计算机组成原理作业 2

毛瑞麟
学号:320230921711
班级：计算机科学与技术 2023 级 1 班

2025 年 5 月 4 日

1

实验环境

实验环境为 Arch Linux x86_64，安装交叉编译工具链的命令如下：

```
sudo pacman -S clang llvm
sudo pacman -S riscv64-unknown-elf-gcc riscv64-unknown-elf-binutils
```

安装完后查看版本并验证支持 RISCV

```
clang --version
clang --print-targets | grep riscv
```

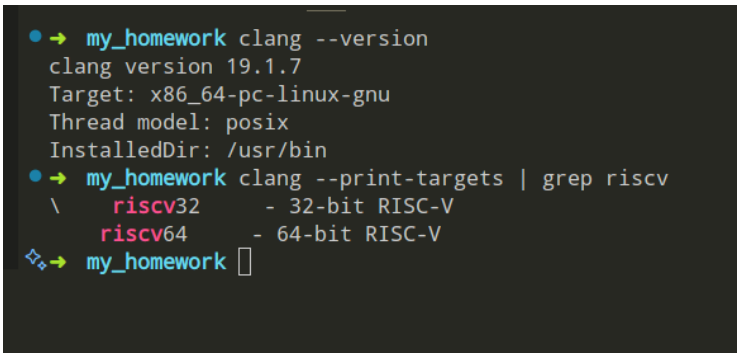

图 1: 安装版本和支持

安装成功

构建工程

构建 code 工程，结构如下：

```
code
  src
     main.c       # 程序的入口点，包含主函数。
     main.s       # 从 main.c 生成的汇编代码。
     main2.s      # 从二进制文件反汇编的代码。
     main.bin     # 为 RISC-V 编译的二进制文件。
  Makefile         # 用于将项目编译为 32 位 RISC-V 汇编的构建规则。
```

- main.s：从 main.c 生成的汇编代码。
- main.bin：为 RISC-V 架构编译的二进制文件。
- main2.s：从二进制文件反汇编的代码。

在 src 文件夹下创建 main.c 文件，写入以下代码：

```c
int main() {
    char string[8] = {'3', 'b', 'a', '?', '#', '@', '<', '8'};
    for (int i = 0; i < 8; i++) {
        char temp = '\0';
```

```
        for (int j = 0; j < 7 - i; j++) {
            if (string[j] <= string[j + 1]) {
                break;
            } else {
                temp = string[j];
                string[j] = string[j + 1];
                string[j + 1] = temp;
            }
        }
    }
    return 0;
}
```

根据要求, 写出 Makefile 文件

```
CC=clang
AS=llvm-mc
OBJDUMP=llvm-objdump
CFLAGS=-S -target riscv32 -march=rv32i -mabi=ilp32 -O0 -g3
SRC=src/main.c
ASM=$(SRC:.c=.s)
BIN=$(SRC:.c=.bin)
DISASM=$(SRC:.c=2.s)

.PHONY: all clean

all: $(ASM) $(BIN) $(DISASM)

$(ASM): $(SRC)
    $(CC) $(CFLAGS) -o $@ $<

$(BIN): $(ASM)
    $(AS) -triple=riscv32 -filetype=obj -o $(BIN) $<
```

```
19
20 $(DISASM): $(BIN)
21     $(OBJDUMP) -d --source $(BIN) > $(DISASM)
22
23 clean:
24     rm -f $(ASM) $(BIN) $(DISASM)
```

