

计算机组成与设计 (RISC-V) 版 (武汉大学)

武汉大学李俊亨

2025 年 12 月 30 日

编者序

本文将辅以大量例子与文字说明，对公式作进一步解释。示例与解释部分会以不同于正文的字体呈现，方便读者快速识别与区分。为了帮助读者对计算机系统有原理上，底层上的理解，本文会对书中内容的顺序做出适当改变，并对内容做出适当扩充和改变。本文同样也会从硬件的角度解释一些计算机组成是如何实现的，以辅助记忆，这些内容将会以下划线的形式呈现，以便读者区分。

排版约定示例如下：

正文 这是正文内容。

例子 例：给出具体算例、类比或代码片段时，用楷体呈现。

解释 解释：对公式或概念背后的“为什么”作进一步说明时，用黑体呈现。

硬件 硬件视角：补充硬件实现细节或记忆要点时，用下划线标出。

目录

编者序	1
1 计算机性能	4
1.1 如何衡量计算机性能	4
1.2 影响计算机性能的因素	4
1.3 衡量计算机改进程度的指标	4
1.4 其他衡量计算机性能的指标	5
2 汇编语言	6
2.1 操作数	6
2.2 指令结构	6
2.3 过程	7
2.4 其他指令的寻址	9
2.5 原子指令	9
3 计算机的数据存储与运算	10
3.1 整数的表示	10
3.2 整数的算术运算	11
3.3 浮点数的表示	13
3.4 浮点数的算术运算	14
4 处理器的单周期与流水线	16
4.1 单周期处理器	16
4.2 流水线控制	18
5 层次化存储——万物皆缓存	22
5.1 存储器层次结构概述	22
5.2 高速缓存	23
5.3 纠错码	25
5.4 虚拟内存	26
5.5 TLB	29
6 计算机并行	30
6.1 引言	30
6.2 并行处理程序的困难	30
6.3 SISD、MIMD、SIMD、SPMD 与向量处理	31
6.4 硬件多线程	33

目录	3
6.5 多核与共享内存多处理器	33
6.6 GPU 简介	34
6.7 多处理器基准测试与性能模型	35
6.8 谬误与陷阱	36
A 记忆指令结构	37
B 链接与重定向	38
B.1 地址重定位	38
B.2 PC 相对寻址的重定向	39
C 其他复杂的算术运算	39
C.1 快速乘法算法: Booth 算法	39
C.2 除法算法	40

1 计算机性能

1.1 如何衡量计算机性能

作为设计者，首要的任务便是判断我们设计出的计算机的性能。

我们可以将计算机性能比作飞机：此时飞机的速度就是 CPU 时钟周期时间 T ，飞机的行程距离就是每条指令所需的周期数 CPI (Cycles Per Instruction，每指令周期数)。

最基础的，我们可以用 CPU 时间来衡量计算机的性能。CPU 时间可以表示为：

$$\text{CPU 时间} = \text{指令数} \times \text{CPI} \times T \quad (1)$$

考虑到时钟周期时间 T 与时钟频率 f 互为倒数，我们也可以将 CPU 时间表示为：

$$\text{CPU 时间} = \frac{\text{指令数} \times \text{CPI}}{f} \quad (2)$$

其中 CPI 可以进一步表示为：

$$\text{CPI} = \frac{\text{时钟周期数}}{\text{指令数}} \quad (3)$$

尽管降低 CPU 时间是我们的最终目标，但我们要清楚，降低 CPU 时间的方式有很多种。我们可以通过减少指令数、降低 CPI 或者提高时钟频率来达到目的。具体来说，我们可以通过以下方式来优化计算机性能：

1.2 影响计算机性能的因素

硬件或软件组件	影响什么？	如何影响？
算法	指令数量、CPI	决定源程序指令数和处理器指令数；影响 CPI（如使用更多除法会提高 CPI）
编程语言	指令数量、CPI	语言特性影响指令数和 CPI（如 Java 的间接调用会使用更高 CPI 的指令）
编译器	指令数量、CPI	编译器效率影响源语言到机器指令的转换
指令集架构	指令数量、时钟周期、CPI	影响性能的所有三个方面

表 1: 影响计算机性能的因素

1.3 衡量计算机改进程度的指标

Amdahl 定律用于计算系统改进后的整体性能提升。当只改进系统的一部分时，改进后的执行时间为：

$$T_{\text{新}} = T_{\text{旧}} \times \left[(1 - F_e) + \frac{F_e}{S_e} \right] \quad (4)$$

整体加速比为：

$$\text{加速比} = \frac{T_{\text{旧}}}{T_{\text{新}}} = \frac{1}{(1 - F_e) + \frac{F_e}{S_e}} \quad (5)$$

其中 F_e 是可改进部分在原系统中所占的比例， S_e 是该部分的改进倍数。

若程序中 40% 的时间用于浮点运算，现将浮点运算速度提升 5 倍，则整体加速比为 $\frac{1}{0.6 + 0.4/5} = 1.47$ 。

上述公式中的 $1 - F_e$ 项理解为未改进部分时间占比不变，而 $\frac{F_e}{S_e}$ 项表示改进部分时间因加速而减少。

1.4 其他衡量计算机性能的指标

MIPS (Million Instructions Per Second, 每秒百万指令数) 是一种性能度量标准：

$$\text{MIPS} = \frac{\text{指令数}}{\text{执行时间} \times 10^6} = \frac{f}{\text{CPI} \times 10^6} \quad (6)$$

注意：MIPS 作为性能指标存在局限性，因为不同指令集架构的指令复杂度不同，单纯比较 MIPS 值可能产生误导。

在这里向读者介绍一些单位前缀和后缀的定义：

当我们谈到前缀 K 的时候，他代指的是数学层次的 10^3 。这是因为 K 是 kilo 的缩写，同时运用这种前缀的有 Kilogram (kg), Kilometer (km)，这些与其基准单位都相差了 1000 倍的关系。同时 M 代表 Mega，代指 10^6 ，G 代表 Giga，代指 10^9 ，T 代表 Tera，代指 10^{12} 。

但当我们使用 Ki (在计算机存储中常常见到这种表达前缀) 时，他指代的则是 2^{10} 。这是因为 Ki 是 Kibi 的缩写，同时 Mi 表示 Mebi，代指 2^{20} ，Gi 表示 Gibi，代指 2^{30} ，Ti 表示 Tebi，代指 2^{40} 。

同时需要向读者说明的是，在计算机存储中，1 Byte (字节) = 8 bits (位)。因此我们用大写的 B 来表示字节，用小写的 b 来表示位。

例如，1 KB = 10^3 Bytes = 1000 Bytes，而 1 Kib = 2^{10} bits = 1024 bits = 128 Bytes。我们将这种符号准确的应用在了后文中，请读者注意辨析这些单位上的区别。

2 汇编语言

本节我们来主要探讨 RISC-V 架构处理器提供的汇编语言指令集。汇编语言与高级编程语言类似，都提供了一些关键字（指令）和操作数（寄存器）来让程序员能够控制计算机的行为。

2.1 操作数

对 RISC-V 来说，总共有三种操作数类型：寄存器、立即数和内存地址。

寄存器 RISC-V 总共规定了 32 个寄存器 (x_0-x_{31})（在后文中，你会看到一些别名，不过这些别名指向的寄存器也属于这 32 个之中），每个寄存器均为 32 位宽。在后文中，我们使用 $R[0]$ 到 $R[31]$ 来表示这 32 个寄存器。

立即数 立即数是指直接在指令中给出的常数值。RISC-V 指令中允许使用的立即数通常为 12 位或 20 位，具体取决于指令类型。

内存地址 内存地址用于指示数据在内存中的位置。RISC-V 使用字节寻址方式，每个地址对应一个字节。在后文中，我们使用 $Memory[i]$ 来表示内存中地址为 i 的字节。

2.2 指令结构

RISC-V 指令集采用固定长度的 32 位指令格式。根据指令的功能和操作数类型，RISC-V 指令可以分为以下几种类型：

R 型指令 用于寄存器之间的算术和逻辑运算。

I 型指令 用于立即数运算、加载数据和某些分支指令。

S 型指令 用于存储数据到内存。

B 型指令 用于条件分支。（也称 SB 型指令）

U 型指令 用于加载高位立即数。

J 型指令 用于无条件跳转。（也称 UJ 型指令）

每种结构的指令都有特定的字段划分，用于表示操作码、寄存器编号、立即数等信息。有关这部分内容我们将在后续的章节中详细探讨。在本章中，我们只需要知道每种指令的功能即可。如果你需要记忆指令结构的帮助，参见附录 A。

B 型指令之所以也成为 SB 型指令，是因为两者的指令结构类似，但立即数的编码方式有所不同。J 型指令亦然。（见章节 4.1.1）

2.3 过程

2.3.1 概念

计算机需要处理复杂的任务，为此，我们的指令架构需要实现过程。过程（Procedure）是一个函数，根据传入的参数给出既定的结果（广义上，我们将 `void` 也理解成一种返回）。

RISC-V 只支持了程序员使用 32 个寄存器（除去 PC 寄存器）来处理他们的数据。但，当我们反复调用过程时，32 个寄存器显然不够用。为此，我们需要新的数据结构——栈，来满足程序员对更多数据存储的需求。

栈是一种后进先出（LIFO, Last In First Out）的数据结构。RISC-V 通过寄存器 `x2`（也称 `sp`, stack pointer, 栈指针）来管理栈。栈通常位于内存的**高地址**部分，并向**低地址**方向增长。

每当我们进入一个过程的时候，程序需要向栈申请一些内存地址来存储更多的数据。每当我们退出一个过程的时候，程序则要释放使用的这些内存地址。

例：当我们调用一个过程后，这个过程会执行类似 `addi sp, sp, -24` 的操作，这个操作便是过程向栈申请了 24 字节的空间。当过程结束时，程序会执行 `addi sp, sp, 24` 来释放这 24 字节的空间。

2.3.2 寄存器的分类与调用约定

为了扩充寄存器的使用，我们需要在栈上保存一些寄存器的值。但是过程的调用者来保存寄存器值还是过程（被调用者）来保存这些值都是可行的。为了保证代码能够正确运行，人们对寄存器的种类进行了划分，并规定了调用约定（Calling Convention）。

RISC-V 寄存器的具体分类如下表所示：

寄存器	ABI 名称	保存方式	用途
<code>x0</code>	<code>zero</code>	—	—
<code>x1</code>	<code>ra</code>	调用者保存	返回地址（Return Address）
<code>x2</code>	<code>sp</code>	被调用者保存	栈指针（Stack Pointer）
<code>x3</code>	<code>gp</code>	—	全局指针（Global Pointer）
<code>x4</code>	<code>tp</code>	—	线程指针（Thread Pointer）
<code>x5-x7</code>	<code>t0-t2</code>	调用者保存	临时寄存器（Temporary）
<code>x8-x9</code>	<code>s0-s1</code>	被调用者保存	保存寄存器（Saved）， <code>s0</code> 也称 <code>fp</code> （Frame Pointer, 帧指针）
<code>x10-x17</code>	<code>a0-a7</code>	调用者保存	函数参数/返回值（Argument）
<code>x18-x27</code>	<code>s2-s11</code>	被调用者保存	保存寄存器（Saved）
<code>x28-x31</code>	<code>t3-t6</code>	调用者保存	临时寄存器（Temporary）

