RISC-V 计算机组成与体系结构入门讲义

第一章：计算机的抽象层次与指令集概述

我们每天使用的计算机，从高级编程语言到硬件执行，中间经历了层层**抽象**。理解这些层次是理解计算机工作原理的关键。

1.1 计算机的抽象层次

计算机系统被设计成一系列的“层次”，每一层都在其下层的基础上构建，并隐藏了下层的复杂细节：

1. **高级语言程序 (High-Level Language Program)**：如C/C++，最接近人类思维。
   - **示例 (C语言)**： `temp = v[k]; v[k] = v[k+1]; v[k+1] = temp;`
2. **编译器 (Compiler)**：将高级语言翻译成汇编语言。
3. **汇编语言程序 (Assembly Language Program)**：低级语言，使用助记符表示机器指令。
   - **示例 (RISC-V 汇编)**：

```
lw x3, 0(x10)     # load word (加载字)
lw x4, 4(x10)
sw x4, 0(x10)     # store word (存储字)
sw x3, 4(x10)
```

4. **汇编器 (Assembler)**：将汇编语言翻译成机器语言。
5. **机器语言程序 (Machine Language Program)**：计算机唯一能直接执行的二进制指令序列（0和1）。
6. **硬件体系结构描述**：描述CPU、内存等硬件部件的设计和互联。
7. **逻辑电路描述**：描述底层的逻辑门电路。
8. **体系结构实现**：具体的物理电路实现。

通过这种分层，我们可以专注于当前层的问题，而不必同时处理所有复杂性。

1.2 指令集体系结构 (ISA) 概述

ISA是CPU能理解和执行的所有指令的集合，定义了CPU的“词汇表”和“语法规则”。

- **指令 (Instructions)**：CPU能执行的最基本操作，对应汇编语言中的一行代码。
- **RISC vs. CISC**：
  - **CISC (Complex Instruction Set Computing)**：指令集庞大复杂，一条指令可以完成多项任务。硬件实现复杂，执行速度可能受限。
  - **RISC (Reduced Instruction Set Computing)**：指令集精简，每条指令只完成最基本的操作。硬件实现简单高效，更利于流水线并行处理。RISC哲学最终取得了成功。

- **选择 RISC-V**：
  - **开放性与免许可**：RISC-V是一个开放标准，任何人都可以免费使用和实现。
  - **简洁性**：设计优雅简洁，非常适合教学和初学者理解核心原理，避免不必要的复杂性。
  - **课程使用**：本课程将使用RISC-V的32位版本（RV32），其中一个指令字长为32位（4字节）。

第二章：RISC-V 寄存器与基本算术操作

2.1 寄存器：CPU 的“极速存储”

寄存器是CPU内部数量有限但速度极快的存储单元。所有CPU操作（如加法、减法）都必须在寄存器上进行。

- **特性**：
  - **速度**：比主内存快 50-500 倍。
  - **数量**：RV32 有 32 个通用寄存器。
  - **大小**：每个寄存器可存储 32 位（4 字节）数据，这 32 位数据被称为一个“字”（Word）。
  - **无类型**：寄存器本身不区分数据类型，指令决定如何解释其内容。
- **RISC-V 寄存器编号与常用命名约定 (ABI Names)**：
  为提高代码可读性，除了 `x0-x31` 的硬编号外，寄存器通常有更具语义的名称。

寄存器号	ABI 名称	约定用途
`x0`	`zero`	**零寄存器**：永远为 0，写入无效。
`x1`	`ra`	**返回地址寄存器** (Return Address)： `jal` (跳转并链接) 指令会将下一条指令地址存入此寄存器，供函数返回时使用。
`x2`	`sp`	**栈指针** (Stack Pointer)：指向栈的顶部，用于管理函数调用栈。
`x3`	`gp`	全局指针 (Global Pointer)。
`x4`	`tp`	线程指针 (Thread Pointer)。
`x5-x7`	`t0-t2`	**临时寄存器** (Temporaries)：函数调用时无需保存，可以被自由覆盖。
`x8`	`s0` / `fp`	**保存寄存器** (Saved) / 帧指针 (Frame Pointer)： 需要被调用者保存和恢复。 `s0` 也常被用作帧指针。
`x9`	`s1`	**保存寄存器** (Saved)。
`x10-x17`	`a0-a7`	**函数参数/返回值寄存器** (Arguments / Return Values)： `a0-a7` 用于传递函数参数， `a0` / `a1` 也用于存储函数返回值。