生成的 main2.s 有汇编和 16 进制机器码、源代码关系

```
src/main.bin: file format elf32-littleriscv

Disassembly of section .text:

00000000 <main>:
; int main() {
       0: fe010113      addi sp, sp, -0x20
       4: 00112e23      sw ra, 0x1c(sp)
       8: 00812c23      sw s0, 0x18(sp)
       c: 02010413      addi s0, sp, 0x20
      10: 00000513      li a0, 0x0
      14: fea42a23      sw a0, -0xc(s0)
      18: 383c45b7      lui a1, 0x383c4
      1c: 02358593      addi a1, a1, 0x23
;     char string[8] = {'3', 'b', 'a', '?', '#', '@', '<', '8'};
      20: feb42823      sw a1, -0x10(s0)
      24: 3f6165b7      lui a1, 0x3f616
      28: 23358593      addi a1, a1, 0x233
      2c: feb42623      sw a1, -0x14(s0)

00000030 <.Ltmp1>:
;     for(int i = 0; i < 8; i++)
      30: fea42423      sw a0, -0x18(s0)
      34: 0040006f      j 0x38 <.Ltmp1+0x8>
```

```
      38: fe842583      lw a1, -0x18(s0)
      3c: 00700513      li a0, 0x7
      40: 0ab54863      blt a0, a1, 0xf0 <.Ltmp16>
      44: 0040006f      j 0x48 <.Ltmp1+0x18>
      48: 00000513      li a0, 0x0

0000004c <.Ltmp4>:
;         char temp = '\0';
      4c: fea403a3      sb a0, -0x19(s0)

00000050 <.Ltmp5>:
;         for(int j = 0; j < 7 - i; j++)
      50: fea42023      sw a0, -0x20(s0)
      54: 0040006f      j 0x58 <.Ltmp5+0x8>
      58: fe042503      lw a0, -0x20(s0)
      5c: fe842603      lw a2, -0x18(s0)
      60: 00700593      li a1, 0x7
      64: 40c585b3      sub a1, a1, a2
      68: 06b55a63      bge a0, a1, 0xdc <.Ltmp14>
      6c: 0040006f      j 0x70 <.Ltmp5+0x20>
;             if(string[j] <= string[j + 1])
      70: fe042583      lw a1, -0x20(s0)
      74: fec40513      addi a0, s0, -0x14
      78: 00b50533      add a0, a0, a1
      7c: 00054583      lbu a1, 0x0(a0)
      80: 00154503      lbu a0, 0x1(a0)
      84: 00b54663      blt a0, a1, 0x90 <.Ltmp5+0x40>
      88: 0040006f      j 0x8c <.Ltmp5+0x3c>
;                 break;
      8c: 0500006f      j 0xdc <.Ltmp14>
;                 temp = string[j];
      90: fe042503      lw a0, -0x20(s0)
      94: fec40613      addi a2, s0, -0x14
```

```
      98: 00a60533      add a0, a2, a0
      9c: 00054503      lbu a0, 0x0(a0)
      a0: fea403a3      sb a0, -0x19(s0)
;                 string[j] = string[j + 1];
      a4: fe042503      lw a0, -0x20(s0)
      a8: 00a605b3      add a1, a2, a0
      ac: 0015c503      lbu a0, 0x1(a1)
      b0: 00a58023      sb a0, 0x0(a1)
;                 string[j + 1] = temp;
      b4: fe744503      lbu a0, -0x19(s0)
      b8: fe042583      lw a1, -0x20(s0)
      bc: 00c585b3      add a1, a1, a2
      c0: 00a580a3      sb a0, 0x1(a1)
      c4: 0040006f      j 0xc8 <.Ltmp5+0x78>
;         }
      c8: 0040006f      j 0xcc <.Ltmp5+0x7c>
;         for(int j = 0; j < 7 - i; j++)
      cc: fe042503      lw a0, -0x20(s0)
      d0: 00150513      addi a0, a0, 0x1
      d4: fea42023      sw a0, -0x20(s0)
      d8: f81ff06f      j 0x58 <.Ltmp5+0x8>

000000dc <.Ltmp14>:
;     }
      dc: 0040006f      j 0xe0 <.Ltmp15>

000000e0 <.Ltmp15>:
;     for(int i = 0; i < 8; i++)
      e0: fe842503      lw a0, -0x18(s0)
      e4: 00150513      addi a0, a0, 0x1
      e8: fea42423      sw a0, -0x18(s0)
      ec: f4dff06f      j 0x38 <.Ltmp1+0x8>
```

```
000000f0 <.Ltmp16>:
; }
      f0: ff442503      lw a0, -0xc(s0)
      f4: 01c12083      lw ra, 0x1c(sp)
      f8: 01812403      lw s0, 0x18(sp)
      fc: 02010113      addi sp, sp, 0x20
     100: 00008067      ret
```

