浙江大学

本科实验报告

project 1

课程名称：	计算机组成与设计
姓名：	
学院：	信息与电子工程学院
专业：	电子科学与技术
学号：	
指导老师：	

January 22, 2024

目　录

一、 docker 实验环境配置

1. 安装 Docker Desktop

(1) 前往 Docker Desktop 官网下载 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。
观察任务栏，若出现鲸鱼图标，即在正常运行。同时下载 Windows Terminal，方便之后镜像文件的安装。

2. 构建/导入镜像

在这一步中，我选择从 Dockerfile 构建。
(1) 解压 Docker.zip。
(2) 打开 PowerShell/Windows Terminal 并且进入到解压后的文件夹
(3) 在 PowerShell/Windows Terminal 中运行命令: docker build -t whisper .
(4) 构建过程大约用时 400 s。
在这之后，于 powershell 运行 docker run -it –name Name whisper 以启动容器

3. 使用 VS Code 连接容器

（1）在 VScode 安装 Dev-container 插件
（2）安装好后，进入下图界面，绿色方框中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。
（3）使用 vscode 进入容器后，需要在容器中安装 C/C++ 插件。然后进入 /work/task1 或者/work/task2 目录即可开始本次工作

二、 仿真器拓展

1. 模拟器的实现思路

在/work/task1/src 目录下，一共提供了七个文件：
(1)instforms.cpp 及 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 及 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；
在 decode.cpp 中：

```
 else if(funct7 == 2)
{
   // cube
   if (funct3 == 0) return instTable_.getEntry(InstId::cube);
   //rotleft
   if (funct3 == 1) return instTable_.getEntry(InstId::rotleft);
   //rotright
   if (funct3 == 2) return instTable_.getEntry(InstId::rotright);
   //reverse
   if (funct3 == 3) return instTable_.getEntry(InstId::reverse);
```

```
    //notand
    if (funct3 == 4) return instTable_.getEntry(InstId::notand);
  }
```

通过判断 funct7 的值来确定将要使用拓展指令，再通过判断 funct3 的值来确定具体使用哪一个拓展指令，并返回函数。

在 Hart.cpp 中：

```
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);
}


template <typename URV>
inline
void
Hart<URV>::execRotleft(const DecodedInst* di)
{
   URV v = ((intRegs_.read(di->op1())<<intRegs_.read(di->op2())) |
       (intRegs_.read(di->op1())>>(32-intRegs_.read(di->op2()))));
   intRegs_.write(di->op0(), v);
}

template <typename URV>
inline
void
Hart<URV>::execRotright(const DecodedInst* di)
{
   u_int32_t rs2;
   rs2= (u_int32_t) (32-intRegs_.read(di->op2()));
   URV v = (u_int32_t) ((intRegs_.read(di->op1())>>intRegs_.read(di->op2())) |
       (intRegs_.read(di->op1()) << rs2));
   intRegs_.write(di->op0(), v);
}

template <typename URV>
inline
void
Hart<URV>::execReverse(const DecodedInst* di)
{
   URV v = (intRegs_.read(di->op1()) >> (24-intRegs_.read(di->op2())*8))&0x000000ff;
   intRegs_.write(di->op0(), v);
}

template <typename URV>
inline
void
Hart<URV>::execNotand(const DecodedInst* di)
{
   URV v = ~(intRegs_.read(di->op1()))&intRegs_.read(di->op2());
```

```
    intRegs_.write(di->op0(), v);
}
```

以上代码实现了拓展函数的实现方法，其对应数学运算如图 1所示，注意在进行右移运算时，需要将数据类型转化为 int 类型以满足 32 位的要求：

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]

Figure 1: 拓展函数运算公式

```
&&cube,
&&rotleft,
&&rotright,
&&reverse,
&&notand,


cube:
execCube(di);
return;

rotleft:
execRotleft(di);
return;


rotright:
execRotright(di);
return;

notand:
execNotand(di);
return;


reverse:
execReverse(di);
return;
```

这一部分代码定义了拓展函数的地址，用于在算法中对其进行调用, 并且修改 maxId 为 notand。同时也定义了拓展指令的 id。

在 Hart.hpp 头文件中：

```
void execCube(const DecodedInst*);
void execRotleft(const DecodedInst*);
void execRotright(const DecodedInst*);
void execReverse(const DecodedInst*);
void execNotand(const DecodedInst*);
```