寄存器号	ABI 名称	约定用途
x18-x27	s2-s11	**保存寄存器** (Saved)。
x28-x31	t3-t6	**临时寄存器** (Temporaries)。

2.2 基本算术操作指令

RISC-V的算术指令通常遵循 `操作名 目的寄存器，源寄存器1，源寄存器2` 的格式。

- **加法 (`add`)**：
  - **指令**： `add rd, rs1, rs2`
  - **含义**： `rd = rs1 + rs2`
  - **C语言等价**： `f = g + h;`
  - **RISC-V 汇编示例**：

```
# C语言: f = g + h;
add x10, x11, x12  # 假设f在x10, g在x11, h在x12
```

- **减法 (`sub`)**：
  - **指令**： `sub rd, rs1, rs2`
  - **含义**： `rd = rs1 - rs2`
  - **C语言等价**： `f = g - h;`
  - **RISC-V 汇编示例**：

```
# C语言: f = g - h;
sub x10, x11, x12  # 假设f在x10, g在x11, h在x12
```

- **加立即数 (`addi`)**：
  - **指令**： `addi rd, rs1, imm`
  - **含义**： `rd = rs1 + imm` (imm 是一个12位有符号立即数常量)
  - **C语言等价**： `f = g + 10;`
  - **RISC-V 汇编示例**：

```
# C语言: f = g + 10;
addi x10, x11, 10  # 假设f在x10, g在x11
```

  - **没有 `subi` 指令**：RISC-V秉持精简原则。因为 `减去一个数` 等同于 `加上它的负数` ，所以不需要单独的 `subi` 。

```
# C语言: f = g - 10;
addi x10, x11, -10 # 假设f在x10, g在x11
```

第三章：内存与数据传输

寄存器数量有限，大量数据（如数组、复杂结构）存储在**内存**中。内存比寄存器慢，但容量大得多。

3.1 内存组织

- **字节可寻址 (Byte-Addressable)**：内存的最小可寻址单元是**字节**（8位）。
- **字 (Word)**：在RV32中，一个字是 4 个字节（32 位）。字地址通常是 4 的倍数（按字对齐）。
- **大小端 (Endianness)**：指多字节数据在内存中的存储顺序。
  - **大端 (Big-Endian)**：高位字节存储在低地址。
  - **小端 (Little-Endian)**：低位字节存储在低地址。RISC-V 默认采用**小端序**。
  - **示例**：一个字 `0x12345678` (高位字节 `12` ，低位字节 `78`)
    - **大端存储**：地址N: `12` , N+1: `34` , N+2: `56` , N+3: `78`
    - **小端存储**：地址N: `78` , N+1: `56` , N+2: `34` , N+3: `12`

3.2 数据传输指令

这些指令用于在寄存器和内存之间移动数据。它们使用**基址 + 偏移量**的寻址模式。

- **加载字 (`lw` - Load Word)**：将内存中的一个字加载到寄存器。
  - **指令**： `lw rd, offset(rs1)`
  - **含义**： `rd = Memory[rs1 + offset]`
  - `rs1` ： 包含基地址的寄存器。 `offset` ： 一个 12 位有符号立即数，表示偏移量（以字节为单位）。
  - **C语言等价**： `g = h + A[3];`
  - **RISC-V 汇编示例**：

    ```
    # 假设数组A的基地址在x15, h在x12, g在x11
    lw x10, 12(x15)    # 将A[3]加载到x10。12是偏移量，因为A[3]在A[0]后的12个字节处 (3 * 4字节/
    add x11, x12, x10  # g = h + A[3]
    ```

- **存储字 (`sw` - Store Word)**：将寄存器中的一个字存储到内存。
  - **指令**： `sw rs2, offset(rs1)`
  - **含义**： `Memory[rs1 + offset] = rs2`
  - **C语言等价**： `A[10] = h + A[3];`
  - **RISC-V 汇编示例**：

```
# 假设数组A的基地址在x15, h在x12
lw x10, 12(x15)    # 临时寄存器x10 = A[3]
add x10, x12, x10  # x10 = h + A[3]
sw x10, 40(x15)    # A[10] = x10。40是偏移量 (10 * 4字节/字)
```