表 2: RISC-V 寄存器分类与调用约定

例：假设函数 `main`（调用者）调用函数 `func`（被调用者）。如果 `main` 在寄存器 `t0`（调用者保

存) 中存储了重要数据, 它必须在调用 `func` 之前将 `t0` 保存到栈中。而如果 `func` 需要使用寄存器 `s0` (被调用者保存), 它必须在使用前保存 `s0` 的值, 并在返回前恢复。

再例: 考虑寄存器 `t0`, 因为这是个调用者寄存器, 因此过程可以放心的使用 `t0` 这个寄存器, 因为在过程结束后, 调用者会负责恢复 `t0` 的值; 而考虑寄存器 `s0`, 因为这是个被调用者寄存器, 因此过程在使用 `s0` 之前必须先将 `s0` 的值保存到栈中, 并在过程结束前恢复 `s0` 的值。因为调用者会假设该寄存器中的内容不会变化。

寄存器 `x0` 总是保存数值 0, 任何对 `x0` 的写操作都会被忽略。

一般过程的结果会被保存在 `a0` 寄存器中, 而 `a1-a7` 用以传递可能的 7 个参数, 超过 8 个参数的函数将会将超出部分的参数存储在栈上的帧中。帧 (Frame) 是每个函数在栈上分配的内存区域。前例中, `main` 和 `func` 各自拥有自己的帧, 且 `main` 的栈帧将位于 `func` 的上方。

2.3.3 分支与跳转

为了实现过程调用与返回, RISC-V 提供了分支和跳转指令。

RISC-V 的跳转指令主要有三类: B 型指令 (条件分支)、`jal` (无条件跳转) 和 `jalr` (寄存器跳转)。这些指令采用不同的寻址方式:

B 型指令 B 型指令采用 **PC 相对寻址 (PC-relative Addressing)**。跳转目标地址的计算方式为:

$$\text{目标地址} = \text{PC} + \text{imm} \times 2 \quad (7)$$

例: 如果当前 PC 为 0x1000, 而我们需要跳转到 0x1010, 则我们需要 `imm` = 0x08 = 0b1000

jal 指令 `jal` 指令同样采用 **PC 相对寻址**。其格式为 `jal rd, imm`, 功能为:

1. 将返回地址 (`PC + 4`) 保存到寄存器 `rd` 中
2. 跳转到 `PC + imm` 处

例如, `jal ra, func` 会将返回地址保存在 `ra` 寄存器中, 然后跳转到 `func`。这是函数调用的典型用法。

jalr 指令 `jalr` 指令采用 **寄存器间接寻址 (Register Indirect Addressing)**。其格式为 `jalr rd, rs1, imm`, 功能为:

1. 将返回地址 (`PC + 4`) 保存到寄存器 `rd` 中
2. 跳转到 `rs1 + imm` 处 (将结果的最低位设为 0 以保证对齐)

`jalr` 的立即数为 12 位有符号数。由于跳转地址由寄存器提供, `jalr` 可以跳转到任意地址, 实现函数指针、虚函数调用等功能。

例如，`jalr zero, ra, 0`（常写作`jalr zero, 0(ra)`）表示跳转到`ra`寄存器中保存的地址，且不保存返回地址（因为保存到了`zero`寄存器）。这是函数返回的典型用法，常简写为`ret`。

之所以有乘 2 是因为 RISC-V 中所有指令都是 2 字节对齐的，因此最后一位总是 0，为了节省编码空间，立即数中省略了这一位。

2.4 其他指令的寻址

并非只有跳转指令需要对内存寻址，存储和加载指令和一些计算指令同样需要对内存进行寻址。RISC-V 支持四种寻址方式：

- 立即数寻址 (Immediate Addressing)
- 寄存器间接寻址 (Register Indirect Addressing)
- 基址加偏移寻址 (Base plus Offset Addressing)
- PC 相对寻址 (PC-relative Addressing)

除去上面提到的 PC 相对寻址与寄存器间接寻址，还有以下两种寻址方式：

立即数寻址 立即数寻址直接使用指令中的立即数作为操作数，无需访问内存或寄存器。

例如，`addi x1, x2, 10` 表示 $x1 = x2 + 10$ ，其中 10 就是立即数，直接编码在指令中。

基址加偏移寻址 基址加偏移寻址通过寄存器值加上立即数偏移来计算内存地址，常用于访问数组和结构体。

例如，`lw x1, 8(x2)` 表示从地址 $x2 + 8$ 处加载一个字到`x1`。如果`x2` 存储数组首地址，则可以访问数组的第 2 个元素（每个元素 4 字节）。

另外，程序的编译过程中需要进行链接，如果读者对此感兴趣，参见附录 B。

2.5 原子指令

RISC-V 提供了原子内存操作指令，用于多处理器环境下的同步操作。主要包括`lr` 和`sc` 两类指令。

`lr` 指令从内存加载数据并在该地址上设置保留标记；`sc` 指令仅在保留标记仍然有效时才将数据写入内存，否则写入失败。这种机制确保了在并发访问时数据的一致性。

例如，`lr.w x1, (x2)` 从地址`x2` 加载一个字到`x1` 并设置保留。随后`sc.w x3, x4, (x2)` 尝试将`x4` 存储到地址`x2`，如果保留仍有效则成功 ($x3 = 0$)，否则失败 ($x3 = 1$)。这常用于实现自旋锁和原子操作。

3 计算机的数据存储与运算

在本章中，我们将深入探讨计算机如何表示和处理数据。理解数据的表示方式是理解计算机运算的基础，这将帮助我们更好地理解处理器内部 ALU 的工作原理。

3.1 整数的表示

计算机内部使用二进制来表示所有数据。对于整数，我们需要考虑两种情况：无符号整数和有符号整数。

3.1.1 无符号整数

无符号整数 (Unsigned Integer) 是最简单的整数表示方式，它只能表示非负整数。一个无符号整数就是数学意义上的二进制数。

n 位无符号整数的表示范围为 0 到 $2^n - 1$ 。

3.1.2 有符号整数的原码表示

计算机是以二进制传递信息的，因此没有专门的符号来供计算机理解。为了表示负数，我们需要引入符号的概念。最直观的方法是原码 (Sign-Magnitude) 表示法：使用最高位作为符号位 (0 表示正数，1 表示负数)，其余位表示数值的绝对值。

对于 n 位原码表示：

$$X_{\text{原}} = \begin{cases} X & \text{若 } X \geq 0 \\ 2^{n-1} + |X| & \text{若 } X < 0 \end{cases} \quad (8)$$

例：8 位原码表示：

- +5 的原码为 00000101 (符号位 0, 数值位 5)
- -5 的原码为 10000101 (符号位 1, 数值位 5)

注意原码存在一个问题：零有两种表示方式——00000000 (+0) 和 10000000 (-0)。

原码在现代计算机中很少使用，原因有二：首先，+0 和 -0 的存在使得零的比较变得复杂；其次，原码的加减法运算需要根据符号位进行不同的处理，这增加了硬件实现的复杂度。例如，计算 $5 + (-3)$ 时，需要先判断两个数的符号，再决定是做加法还是减法。

3.1.3 有符号整数的补码表示

补码是现代计算机中最常用的有符号整数表示方式。

补码的另一种等价定义是：

$$X_{\text{补}} = \begin{cases} X & \text{若 } X \geq 0 \\ 2^n + X & \text{若 } X < 0 \end{cases} \quad (9)$$

或者说，第一位的权重是 -2^n ，这在二进制数中则是 2^n 。

例如：一个八位补码 $1000\ 1111 = -2^8 + 15 = -241$

n 位补码的表示范围为 -2^{n-1} 到 $2^{n-1} - 1$ 。

例：特殊值 -128 的 8 位补码表示为 10000000 。这是 8 位补码能表示的最小值，它没有对应的正数（因为 $+128$ 超出了 8 位补码的表示范围）。

需要注意的是，对补码取相反数，需要对所有位取反加一。

而对原码和补码进行相互转换的时候，则需要根据符号位判断：**如果是正数，则原码即补码；如果是负数，应该将除去最高位（符号位）的其他位都取反加一。**

需要特别的注意是，这里面的原码与补码转换可能会溢出至更多的位数，只要不做截断，那么上述的转换规则仍然成立

例如：当我们对 $1000\ 0000$ 这个补码转换成原码的过程中，先对非符号位 $000\ 0000$ 取反得到 $111\ 1111$ ，再加一得到 $1000\ 0000$ ，这时我们发现结果为 $1\ 1000\ 0000$ 溢出至 9 位了。如果我们不做截断，那么这个结果仍然是正确的。

3.1.4 符号扩展与截断

当我们需要将一个较短的有符号整数转换为较长的表示时，需要进行**符号扩展 (Sign Extension)**：将符号位复制到所有新增的高位。

例：将 8 位补码 11111011 (-5) 扩展为 16 位：

- 符号位为 1，将其复制到高 8 位
- 结果为 111111111111011 ，仍然表示 -5

对于正数 00000101 ($+5$)，扩展后为 0000000000000101 。

相对地，对于无符号数，我们进行**零扩展 (Zero Extension)**：高位填充 0。

在 RISC-V 中，`1b` (load byte) 指令会对加载的字节进行符号扩展，而 `1bu` (load byte unsigned) 指令则进行零扩展。例如，从内存加载字节 `0xFF` (-1 作为有符号数， 255 作为无符号数)：

- `1b` 将其扩展为 32 位的 `0xFFFFFFFF` (-1)
- `1bu` 将其扩展为 32 位的 `0x000000FF` (255)

当我们需要将一个较长的有符号整数转换为较短的表示时，需要进行**截断 (Truncation)**：直接丢弃高位，保留低位。

在前面原码与补码转换的例子中， $1\ 1000\ 0000$ 将会被截断成 $1000\ 0000$ ，也就是 -256 。因为截断的存在，导致原码与补码的转换出现错误。

3.2 整数的算术运算

3.2.1 加法运算

二进制加法遵循与十进制加法相同的原理：从最低位开始逐位相加，产生进位时向高位传递。

例：计算 4 位补码加法 $5 + 3$: $0101 + 0011 = 1000$ 。结果 1000 在 4 位无符号解释下为 8，但在 4 位有符号补码解释下为 -8：这是一个溢出的例子，我们稍后讨论。

再例：计算 $(-3) + 5$ (使用 4 位补码): 1101 (-3 的补码) $+ 0101$ (5) $= 10010$ ，丢弃最高位进位得到了正确的结果 0010 (2)。

3.2.2 减法运算

在补码表示下，减法可以转换为加法：

$$A - B = A + (-B) = A + (\bar{B} + 1) \quad (10)$$

其中 \bar{B} 表示 B 的按位取反。

3.2.3 溢出检测

当运算结果超出表示范围时，就会发生溢出 (Overflow)。对于有符号数的加减法，溢出的检测规则如下：

$$\text{有符号溢出} = C_{n-1} \oplus C_n \quad (11)$$

其中 C_{n-1} 是倒数第二位向最高位的进位， C_n 是最高位向外的进位。

等价地，溢出发生在以下情况：

- 正溢出：两个正数相加，结果为负数
- 负溢出：两个负数相加，结果为正数

例 (正溢出)：4 位补码计算 $7 + 1$: $0111 + 0001 = 1000$ (-8?)。两个正数相加得到负数，发生正溢出。正确结果 8 超出了 4 位补码的表示范围 $[-8, 7]$ 。