这一部分代码对拓展函数的执行函数进行了声明。
在 InstEntry.cpp 中：

```
    { "cube", InstId::cube, 0x4000033, top7Funct3Low7Mask,
       InstType::Int,
       OperandType::IntReg, OperandMode::Write, rdMask,
       OperandType::IntReg, OperandMode::Read, rs1Mask,
       OperandType::IntReg, OperandMode::Read, rs2Mask },

    { "rotleft", InstId::rotleft, 0x4001033, top7Funct3Low7Mask,
       InstType::Int,
       OperandType::IntReg, OperandMode::Write, rdMask,
       OperandType::IntReg, OperandMode::Read, rs1Mask,
       OperandType::IntReg, OperandMode::Read, rs2Mask },

    { "rotright", InstId::rotright, 0x4002033, top7Funct3Low7Mask,
       InstType::Int,
       OperandType::IntReg, OperandMode::Write, rdMask,
       OperandType::IntReg, OperandMode::Read, rs1Mask,
       OperandType::IntReg, OperandMode::Read, rs2Mask },

    { "reverse", InstId::reverse, 0x4003033, top7Funct3Low7Mask,
       InstType::Int,
       OperandType::IntReg, OperandMode::Write, rdMask,
       OperandType::IntReg, OperandMode::Read, rs1Mask,
       OperandType::IntReg, OperandMode::Read, rs2Mask },

    { "notand", InstId::notand, 0x4004033, top7Funct3Low7Mask,
       InstType::Int,
       OperandType::IntReg, OperandMode::Write, rdMask,
       OperandType::IntReg, OperandMode::Read, rs1Mask,
       OperandType::IntReg, OperandMode::Read, rs2Mask },
```

这一部分代码定义指令集中每一条指令对应条目。
在 instform.hpp 中：

```
    /// Encode "Cube rd, rs1, rs2" into this object.
    bool encodeCube(unsigned rd, unsigned rs1, unsigned rs2);

    /// Encode "Rotleft rd, rs1, rs2" into this object.
    bool encodeRotleft(unsigned rd, unsigned rs1, unsigned rs2);

    /// Encode "Rotright rd, rs1, rs2" into this object.
    bool encodeRotright(unsigned rd, unsigned rs1, unsigned rs2);

    /// Encode "Reverse rd, rs1, rs2" into this object.
    bool encodeReverse(unsigned rd, unsigned rs1, unsigned rs2);

    /// Encode "Notand rd, rs1, rs2" into this object.
    bool encodeNotand(unsigned rd, unsigned rs1, unsigned rs2);
```

该部分对拓展指令的编码函数进行了声明。
在在 instform.cpp 中：

```
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;
}

bool
RFormInst::encodeRotleft(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 = 1;
   bits.rs1 = rs1v & 0x1f;
   bits.rs2 = rs2v & 0x1f;
   bits.funct7 = 2;
   return true;
}

bool
RFormInst::encodeRotright(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 = 2;
   bits.rs1 = rs1v & 0x1f;
   bits.rs2 = rs2v & 0x1f;
   bits.funct7 = 2;
   return true;
}

bool
RFormInst::encodeReverse(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 = 3;
   bits.rs1 = rs1v & 0x1f;
   bits.rs2 = rs2v & 0x1f;
   bits.funct7 = 2;
   return true;
}
```

```
bool
RFormInst::encodeNotand(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 = 4;
   bits.rs1 = rs1v & 0x1f;
   bits.rs2 = rs2v & 0x1f;
   bits.funct7 = 2;
   return true;
}
```

