# RISCV 领域加速指令设计与仿真

浙江大学信息与电子工程学院
孙佳科　　22131087@zju.edu.cn

在本次编程训练中，将从软硬件协同设计的思想出发，利用拓展指令实现特定程序的优化。你的任务是根据拓展指令集，完成仿真器编码、译码与执行过程中的拓展内容，从而得到一个功能完整的指令集仿真器。然后，在此基础上，修改汇编代码添加拓展指令，利用仿真器进行仿真，比较拓展指令减少的指令条目，优化性能。

## 一、文件说明

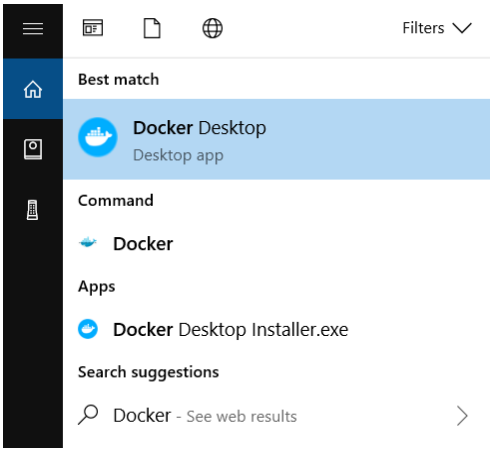
除去本文档外，文件夹下还包括以下文件：

1. Docker.zip：用于构建 docker 镜像的 Dockerfile 以及相关资源。
2. Whisper.tar：构建好的镜像，可以直接导入。（过大，钉钉群提供）
3. SM3 密码杂凑算法.pdf:SM3 密码杂凑算法说明
4. 两份参考文献：《RISC-V 领域加速指令设计与仿真》《RISC-V 领域加速指令设计与仿真》

## 二、建立实验环境

### 2.1 安装 Docker Desktop

1. 前往Docker Desktop 下载Docker Desktop for Windows
2. 下载之后双击 Docker Desktop Installer.exe 开始安装
3. 安装完成后在搜索栏中输入 Docker 点击Docker Desktop 开始运行。




Docker 启动之后会在 Windows 任务栏出现鲸鱼图标。

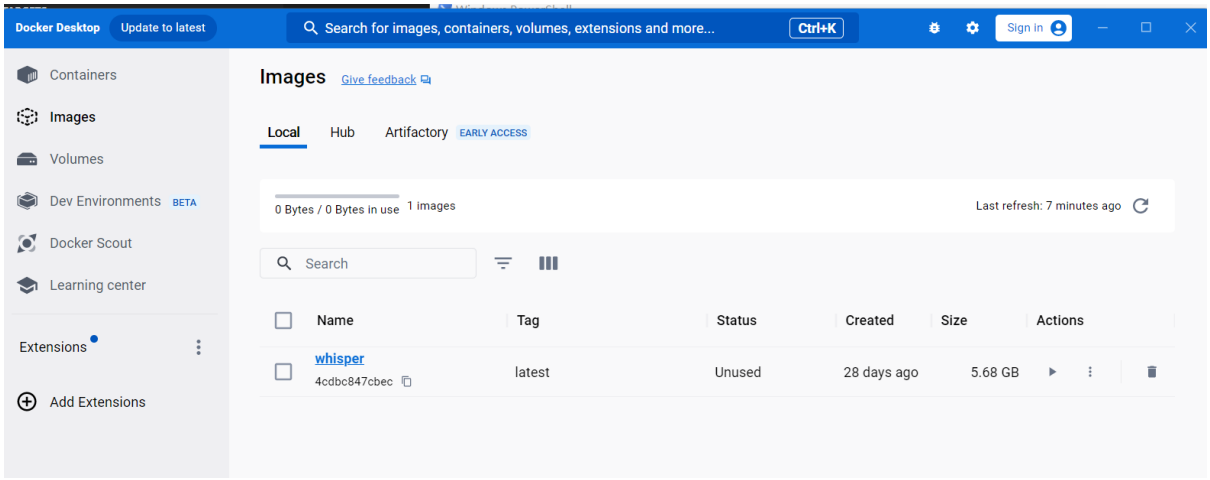
等待片刻，当鲸鱼图标静止时，说明 Docker 启动成功，之后你可以打开 **PowerShell** 使用 Docker。