例 (负溢出)：4 位补码计算 $(-8) + (-1)$: $1000 + 1111 = 10111$ ，丢弃进位得 0111 (7?)。两个负数相加得到正数，发生负溢出。正确结果 -9 超出了表示范围。

溢出检测的本质是：如果两个操作数的符号相同，但结果的符号与它们不同，则发生溢出。从进位的角度看，当最高位的进入进位和输出进位不一致时，说明符号位被错误地改变了。

3.2.4 乘法运算

二进制乘法的基本原理与十进制乘法相似：将乘数的每一位与被乘数相乘，得到部分积，然后将所有部分积相加。

$$P = A \times B = \sum_{i=0}^{n-1} (A \times b_i) \times 2^i \quad (12)$$

其中 b_i 是乘数 B 的第 i 位。由于 b_i 只能是 0 或 1，所以 $A \times b_i$ 要么是 0，要么是 A 。

基本的乘法器需要 n 个时钟周期来完成一次乘法，效率较低。关于更快速的乘法算法 (如 Booth 算法) 或除法运算，请参见附录 C。

3.3 浮点数的表示

整数虽然精确，但无法表示小数。为了表示实数，计算机使用浮点数（Floating-Point Number）表示法，其基本形式类似于科学计数法。

3.3.1 IEEE 754 标准

IEEE 754 标准是目前最广泛使用的浮点数表示标准。

该标准规定浮点数由三个部分组成：

- **符号位 S** : 0 表示正数，1 表示负数
- **指数 E** : 表示 2 的幂次
- **尾数 M** （也称有效数字、小数部分）: 表示有效数字

3.3.2 单精度浮点数

IEEE 754 单精度浮点数使用 32 位存储，格式如下：

字段	符号位 S	指数 E	尾数 M
位数	1 位	8 位	23 位
位置	[31]	[30:23]	[22:0]

表 3: IEEE 754 单精度浮点数格式

为了能够表示正负指数，IEEE 754 使用**偏移指数**：存储的指数值等于实际指数加上一个偏移量。对于单精度，偏移量为 $127 = 2^7 - 1$ 。

此外，IEEE 754 采用**规格化表示**：尾数的整数部分总是 1（称为隐含的整数位），只存储小数部分。这样可以多获得一位精度。

因此，单精度浮点数的值为：

$$V = (-1)^S \times 1.M \times 2^{E-127} \quad (13)$$

其中 $1.M$ 表示在存储的尾数前面加上隐含的 1。

3.3.3 双精度浮点数

双精度浮点数使用 64 位存储，有 11 个指数位和 52 个尾数位，其偏移量为 $1023 = 2^{10} - 1$ 。

双精度浮点数的值为：

$$V = (-1)^S \times 1.M \times 2^{E-1023} \quad (14)$$

3.3.4 特殊值与非规格化数

IEEE 754 定义了几种特殊的编码来处理边界情况：

类型	指数 E	尾数 M	表示值
零	全 0	全 0	± 0
非规格化数	全 0	$\neq 0$	$\pm 0.M \times 2^{1-\text{Bias}}$
规格化数	$1 \sim 2^k - 2$	任意	$\pm 1.M \times 2^{E-\text{Bias}}$
无穷大	全 1	全 0	$\pm \infty$
NaN	全 1	$\neq 0$	NaN

表 4: IEEE 754 特殊值编码 (单精度: $k = 8$, Bias= 127)

非规格化数与渐进下溢 非规格化数是指数全为 0 但尾数不为 0 的浮点数。与规格化数不同，非规格化数的隐含整数位是 0 而不是 1:

$$V = (-1)^S \times 0.M \times 2^{1-\text{Bias}} \quad (15)$$

注意指数使用 $1 - \text{Bias}$ 而不是 $0 - \text{Bias}$ ，这是为了使非规格化数能够平滑地过渡到最小的规格化数。

例：对于单精度浮点数，最小的正规格化数是 $1.0 \times 2^{-126} \approx 1.18 \times 10^{-38}$ 。如果没有非规格化数，任何绝对值小于这个数的非零值都会被强制变为 0（称为“突然下溢”）。有了非规格化数，我们可以表示 0 到 2^{-126} 之间的值，实现渐进下溢。

NaN 与无穷大 产生特殊值的运算示例：

- 无穷大: $1.0/0.0 = +\infty$, $-1.0/0.0 = -\infty$
- NaN: $0.0/0.0$, $\infty - \infty$, $\sqrt{-1}$, $\infty \times 0$

NaN 有一个特殊性质：它与任何值（包括自身）的比较都返回 false，即 $\text{NaN} \neq \text{NaN}$ 。

3.4 浮点数的算术运算

3.4.1 浮点加减法

浮点加减法与整数加减法不同，因为他需要保证两个数的指数相同再处理运算才能得到正确的结果，因此我们需要将**指数较小的数的尾数右移**，使两个指数相等（对齐）：

例：使用 8 位浮点数格式计算 $1.5 + 0.125$ 。
 1.5 的二进制表示为 $1.0111\ 100$ ，
 0.125 的二进制表示为 $0.0100\ 000$ 。
对阶时，将小指数向大指数看齐， 0.125 的尾数右移 3 位： $1.000_2 \rightarrow 0.001_2$ 。
尾数相加： $1.100_2 + 0.001_2 = 1.101_2$ 。
结果 $1.101_2 \times 2^0 = 1.625$ ，已规格化，其 8 位表示为 $0\ 0111\ 101$ 。

我们采取**指数较小的数向指数较大的数对齐**的主要原因是，右移尾数只会丢失低位，而左移尾数会使得高位信息丢失，从而影响结果。

3.4.2 浮点乘除法

浮点乘除法的计算步骤相似：符号位通过异或运算得到 $S = S_1 \oplus S_2$ ；指数相加或相减后需调整偏移量，乘法为 $E = E_1 + E_2 - \text{Bias}$ ，除法为 $E = E_1 - E_2 + \text{Bias}$ ；尾数直接相乘或相除 $M = M_1 \times M_2$ 或 $M = M_1/M_2$ ；最后进行规格化与舍入。

例：计算 1.5×2.0 。将两数转换为二进制浮点： $1.5 = 1.1_2 \times 2^0$, $2.0 = 1.0_2 \times 2^1$ 。符号均为正，结果为正；指数相加 $0 + 1 = 1$ ；尾数相乘 $1.1_2 \times 1.0_2 = 1.1_2$ 。结果 $1.1_2 \times 2^1 = 11_2 = 3.0$ 。

3.4.3 精度问题与舍入

浮点运算中，精度损失是不可避免的。IEEE 754 定义了四种舍入模式：

- **向最近偶数舍入**（默认）：舍入到最近的可表示值；若恰好在中间，则舍入到最近的偶数
- **向零舍入**：直接截断，向零方向舍入
- **向正无穷舍入**：向 $+\infty$ 方向舍入
- **向负无穷舍入**：向 $-\infty$ 方向舍入

例（向偶数舍入）：假设尾数只能保留 2 位小数。对于 1.101_2 （恰好在 1.10_2 和 1.11_2 的中点），由于 1.10_2 的末位是 0（偶数），故舍入为 1.10_2 ；对于 1.111_2 （恰好在 1.11_2 和 10.00_2 的中点），由于 10.00_2 的末位是 0（偶数），故舍入为 10.00_2 。

4 处理器的单周期与流水线

从本章开始，我们将介绍处理器的设计与实现。本章中会涵盖一些硬件设计上的原理解释，这些内容均已用下划线标出，以便读者区分。

4.1 单周期处理器

4.1.1 数据通路

计算机之所以能够处理复杂的任务，是因为在处理器内部，我们建立起了符合逻辑的数据通路。我们可以将数据沿着我们所希望的方向传递，并得到正确的加工。

在完成了计算机组成与设计实验课后，我们知道，计算机处理指令分为五个步骤，分别是取指 (IF, Instruction Fetch)、译码 (ID, Instruction Decode)、执行 (EX, EXecute)、访存 (MEM, MEMory Access) 和写回 (WB, Write Back)。在单周期处理器中，这五个步骤会在一个时钟周期内完成。

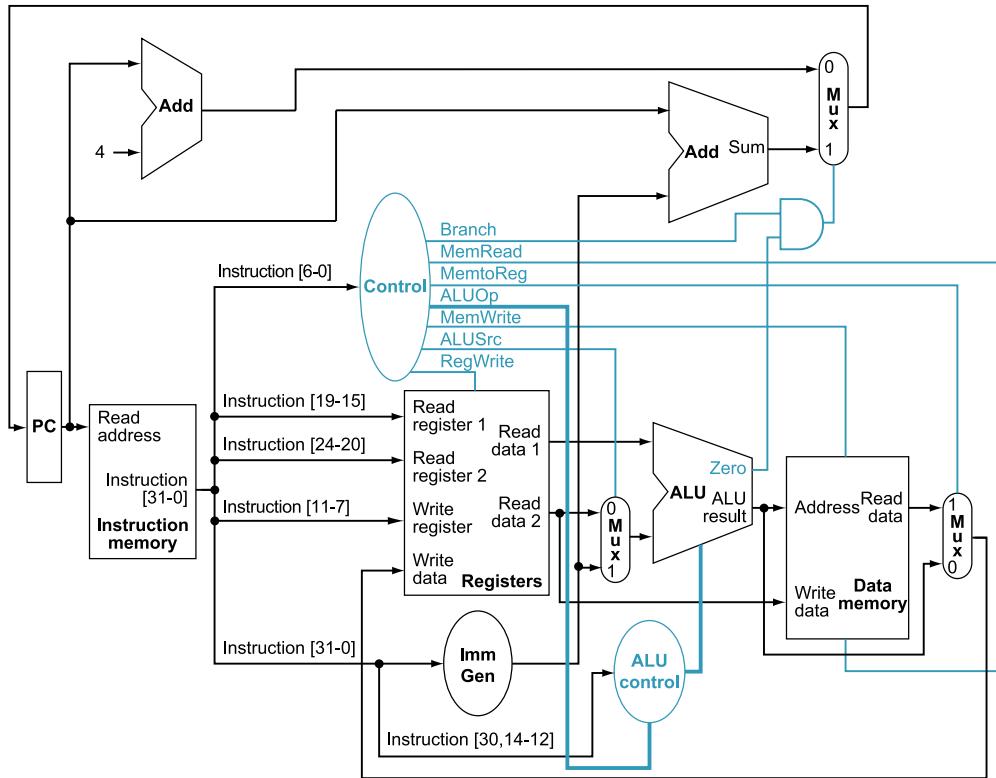


图 1: 单周期处理器数据通路

取指 首先，我们先找到当前程序执行到了哪个位置 (PC)，并根据 PC 的值从指令存储器 (Im) 中取出对应的指令 (Instruction)。在 RISC-V 中，每条指令共有 4 字节 32 位宽。