- **加载字节 (`lb` , `lbu`) 和存储字节 (`sb`)**:
  - `lb rd, offset(rs1)`： 加载一个字节并进行有符号扩展到 32 位。
  - `lbu rd, offset(rs1)`： 加载一个字节并进行无符号扩展到 32 位。
  - `sb rs2, offset(rs1)`： 存储一个字节。

第四章：逻辑运算与移位指令

逻辑指令用于对寄存器中的数据进行位级别的操作。

4.1 逻辑运算指令

- **按位与 (`and` , `andi`)**:
  - `and rd, rs1, rs2`： `rd = rs1 & rs2`
  - `andi rd, rs1, imm`： `rd = rs1 & imm` (常用于位掩码操作)
- **按位或 (`or` , `ori`)**:
  - `or rd, rs1, rs2`： `rd = rs1 | rs2`
  - `ori rd, rs1, imm`： `rd = rs1 | imm`
- **按位异或 (`xor` , `xori`)**:
  - `xor rd, rs1, rs2`： `rd = rs1 ^ rs2`
  - `xori rd, rs1, imm`： `rd = rs1 ^ imm`
- **没有 `NOT` 指令**：在RISC-V中，通过 `xori rd, rs1, -1` （或 `xori rd, rs1, 0xFFFFFFFF` ）可以实现按位非操作，因为任何数与全1异或会得到其按位取反。

4.2 移位指令

- **逻辑左移 (`sll` , `slli`)**:
  - `sll rd, rs1, rs2`： `rd = rs1 << rs2`
  - `slli rd, rs1, imm`： `rd = rs1 << imm`
  - 右侧空出的位用 0 填充。常用于乘以 2 的幂次。
- **逻辑右移 (`srl` , `srli`)**:
  - `srl rd, rs1, rs2`： `rd = rs1 >> rs2`
  - `srli rd, rs1, imm`： `rd = rs1 >> imm`
  - 左侧空出的位用 0 填充。
- **算术右移 (`sra` , `srai`)**:
  - `sra rd, rs1, rs2`： `rd = rs1 >> rs2`

- `srai rd, rs1, imm`： `rd = rs1 >> imm`
- 左侧空出的位用**符号位**（最高有效位）填充。这对于有符号整数的除法运算至关重要。
- **示例**：
  - 正数： `0000 1000` (8) `srai` 1位 -> `0000 0100` (4)
  - 负数： `1111 1000` (-8) `srai` 1位 -> `1111 1100` (-4)。如果使用 `srli` 会变成 `0111 1100`，改变了符号。

第五章：控制流：分支与跳转

控制流指令改变程序执行的顺序，是实现 `if-else` 、循环等结构的关键。

- **程序计数器 (PC)**：CPU内部一个特殊寄存器，存储下一条要执行指令的**字节地址**。通常PC会自动递增 4，指向顺序执行的下一条指令。

5.1 条件分支指令

根据条件判断决定是否跳转。

- `beq rs1, rs2, Label`： **相等则分支** (`Branch if Equal`)。如果 `rs1 == rs2`，PC 跳转到 `Label`。
- `bne rs1, rs2, Label`： **不等则分支** (`Branch if Not Equal`)。如果 `rs1 != rs2`，PC 跳转到 `Label`。
- `blt rs1, rs2, Label`： **小于则分支** (`Branch on Less Than`)。如果 `rs1 < rs2` （有符号），PC 跳转到 `Label`。
- `bge rs1, rs2, Label`： **大于等于则分支** (`Branch on Greater or Equal`)。如果 `rs1 >= rs2` （有符号），PC 跳转到 `Label`。
- `bltu rs1, rs2, Label`： **无符号小于则分支** (`Branch on Less Than Unsigned`)。如果 `rs1 < rs2` （无符号），PC 跳转到 `Label`。
- `bgeu rs1, rs2, Label`： **无符号大于等于则分支** (`Branch on Greater or Equal Unsigned`)。如果 `rs1 >= rs2` （无符号），PC 跳转到 `Label`。
- **注意**：RISC-V 没有 `bgt` (大于) 或 `ble` (小于等于) 等指令。这些可以通过组合现有指令（如反转比较操作数或改变跳转条件）来模拟。