推荐大家使用 Windows Terminal 使用 Docker。Windows 终端安装 | Microsoft Learn

## 2.2 构建/导入镜像

这一步，我们提供了两种方式，大家可以任选其一进行构建，两种方式最后得到的镜像完全一样。完成后在 Docker Desktop的Images中会有 whisper 镜像。

1. 从Dockerfile 构建
   a) 解压 Docker.zip
   b) 打开 PowerShell / Windows Terminal 并且进入到解压后的文件夹
   c) 在 PowerShell / Windows Terminal 中运行命令: Docker build -t whisper .
      (注意 “whisper” 后面空格再加 “.” 即 whisper .)
   d) 构建过程大约用时 400 s
2. 从Whisper.tar 中导入
   a) 打开 PowerShell / Windows Terminal 进入Whisper.tar 文件所在目录
   b) 运行命令 docker load -i whisper.tar 完成导入



## 2.3 启动容器

完成2.2 中的步骤后，运行: docker run -it --name Name whisper 即可启动容器并且进入容器中 (注意 name前面是两个 -)。

Name 是你启动的容器的名字，可以自定义，列如取容器名为 Cod,则

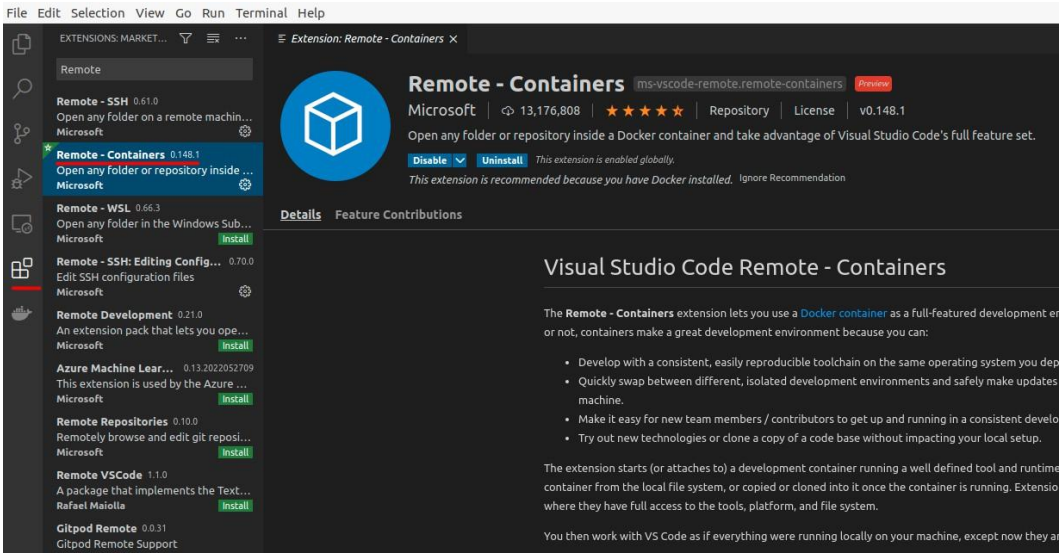
docker run -it --name Cod whisper

**Docker 常用命令：**

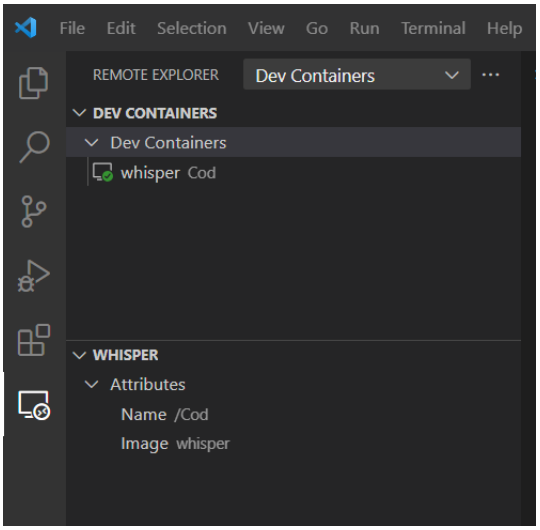
1. **查看目前拥有的容器: docker container ls -la**
2. **查看目前拥有的镜像: docker image ls**
3. **启动容器: docker start name**
4. **关闭容器: docker stop name**
5. **退出容器: Ctrl+A,D**

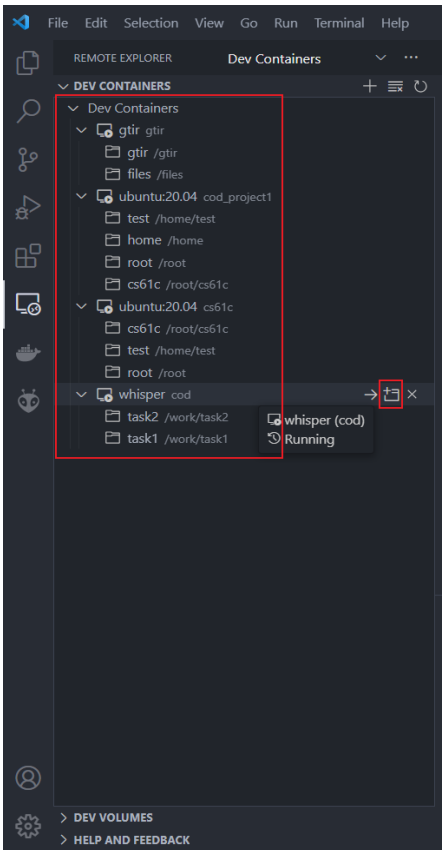
## 2.4使用VS Code 连接容器

1. 从Download Visual Studio Code 下载VS-Code, 并进行安装
2. 安装Remote-container 插件 或者 Dev Containers插件

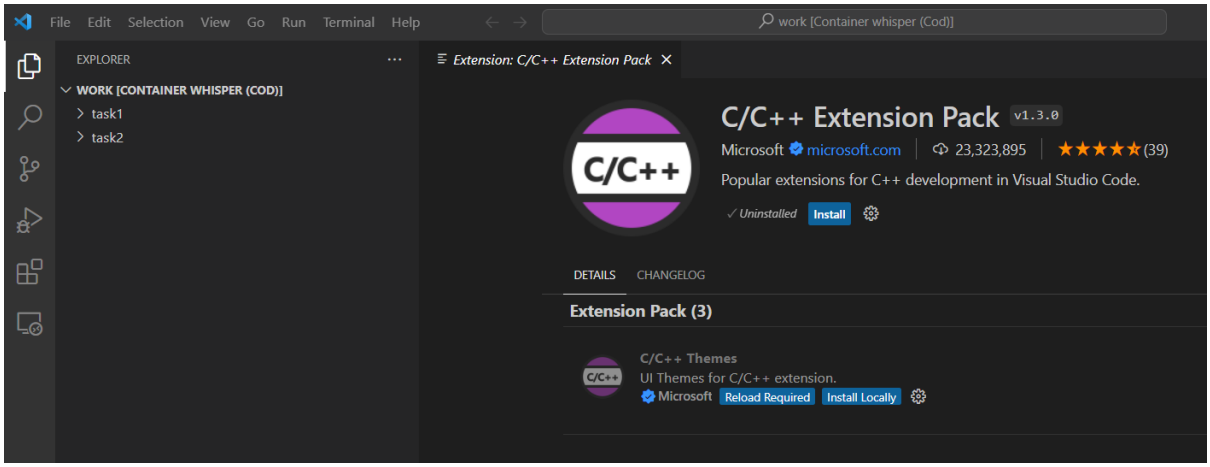



3. 安装好后，进入下图界面，whisper为目前电脑中正在运行的容器，红色方框处按钮用于打开对应容器，打开容器后可以使用 Ctrl+K, O 快捷键打开容器目录。