译码 知道指令的格式后，我们需要根据指令的一些信息来判断指令的类型，并根据类型提取出对应的数据和控制信号。

RISC-V 指令集定义了六种基本指令格式，每种格式的字段划分如下表所示：

名称 (字段大小)	字段 Instruction[31:0]					
	7 位	5 位	5 位	3 位	5 位	7 位
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode
U-type	immediate[31:12]				rd	opcode

表 5: RISC-V 指令格式

实际上，计算机硬件在处理此类问题的时候会先通过 `opcode` 来解析该条指令所述的指令名称（甚至是具体内容，比如 U 和 UJ 型指令就不需要通过 `funct3` 和 `funct7` 来进一步判断，因为他们没有这些字段）

对于需要构建立即数的指令，我们将指令格式的信息和立即数字段传递给立即数生成器（ImmGen），以生成正确的立即数。

执行 在执行阶段，我们会将指令的两个操作数传送给算术逻辑单元（ALU，Arithmetic Logic Unit）。ALU 会根据指令的功能码（`funct3` 和 `funct7`）执行相应的运算，并将结果输出。

访存 对于需要访问内存的指令（如加载和存储指令），我们会使用 ALU 的结果作为内存地址，从数据存储器（Dm）中读取或写入数据。

写回 最后，在写回阶段，我们会将 ALU 的结果或从内存中读取的数据写回到寄存器文件（RF）中指定的寄存器。

4.1.2 控制信号

如果不对数据通路进行控制，那么每一条指令都会访存、写回寄存器，甚至会传递给 ALU 错误的输入源。为此，我们需要控制信号来指导数据通路的行为。我们会根据 `funct3`, `funct7` 和 `opcode` 生成控制信号，接下来让我们看看这些控制信号是如何控制数据通路的行为的。

Branch Branch 控制信号用于控制分支指令的执行。当 Branch 信号为高电平时，表示当前指令是一个分支指令，处理器需要根据 ALU 的比较结果决定是否跳转。当 Branch 信号为低电平时，ALU 的计算结果对 PC 单元将会被屏蔽，也即 ALU 的计算结果不能影响 PC 的值。

MemRead MemRead 控制信号用于控制数据存储器的读取操作。当 MemRead 信号为高电平时，Data Memory 的输出口会输出从内存中读取的数据。也即 `Read data = Memory[Address]` 当 MemRead 控制信号为低电平时，Data Memory 的输出口将不会输出有意义的数据。

MemtoReg 对于写回这个阶段来说，我们可以选择是写回 ALU 的结果（像是 `add`, `addi` 这样的指令）还是写回从内存中读取的数据（像是 `lw`, `lb` 这样的指令）。MemtoReg 控制信号用于选择写回的数据源。当 MemtoReg 信号为高电平时，写回的数据来自数据存储器（Data Memory）；当 MemtoReg 信号为低电平时，写回的数据来自 ALU 的输出。

ALUOp ALUOp 控制信号用于指导 ALU 执行特定的运算。ALUOp 通常是一个多位信号，不同的组合对应不同的运算类型。例如，ALUOp 可以指示 ALU 执行加法、减法、逻辑与、逻辑或等操作。

在 RISC-V 中，我们的 ALUOp 共有 2 位，但 ALU 需要接受的控制信号 ALUCtrl 有 4 位，因此我们需要一个 ALU 控制单元（ALU Control Unit）来将 ALUOp 转换为 ALUCtrl。

ALUCtrl 的四位分别是 Ainvert, Bnegate, Operation[1:0]。其中 Ainvert 用于控制第一个操作数是否取反，Bnegate 用于控制第二个操作数是否取负，Operation[1:0] 用于指定具体的运算类型。见下表：

Operation[1:0]	操作类型
00	AND
01	OR
10	ADD
11	LESS

表 6: Operation[1:0] 与 ALU 操作类型的对应关系

Operation 这个控制信号本质上是控制最后的 Mux（多路选择器）所用的，这个选择器会根据 Operation 的值选择 ALU 的最终输出结果。如果没有这个选择器，ALU 将会同时混淆输出这四种输出（实际上不可能）。

因此，想要执行 `sub ans, a, b` 则需要将 ALUCtrl 设置为 0110，表示对第一个操作数不取反，对第二个操作数取负，并执行加法运算（即 $a + (-b)$ ）。

MemWrite ALU 结果的输出槽连向数据存储器的地址输入槽，而寄存器 `Read data 2` 连向。为当 MemWrite 信号为高电平时，。当 MemWrite 信号为低电平时，数据存储器不会进行写操作。

4.2 流水线控制

流水线（Pipelining）是一种提高处理器性能的技术，它允许多条指令同时在处理器的不同阶段执行。在 RISC-V 中，我们通常使用五级流水线，言下之意，我们将处理器的数据通路和控制拆

成五个步骤，每个时钟周期这五个步骤要同时分别执行完成，因此，时钟周期的长度由最慢的步骤决定。

4.2.1 流水线寄存器

为了在流水线的各个阶段之间传递数据和控制信号，我们需要在每两个阶段之间插入流水线寄存器（Pipeline Registers）。这些寄存器在每个时钟周期结束时保存当前阶段的输出，并在下一个时钟周期开始时将这些值传递给下一个阶段。

RISC-V 的五级流水线需要四组流水线寄存器：IF/ID、ID/EX、EX/MEM 和 MEM/WB。每组寄存器存储该阶段需要传递给下一阶段的所有数据和控制信号。

4.2.2 流水线冒险

流水线冒险（Pipeline Hazards）是指在流水线执行过程中可能出现的导致流水线无法正常工作的情况。主要有三类冒险：

数据冒险（Data Hazards） 数据冒险发生在一条指令依赖于前一条指令的结果，但前一条指令还没有将结果写回寄存器时。

考虑以下指令序列，其中 `sub` 指令需要使用 `add` 指令计算得到的 `x1` 值，但当 `sub` 指令在 ID 阶段读取 `x1` 时，`add` 指令可能还在 EX 或 MEM 阶段，尚未将结果写回 `x1`：

```

1 add x1, x2, x3      # x1 = x2 + x3
2 sub x4, x1, x5      # x4 = x1 - x5

```

控制冒险（Control Hazards） 控制冒险发生在遇到分支指令时，因为处理器在分支指令执行完成之前无法确定下一条要执行的指令是什么。

考虑分支指令 `beq x1, x2, label`。处理器需要等到 EX 阶段才能确定分支是否成立，但此时后续的指令可能已经进入流水线。如果分支成立，这些已经取出的指令需要被丢弃。

结构冒险（Structural Hazards） 结构冒险发生在硬件资源不足以同时支持所有指令执行时。

在 RISC-V 的五级流水线设计中，我们通常将指令存储器和数据存储器分离（哈佛架构，Harvard Architecture），因此几乎不会出现结构冒险。

4.2.3 冒险检测与解决方案

前递（Data Forwarding/Bypassing） 前递是解决数据冒险的主要技术。其基本思想是：如果一条指令需要的数据刚刚由前面的指令计算出来但还未写回寄存器，那么可以直接将这个数据从流水线寄存器中转发到需要它的阶段。

前递单元（Forwarding Unit）会检测以下情况：

- **EX 冒险**: 如果 ID/EX 阶段的源寄存器与 EX/MEM 阶段的目标寄存器相同，则从 EX/MEM 寄存器转发
- **MEM 冒险**: 如果 ID/EX 阶段的源寄存器与 MEM/WB 阶段的目标寄存器相同，则从 MEM/WB 寄存器转发

前述单元的硬件逻辑可以用以下 Python 代码表示：

```

1 def forwarding_unit(rs: int, ex_rd: int, ex_wr: bool,
2                     mem_rd: int, mem_wr: bool) -> int:
3     """
4     Args: rs: 当前指令的源寄存器编号
5             ex_rd: EX/MEM 阶段指令的目标寄存器编号
6             ex_wr: EX/MEM 阶段指令是否写寄存器
7             mem_rd: MEM/WB 阶段指令的目标寄存器编号
8             mem_wr: MEM/WB 阶段指令是否写寄存器
9     Returns: 0=无前递，1=从EX/MEM前递，2=从MEM/WB前递
10    """
11    # EX 冒险（优先）
12    if ex_wr and ex_rd != 0 and ex_rd == rs:
13        return 1
14    # MEM 冒险
15    if mem_wr and mem_rd != 0 and mem_rd == rs:
16        return 2
17    return 0

```

流水线暂停 (Pipeline Stalling) 对于某些数据冒险，仅靠转发无法解决。例如，当一条加载指令 (lw) 后紧跟一条使用该加载结果的指令时，因为加载指令在 MEM 阶段才能获得数据，而后续指令在 EX 阶段就需要这个数据，此时无法通过转发解决。

此时需要暂停 (Stall) 流水线，即在流水线中插入一个或多个空操作 (NOP, No Operation)，也称为气泡 (Bubble)。冒险检测单元 (Hazard Detection Unit) 负责检测此类情况并插入暂停。

这里的意思是说，当我们在汇编代码层面上避免流水线暂停的时候应该引入 NOP。例如，如果不加这条指令，处理器会自动的进行流水线暂停，为此，处理器会将 Ex 执行阶段前的寄存器冻结，并向后面的寄存器注入 NOP 指令，从而实现暂停的效果。

分支预测 (Branch Prediction) 为了减少控制冒险的影响，现代处理器通常采用分支预测技术。最简单的策略是：

- **静态预测**: 始终预测分支不成立（或成立）
- **动态预测**: 根据分支指令的历史行为进行预测

如果预测错误，需要清空（Flush）流水线中已经取出的错误指令，并从正确的地址重新开始取指。这个操作由流水线刷新单元（Pipeline Flush Unit）完成。并且，我们应该在预测后指令执行到 MEM 阶段前完成分支判断，否则内存或者寄存器将会被写回错误的数据。

延迟分支（Delayed Branch） RISC-V 还可以使用延迟分支技术：编译器将分支指令后的一条或多条与分支结果无关的指令移到分支指令之后，这样即使分支预测错误，这些指令的执行也不会浪费。

4.2.4 流水线控制

在流水线处理器中，控制单元需要为每个阶段生成相应的控制信号。由于指令在流水线中移动，控制信号也需要随着指令一起在流水线寄存器间传递。

控制信号的生成与传递 控制单元在 ID 阶段译码指令后生成所有控制信号，然后这些信号随着指令通过流水线寄存器向后传递。（实际上，除去这些信号，一些其他信息，如某个阶段的寄存器信息也会装入流水线寄存器中，并向后传递）不同阶段需要的控制信号如下：

阶段	控制信号	作用
EX	ALUSrc	选择 ALU 第二个操作数来源 (0= 寄存器, 1= 立即数)
	ALUOp	指定 ALU 执行的操作类型
MEM	MemRead	是否从内存读取数据
	MemWrite	是否向内存写入数据
WB	RegWrite	是否写回寄存器
	MemtoReg	选择写回寄存器的数据来源 (0=ALU 结果, 1= 内存数据)

表 7: 流水线各阶段的主要控制信号

控制信号在流水线寄存器中占用的空间很小，通常只需要几个比特。例如，上述 6 个主要控制信号加起来可能只需要 10 位左右。这些信号随着指令一起在流水线中流动，确保每条指令在正确的阶段执行正确的操作。