源程序第四行为:

```
1     char tmp = '\0';
```

对应的汇编为

```
4c: fea403a3      sb a0, -0x19(s0)
```

sb 指令解析

sb 是 RISC-V 的存储指令，表示 **Store Byte**（存储一个字节）。它将一个寄存器中的低 8 位（1 字节）存储到内存中。

- a0 是源寄存器，表示要存储的数据来源。这里是寄存器 a0 的低 8 位。
- -0x19(s0) 是目标内存地址，表示从寄存器 s0 的值减去 0x19（十六进制 25）得到的地址。

这条指令的作用是将寄存器 a0 中的一个字节数据存储到内存地址 [s0 - 0x19] 中。它通常用于将临时变量或局部变量存储到栈帧中的特定位置。

结合 C 语言代码，temp 是一个局部变量，分配在栈上。s0 通常是栈帧指针（frame pointer），-0x19 是 temp 在栈帧中的偏移量。指令 sb 将初始化的值 '\0'（即 0）存储到 temp 的内存位置。源程序第十一行为

```
else
```

对应的汇编为

```
88: 0040006f      j 0x8c <.Ltmp5+0x3c>
```

else 语句对应的汇编解释

在源代码中，else 语句的作用是当条件 string[j] <= string[j+1] 不成立时执行。**指令解析**：

- j 是无条件跳转指令，表示直接跳转到指定的地址。
- 0x8c 是跳转目标地址，表示程序将跳转到标签 .Ltmp5+0x3c 处继续执行。

结合上下文：

- 在之前的条件判断中，blt 指令（位于地址 84）判断 string[j+1] 是否小于 string[j]。
- 如果条件成立，则跳转到 0x90 执行 if 语句的内容。
- 如果条件不成立，则执行当前的 j 指令，跳转到 0x8c，即进入 else 分支。

总结：这条指令的作用是确保在条件判断失败时，程序能够正确跳转到 else 分支的代码位置。

源程序第十五行为

```
1         string[j+1] = temp;
```

```
b4: fe744503      lbu a0, -0x19(s0)
b8: fe042583      lw a1, -0x20(s0)
bc: 00c585b3      add a1, a1, a2
c0: 00a580a3      sb a0, 0x1(a1)
c4: 0040006f      j 0xc8 <.Ltmp5+0x78>
```

string[j+1] = temp 对应的汇编解释

在源代码中，string[j+1] = temp; 的作用是将变量 temp 的值赋给数组 string 的第 j+1 个元素。

指令解析：

- `lbu a0, -0x19(s0)`：从内存地址 `[s0 - 0x19]` 加载一个无符号字节到寄存器 `a0`。这里的 `s0 - 0x19` 是变量 `temp` 在栈帧中的偏移地址。
- `lw a1, -0x20(s0)`：从内存地址 `[s0 - 0x20]` 加载一个字到寄存器 `a1`。这里的 `s0 - 0x20` 是变量 `j` 在栈帧中的偏移地址。
- `add a1, a1, a2`：将寄存器 `a1` 和 `a2` 的值相加，结果存储到 `a1` 中。这里 `a2` 是数组 `string` 的基地址，`a1` 是索引 `j`。
- `sb a0, 0x1(a1)`：将寄存器 `a0` 的低 8 位（即 `temp` 的值）存储到内存地址 `[a1 + 0x1]` 中。这里的 `a1 + 0x1` 是数组 `string[j+1]` 的地址。
- `j 0xc8`：无条件跳转到地址 `0xc8`，继续执行后续代码。

结合上下文：

- 这段汇编代码的作用是将变量 `temp` 的值存储到数组 `string` 的第 `j+1` 个元素中。
- `lbu` 和 `sb` 指令分别用于加载和存储字节数据，确保操作的是单个字符。
- `add` 指令用于计算目标数组元素的地址。