4. 使用 vscode 进入容器后，需要在容器中安装 C/C++插件，请在容器界面安装C/C++插件，不是在容器中安装的C/C++插件无法在容器中使用。然后进入 /work/task1 或者 /work/task2 目录即可开始本次工作。

# 三、whisper 容器中文件介绍

## 3.1 /whisper 目录

此目录下为whisper 模拟器相关文件。

**此目录无需修改**

## 3.2 /opt/riscv 目录

此目录下为RISC-V 编译工具链，包括了本次实验中所需的所有交叉编译器。可以查看/opt/riscv/bin 目录下的文件。

**此目录无需修改。**

## 3.3 /work/task1/src 目录

此目录下包含了任务一所需的七个文件。

1) instforms.cpp & instform.hpp：根据头文件中不同指令类型的结构，实现了指令对应结构体中的编码函数；
2) decode.cpp：实现仿真器的译码函数，根据指令操作码和功能码得到译码结果；
3) Hart.cpp & Hart.hpp：实现硬件线程的模拟，需要补充部分指令的执行函数；
4) InstEntry.cpp：定义指令集中每一条指令对应条目；
5) InstId.hpp：定义指令集中每一条指令对应 id；

## 3.4 /work/task1/test 目录

此目录下包含了任务一所需要的一些测试文件。

**此目录无需修改**

### 3.5 /work/task2 目录

此文件中包含了任务二所需的相关文件，包括 SM3 算法的相关文件，需要修改 sm3opt.S 文件。

## 四、任务 1：仿真器拓展

本节将从拓展指令 cube 出发，引导你实现仿真器的指令拓展。你的任务是完成以下 R-Type 指令的拓展：

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000010 | rs2 | rs1 | 000 | rd | 0110011 | cube |
| 0000010 | rs2 | rs1 | 001 | rd | 0110011 | rotleft |
| 0000010 | rs2 | rs1 | 010 | rd | 0110011 | rotrigt |
| 0000010 | rs2 | rs1 | 011 | rd | 0110011 | reverse |
| 0000010 | rs2 | rs1 | 100 | rd | 0110011 | notand |

| | | |
|---|---|---|
| cube | rd, rs1, rs2: | $R[rd] = R[rs1]^3$ |
| rotleft | rd, rs1, rs2: | R[rd] = ((R[rs1] << R[rs2]) \| (R[rs1] >> (32- R[rs2]))) |
| rotright | rd, rs1, rs2: | R[rd] = ((R[rs1] >> R[rs2]) \| (R[rs1] << (32- R[rs2]))) |
| reverse | rd, rs1, rs2: | R[rd] = (R[rs1] >> (24 - R[rs2] * 8)) & 0x000000ff; |
| notand | rd, rs1, rs2: | R[rd] = ~(R[rs1]) & R[rs2] |

### 1. instforms: 指令编码

仿真器在 instforms.hpp 文件中针对不同指令类型构建了用于编码和译码的结构体，你需要在 instforms.cpp 中实现对应的编码函数。编码函数的输入参数是寄存器操作数（和立即数操作数），根据编码规则确定最终的编码。实验中用到的拓展指令集已在此前文档给出。

R-Type 对应的结构体为

```
struct
{
unsigned opcode : 7;
unsigned rd : 5;
unsigned funct3 : 3;
unsigned rs1 : 5;
unsigned rs2 : 5;
unsigned funct7 : 7;
} bits;
```

