专业：电子信息工程
姓名：王涵
学号：3200104515
日期：2022.12.4
地点：寝室

浙江大学实验报告

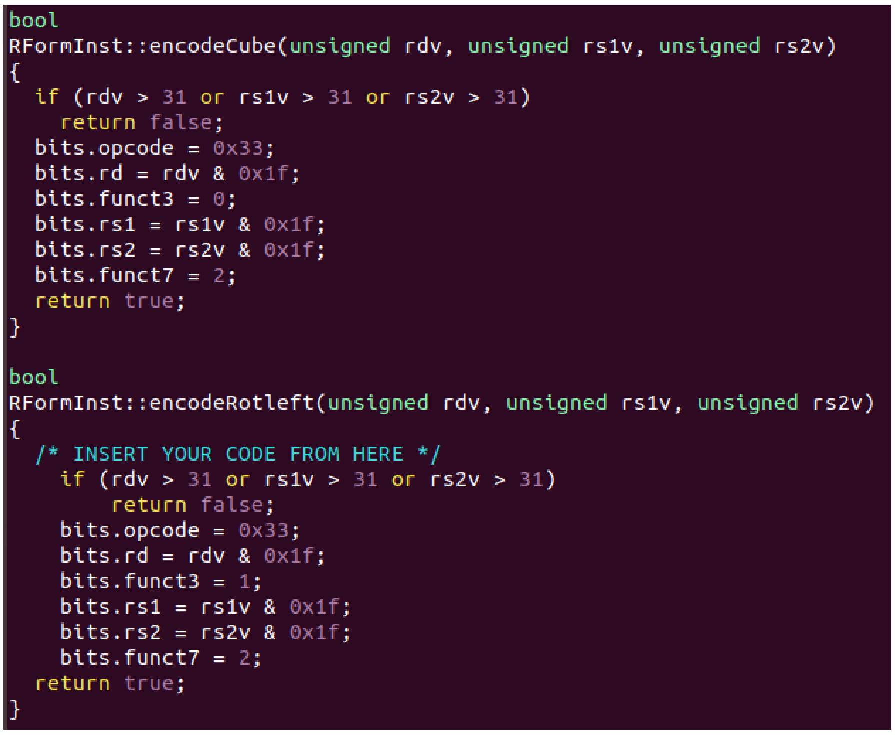
课程名称：计算机组成与设计　指导老师：沈继忠,赵武锋,唐奕,屈民军　成绩：

实验名称：RISC-V 领域加速指令设计与仿真　实验类型：设计型　同组学生姓名：无

一、　仿真器拓展

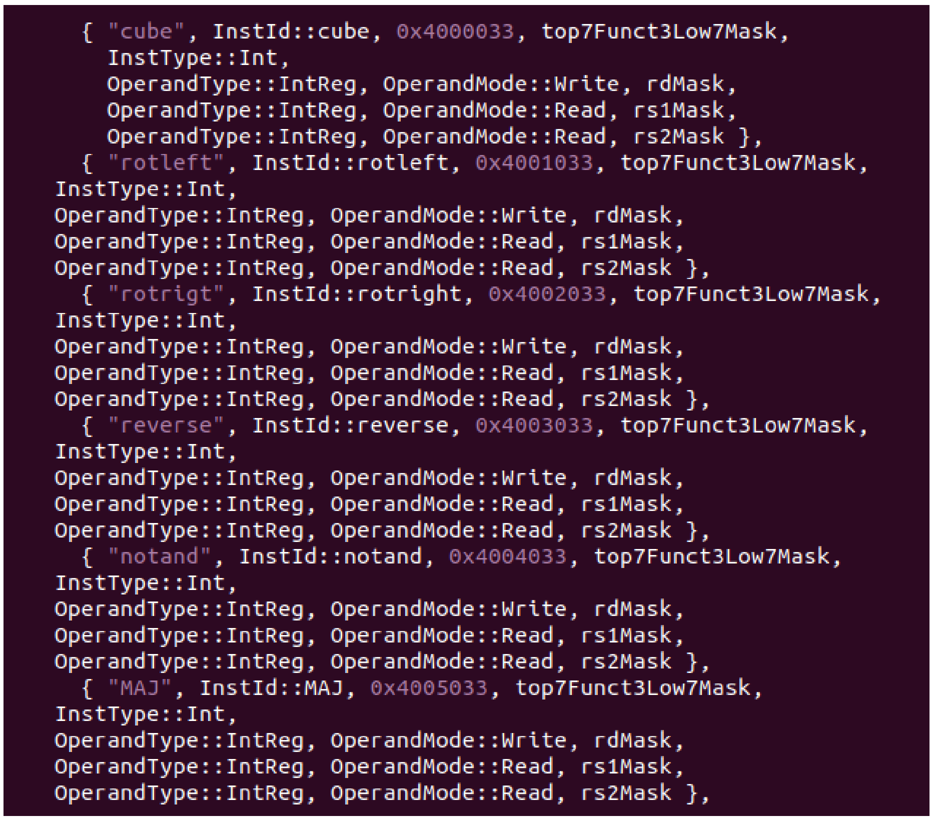
1. 模拟器的实现思路

按照文档，修改代码，其中 instforms.cpp&instform.hpp 的作用是根据头文件中的不同指令类型的结构，实现指令对应结构体中的编码函数，其中首先要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，操作码和功能码右指令确定，操作数由输入参数确定，hpp 文件中生成拓展的编码函数，实验的部分代码如下



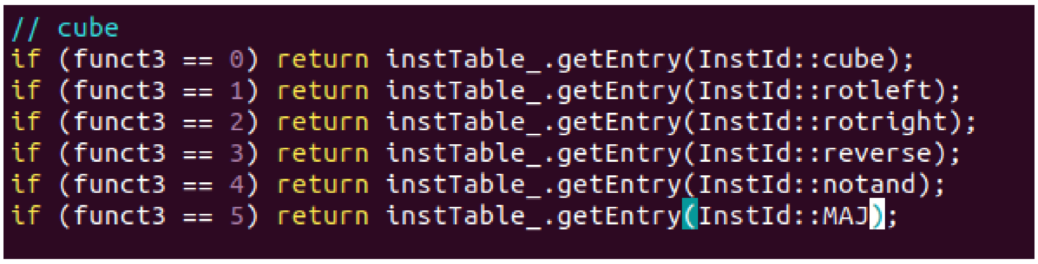
```
bool
RFormInst::encodeCube(unsigned rdv, unsigned rs1v, unsigned rs2v)
{
  if (rdv > 31 or rs1v > 31 or rs2v > 31)
    return false;
  bits.opcode = 0x33;
  bits.rd = rdv & 0x1f;
  bits.funct3 = 0;
  bits.rs1 = rs1v & 0x1f;
  bits.rs2 = rs2v & 0x1f;
  bits.funct7 = 2;
  return true;
}

bool
RFormInst::encodeRotleft(unsigned rdv, unsigned rs1v, unsigned rs2v)
{
  /* INSERT YOUR CODE FROM HERE */
    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;