总结：这段汇编代码实现了 C 语言中的赋值语句 `string[j+1] = temp;`，通过加载、地址计算和存储操作完成了对数组元素的更新。

2

选择第一种处理器架构。上述代码中，

```
4c: fea403a3      sb a0, -0x19(s0)
c0: 00a580a3      sb a0, 0x1(a1)
```

为 S 型数据通路, 如下图所示

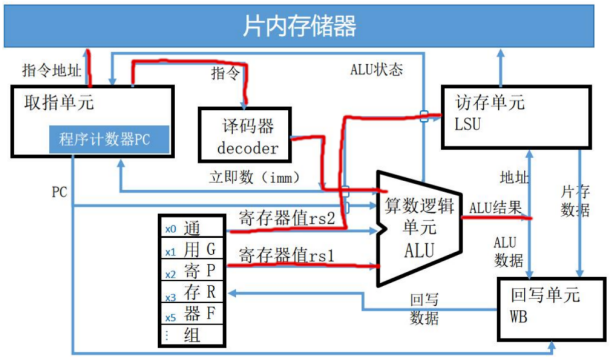

图 2: sb 数据通路

```
b4: fe744503        lbu a0, -0x19(s0)
b8: fe042583        lw a1, -0x20(s0)
```

为 I 型数据通路 (读存储器), 如下图所示

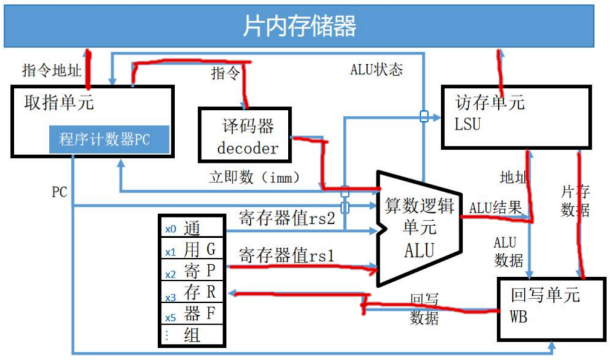

图 3: I 型数据通路 (读存储器)

```
bc: 00c585b3        add a1, a1, a2
```

上述代码是 I 型数据通路 (算术运算), 如下图所示

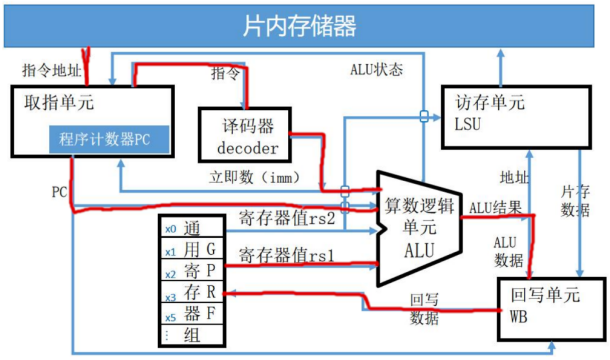

图 4: I 型数据通路 (算术运算)

```
c4: 0040006f       j 0xc8 <.Ltmp5+0x78>
```

是 J 型数据通路, 如下图所示

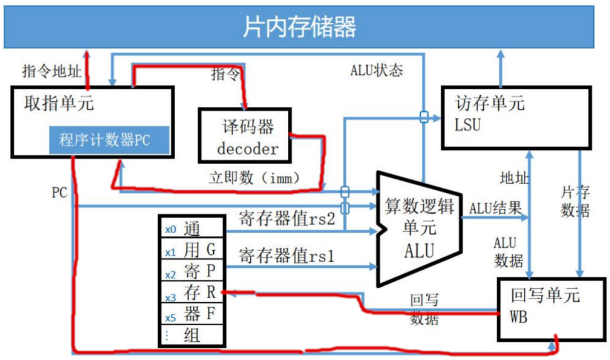

图 5: J 型数据通路