该部分代码根据头文件中不同指令类型的结构，实现了拓展指令对应结构体中的编码函数。

2. 测试结果

在终端中打开 task1 的路径，并在其中运行 make build 指令，出现如下图的情况即为编译成功，在此基础上，即可对原有指令和拓展指令进行测试。

```
g++  -MMD -MP -mfma -std=c++17 -O3 -isystem /opt/boost_1_83_0 -I. -fPIC -pedantic -Wall -Wextra -c -o build-Linux/vector.cpp.o vector.cpp
g++  -MMD -MP -mfma -std=c++17 -O3 -isystem /opt/boost_1_83_0 -I. -fPIC -pedantic -Wall -Wextra -c -o build-Linux/float.cpp.o float.cpp
g++  -MMD -MP -mfma -std=c++17 -O3 -isystem /opt/boost_1_83_0 -I. -fPIC -pedantic -Wall -Wextra -c -o build-Linux/bitmanip.cpp.o bitmanip.cpp
g++  -MMD -MP -mfma -std=c++17 -O3 -isystem /opt/boost_1_83_0 -I. -fPIC -pedantic -Wall -Wextra -c -o build-Linux/amo.cpp.o amo.cpp
ar cr build-Linux/librvcore.a build-Linux/IntRegs.cpp.o build-Linux/CsRegs.cpp.o build-Linux/FpRegs.cpp.o build-Linux/instforms.cpp.o build-Li
nux/Memory.cpp.o build-Linux/Hart.cpp.o build-Linux/InstEntry.cpp.o build-Linux/Triggers.cpp.o build-Linux/PerfRegs.cpp.o build-Linux/gdb.cpp.
o build-Linux/HartConfig.cpp.o build-Linux/Server.cpp.o build-Linux/Interactive.cpp.o build-Linux/decode.cpp.o build-Linux/disas.cpp.o build-L
inux/Syscall.cpp.o build-Linux/PmaManager.cpp.o build-Linux/DecodedInst.cpp.o build-Linux/snapshot.cpp.o build-Linux/PmpManager.cpp.o build-Li
nux/VirtMem.cpp.o build-Linux/Core.cpp.o build-Linux/System.cpp.o build-Linux/Cache.cpp.o build-Linux/Tlb.cpp.o build-Linux/VecRegs.cpp.o buil
d-Linux/vector.cpp.o build-Linux/wideint.cpp.o build-Linux/float.cpp.o build-Linux/bitmanip.cpp.o build-Linux/amo.cpp.o build-Linux/SparseMem.
cpp.o
g++ -o build-Linux/whisper build-Linux/whisper.cpp.o build-Linux/librvcore.a  -l:libboost_program_options.a -lpthread -lz -static-libstdc++ -l
stdc++fs
make[2]: Leaving directory '/whisper'
make[1]: Leaving directory '/work/task1/src'
root@7f9eb2ef1755:/work/task1#
```

Figure 2: task1 编译结果

在这之后运行 make test_origin 指令，即可测试原有指令是否正确，最终出现如图 3所示情况，说明测试通过。

```
root@7f9eb2ef1755:/work/task1# make test_origin
make -C ./test -f test1.mk
make[1]: Entering directory '/work/task1/test'
/whisper/build-Linux/whisper  test1
Test Pass!
Target program exited with code 0
Retired 1864 instructions in 0.00s  4304849 inst/s
make[1]: Leaving directory '/work/task1/test'
```

Figure 3: task1 原有指令测试

最后，运行 make test_expand 指令，即可测试拓展指令是否正确，得到如图 4所示的测试结果，说明五种拓展指令都能正确执行。

3. task1 部分总结

在 task1 中，我们搭建了一个用于模拟运行 RISC-V 汇编代码的环境，通过 c++ 将指令的编码搭建出来，构成完整的汇编代码指令集。

```
root@7f9eb2ef1755:/work/task1# make test_expand
make -C ./test -f test2.mk
make[1]: Entering directory '/work/task1/test'
/whisper/build-Linux/whisper  test2
cube test pass!
rotleft test pass!
rotright test pass!
reverse test pass!
notand test pass!
Target program exited with code 0
Retired 3926 instructions in 0.00s  10118556 inst/s
make[1]: Leaving directory '/work/task1/test'
```

Figure 4: task1 拓展指令测试

在这一过程中，我们仿照示例，设计得到用于循环右移，循环左移等 5 条拓展指令，并通过测试代码保证其能正常运行。

三、 优化 SM3 加密算法

1. 通过已有拓展指令进行算法优化

(1) 循环左移

rotleft 指令的逻辑为在操作数左移后右移的基础上进行或运算，对应的汇编语句为：

```
#URV v = ((intRegs_.read(di->op1())<<intRegs_.read(di->op2())) |
    (intRegs_.read(di->op1())>>(32-intRegs_.read(di->op2()))));
#intRegs_.write(di->op0(), v);

#slli a3,a5,15
#srli a5,a5,17
#or a5,a5,a3
addi a3, x0, 15
.insn r 0x33, 1, 2, a5, a5, a3
```