}
```

之后实现仿真器的译码功能，首先修改 InstEntry.cpp&InstId.hpp，定义指令集中每一条指令对应的条目



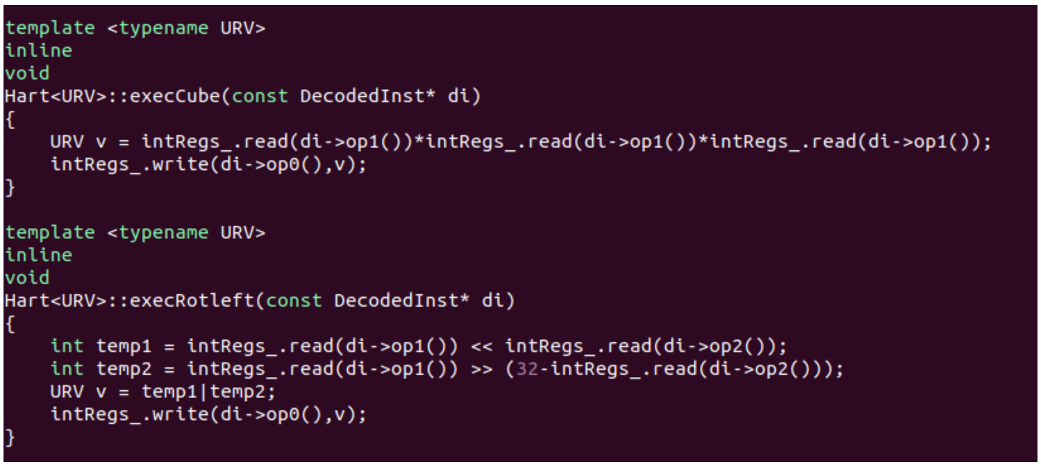
```
    { "cube", InstId::cube, 0x4000033, top7Funct3Low7Mask,
      InstType::Int,
      OperandType::IntReg, OperandMode::Write, rdMask,
      OperandType::IntReg, OperandMode::Read, rs1Mask,
      OperandType::IntReg, OperandMode::Read, rs2Mask },
    { "rotleft", InstId::rotleft, 0x4001033, top7Funct3Low7Mask,
  InstType::Int,
  OperandType::IntReg, OperandMode::Write, rdMask,
  OperandType::IntReg, OperandMode::Read, rs1Mask,
  OperandType::IntReg, OperandMode::Read, rs2Mask },
    { "rotrigt", 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 },
    { "MAJ", InstId::MAJ, 0x4005033, top7Funct3Low7Mask,
  InstType::Int,
  OperandType::IntReg, OperandMode::Write, rdMask,
  OperandType::IntReg, OperandMode::Read, rs1Mask,
  OperandType::IntReg, OperandMode::Read, rs2Mask },
```

