8
5.4 Windows 系统中可执行程序的格式及其简述 (不是指文件后缀名)	8

## 1 关于 RISC-V 指令

```
addi x30, x10, 8
addi x31, x10, 0
sd   x31, 0(x30)
ld   x30, 0(x30)
add  x5, x30, x31
```



- 2.9** [20] <2.2, 2.5> 对于练习 2.8 中的每条 RISC-V 指令，写出操作码 (op)、源寄存器 (rs1) 和目标寄存器 (rd) 字段的值。对于 I 型指令，写出立即数字段的值，对于 R 型指令，写出第二个源寄存器 (rs2) 的值。对于非 U 型和 UJ 型指令，写出 funct3 字段，对于 R 型和 S 型指令，写出 funct7 字段。

图 1: T9——基本指令类型

### 1.1 回顾指令类型

- I 型
- R 型
- U 型和 UJ 型
- S 型

## 1.2 具体指令的分析

表 1: 相关答案

	type	op	rs1	rd	immediate	rs2	funct7	funct3
addi	I	0010011	01010	11110	0000_0000_1000	no		000
addi	I	0010011	01010	11111	0000_0000_0000	no		000
sd	S	0100011	11110	11111	0000_0000_0000	no	?	011
ld	I	0000011	11110	11110	0000_0000_0000	no		011
add	R	0110011	11110	00101	no	11111	0000000	000

## 2 读懂 RISC-V 程序代码

### 2.1 例题

**2.24** 考虑以下 RISC-V 循环：

```

LOOP: beq x6, x0, DONE
      addi x6, x6, -1
      addi x5, x5, 2
      jal x0, LOOP
DONE:

```

- 2.24.1** [5] <2.7> 假设寄存器 x6 初始化为 10。寄存器 x5 的最终值是多少（假设 x5 初始值为零）？
- 2.24.2** [5] <2.7> 对于上面的循环，编写等效的 C 代码。假设寄存器 x5 和 x6 分别是整型 acc 和 i。
- 2.24.3** [5] <2.7> 对于上面用 RISC-V 汇编语言编写的循环，假设寄存器 x6 初始化为 N。总共执行了多少条 RISC-V 指令？
- 2.24.4** [5] <2.7> 对于上面用 RISC-V 汇编语言编写的循环，将指令“beq x6, x0, DONE”替换为“blt x6, x0, DONE”指令并写出等效的 C 代码。

图 2: RISC-V 循环

#### 2.1.1 x6 寄存器初始化为 10, x5 的最终值是？

当 x6 递减到 0，循环结束，执行十次，因此 x5 的最终值为 20。

### 2.1.2 编写等效 C 代码

```

int main() {
    int x5 = 0; // Initialize x5
    int x6 = 10; // Initialize x6
    while (x6 != 0) {
        x6 = x6 - 1;
        x5 = x5 + 2;
    } // Done with loop
    return 0;
}

```

### 2.1.3 理解循环次数

$x6$  初始化为  $N$ ，则总共执行了多少条指令？

$$num = N * 4 + 1$$

### 2.1.4 替换指令后再写等效 C 代码

将 `beq` 改为 `blt`

```

int main() {
    int x5 = 0; // Initialize x5
    int x6 = 10; // Initialize x6

    while (x6 > 0) {
        x6 = x6 - 1;
        x5 = x5 + 2;
    } // Done with loop
}

```

```
    return 0;
}
```

### 3 计算机双字地址存储——大端对齐和小端对齐

2.35 考虑以下代码：

```
lb x6, 0(x7)
sd x6, 8(x7)
```

假设寄存器 x7 包含地址 0x10000000，且地址中的数据是 0x1122334455667788。

- 2.35.1 [ 5 ] <2.3, 2.9> 在大端对齐的机器上 0x10000008 中存储的是什么值？
- 2.35.2 [ 5 ] <2.3, 2.9> 在小端对齐的机器上 0x10000008 中存储的是什么值？

图 3: T35——大端对齐和小端对齐

$x$  是 64 位寄存器； $0 \times 1122334455667788$  是数据，约  $1.2 \times 10^{18}$ ，存储方式和小题有关。

我们假设从 0x1000 作为小端起始：

表 2: 大端对齐和小端对齐的存储结果

地址	0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006
大端对齐	0x11	0x22	0x33	0x44	0x55	0x66	0x77
小端对齐	0x88	0x77	0x66	0x55	0x44	0x33	0x22