冒险控制单元的协作 流水线处理器需要三个主要的控制单元协同工作：

- **主控制单元（Main Control Unit）**: 在 ID 阶段根据指令操作码生成基本控制信号
- **冒险检测单元（Hazard Detection Unit）**: 检测加载使用型（Load-Use）数据冒险和控制冒险，必要时插入气泡（暂停流水线）
- **前递单元（Forwarding Unit）**: 检测其他类型的数据冒险，通过前递路径解决

5 层次化存储——万物皆缓存

5.1 存储器层次结构概述

在理想情况下，我们希望拥有一个既快速又容量巨大的存储器。然而，物理规律和经济成本使得这一愿望难以实现——速度越快的存储器，单位容量的成本越高，容量也越受限。为了在速度和容量之间取得平衡，现代计算机采用了层次化存储结构：将少量快速但昂贵的存储器与大量慢速但廉价的存储器结合使用。

从处理器核心向外看，存储器层次依次为：寄存器、多级 Cache 缓存、主存（DRAM）、以及磁盘或固态硬盘。越靠近处理器的层次，访问速度越快、容量越小、成本越高；反之则速度越慢、容量越大、成本越低。

层次化存储的核心思想是“用小而快的存储器缓存大而慢的存储器中的热点数据”。从 CPU 的角度看，它希望获得接近寄存器的访问速度和接近磁盘的存储容量——这两个目标本身是矛盾的，而层次化存储正是这一矛盾的折中方案。

5.1.1 局部性原理

层次化存储之所以有效，根本原因在于程序访问数据时表现出的局部性（Locality）。局部性原理分为两种形式：

时间局部性（Temporal Locality）指的是：如果一个数据项被访问，那么它在不久的将来很可能再次被访问。例如，在一个循环中，循环变量 i 会被反复读取和修改，循环体内的指令也会被重复执行。

空间局部性（Spatial Locality）指的是：如果一个数据项被访问，那么与它地址相邻的数据项很可能也会被访问。例如，当程序遍历数组 $A[0], A[1], A[2], \dots$ 时，访问完 $A[0]$ 后很快就会访问 $A[1]$ 。

5.1.2 平均访问时间

衡量存储系统性能的关键指标是平均访问时间（AMAT，Average Memory Access Time）。对于一个包含缓存的存储层次，平均访问时间由以下公式计算：

$$T_{\text{avg}} = T_{\text{hit}} + \text{Miss Rate} \times T_{\text{miss}} \quad (16)$$

其中 T_{hit} 是缓存命中时的访问时间，Miss Rate 是缓存缺失率（未命中的访问比例）， T_{miss} 是缓存缺失时从下一级存储获取数据的额外时间。

例：假设 L1 缓存的命中时间为 1 个周期，缺失率为 5%，缺失时需要额外 20 个周期从 L2 获得数据。则平均访问时间为 $T_{\text{avg}} = 1 + 0.05 \times 20 = 2$ 个周期。

对于多级缓存系统，平均访问时间可以递归计算。以两级缓存为例：

$$T_{\text{avg}} = T_{L1} + M_{L1} \times (T_{L2} + M_{L2} \times T_{\text{mem}}) \quad (17)$$

其中 T_{L1} 和 T_{L2} 分别是 L1 和 L2 缓存的命中时间, M_{L1} 和 M_{L2} 是各级缓存的缺失率, T_{mem} 是访问主存的时间。

5.2 高速缓存

缓存通常通过标签 (Tag)、索引 (Index) 和块内偏移 (Block Offset) 这三个字段来理解地址。每当我们产生一次内存访问请求时, 我们会传递一个地址, 我们需要先判断这个地址是否在缓冲中, 对于缓存来说, 通常采取不一样的视角来看待地址:

Tag (标签)	Index (索引)	Block Offset (块内偏移)
----------	------------	---------------------

对于缓存控制器来说, 他会进行如下的操作:

```

1 def cache_access(address: int) -> Data:
2     """
3         Args: address: 访问的内存地址
4         Returns: True=命中, False=缺失
5     """
6     tag, index, offset = reading(address) # 解析地址
7
8     # 获取缓存行
9     cache_line = cache.find(line = index)
10
11    # 检查标签和有效位
12    if cache_line.valid and cache_line.tag == tag:
13        # 命中, 返回数据
14        data = cache_line[offset]
15        return data
16    else:
17        # 缺失, 需要从下一级存储加载数据
18        load(address, cache_line)
19        return None

```

其中, 缓存行的结构类似于:

```

1 @dataclass
2 class CacheLine:
3     tag: int          # 标签
4     valid: bool       # 有效位
5     data: Data        # 数据块
6     ...              # eg. dirty: bool

```

5.2.1 缓存的映射方式

确定标签，索引和块内偏移的位大小是直观重要的，程序员们设计了三种不同的缓存映射方式来处理这些字段：直接映射缓存、全相联缓存和组相联缓存。

在后续的讨论中，我们假设我们有 2^n 个缓存行，每行保存 2^m 字节，每次访问的地址有 32 位宽。

直接映射缓存 直接映射（Direct-Mapped）缓存是最简单的映射方式：每个内存地址只能映射到缓存中的一个固定位置。

在这种映射模式下，我们的 $\text{index} = n$, $\text{offset} = m$, 标签占剩余的位数，即 $\text{tag} = 32 - m - n$ 。

直接映射缓存的优点是硬件简单、访问速度快——只需要一次比较即可判断命中。但其缺点同样明显：如果两个频繁访问的地址恰好映射到同一行，就会发生严重的冲突。

全相联缓存 全相联（Fully-Associative）缓存走向另一个极端：任意内存块可以放在缓存的任意位置。此时 $\text{index} = 0$, $\text{offset} = m$, $\text{tag} = 32 - m$ 。每次访问时，缓存控制器必须同时比较所有缓存行的标签，以判断是否命中。换言之，在前面代码中的 `cache.find()` 中我们得到了整个缓存所拥有的缓存行，即：

```

1 lines: List[CacheLine] = cache.find() # index = None
2 for line in lines:
3     ...

```

组相联缓存 组相联(Set-Associative)缓存是直接映射和全相联的折中。缓存被划分为若干组(Set)，每组包含 t 个缓存行，称为 t 路组相联。每个内存地址根据索引字段映射到一个固定的组，但在组内可以放在任意一行。此时 $\text{index} = t$, $\text{offset} = m$, $\text{tag} = 32 - m - t$ 对于这种情况，我们会有：

```

1 Set: List[CacheLine] = cache.find(line = index) # size = 2^t, index = t
2 for line in lines:
3     ...

```

5.2.2 缓存访问

当处理器发出内存访问请求时，缓存控制器执行以下步骤：首先根据地址的索引字段定位到对应的缓存组；然后将地址的标签与组内所有行的标签并行比较；若某行标签匹配且有效位为 1，则命中，根据块内偏移取出数据；若无匹配，则缺失，需要从下一级存储取回数据并替换某一行。

缓存缺失可分为三类，称为 3C 模型：强制缺失（Compulsory Miss）是首次访问某块时不可避免的缺失；容量缺失（Capacity Miss）是因为缓存容量不足以容纳工作集而导致的缺失；冲突缺失（Conflict Miss）是因为多个地址映射到同一位置而导致的缺失，增加相联度可以减少冲突缺失。

当缓存缺失且需要替换某一行时，替换策略决定了选择哪一行被驱逐。常见的策略包括：LRU（Least Recently Used，最近最少使用）选择最长时间未被访问的行；FIFO（First In First Out，先

进先出) 选择最早被加载的行; 随机替换则随机选择一行。LRU 通常效果最好, 但硬件实现较复杂; 实际系统中常使用近似 LRU 算法。

5.2.3 缓存写策略

处理器不仅读取数据, 还会写入数据。缓存的写策略决定了写操作如何处理。

写直达 写直达 (Write-Through) 策略在每次写操作时同时更新缓存和主存。这保证了缓存与主存的数据始终一致, 简化了一致性管理。但每次写操作都需要访问主存, 带宽消耗大、延迟高。为缓解这一问题, 通常配合使用写缓冲区 (Write Buffer): 写操作先写入缓冲区, 处理器无需等待主存完成即可继续执行。

写回 写回 (Write-Back) 策略只更新缓存, 不立即更新主存。被修改的缓存行用脏位 (Dirty Bit) 标记。只有当该行被替换时, 才将脏数据写回主存。写回策略减少了主存访问次数, 提高了性能, 但增加了一致性管理的复杂度。

写分配与非写分配 当写操作发生缺失时, 还需决定是否将目标块加载到缓存。写分配 (Write-Allocate) 会先将块放入缓存再写入, 利用了空间局部性——后续可能还会访问同一块。非写分配 (No-Write-Allocate) 直接写入下一级存储, 不加载到缓存。通常写回策略配合写分配使用, 写直达策略配合非写分配使用。

5.2.4 缓存一致性

在多核处理器中, 每个核心通常拥有独立的 L1 缓存, 可能同时缓存同一内存地址的数据副本。当一个核心修改了自己缓存中的数据, 其他核心的缓存副本就变得陈旧。缓存一致性协议确保所有核心看到的内存视图是一致的。

MESI 协议是最经典的缓存一致性协议之一。每个缓存行处于四种状态之一: Modified (M, 已修改) 表示该行已被本核心修改, 是唯一有效副本, 与主存不一致; Exclusive (E, 独占) 表示该行未被修改, 是唯一副本, 与主存一致; Shared (S, 共享) 表示该行未被修改, 可能存在于多个核心的缓存中; Invalid (I, 无效) 表示该行不包含有效数据。

当一个核心要写入 Shared 状态的缓存行时, 必须先向总线广播使无效 (Invalidate) 消息, 使其他核心将对应行标记为 Invalid。这称为写使无效协议。只有当其他核心确认使无效完成后, 该核心才能将状态改为 Modified 并执行写操作。

5.3 纠错码

存储器中的数据可能因为各种原因发生错误: 宇宙射线、电源噪声、器件老化等都可能导致存储单元的值意外翻转。对于关键应用场景, 仅仅检测错误是不够的, 还需要能够纠正错误。ECC (Error-Correcting Code, 纠错码) 通过添加冗余位来实现错误检测和纠正。

5.3.1 基本原理

最简单的错误检测方法是奇偶校验：为每组数据添加一个校验位，使得所有位（包括校验位）的异或结果为 0（偶校验）或 1（奇校验）。如果传输过程中某一位发生翻转，校验结果就会改变，从而检测到错误。但奇偶校验无法定位错误位置，也无法检测偶数个错误。

为了纠正错误，需要更多的冗余信息。关键概念是汉明距离（Hamming Distance）：两个等长码字（符合要求的编码都是码字）之间不同位（差异）的数量。如果一个编码方案的最小汉明距离为 d ，则该方案可以检测任意 $d - 1$ 个错误，或纠正任意 $\lfloor(d - 1)/2\rfloor$ 个错误。要纠正单比特错误，最小汉明距离至少为 3。

5.3.2 汉明码

汉明码（Hamming Code）是一种经典的单比特纠错码。其核心思想是：将校验位放在位置编号为 2 的幂次的位置（1, 2, 4, 8, ...），每个校验位负责检验一组特定的数据位。