之后修改 decode.cpp，主要实现的是指令的译码



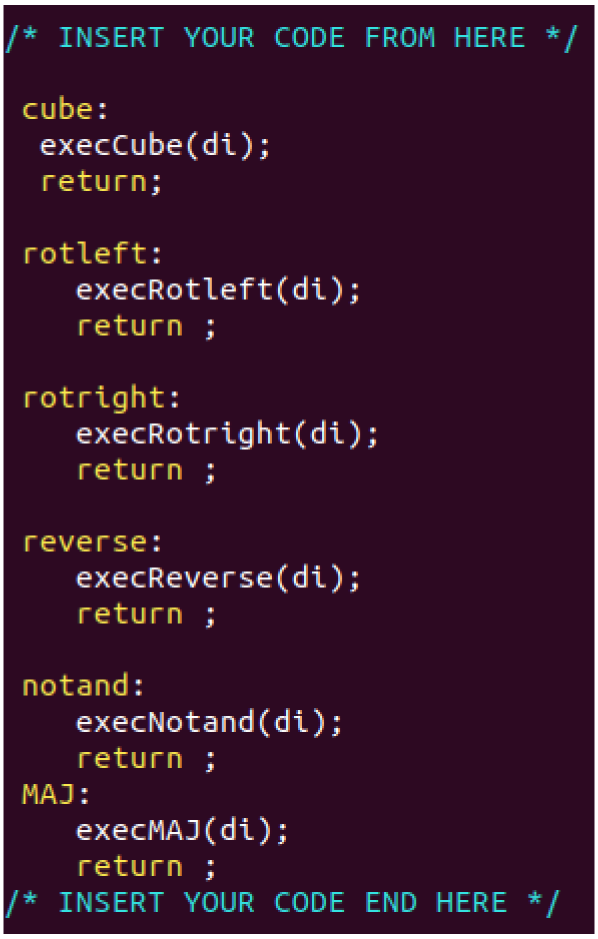
```
// cube
if (funct3 == 0) return instTable_.getEntry(InstId::cube);
if (funct3 == 1) return instTable_.getEntry(InstId::rotleft);
if (funct3 == 2) return instTable_.getEntry(InstId::rotright);
if (funct3 == 3) return instTable_.getEntry(InstId::reverse);
if (funct3 == 4) return instTable_.getEntry(InstId::notand);
if (funct3 == 5) return instTable_.getEntry(InstId::MAJ);
```

再之后，修改 Hart.cpp&Hart.hpp 指令执行部分，根据不同指令实现的功能，编写代码，部分代码的截图如下



```
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)
{
    int temp1 = intRegs_.read(di->op1()) << intRegs_.read(di->op2());
    int temp2 = intRegs_.read(di->op1()) >> (32-intRegs_.read(di->op2()));
    URV v = temp1|temp2;
    intRegs_.write(di->op0(),v);
}
```

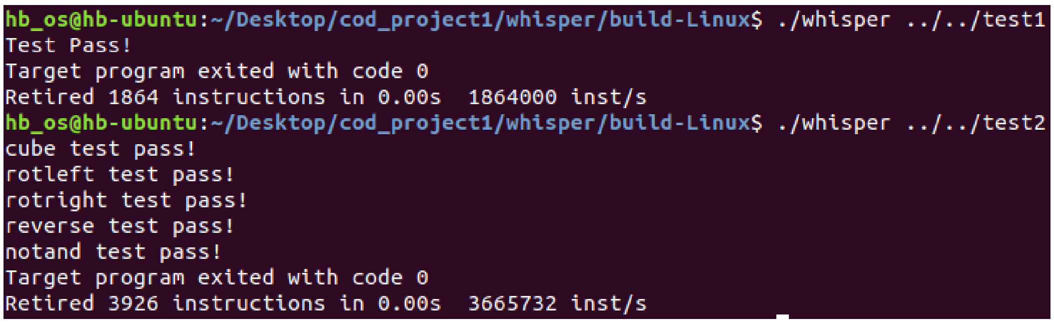
然后，execute()函数中，添加拓展指令的 label 以及相应的跳转执行



```
/* INSERT YOUR CODE FROM HERE */

 cube:
  execCube(di);
  return;

 rotleft:
    execRotleft(di);
    return ;

 rotright:
    execRotright(di);
    return ;

 reverse:
    execReverse(di);
    return ;

 notand:
    execNotand(di);
    return ;
 MAJ:
    execMAJ(di);
    return ;
/* INSERT YOUR CODE END HERE */
```