## 4 当各种指令的时钟周期数不再看作相等时, 求 CPI

- 2.40** 假设对于一个给定程序, 70% 的执行指令是算术指令, 10% 是加载 / 存储指令, 20% 是分支指令。
- 2.40.1** [5] <1.6, 2.13> 假设算术指令需要 2 周期, 加载 / 存储指令需要 6 周期, 而一条分支指令需要 3 周期, 求平均 CPI。
- 2.40.2** [5] <1.6, 2.13> 对于性能提高 25%, 如果加载 / 存储和分支指令都没有改进, 一条算术指令平均需要多少周期?
- 2.40.3** [5] <1.6, 2.13> 对于性能提高 50%, 如果加载 / 存储和分支指令都没有改进, 一条算术指令平均需要多少周期?

图 4: 平均 CPI

### 4.1 平均 CPI

假设指令数  $N = 100$

$$CPI = \frac{70 \times 2 + 10 \times 6 + 20 \times 3}{100} = 2.6$$

### 4.2 性能优化后

设计算指令需要周期数为  $x$

$$\begin{aligned} CPI_1 &= CPI \times 0.75 \\ &= 1.95 \\ &= \frac{70 \times x + 10 \times 6 + 20 \times 3}{100} \end{aligned}$$

解出

$$x = 1.07$$

### 4.3 再次优化

设计算指令需要周期数为 $x_1$

$$\begin{aligned} CPI_2 &= CPI \times 0.5 \\ &= 1.3 \\ &= \frac{70 \times x_1 + 10 \times 6 + 20 \times 3}{100} \end{aligned}$$

解出

$$x_1 \approx 0.143$$

## 5 Thinking

### 5.1 CPU 的 ISA 要定义哪些内容？

CPU 的 ISA (Instruction Set Architecture, **指令集架构**) 定义了 CPU 支持的指令集合、指令的格式、编码方式、寄存器的数量和功能、内存地址空间的组织结构等关键方面。ISA 定义了软件和硬件之间的接口。

通常，一个完整的 ISA 包含以下内容：

1. **指令集合 (Instruction Set)**: 定义了 CPU 支持的所有指令，例如算术运算指令、逻辑运算指令、数据传输指令等。
2. **指令格式 (Instruction Format)**: 指定了每条指令在存储器中的表示格式，包括操作码、寄存器地址、立即数等字段的排列和长度。
3. **寄存器 (Registers)**: 定义了 CPU 内部的寄存器，包括通用目的寄存器、特殊用途寄存器等。ISA 规定了寄存器的数量、位宽、用途和访问方式。
4. **内存模型 (Memory Model)**: 描述了 CPU 如何访问内存，包括地址空间的组织结构、内存访问方式（例如字节访问、字访问）、对齐要求等。



5. **异常和中断 (Exception and Interrupt)**: 规定了 CPU 如何处理异常 (例如除零、非法指令) 和中断 (例如时钟中断、外部中断)。
6. **指令执行流程 (Execution Flow)**: 定义了指令的执行流程、操作数的获取方式、指令的执行顺序等。
7. **特权级别 (Privilege Levels)**: 定义了不同特权级别下 CPU 的行为和权限, 通常包括用户态和内核态。
8. **扩展和兼容性 (Extensions and Compatibility)**: 规定了 ISA 的扩展机制和兼容性规则, 以支持后续的扩展和版本升级。
9. **调试和性能监控 (Debugging and Performance Monitoring)**: 定义了 CPU 的调试接口、性能计数器和监控功能, 以支持软件调试和性能分析。

## 5.2 main() 与 swap() 状态保存异同

- **相同点:**

- 在函数调用时, 都会将当前函数的状态保存到栈上, 以便在函数执行完毕后能够恢复到调用前的状态。
- 保存的状态包括函数的局部变量、参数、返回地址等。

- **不同点:**

- 当 `main()` 函数被调用时, 通常会涉及更多的程序状态, 因为它是程序的入口点, 需要保存整个程序的状态, 包括函数调用栈、全局变量、参数等。这样, 程序执行完 `main()` 函数后, 可以继续执行其他的函数或代码。
- 而 `swap()` 函数通常只需要保存局部变量、参数和返回地址等最基本的状态信息。这是因为 `swap()` 函数的功能不需要涉及整个程序的状态。

### 5.3 过程调用时程序的内存数据是否需要保存？

在 search 的过程中发现可能需要研读 csapp？遂不答，先看书。

### 5.4 Windows 系统中可执行程序的格式及其简述（不是指文件后缀名）