通过该指令可以将每一循环中的三条指令减少为 2 条，实现循环左移的优化。

(2) 循环右移

rotright 指令的逻辑时在操作数右移后左移的基础上同样进行或运算，对应的汇编代码为：

```
#u_int32_t rs2;
#rs2= (u_int32_t) (32-intRegs_.read(di->op2()));
#URV v = (u_int32_t) ((intRegs_.read(di->op1())>>intRegs_.read(di->op2())) |
    (intRegs_.read(di->op1()) << rs2));
#intRegs_.write(di->op0(), v);

#srli a3,a5,9
#slli a5,a5,23
#or a5,a5,a3
addi a3, x0, 9
.insn r 0x33, 2, 2, a5, a5, a3
```

通过该指令可以将每一循环中的三条指令减少为 2 条，实现循环右移的优化。

(3) notand

notand 指令的逻辑为一个操作数的取反同另一个操作数进行与运算，但在汇编代码中，两个 lw 的操作数都是 a5，所以在进行汇编语言的优化时，还需要更改 lw 的操作数，其对应的代码如下：

```
#URV v = ~(intRegs_.read(di->op1()))&intRegs_.read(di->op2());
#intRegs_.write(di->op0(), v);

#lw a5,-556(s0)
#not a3,a5
#lw a5,-548(s0)
#and a5,a3,a5
lw a3,-556(s0)
lw a5,-548(s0)
.insn r 0x33,4,2,a5,a3,a5
```

通过该指令可以将每一循环中的 4 条指令减少为 3 条，实现 notand 的优化。

(4) 测试结果

对”Zhejiang University”,”COD”,”3210105209” 进行测试后，可得到以下的三组结果：

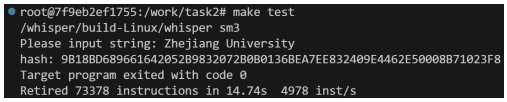

Figure 5: 原代码 ZJUhash 测试

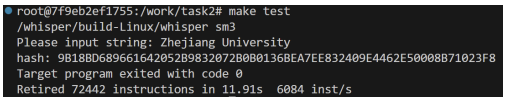

Figure 6: 优化代码 ZJUhash 测试

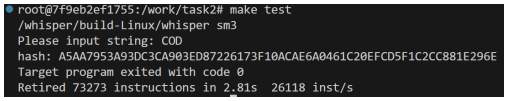

Figure 7: 原代码 CODhash 测试

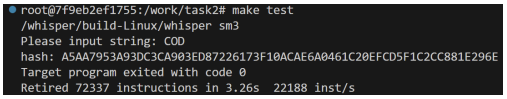

Figure 8: 优化代码 CODhash 测试

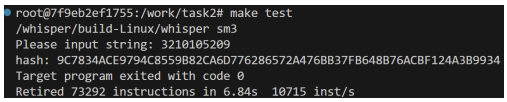

Figure 9: 原代码学号 hash 测试

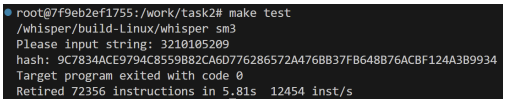

Figure 10: 优化代码学号 hash 测试

	Zhejiang University	COD	3210105209
原代码	73378	73273	73292
优化代码	72442	72337	72356

通过以上测试，可以看出，优化算法平均减少了 936 条指令。

2. 自定拓展指令进行算法优化

(1) selfopt 指令优化

通过对 sm3 加密算法阅读，我发现

```
addi a4,s0,-368
slli a5,a5,2
add a1,a4,a5
```

这部分类似代码的重复出现程度很高，因此想对这一部分代码进行优化，该部分代码的数学含义为：

$$rd = (rd_1 - 360) + (rs_2 << 2)$$

根据该段代码的数学含义，可以得到自定指令的实现代码如下所示：

```
template <typename URV>
inline
void
Hart<URV>::execSelfopt(const DecodedInst* di)
{
   URV v = (intRegs_.read(di->op1())-368) + (intRegs_.read(di->op2())<<2);
   intRegs_.write(di->op0(), v);
}
```

其具体在 sm3opt.s 中的表现为：

```
#addi a4,s0,-368
#slli a5,a5,2
#add a1,a4,a5
.insn r 0x33, 5, 2, a1, s0, a5
```