至此，完成全部的修改部分

2. 测试结果



```
hb_os@hb-ubuntu:~/Desktop/cod_project1/whisper/build-Linux$ ./whisper ../../test1
Test Pass!
Target program exited with code 0
Retired 1864 instructions in 0.00s  1864000 inst/s
hb_os@hb-ubuntu:~/Desktop/cod_project1/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  3665732 inst/s
```

编译运行，测试 test1&test2 均正确

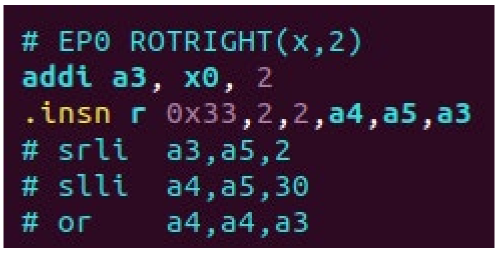
二、 优化哈希加密

1. 汇编文件的修改思路

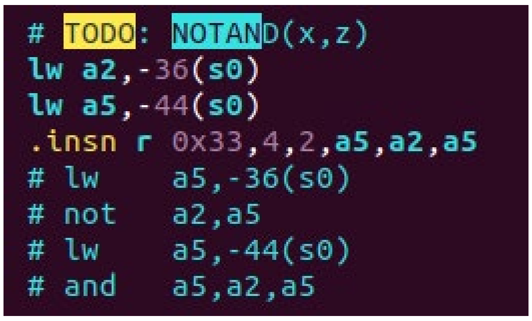
基本指令的实现：

观察 sha256.c 文件，发现一些宏定义的地方正好对应了拓展的汇编指令的部分，因此找到相应的代码段，探究清楚输入和输出的变量，进行代码替换，以下是部分的替换代码，因为替换的代码逻辑一致，所以就不全部列出

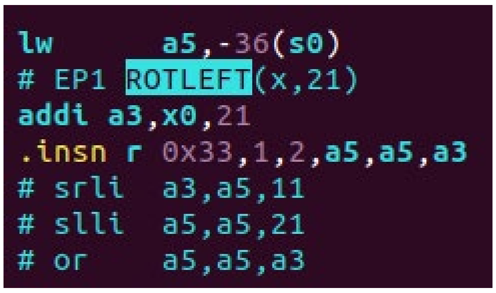
ROTRIGHT(x,2)


```
# EP0 ROTRIGHT(x,2)
addi a3, x0, 2
.insn r 0x33,2,2,a4,a5,a3
# srli  a3,a5,2
# slli  a4,a5,30
# or    a4,a4,a3
```

NOTAND(x, z)