与编码格式类似，结构体中将 32 位指令分为 6 个部分，包括三个寄存器操作数，操作码和功能码。拓展指令 cube 的编码函数为

```
bool
RFormInst::encodeCube(unsigned rdv, unsigned rs1v,
unsigned rs2v)
{
   if (rdv > 31 or rs1v > 31 or rs2v > 31)
   return false;
   bits.opcode = 0x33;
   bits.rd = rdv & 0x1f;
   bits.funct3 = 0;
   bits.rs1 = rs1v & 0x1f;
   bits.rs2 = rs2v & 0x1f;
   bits.funct7 = 2;
   return true;
}
```

首先判断输入参数是否超出了理论范围，然后为每个结构体内容赋相应的值，操作码和功能码由指令确定，操作数由输入参数确定。**根据 cube 指令示例，你需要完成其他四条拓展指令的编码函数，除此之外，请不要改变其余部分代码。**
注：instforms.hpp 文件中 84 行起已添加拓展编码函数的声明。

## 2. decode: 指令译码

本部分实现了仿真器的译码功能，实验主要关注从 decode.cpp 文件 1368 行 开始的 decode 函数，输入一条指令 inst，需要给出其对于的操作数 op 和指令条 目 InstEntry。InstEntry 在 InstEntry.hpp 中定义，指令集中的每一条指

令都对应了一个条目，cube 指令声明于 InstEntry.cpp 中 3891 行。指令同时需要在 InstId.hpp 中声明，cube 指令声明于 634 行。**你需要参照 cube 及其他指令格式，在 InstEntry.cpp 和 InstId.hpp 中添加其他四条拓展指令内容，注意及时修改 InstId.hpp 中 maxId 的值。**

操作数 op 定义在 DecodeInst.hpp 文件中，对于一般的指令，如 add x2, x1, x0，op0 为 x2，op1 为 x1，op2 为 x0，特殊的指令主要是 load 和 store指令。对于 load 指令，将“load rd, offset(rs1)”映射为“load rd, rs1, offset”，因此 op0 为 rd，op2 为 offset；对于 store 指令，将“store rs2, offset(rs1)”映射为“store rs2, rs1, offset”，因此 op0 为 rs2，op2 为 offset。

下面来看 decode 具体操作。可以将 RISC-V 指令集根据操作码 opcode的高五位（低两位恒为 11）分为 32 个操作码空间，每个操作码空间对应了同一种类型的指令，在每个操作码空间确定指令对应操作数，并根据功能码进行译码，确定对应的指令条目。cube 指令的译码已于 decode.cpp 文件 1839 行给出，**你需要参照 cube 及其他指令格式，在 decode.cpp 中添加其他四条拓展指令译码内容。**

## 3. Hart: 指令执行

Hart 是 Hardware Thread 的缩写，本实验主要关注从 Hart.cpp 文件 979 行开始的内容，即指令的执行过程。下面同样以拓展指令 cube 指令为例：

```
template <typename URV>
inline
void
Hart<URV>::execCube(const DecodedInst* di)
{
  URV v = intRegs_.read(di->op1()) * intRegs_.read(di->op1()) *
          intRegs_.read(di->op1());
  intRegs_.write(di->op0(), v);
}
```

从指令的译码过程中我们知道，op0 对应目的寄存器 rd，op1 和 op2 对应源寄存器 rs1 和 rs2。执行过程中从 op1 对应寄存器中读出相应内容并立方计算，然后将结果写回 op0 对应寄存器中。

根据这些示例，你需要完成其他四条拓展指令的执行过程，**同时注意左移操作时寄存器中数据的比特位长度应设为 32 位，可使用强制类型转换为 int。**注：Hart.hpp 文件中 1848 行起已添加拓展执行函数的声明。