5.2 非条件跳转指令

无条件改变PC值，即无条件跳转。

- `j Label` (伪指令，实际是 `jal zero, Label`)：无条件跳转到 `Label` 处执行。

示例：实现 If-Else 语句

```c
// C语言:
if (i == j) {
    f = g + h;
} else {
    f = g - h;
}
```

```
# RISC-V 汇编
# 假设i在x10, j在x11, f在x12, g在x13, h在x14
bne x10, x11, Else # 如果 i != j，则跳到 Else 标签处
add x12, x13, x14  # f = g + h (当 i == j 时执行)
j Exit             # 跳过 Else 分支，继续执行后续代码

Else:
sub x12, x13, x14  # f = g - h (当 i != j 时执行)

Exit:
# 程序继续执行这里的代码
```

示例：实现 While 循环

```c
// C语言:
while (save[i] == k) {
    i = i + 1;
}
```

```
# RISC-V 汇编
# 假设i在x10, k在x11, 数组save的基地址在x12
Loop:
    slli x13, x10, 2    # 计算i * 4 (因为每个字4字节)
    add x14, x12, x13   # 计算save[i]的内存地址 (基地址 + 偏移量)
    lw x15, 0(x14)      # 加载save[i]的值到x15

    bne x15, x11, Exit  # 如果 save[i] != k, 跳出循环到Exit
    addi x10, x10, 1    # i = i + 1
    j Loop              # 无条件跳转回 Loop 标签处，继续循环

Exit:
# 循环结束后的代码
```

第六章：函数调用与栈

函数（或过程）是结构化编程的关键。函数调用涉及到一套约定来管理参数、返回地址和局部变量。

6.1 函数调用六步法

1. **传递参数**：将参数放入函数可以访问的地方（通常是寄存器，如 `a0-a7`）。
2. **控制转移**：使用 `jal` (Jump and Link) 指令将程序控制权转移给被调用函数。
3. **获取资源**：被调用函数获取所需的本地存储资源（例如，在栈上分配空间）。
4. **执行任务**：被调用函数执行其核心逻辑。
5. **返回结果并恢复**：将返回值放入调用者可访问的地方（如 `a0`），恢复任何被使用的寄存器，并释放本地存储。
6. **返回控制权**：返回到调用函数的原始位置。

6.2 跳转与链接指令

- `jal rd, Label` **(Jump and Link)**：
  - **功能**：跳转到 `Label` 处执行指令，同时将**下一条指令的地址**（即**返回地址**）保存到 `rd` 寄存器中。
  - **约定**：`rd` 通常是 `ra` (即 `x1`)。
  - **示例**：`jal ra, my_function` 会跳转到 `my_function`，并将 `jal` 指令后面的指令地址存入 `ra`。
- `jr rs1` **(Jump Register)**：
  - **功能**：无条件跳转到 `rs1` 寄存器中存储的地址。
  - **约定**：通常用于函数返回，即 `jr ra` 会跳转回 `ra` 中保存的返回地址，回到调用点。

6.3 栈 (Stack)：管理函数调用的利器

当一个函数被调用时，它可能需要：

1. 保存调用方的一些寄存器值，以免在使用中被覆盖。
2. 为自己的局部变量分配存储空间。

栈是一种**后进先出 (LIFO - Last-In, First-Out)** 的数据结构，非常适合管理函数调用。

- **栈指针 (`sp` - Stack Pointer, `x2`)**：专门的寄存器，指向栈的当前“顶部”。
- **栈的增长方向**：RISC-V 中，栈通常从高内存地址向低内存地址增长。
  - **压栈 (Push)**：向栈中放入数据，`sp` 的值会**减小**。
    - **操作**：先 `addi sp, sp, -[size]` 分配空间，再 `sw/sb` 存储数据。
  - **弹栈 (Pop)**：从栈中取出数据，`sp` 的值会**增大**。
    - **操作**：先 `lw/lb` 读取数据，再 `addi sp, sp, [size]` 释放空间。