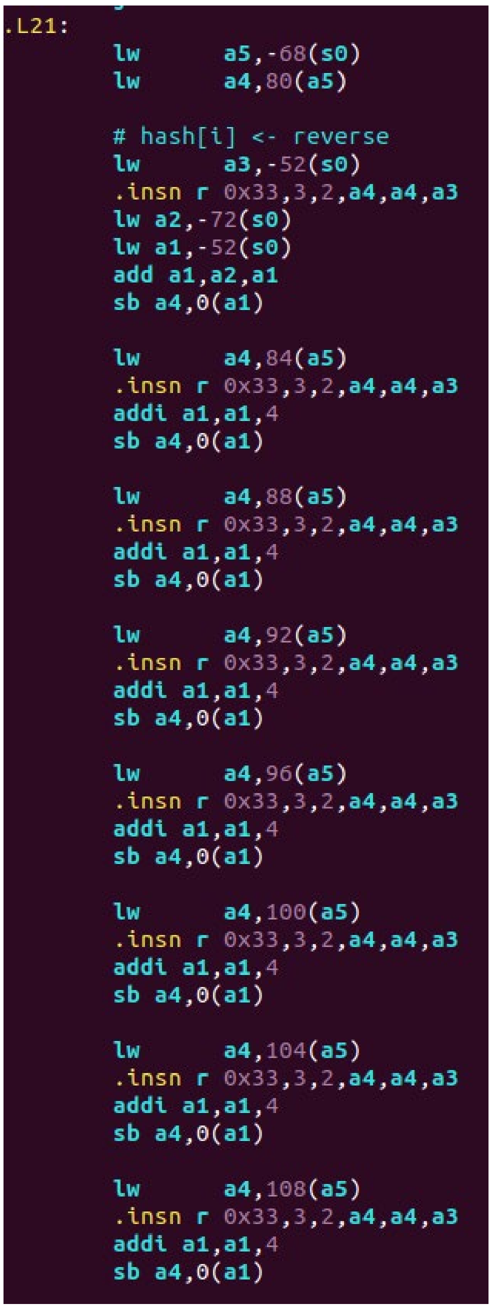

```
# TODO: NOTAND(x,z)
lw a2,-36(s0)
lw a5,-44(s0)
.insn r 0x33,4,2,a5,a2,a5
# lw    a5,-36(s0)
# not   a2,a5
# lw    a5,-44(s0)
# and   a5,a2,a5
```

ROTLEFT(x,21)


```
lw      a5,-36(s0)
# EP1 ROTLEFT(x,21)
addi a3,x0,21
.insn r 0x33,1,2,a5,a5,a3
# srli  a3,a5,11
# slli  a5,a5,21
# or    a5,a5,a3
```

REVERSE:

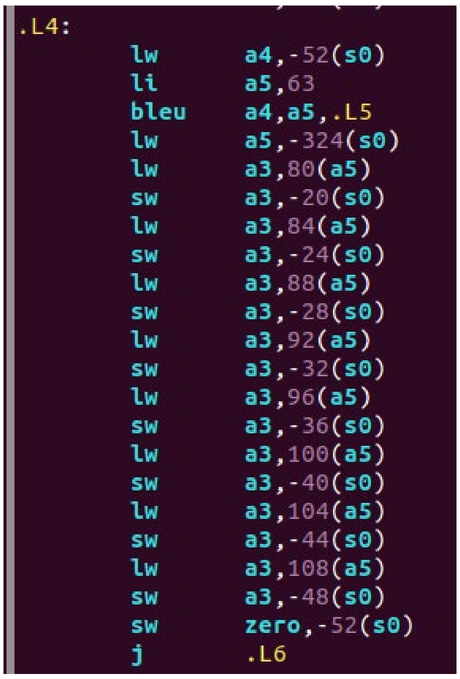
观察到 sha256_final()部分存在一段代码与 reverse 实现的功能一致，因此进行替换，替换时注意寄存器的复用要求，可以进一步优化代码，优化代码如下



```
.L21:
        lw      a5,-68(s0)
        lw      a4,80(a5)

        # hash[i] <- reverse
        lw      a3,-52(s0)
        .insn r 0x33,3,2,a4,a4,a3
        lw a2,-72(s0)
        lw a1,-52(s0)
        add a1,a2,a1
        sb a4,0(a1)

        lw      a4,84(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,88(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,92(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,96(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,100(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,104(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)

        lw      a4,108(a5)
        .insn r 0x33,3,2,a4,a4,a3
        addi a1,a1,4
        sb a4,0(a1)
```

寄存器的复用：

观察到许多处的都有重复加载寄存器的操作，因此如果我们在加载完一次该寄存器之后，就不再更改该寄存器的值，那么就可以减少重复加载寄存器的汇编代码

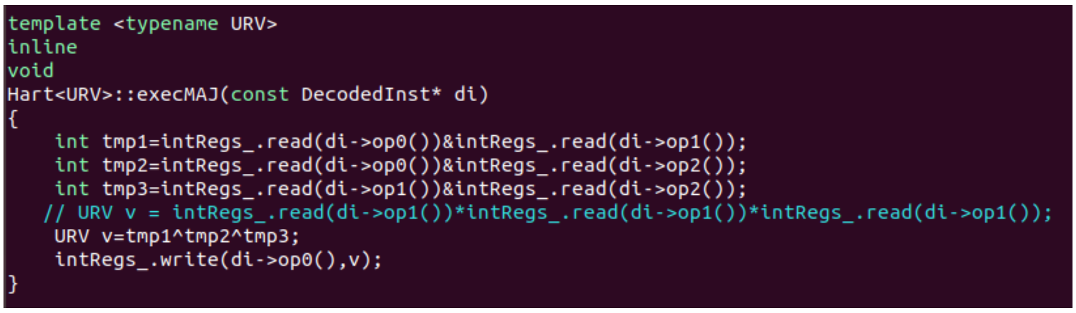

```
.L4:
        lw      a4,-52(s0)
        li      a5,63
        bleu    a4,a5,.L5
        lw      a5,-324(s0)
        lw      a3,80(a5)
        sw      a3,-20(s0)
        lw      a3,84(a5)
        sw      a3,-24(s0)
        lw      a3,88(a5)
        sw      a3,-28(s0)
        lw      a3,92(a5)
        sw      a3,-32(s0)
        lw      a3,96(a5)
        sw      a3,-36(s0)
        lw      a3,100(a5)
        sw      a3,-40(s0)
        lw      a3,104(a5)
        sw      a3,-44(s0)
        lw      a3,108(a5)
        sw      a3,-48(s0)
        sw      zero,-52(s0)
        j       .L6
```