具体地，位置 i 的数据由位置编号的二进制表示中对应位为 1 的所有校验位共同检验。例如，位置 7 的二进制是 0111，因此它被校验位 P_1 （位置 $1 = 2^0$ ）、 P_2 （位置 $2 = 2^1$ ）和 P_4 （位置 $4 = 2^2$ ）检验。

当接收方收到编码后，重新计算各校验位。若某些校验位不匹配，将这些校验位的位置编号相加，得到的和就是出错位的位置。

在实际的存储系统中，常用 SECDED (Single Error Correction, Double Error Detection) 编码，即在汉明码基础上再增加一个总体校验位（第 0 位， P_0 ），既能纠正单比特错误，又能检测双比特错误。现代 ECC 内存通常使用 72 位编码保护 64 位数据。

纠正两位错误的编码不太现实，并且成本极高，因此一般生活中都采用 SECDED 的编码格式。

5.4 虚拟内存

虚拟内存是操作系统提供的一种抽象，它让每个进程都认为自己独占整个地址空间。程序使用的地址称为虚拟地址（Virtual Address），而物理内存的实际地址称为物理地址（Physical Address）。MMU（Memory Management Unit，内存管理单元）负责在程序运行时将虚拟地址转换为物理地址。

简单地理解，虚拟内存就是硬盘的 cache。（缓存是内存的 cache）

虚拟内存带来了多重好处：首先，它实现了进程隔离——每个进程拥有独立的虚拟地址空间，无法直接访问其他进程的内存；其次，它简化了内存管理——程序可以假设从固定地址开始，无需关心物理内存的实际分配；最后，它支持按需分配——程序可以使用比物理内存更大的地址空间，不常用的数据可以暂存到磁盘。

5.4.1 基本概念

虚拟内存将地址空间划分为固定大小的页（Page），物理内存则划分为同样大小的页框（Page Frame）。常见的页大小为 4 KiB。虚拟地址被分为两部分：高位的 VPN（Virtual Page Number，

虚拟页号) 指示属于哪一页, 低位的页内偏移 (Page Offset) 指示在页内的位置。

地址转换的核心是将 VPN 映射到 PPN (Physical Page Number, 物理页号)。这个映射关系存储在页表 (Page Table) 中。物理地址由 PPN 和页内偏移拼接而成:

$$\text{物理地址} = \text{PPN} \times \text{页大小} + \text{页内偏移} \quad (18)$$

页表存储在主存中, 每个进程拥有独立的页表。页表的每一项称为 PTE (Page Table Entry, 页表项), 包含以下信息: PPN (物理页号)、有效位 (指示该页是否在物理内存中)、访问权限位 (读/写/执行)、以及其他状态位 (如是否被访问、是否被修改)。

当程序访问某个虚拟地址, 而对应的页不在物理内存中时 (有效位为 0), MMU 触发缺页异常 (Page Fault)。操作系统的异常处理程序将请求的页从磁盘调入物理内存, 更新页表, 然后重新执行被中断的指令。

当一个进程访问内存的时候, 我们将先去寻找这个内存对应的页表, 然后解析虚拟地址, 找到对应的物理地址。类似缓存的, 虚拟内存也会以不一样的方式理解内存:

VPN (虚拟页号)	Block Offset (页内偏移)
------------	---------------------

```

1 def memory_access(virtual_address: int) -> Data:
2     """
3     Args: virtual_address: 访问的虚拟地址
4     Returns: 访问的数据
5     """
6     vpn, offset = reading(virtual_address)
7     page_table = get_page(pid) # 获取进程对应的页表
8
9     # 查找页表项
10    pte = page_table.find(vpn)
11
12    if not pte:
13        # 触发缺页异常
14        raise PageFault(virtual_address)
15
16    # 计算物理地址
17    physical_address = pte.ppn * PAGE_SIZE + offset
18
19    # 访问物理内存
20    data = physical_memory.read(physical_address)
21    return data

```

5.4.2 页表

单级页表 最直接的页表实现是线性页表：为虚拟地址空间的每一页分配一个页表项。对于 32 位地址空间、4 KiB 页面，共有 2^{20} 个页，每个 PTE 占 4 字节，则页表大小为 4 MiB。这对于单个进程尚可接受，但当系统运行大量进程时，页表占用的内存将变得不可接受。

更严重的问题是：大多数程序只使用地址空间的一小部分，但线性页表必须为整个地址空间预留空间。对于 64 位系统，理论上的虚拟地址空间高达 2^{64} 字节，线性页表根本不可行。

多级页表 多级页表通过将页表本身分页来解决上述问题。以两级页表为例：虚拟地址的 VPN 被进一步划分为一级页号和二级页号。一级页表（页目录）的每一项指向一个二级页表；二级页表的每一项才包含实际的 PPN。

但是，将页表分成几级成了一个复杂的问题。由于所有的页表会储存在主存中，而主存又是分页存储的，所以，当页表大小刚刚好等于物理页的大小时，效率最高。对于 64 位系统来说，一个页表通常有 4KiB 大，一个页表项 8 字节。假设虚拟内存有 44 位，那么每一个页表可以存储 $2^{12}/2^3 = 2^9$ 个页表项。因此，我们有 $44 - 9 = 35$ 个虚拟页号，我们需要 $\lceil 35/9 \rceil = 4$ 级页表。

5.4.3 反向页表

传统页表以虚拟页号为索引，表的大小与虚拟地址空间成正比。反向页表（Inverted Page Table）采用相反的思路：以物理页框号为索引，表的大小与物理内存成正比。

反向页表的每一项记录：哪个进程的哪个虚拟页当前占用该物理页框。查找时，给定虚拟地址和进程标识，需要搜索整个反向页表找到匹配项。为了加速搜索，通常使用哈希表：

$$\text{哈希索引} = h(\text{进程 ID}, \text{VPN}) \quad (19)$$

由于哈希冲突，每个哈希桶可能包含多个表项，需要链式查找。

反向页表特别适合 64 位系统。在 64 位系统中，虚拟地址空间高达 2^{64} 字节，即使使用多级页表，表的规模也难以接受。而反向页表的大小只取决于物理内存容量——如果物理内存只有 16 GB (2^{34} 字节)，反向页表只需约 2^{22} 个表项（假设 4 KiB 页面），这是完全可行的。

5.4.4 页面置换策略

当物理内存已满而需要调入新页时，操作系统必须选择一个页面驱逐到磁盘。由于缺页代价极高，选择合适的置换策略至关重要。

最优策略（OPT）是驱逐未来最长时间不会被访问的页面。但由于无法预知未来，这只能作为理论基准。LRU 策略驱逐最近最少使用的页面，基于时间局部性假设——最近未被使用的页面将来也不太可能被使用。LRU 通常效果接近最优，但需要记录每次访问的时间戳或维护访问顺序链表。

5.5 TLB

页表存储在主存中，每次地址转换都需要访问页表。对于多级页表，一次地址转换可能需要多次内存访问（三级页表需要 3 次），这将严重拖慢程序执行速度。TLB（Translation Lookaside Buffer，转换后备缓冲区）是页表的硬件缓存，用于加速地址转换。

5.5.1 基本原理

TLB 缓存最近使用过的页表项。每个 TLB 表项包含：VPN、PPN、有效位、访问权限位，以及 ASID（Address Space Identifier，地址空间标识符，用于区分不同进程的页表项）。当处理器访问某个虚拟地址时，MMU 首先在 TLB 中查找对应的 VPN。若命中，直接获得 PPN，无需访问页表；若缺失，则访问页表获取映射，并将结果填入 TLB。

由于 TLB 容量很小（通常只有几十到几百项），它通常采用全相联或高相联度的组相联结构，以最大化命中率。全相联 TLB 需要并行比较所有表项的 VPN，但由于表项数量有限，硬件开销可以接受。

例：假设 TLB 有 64 项，采用全相联结构。处理器访问虚拟地址 0x00401234，页大小 4 KiB。VPN 为 0x00401，页内偏移为 0x234。TLB 控制器将 0x00401 与所有 64 项的 VPN 并行比较。若第 17 项匹配且有效，则读取该项的 PPN（假设为 0x12345），物理地址为 0x12345234。

TLB 的命中率直接影响系统性能。幸运的是，由于程序访问的局部性，TLB 的命中率通常可以达到 99% 以上。一个 64 项的 TLB 配合 4 KiB 页面，可以覆盖 256 KiB 的工作集——对于大多数程序的热点代码和数据已经足够。

6 计算机并行

在前面的章节中，我们学习了如何通过流水线、缓存等技术提升单个处理器的性能。然而，随着晶体管尺寸接近物理极限，单纯依靠提高时钟频率来提升性能变得越来越困难。本章将介绍另一条提升计算机性能的道路——并行计算。

6.1 引言

6.1.1 功耗墙与并行的必要性

长期以来，处理器性能的提升主要依赖于时钟频率的增加。然而，功耗与频率的关系使这一策略遇到了瓶颈。动态功耗的计算公式为：

$$P_{\text{动态}} = C \times V^2 \times f \quad (20)$$

其中 C 是负载电容， V 是工作电压， f 是时钟频率。

由于电压与频率正相关（提高频率需要提高电压以保证信号稳定），功耗实际上与频率的三次方近似成正比。这意味着频率翻倍，功耗将增加约 8 倍。这就是所谓的“功耗墙”（Power Wall）。

面对功耗墙的限制，处理器设计者转向了另一种策略：在相同功耗预算下，使用多个较低频率的核心代替单个高频核心。

例：假设单核处理器频率为 4 GHz，功耗为 100 W。如果将频率降至 2 GHz，功耗约降为 $100 \times (2/4)^3 = 12.5$ W。这意味着在 100 W 的功耗预算下，可以放置约 8 个 2 GHz 的核心。

6.1.2 并行的层次

并行可以在多个层次上实现：

- **指令级并行 (ILP, Instruction-Level Parallelism)**：在单个处理器内部，通过流水线、超标量、乱序执行等技术同时执行多条指令。这在第 4 章已有介绍。
- **数据级并行 (DLP, Data-Level Parallelism)**：对大量数据同时执行相同的操作，如向量处理和 SIMD 指令。
- **线程级并行 (TLP, Thread-Level Parallelism)**：多个线程同时执行不同的任务，可通过多核处理器或硬件多线程实现。
- **请求级并行 (RLP, Request-Level Parallelism)**：在服务器和数据中心中，同时处理多个独立的用户请求。

6.2 并行处理程序的困难

编写高效的并行程序比串行程序困难得多。本节介绍并行编程面临的主要挑战。

6.2.1 Amdahl 定律的限制

在第 1 章中，我们介绍了 Amdahl 定律。对于并行计算，该定律同样适用。假设程序中可并行化的比例为 F_p ，使用 n 个处理器时的加速比为：

$$\text{加速比} = \frac{1}{(1 - F_p) + \frac{F_p}{n}} \quad (21)$$

当处理器数量趋向无穷大时，加速比的上限为：

$$\lim_{n \rightarrow \infty} \text{加速比} = \frac{1}{1 - F_p} \quad (22)$$

例：若程序中 90% 的部分可以并行化 ($F_p = 0.9$)，则无论使用多少处理器，加速比的上限仅为 $1/(1 - 0.9) = 10$ 倍。

Amdahl 定律揭示了一个残酷的现实：程序中的串行部分会成为并行加速的瓶颈。即使只有 1% 的串行代码，加速比也不可能超过 100 倍。因此，优化串行部分与增加处理器数量同样重要。