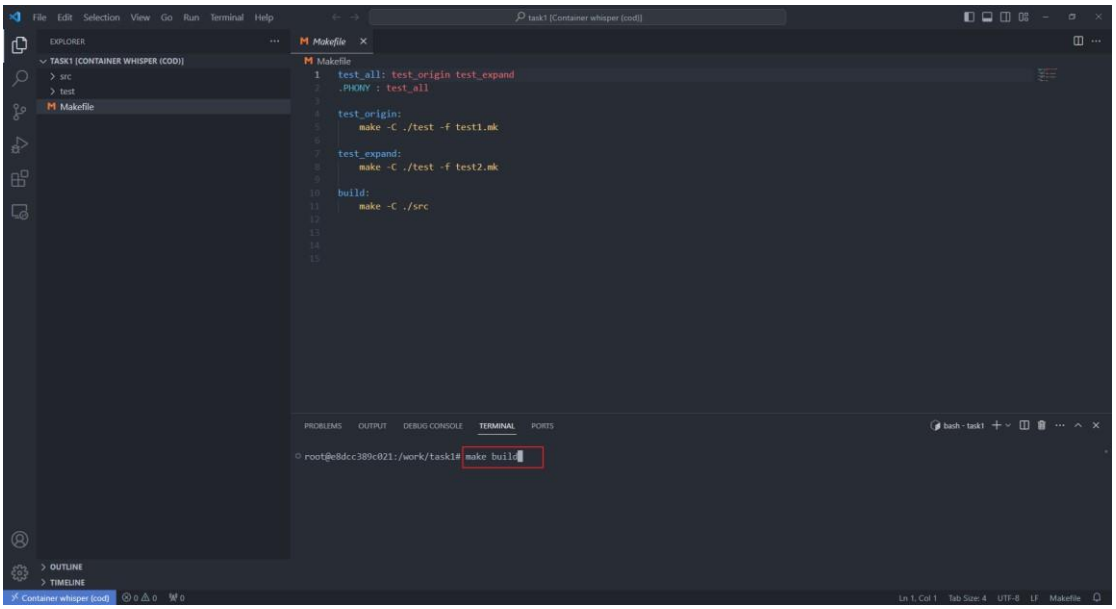
在完成执行函数后，你需要参照 cube 指令在 Hart.cpp 文件 5442 行开始的 execute()函数中添加拓展指令的 label 以及相应的跳转执行，cube 指令的 label 已于 6007 行给出，执行跳转已于 6171 行给出。

**即，你需要在 Hart.cpp 完成其他四条拓展指令的执行过程，添加拓展指令的 label 以及相应的跳转执行。**

## 4. 测试

在/work/task1/src 下完成相关文件的修改之后即可开始进行测试工作。

在VS Code 终端中输入 make build 命令即可完成 whisper 的编译。




我们提供了两个测试文件 test1 与 test2，同时也提供了相应的 Makefile 用于测试。

test1 用于测试原有 RISC-V 指令的执行情况， test2 用于测试拓展指令的执行情况。可以在 VS Code 终端中使用 make test_origin 或者 make test_expand 命令测试原有指令/拓展指令的执行状况。

如果对于每条拓展指令终端均打印出对应的“test pass”， 说明仿真器正确执行了 RISC-V 程序中的拓展指令。

## 五、任务 2：优化 SM3 加密算法

SM3 算法是散列算法的一种，是我国国家密码管理局编制的商用算法。 为满足电子认证服务系统等应用的需求，国家密码管理局在 2010 年 12 月发布了 SM3 算法，用于密码应用中的数字签名和验证、消息认证码的生成与验证以及随机数的生成，可满足多种密码应用的安全需求。

我们在 /work/task2 文件夹中提供了实现 SM3 算法的 c 文件 sm3.c，汇编文件 sm3.s。

你的任务是利用拓展指令 rotleft、rotright、reverse、notand，对照 sm3.c 修改 sm3.s 汇编文件，将原来汇编文件中的部分指令等价转换为拓展指令，对 SM3 算法实现指令级的优化， 并将修改后的结果保存到 sm3opt.s 中。实验重点关注sm3_str_summ()函数。

汇编文件中 R-Type 扩展指令集的格式如下：

- .insn r opcode, func3, func7, rd, rs1, rs2

下以 cube 指令为例，说明如何进行指令等价转换:

原汇编：

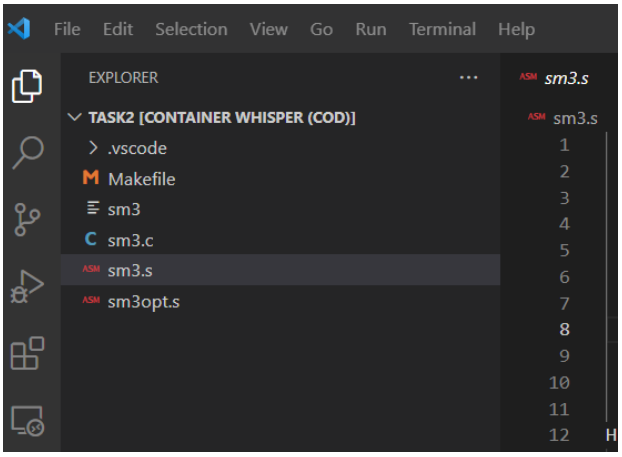
```
mul     a4, a5, a5
mul     a4, a4, a5
```

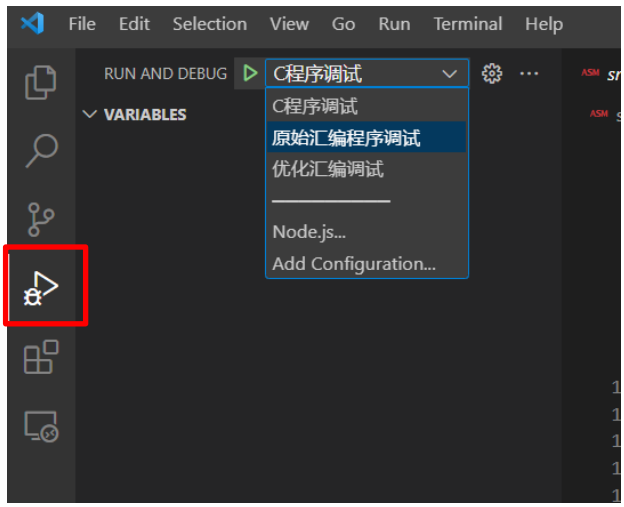
优化汇编：

```
.insn r 0x33, 0, 2, a4, a5, x0     # a4=a5*a5*a5
```

汇编文件修改完成后，可以使用 vscode 的 debug 选项逐行调试代码，运行结束后终端会输出程序执行过程中的指令条目数“Retired XXX instructions in X s”。

汇编调试请在task2文件夹下进行，不要在work文件夹下，有三种调试方式。



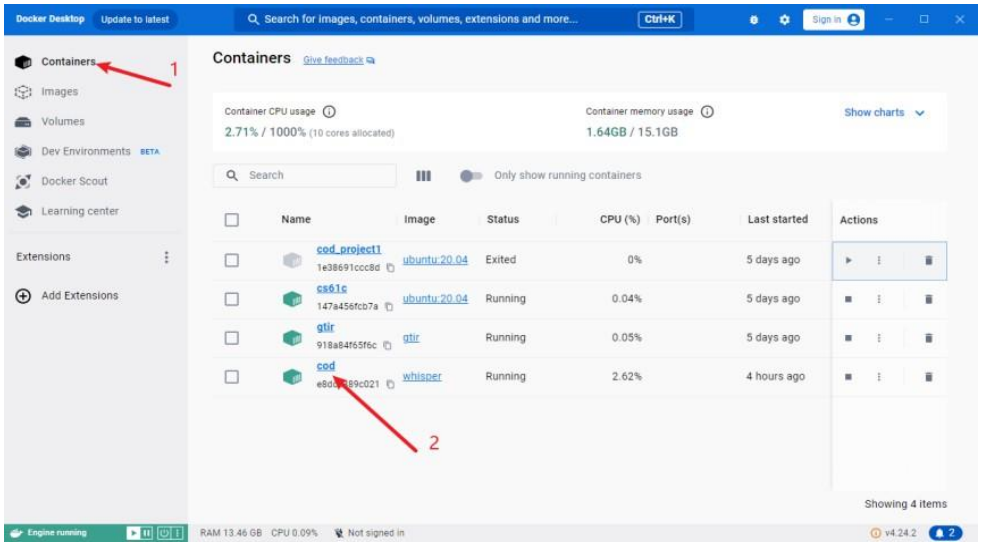
在对比过程中， 你需要分别输入”Zhejiang University”、 ”COD”、 你的学号，计算其哈希值，在验证你汇编的正确性同时比较优化效果。