例如 L4 部分，源代码存在多次 lw a5,-324(s0)的部分，但如果把之后 lw 和 sw 的寄存器改成 a3，就可以减少重复加载寄存器的指令

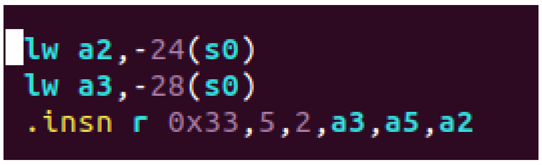
2. 自主拓展指令的设计思路及实现

MAJ:

发现宏定义中，MAJ(x,y,z)没有对应的拓展汇编代码，因此就按照实验一的顺序，重新编译了 whisper，其中 MAJ(x,y,z)中核心代码为，


```
template <typename URV>
inline
void
Hart<URV>::execMAJ(const DecodedInst* di)
{
    int tmp1=intRegs_.read(di->op0())&intRegs_.read(di->op1());
    int tmp2=intRegs_.read(di->op0())&intRegs_.read(di->op2());
    int tmp3=intRegs_.read(di->op1())&intRegs_.read(di->op2());
   // URV v = intRegs_.read(di->op1())*intRegs_.read(di->op1())*intRegs_.read(di->op1());
    URV v=tmp1^tmp2^tmp3;
    intRegs_.write(di->op0(),v);
}
```

因为 MAJ(x,y,z)对应到汇编层面为四操作数，因此就复用了 rd 寄存器，既作为输入又作为输出

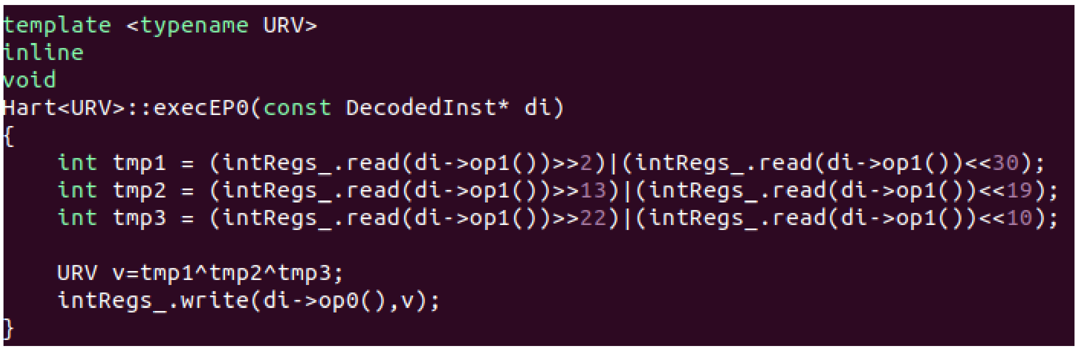

```
lw a2,-24(s0)
 lw a3,-28(s0)
 .insn r 0x33,5,2,a3,a5,a2
```

MAJ 宏定义函数 sha256opt.s 修改部分如上

EP0:

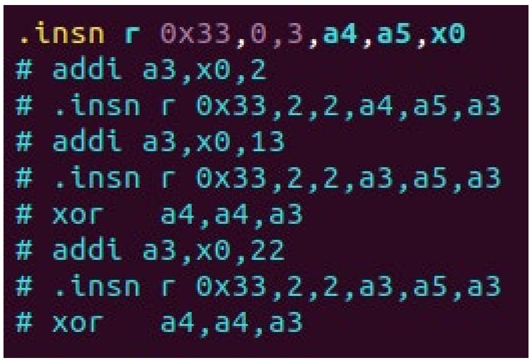
注意到即使实现了 ROTRIGHT 的拓展指令，重复三次一致的操作仍显得臃肿，所以我额外封装了 EP0，进一步优化代码