6.2.2 负载均衡

负载均衡 (Load Balancing) 是指将计算任务均匀分配给各个处理器。如果某些处理器的任务量远大于其他处理器，那么快完成任务的处理器将空闲等待，导致整体效率下降。

例：将 100 个任务分配给 4 个处理器。若任务耗时不均，处理器 1 分到 10 个耗时 1 秒的任务，处理器 2 分到 30 个耗时 0.1 秒的任务，则处理器 2 仅需 3 秒完成，但必须等待处理器 1 完成其 10 秒的工作。

6.2.3 同步开销

当多个线程需要访问共享数据时，必须使用同步机制（如锁、屏障）来保证正确性。然而，同步操作本身会带来开销：

- 获取和释放锁需要时间
- 等待锁的线程会阻塞
- 过多的同步会降低并行度

6.3 SISD、MIMD、SIMD、SPMD 与向量处理

Michael Flynn 于 1966 年提出了一种根据指令流和数据流对计算机体系结构进行分类的方法，称为 Flynn 分类法 (Flynn's Taxonomy)。

6.3.1 Flynn 分类法

	单数据流	多数据流
单指令流	SISD	SIMD
多指令流	MISD	MIMD

表 8: Flynn 分类法

SISD SISD (Single Instruction, Single Data, 单指令单数据) 是传统的单处理器架构。在任意时刻，只有一条指令在一个数据上执行。这是我们在前几章中主要讨论的架构。

SIMD SIMD (Single Instruction, Multiple Data, 单指令多数据) 架构使用一条指令同时对多个数据进行操作。这种架构特别适合处理数组、矩阵等规则数据结构。

例：对两个长度为 4 的向量 $\mathbf{A} = [a_1, a_2, a_3, a_4]$ 和 $\mathbf{B} = [b_1, b_2, b_3, b_4]$ 进行加法。SISD 需要 4 条加法指令，而 SIMD 只需 1 条向量加法指令即可同时完成全部 4 次加法。

现代处理器普遍支持 SIMD 扩展指令集，如 Intel 的 SSE/AVX、ARM 的 NEON 等。RISC-V 也定义了向量扩展 (V 扩展) 来支持 SIMD 操作。

MISD MISD (Multiple Instruction, Single Data, 多指令单数据) 指多条指令对同一数据进行操作。这种架构在实际中很少见，主要用于容错系统（多个处理器对同一输入进行计算并比较结果）。

MIMD MIMD (Multiple Instruction, Multiple Data, 多指令多数据) 是最通用的并行架构。每个处理器可以独立执行不同的指令，处理不同的数据。多核处理器和计算机集群都属于 MIMD 架构。

6.3.2 SPMD 编程模型

SPMD (Single Program, Multiple Data, 单程序多数据) 是一种在 MIMD 硬件上广泛使用的编程模型。所有处理器执行相同的程序，但每个处理器根据其编号 (ID) 处理不同的数据分区。

例：假设有 4 个处理器要计算数组 $\mathbf{A}[0:999]$ 的和。在 SPMD 模型下，每个处理器执行相同的程序，但处理器 i 只计算 $\mathbf{A}[250*i : 250*(i+1)-1]$ 的部分和，最后再将 4 个部分和汇总。

6.3.3 向量处理器

向量处理器是 SIMD 的一种经典实现。与标量处理器每次处理一个数据不同，向量处理器使用向量寄存器存储多个数据元素，并使用向量指令一次处理整个向量。

向量处理器的主要优势包括：

- 减少指令获取和译码的开销
- 利用数据局部性优化内存访问

- 隐藏内存延迟

6.4 硬件多线程

硬件多线程 (Hardware Multithreading) 是一种在单个处理器核心上同时维护多个线程上下文的技术。当一个线程因等待内存访问等原因而停顿时，处理器可以切换到另一个线程继续执行，从而提高处理器的利用率。

6.4.1 细粒度多线程

细粒度多线程 (Fine-Grained Multithreading) 在每个时钟周期都切换线程。处理器以轮询的方式依次执行各个线程的指令。

硬件上，细粒度多线程需要为每个线程维护独立的程序计数器 (PC) 和寄存器文件，或者为寄存器文件增加额外的读写端口。

优点：能够有效隐藏流水线停顿，保持流水线满载。

缺点：单个线程的执行速度降低，因为即使没有停顿也要让出时钟周期给其他线程。

6.4.2 粗粒度多线程

粗粒度多线程 (Coarse-Grained Multithreading) 只在当前线程发生长延迟事件 (如缓存缺失) 时才切换线程。

优点：单个线程的执行效率更高。

缺点：线程切换有一定延迟，不能完全隐藏短延迟停顿。

6.4.3 同时多线程

同时多线程 (SMT, Simultaneous Multithreading) 结合了超标量和多线程技术。在每个时钟周期内，处理器可以从多个线程中选择指令发射到不同的功能单元，从而最大化功能单元的利用率。

例：Intel 的超线程技术 (Hyper-Threading) 就是 SMT 的一种实现。一个物理核心可以同时执行两个线程，操作系统会将其识别为两个逻辑处理器。

SMT 的核心思想是利用超标量处理器中未被充分利用的执行资源。当一个线程由于数据依赖或缓存缺失而无法发射指令时，另一个线程的指令可以填补这些空闲的执行单元。

6.5 多核与共享内存多处理器

多核处理器 (Multicore Processor) 在单个芯片上集成多个独立的处理器核心。每个核心可以独立执行指令，核心之间通过共享的缓存或片上网络进行通信。

6.5.1 共享内存架构

在共享内存多处理器（Shared Memory Multiprocessor）中，所有处理器共享同一个物理内存地址空间。根据内存访问时间是否均匀，可分为两类：

UMA UMA (Uniform Memory Access, 统一内存访问) 架构中，所有处理器访问内存的延迟相同。这种架构也称为 SMP (Symmetric Multiprocessing, 对称多处理)。

NUMA NUMA (Non-Uniform Memory Access, 非统一内存访问) 架构中，每个处理器有自己的本地内存，访问本地内存比访问远程内存更快。现代多路服务器通常采用 NUMA 架构。

6.5.2 缓存一致性

在共享内存多处理器中，每个核心通常有自己的私有缓存。当多个核心缓存了同一内存地址的数据时，需要保证缓存之间的一致性。这在第 5 章的 MESI 协议部分已有介绍。

6.5.3 同步原语

为了协调多个线程对共享数据的访问，需要使用同步原语。最基本的同步原语是锁（Lock），其实现依赖于原子操作。

RISC-V 提供了两种原子操作指令来支持同步：

- **lr.w** (Load-Reserved Word): 从内存加载数据并设置保留标记
- **sc.w** (Store-Conditional Word): 条件存储，仅当保留标记有效时才写入

例：使用 lr.w/sc.w 实现自旋锁：

```

1 spin:    lr.w      t0, (a0)          # 加载锁变量并保留
2           bnez     t0, spin        # 若已锁定则重试
3           li       t1, 1          # 伪指令, load immediate, 等价于 addi t1, zero, 1
4           sc.w      t1, t1, (a0)    # 尝试设置锁
5           bnez     t1, spin        # 若存储失败则重试
6           # 临界区代码
7           sw       zero, (a0)      # 释放锁

```

6.6 GPU 简介

图形处理单元（GPU, Graphics Processing Unit）最初设计用于图形渲染，但其大规模并行的架构使其成为通用计算的有力工具。这种将 GPU 用于非图形计算的方式称为 GPGPU (General-Purpose computing on GPU, 通用 GPU 计算)。

6.6.1 GPU 与 CPU 的架构差异

GPU 和 CPU 的设计理念有根本差异：

特性	CPU	GPU
核心数量	几个到几十个	数千个
单核性能	高	低
适合任务	串行、分支复杂	并行、数据密集
内存带宽	较低	很高

表 9: CPU 与 GPU 架构对比

6.6.2 GPU 编程模型

以 NVIDIA 的 CUDA (Compute Unified Device Architecture, 统一计算设备架构) 为例, GPU 程序的执行遵循以下层次结构：

- **线程 (Thread)**: 最基本的执行单元, 每个线程执行相同的代码 (称为内核, Kernel)
- **线程块 (Block)**: 一组线程, 同一块内的线程可以共享内存并同步
- **网格 (Grid)**: 一组线程块, 构成完整的并行任务

GPU 的执行模型本质上是 SIMT (Single Instruction, Multiple Threads, 单指令多线程)。一组线程 (称为 warp, 通常 32 个线程) 在同一时刻执行相同的指令。当遇到分支时, 如果线程发生分歧, 则需要串行执行两个分支路径, 这会显著降低效率。因此, GPU 更适合分支较少的规则计算。

6.7 多处理器基准测试与性能模型

评估并行系统性能需要专门的方法论和基准测试。

6.7.1 强比例缩放与弱比例缩放

强比例缩放 强比例缩放 (Strong Scaling) 衡量在问题规模固定的情况下, 增加处理器数量能带来多少加速。理想情况下, 处理器数量翻倍, 执行时间减半。

弱比例缩放 弱比例缩放 (Weak Scaling) 衡量在每个处理器的负载固定的情况下, 增加处理器数量是否能保持恒定的执行时间。理想情况下, 处理器数量和问题规模同比例增长时, 执行时间不变。

6.7.2 Roofline 模型

Roofline 模型是一种直观的性能分析工具，用于评估程序的计算效率和内存带宽利用情况。

模型的核心是算术强度 (Arithmetic Intensity)，定义为：

$$\text{算术强度} = \frac{\text{浮点运算次数}}{\text{内存访问字节数}} \quad (\text{单位: FLOP/Byte}) \quad (23)$$

程序的最大可达性能受两个因素限制：

$$\text{可达性能} = \min(\text{峰值计算性能}, \text{内存带宽} \times \text{算术强度}) \quad (24)$$

Roofline 模型将程序分为两类：计算受限（算术强度高）和内存受限（算术强度低）。对于内存受限的程序，增加计算能力无济于事，需要优化内存访问；对于计算受限的程序，则应关注如何更好地利用计算资源。

6.8 谬误与陷阱

谬误：Amdahl 定律意味着并行计算终将失效。

虽然 Amdahl 定律确实限制了固定规模问题的加速比，但 Gustafson 定律 (Gustafson's Law) 提供了另一个视角：随着处理器数量增加，人们往往会求解更大规模的问题。在弱比例缩放场景下，并行计算仍然能够带来巨大的价值。

谬误：峰值性能是评估并行系统的最佳指标。

峰值性能只是理论上限，实际应用很难达到。更有意义的指标是持续性能 (Sustained Performance) 和实际应用的执行时间。

陷阱：忽视通信开销。

在并行编程中，通信开销常常被低估。频繁的同步和数据交换可能会抵消并行带来的性能提升。设计并行算法时，应尽量减少通信次数，增大计算与通信的比率。

陷阱：过度同步。

使用过多的锁和屏障会序列化程序执行，降低并行度。应该仔细分析数据依赖关系，只在必要时进行同步。

A 记忆指令结构

记下所有指令是极其困难的，对指令进行分类是有必要的，尽管 RISC-V 的设计者已经为我们对指令进行了简单的分类，但是许多指令名称对于初学者（尤其是非英语母语者）来说依旧难以理解。在本小节，我们会将有关联的指令进行成对记忆，帮助读者理解。