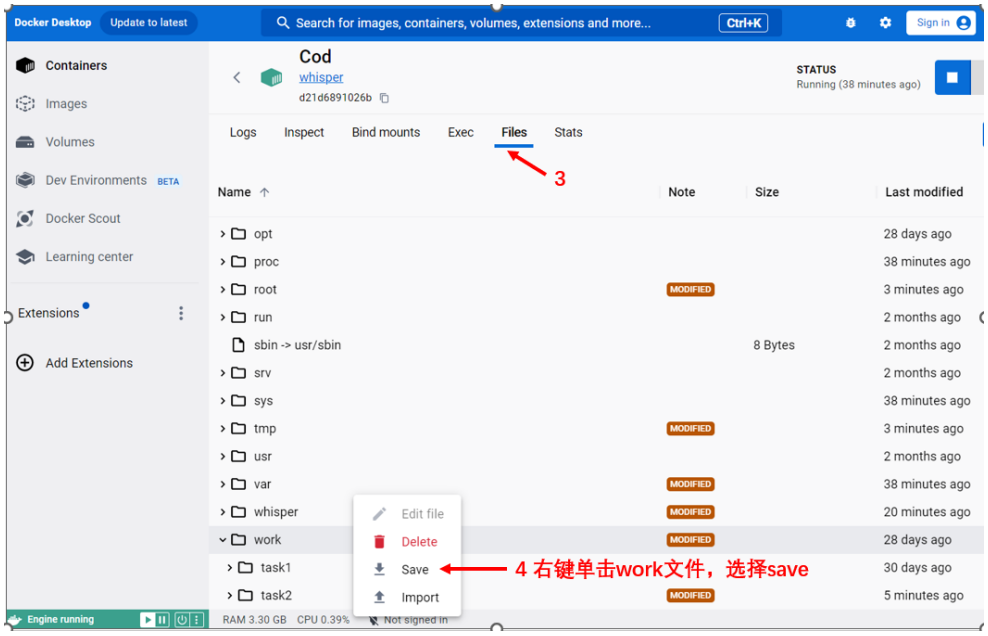
**最后，你需要按照此前的设计思路， 在已提供的五条拓展指令以外， 至少自主设计并完成一条拓展指令， 同时对仿真器和哈希加密的汇编文件进行修改，从而实现程序的进一步优化。 注： 自定拓展指令添加函数时需要修改对应的头文件.hpp。** 你的优化结果将会在一定程度上影响你的最终得分。

## 六、提交

只需提交修改后的 work 目录的压缩包文件，以及一份报告。报告中应说明模拟器的实现思路及测试结果、汇编文件的修改思路、 自定拓展指令的设计思路及实现， 程序最终的优化结果。将所有文件打包后以学号加姓名命名，上传到学在浙大，并在 2024.1.4 课堂上上交一份纸质报告。学在浙大的截止时间是 2023.1.4 日 23:59。

work 目录可以通过 docker desktop 保存到自己电脑，方法如下：



# 参考文献

[1] 席宇浩, RISC-V 指令集仿真器, 浙江大学信息与电子工程学院, 计算机组成与设计(2021 年)

[2] 刘岸林, RISC-V 领域加速指令设计与仿真, 浙江大学信息与电子工程学院, 计算机组成与设计(2022 年)