在此基础上，可以对其进行测试，测试结果如图所示：

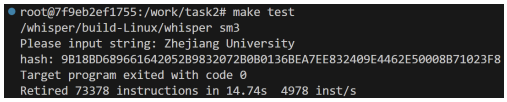

Figure 11: 原代码 ZJUhash 测试

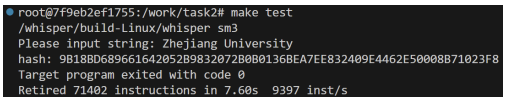

Figure 12: 自定指令优化 ZJUhash 测试

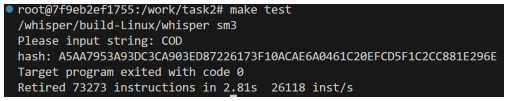

Figure 13: 原代码 CODhash 测试

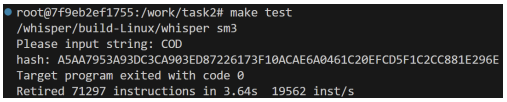

Figure 14: 自定指令优化 CODhash 测试

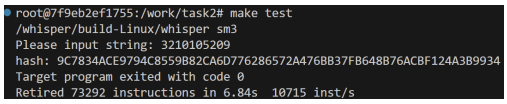

Figure 15: 原代码学号 hash 测试

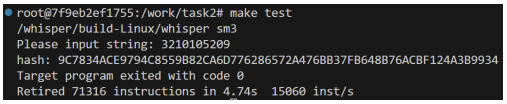

Figure 16: 自定指令优化学号 hash 测试

其优化效果如表所示：

	Zhejiang University	COD	3210105209
原代码	73378	73273	73292
优化代码	72442	72337	72356
自定指令代码	71420	71297	71316

从中可以看出，经过了自定指令优化之后，相较于原代码，指令数量减少了 1976 条，相较于拓展指令优化后的代码，指令数量减少了 1040 条。可以看出，优化的效果是较为明显的。

(2) addlli 指令

除此之外，

```
addi a4,s0,-540
slli a5,a5,2
add a5,a4,a5
```

该段代码也可以进行优化，其数学含义为：

$$rd = (rd_1 - 540) + (rs_2 << 2)$$

因此，可以进行数学运算的代码简化，其对应指令代码为：

```
template <typename URV>
inline
void
Hart<URV>::execAddlli(const DecodedInst* di)
{
   URV v = (intRegs_.read(di->op1()) - 540) + (intRegs_.read(di->op2())<<2);
   intRegs_.write(di->op0(), v);
}
```

其测试结果，如下图所示：

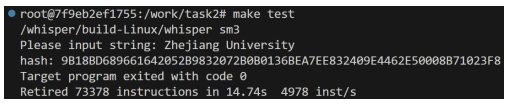

Figure 17: 原代码 ZJUhash 测试

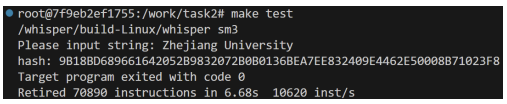

Figure 18: 自定指令优化 2ZJUhash 测试

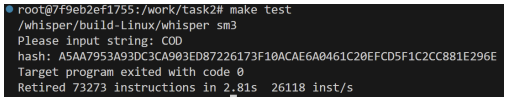

Figure 19: 原代码 CODhash 测试

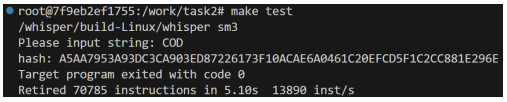

Figure 20: 自定指令优化 2CODhash 测试

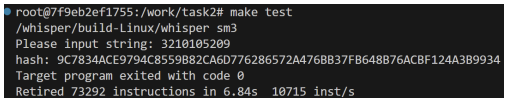

Figure 21: 原代码学号 hash 测试

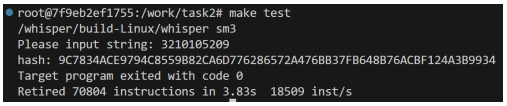

Figure 22: 自定指令优化学号 2hash 测试

其优化效果可以由下表看出：

	Zhejiang University	COD	3210105209
原代码	73378	73273	73292
优化代码	72442	72337	72356
自定指令代码	71420	71297	71316
自定指令代码 2	70890	70785	70804

从中我们可以看出，该优化又优化了 530 条指令，相比于源代码共优化了 2488 条指令，相较于拓展指令共优化了 1552 条指令，优化效果明显。

四、 总结与感悟

通过对 task1 和 task2 两部分代码的完善和优化，我一定程度上理解了汇编模拟器的实现思路，并根据自己的理解在其中设计了自定指令。同时对汇编相较于 C 语言能够进行更好的优化有了更深的理解，并根据实际算法对其进行了优化，加强了自身对 RISC-V 汇编的掌握。