我给 EP0 设置了 funct7=3, funct3=0，在 Hart.cpp 中核心的逻辑如下，



```
template <typename URV>
inline
void
Hart<URV>::execEP0(const DecodedInst* di)
{
    int tmp1 = (intRegs_.read(di->op1())>>2)|(intRegs_.read(di->op1())<<30);
    int tmp2 = (intRegs_.read(di->op1())>>13)|(intRegs_.read(di->op1())<<19);
    int tmp3 = (intRegs_.read(di->op1())>>22)|(intRegs_.read(di->op1())<<10);

    URV v=tmp1^tmp2^tmp3;
    intRegs_.write(di->op0(),v);
}
```

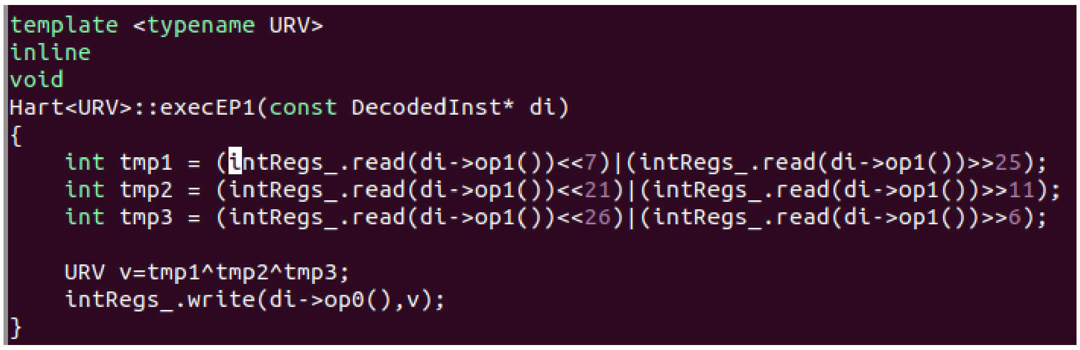
对应到汇编代码有如下的改动，



```
.insn r 0x33,0,3,a4,a5,x0
# addi a3,x0,2
# .insn r 0x33,2,2,a4,a5,a3
# addi a3,x0,13
# .insn r 0x33,2,2,a3,a5,a3
# xor   a4,a4,a3
# addi a3,x0,22
# .insn r 0x33,2,2,a3,a5,a3
# xor   a4,a4,a3
```

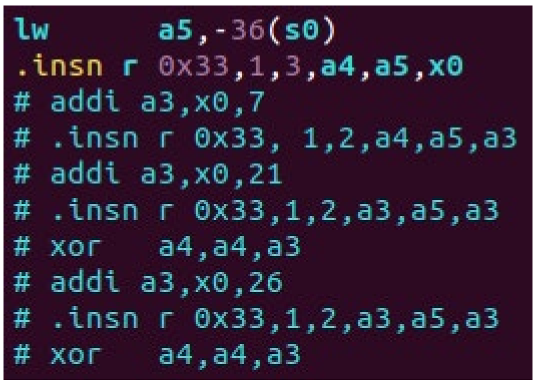
EP1:

EP1 的 funct7=3, funct3=1，也是希望将三次一致的操作封装到一条拓展指令中，



```
template <typename URV>
inline
void
Hart<URV>::execEP1(const DecodedInst* di)
{
    int tmp1 = (intRegs_.read(di->op1())<<7)|(intRegs_.read(di->op1())>>25);
    int tmp2 = (intRegs_.read(di->op1())<<21)|(intRegs_.read(di->op1())>>11);
    int tmp3 = (intRegs_.read(di->op1())<<26)|(intRegs_.read(di->op1())>>6);

    URV v=tmp1^tmp2^tmp3;
    intRegs_.write(di->op0(),v);
}
```