- **栈帧 (Stack Frame)**：每次函数调用，都会在栈上创建一个新的**栈帧**。一个栈帧通常包含：
  - 返回地址（通常保存在 `ra` 中，但如果函数内部再次调用其他函数， `ra` 的值会被覆盖，因此需要压栈保存）
  - 函数参数（如果参数数量超过寄存器数量）
  - 局部变量
  - 被调用者保存的寄存器值（ `s0-s11` 等）

6.4 寄存器使用约定 (Register Conventions)

为了确保函数之间的正确交互，RISC-V 定义了寄存器的使用约定：

- **调用者保存 (Caller-Saved Registers)**：
  - 如果**调用函数 (Caller)** 期望这些寄存器在被调用函数返回后保持不变，则它必须在调用前将这些寄存器的值保存到栈上，并在返回后恢复。
  - **包括**： `ra` （ `x1` ）、参数寄存器 `a0-a7` （ `x10-x17` ）、临时寄存器 `t0-t6` （ `x5-x7` ， `x28-x31` ）。
- **被调用者保存 (Callee-Saved Registers)**：
  - 如果**被调用函数 (Callee)** 在执行过程中使用了这些寄存器，则它必须在函数入口处将其原始值保存到栈上，并在函数返回前恢复这些值。
  - **包括**：栈指针 `sp` （ `x2` ）、保存寄存器 `s0-s11` （ `x8-x9` ， `x18-x27` ）。

函数调用示例 (C代码与RISC-V汇编)

C 语言函数：

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

RISC-V 汇编实现：

```
# 假设：g在a0，h在a1，i在a2，j在a3。f将使用s0，并需要一个临时寄存器s1。
Leaf:
    # 步骤 3: 获取资源 (在栈上为s0和s1分配空间并保存它们)
    addi sp, sp, -8     # sp = sp - 8 (栈向下增长，分配8字节空间)
    sw s1, 4(sp)        # 将s1的值保存到栈顶上方4字节处
    sw s0, 0(sp)        # 将s0的值保存到栈顶处

    # 步骤 4: 执行任务
    add s0, a0, a1      # s0 = g + h (使用s0存储中间结果)
    add s1, a2, a3      # s1 = i + j (使用s1存储中间结果)
    sub a0, s0, s1      # a0 = s0 - s1 (最终结果存入a0，作为返回值)

    # 步骤 5: 返回结果并恢复 (从栈中恢复s0和s1的值)
    lw s0, 0(sp)        # 从栈顶加载s0的原始值
    lw s1, 4(sp)        # 从栈顶上方4字节处加载s1的原始值
    addi sp, sp, 8      # sp = sp + 8 (释放栈空间)

    # 步骤 6: 返回控制权
    jr ra               # 跳转回ra中保存的返回地址
```

第七章：内存分配区域

当一个C/C++程序运行时，其内存通常被操作系统划分为几个逻辑区域：

1. **代码区 (Text Segment)**：
   - 存放程序的可执行机器代码指令。
   - 通常是只读的，防止程序意外修改自身代码。
   - 在程序加载时被操作系统载入内存。
2. **静态数据区 (Static Data Segment)**：
   - 存放程序中的全局变量和静态变量。
   - 这些变量在程序整个生命周期中都存在。
   - 包括初始化的数据段（如已初始化的全局变量）和未初始化的数据段（BSS段，如未初始化的全局变量，运行时初始化为0）。
   - RISC-V 有约定使用 `gp` (全局指针 `x3`) 指向静态数据区。
3. **堆区 (Heap Segment)**：
   - 用于程序运行时的**动态内存分配**（例如，C语言的 `malloc()` / `free()`，C++的 `new` / `delete`）。
   - 程序运行时可以按需增长（通常向上增长到高地址）和收缩。
   - 由程序员手动管理内存的分配和释放。
4. **栈区 (Stack Segment)**：

- 用于存储函数调用相关的信息：
  - **局部变量**（非静态局部变量）。
  - **函数参数**（如果参数过多，无法全部通过寄存器传递）。
  - **返回地址**（由 `jal` 指令存储在 `ra` 中，如果 `ra` 需要被覆盖，则将其压入栈）。
  - **被调用者保存的寄存器**。
- 通常从高内存地址向低内存地址增长。
- 由编译器和操作系统自动管理内存的分配和释放（通过栈指针 `sp` ）。