首先，先来考察 R (Register) 型指令，这类指令都是针对寄存器进行操作的指令。这类指令通常好理解，是因为他们的名字往往直接反映了指令的功能。例如，`add` 指令表示加法，而 `sub` 指令表示减法。但是，左移、右移等指令的名称不太直观，这里引入这些指令的英文原名来辅助记忆：

指令	英文全称	功能描述
<code>sll</code>	Shift Left Logical (逻辑左移)	$R[rd] = R[rs1] \ll R[rs2]$
<code>srl</code>	Shift Right Logical (逻辑右移)	$R[rd] = R[rs1] \gg R[rs2]$
<code>sra</code>	Shift Right Arithmetic (算术右移)	$R[rd] = R[rs1] \ggg R[rs2]$
<code>slt</code>	Set Less Than (小于则置位)	$R[rd] = (R[rs1] < R[rs2]) ? 1 : 0$

表 10: R 型移位与比较指令

逻辑移位默认高低位补零，而算术右移则保持符号位不变。对补码正数来说，两者效果相同；对补码负数来说，逻辑右移高位补零，算术右移高位补符号位。或者说，我们可以简单地将逻辑移位存在的原因归结到我们需要对无符号数进行移位操作。

而这类指令常常带有指令后缀（或者前缀），例如，我们可以看到 `mulh` 这样子的指令，其中的 `h` 表示 `high`（高位），也即保存乘法结果中的高位，再比如一些指令可能会有 `u` 后缀，例如 `sltu`，其中的 `u` 表示 `unsigned`（无符号），也即该指令会将操作数视为无符号数进行比较。

常见的前后缀如下：

前后缀	英文全称	意义	例子
<code>h</code>	<code>High</code> (高位)	操作或保存高位结果	<code>mulh</code> (乘法高位)、 <code>mulhu</code>
<code>u</code>	<code>Unsigned</code> (无符号)	按无符号数处理	<code>sltu</code> 、 <code>mulhu</code>
<code>f.s</code>	<code>Float Single</code> (浮点单精度)	单精度浮点运算	<code>fadd.s</code> 、 <code>fmul.s</code>
<code>f.d</code>	<code>Float Double</code> (浮点双精度)	双精度浮点运算	<code>fadd.d</code> 、 <code>fmul.d</code>
<code>amo</code>	<code>Atomic Memory Operation</code> (原子操作)	原子内存操作	<code>amoadd.w</code> 、 <code>amoswap.d</code>

表 11: 常见指令前后缀

如果我们将一些 R 型指令中第二个操作数替换为立即数，我们就得到了 I 型指令，为此我们需要添加后缀 `i` (Immediate)。例如，`addi` ($R[rd] = R[rs1] + imm$) 指令就是将 `add` 指令的第二个操作数替换为立即数后的结果。

RISC-V 指令集采用 `[op] [rd], [rs1], [rs2/imm]` 的格式来表示指令操作数，一般来说，你可以简单的用等号与 `op` 连接操作数即可（采用了符合数学习惯的顺序），例如 `add R[1], R[2],`

$R[3]$ 可以表示为 $R[1] = R[2] + R[3]$ 。

除了上述这些计算指令, RISC-V 还提供了一些用于数据传送 (将数据在不同的存储结构中传递) 的指令, 例如 `lw` (load word) 和 `sw` (store word) 指令。我们不仅可以以字为单位进行读写, 还可以以字节 (后缀 `b`, byte)、半字 (后缀 `h`, halfword) 为单位进行读写。一般来讲, 加载指令因为只需要给出内存地址 (一种立即数) 和用于保存的 `rd` 结果寄存器, 因此是 I 型指令, 而存储指令需要给出内存地址和用于存储数据的 `rs` 源寄存器, 因此是 S (SB) 型指令。

除了一般的数据传送指令, RISC-V 还提供了一些原子内存操作指令, 有 `lrc.w`, `lrc.d`, `sc.w`, `sc.d`。这里面的 `lrc` 表示 Load Reserved (加载保留), `sc` 表示 Store Conditional (条件存储)。这些指令用于多处理器环境下的同步操作, 确保在并发访问内存时数据的一致性。具体内容见第 2.5 节。

除去这些指令, 剩下的就是跳转指令, 跳转指令总共有 I、B 和 J 型三种, 其中 B 型指令 (branch) 用于条件跳转。J 型与 I 型指令用于无条件叫转, 分别有 `jal` 和 `jalr` 两条指令, 后者多出一个 `r`, 表示 register (寄存器), 也即跳转地址由寄存器提供。

分支 (Branch), 即有两种选择, 因此要进行条件判断进而确定是否跳转; 跳转 (Jump), 则是直接跳转到指定的位置

B 型指令则有多种跳转指令, 但无论怎么组合, b 后面的后缀类型总是有限的, 见下表:

后缀	英文全称	含义
<code>e(q)</code>	Equal (相等)	当 <code>rs1 == rs2</code> 时跳转
<code>ne</code>	Not Equal (不等)	当 <code>rs1 != rs2</code> 时跳转
<code>l(t)</code>	Less Than (小于)	当 <code>rs1 < rs2</code> 时跳转 (有符号比较)
<code>g(t)</code>	Greater Than (大于)	当 <code>rs1 >= rs2</code> 时跳转 (有符号比较)

表 12: B 型指令后缀

这些后缀是可以进行组合的, 比如 `bge` 表示 Branch if Greater than or Equal (大于等于), 即当 `rs1 >= rs2` 时跳转。

B 链接与重定向

在程序的编译过程中, 链接 (Linking) 是将多个目标文件和库文件组合成可执行文件的过程。链接器 (Linker) 需要解决符号引用、分配内存地址等问题。在 RISC-V 架构中, 由于指令长度固定为 32 位, 某些操作需要特殊处理。

B.1 地址重定位

当程序引用全局变量或函数时, 编译器在编译阶段无法确定这些符号的最终地址。链接器在链接时会进行重定位 (Relocation), 将符号引用替换为实际地址。

对于 32 位地址的加载, RISC-V 使用 `lui` (Load Upper Immediate, 加载高位立即数) 和 `addi` 指令的组合:

- `lui rd, imm`: 将 20 位立即数加载到寄存器 `rd` 的高 20 位，低 12 位清零
- `addi rd, rd, imm`: 将 12 位立即数加到 `rd` 上

例如，要加载地址 0x12345678 到寄存器 `x1`:

```
1 lui x1, 0x12345      # x1 = 0x12345000
2 addi x1, x1, 0x678    # x1 = 0x12345678
```

需要注意的是，当低 12 位的立即数是负数（最高位为 1）时，`addi` 会进行符号扩展，导致减法操作。因此链接器在处理时需要对 `lui` 的立即数进行调整（加 1），以补偿这个符号扩展效果。

例如，要加载地址 0x12345800 到寄存器 `x2`:

```
1 lui x2, 0x12346      # x2 = 0x12346000
2 addi x2, x2, -0x800   # x2 = 0x12346000 + (-0x800) = 0x12345800
```

B.2 PC 相对寻址的重定向

对于函数调用和分支跳转，RISC-V 使用 PC 相对寻址。链接器需要计算目标地址与当前 PC 的偏移量，并将其编码到指令中。

例如，`auipc` (Add Upper Immediate to PC, PC 加立即数高位) 指令常与 `jalr` 配合使用来实现远距离调用：

```
1 auipc x1, offset_high    # x1 = PC + (offset_high << 12)
2 jalr x1, offset_low(x1)  # 跳转到 x1 + offset_low
```

`auipc` 指令将 20 位立即数左移 12 位后加到当前 PC 上，结果保存到目标寄存器。这使得程序可以访问相对于 PC 的 ± 2 GB 范围内的地址，实现位置无关代码 (PIC, Position Independent Code)。

C 其他复杂的算术运算

本附录介绍一些更复杂的算术运算算法，包括快速乘法和除法。这些算法在硬件实现中被广泛使用，以提高运算效率。

C.1 快速乘法算法：Booth 算法

在第三章中，我们介绍了基本的移位-加法乘法算法，它需要 n 个时钟周期来完成 n 位乘法。**Booth 算法**是一种更高效的乘法算法，通过减少部分积的数量来加速乘法运算。

C.1.1 Booth 算法原理

Booth 算法的核心思想是：观察乘数中连续的 1 序列，可以将多次加法转换为一次减法和一次加法。

例如，考虑乘数 $B = 0111_2 = 7$ 。在基本算法中，我们需要计算：

$$A \times 7 = A \times (4 + 2 + 1) = A \times 2^2 + A \times 2^1 + A \times 2^0$$

即三次加法。但我们也同样可以这样计算：

$$A \times 7 = A \times (8 - 1) = A \times 2^3 - A \times 2^0$$

即一次加法和一次减法。

Booth 算法通过扫描乘数的相邻两位来决定操作：

b_i	b_{i-1}	Booth 编码	操作
0	0	0	无操作（连续 0 的中间）
0	1	+1	加被乘数（连续 1 的结束）
1	0	-1	减被乘数（连续 1 的开始）
1	1	0	无操作（连续 1 的中间）

表 13: Booth 算法编码规则

用数学公式表示：

$$B_i = b_{i-1} - b_i \quad (25)$$

其中 $b_{-1} = 0$ （乘数最低位右边虚拟添加一个 0）。

Booth 算法的优势在于：对于含有连续 1 的乘数，它可以减少加法操作的次数。在最好情况下（乘数为交替的 01 模式），操作次数与基本算法相同；在平均情况下，操作次数约为基本算法的一半。

C.2 除法算法

除法是四则运算中最复杂的一种。与乘法类似，除法也可以通过移位和减法来实现。

C.2.1 恢复余数除法

恢复余数除法 (Restoring Division) 是最基本的除法算法，其思想是：尝试用除数去减被除数的一部分，如果结果为负，则恢复原值并商 0；如果结果为非负，则保留结果并商 1。

C.2.2 不恢复余数除法

不恢复余数除法 (Non-Restoring Division) 是恢复余数除法的优化版本。其核心思想是：当减法结果为负时，不立即恢复，而是在下一步执行加法来补偿。

优化原理：在恢复余数除法中，当结果为负时，我们执行“恢复”操作（加回除数），然后在下一轮左移后再减。不恢复余数除法观察到：

$$2 \times (R + D) - D = 2R + D$$

其中 R 是负的部分余数， D 是除数。也就是说，我们可以跳过恢复步骤，直接在下一轮执行加法。

在 RISC-V 中，除法指令包括：

- `div rd, rs1, rs2`: 有符号除法，商存入 `rd`
- `divu rd, rs1, rs2`: 无符号除法，商存入 `rd`
- `rem rd, rs1, rs2`: 有符号除法，余数存入 `rd`
- `remu rd, rs1, rs2`: 无符号除法，余数存入 `rd`

除法是所有基本算术运算中最慢的。即使使用优化的算法，除法通常也需要比乘法更多的时钟周期。因此，在编写高性能代码时，程序员通常会尽量避免除法，或者用移位操作（对于 2 的幂次除法）来替代。