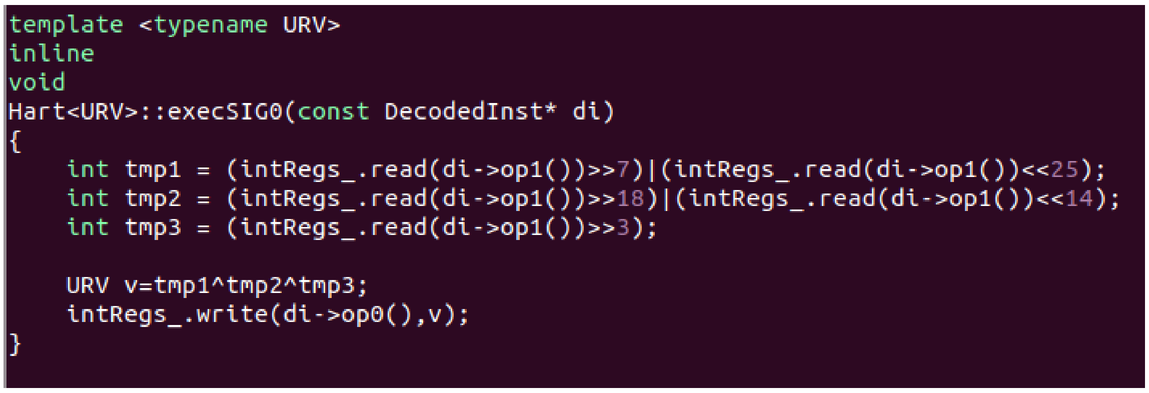
对应到汇编代码有如下的优化



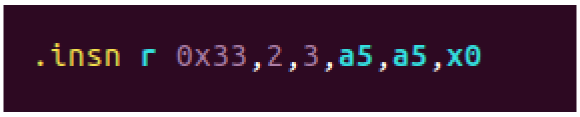
```
lw      a5,-36(s0)
.insn r 0x33,1,3,a4,a5,x0
# addi a3,x0,7
# .insn r 0x33, 1,2,a4,a5,a3
# addi a3,x0,21
# .insn r 0x33,1,2,a3,a5,a3
# xor   a4,a4,a3
# addi a3,x0,26
# .insn r 0x33,1,2,a3,a5,a3
# xor   a4,a4,a3
```

SIG0：

SIG0 的 funct7=3, funct3=2



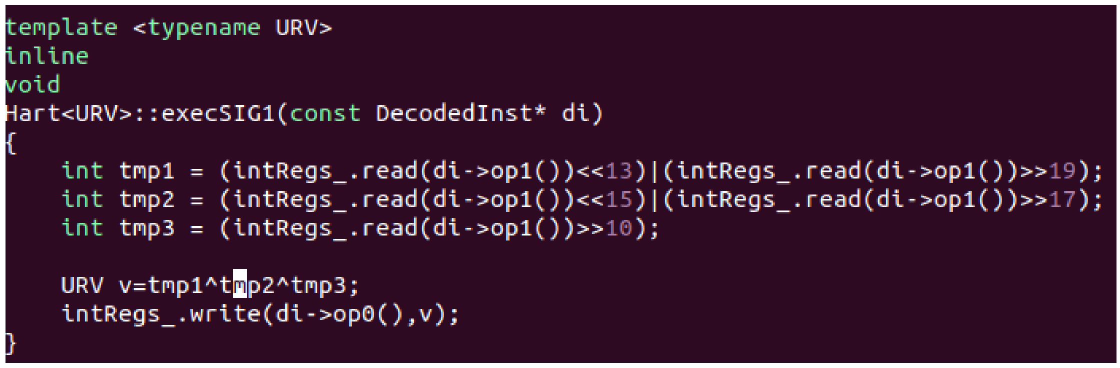
```
template <typename URV>
inline
void
Hart<URV>::execSIG0(const DecodedInst* di)
{
    int tmp1 = (intRegs_.read(di->op1())>>7)|(intRegs_.read(di->op1())<<25);
    int tmp2 = (intRegs_.read(di->op1())>>18)|(intRegs_.read(di->op1())<<14);
    int tmp3 = (intRegs_.read(di->op1())>>3);

    URV v=tmp1^tmp2^tmp3;
    intRegs_.write(di->op0(),v);
}
```



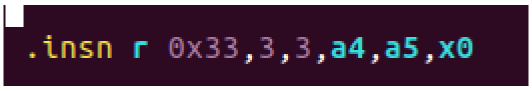
```
.insn r 0x33,2,3,a5,a5,x0
```

SIG1:

SIG1 的 funct7=3, funct3=3



```
template <typename URV>
inline
void
Hart<URV>::execSIG1(const DecodedInst* di)
{
    int tmp1 = (intRegs_.read(di->op1())<<13)|(intRegs_.read(di->op1())>>19);
    int tmp2 = (intRegs_.read(di->op1())<<15)|(intRegs_.read(di->op1())>>17);
    int tmp3 = (intRegs_.read(di->op1())>>10);

    URV v=tmp1^tmp2^tmp3;
    intRegs_.write(di->op0(),v);
}
```

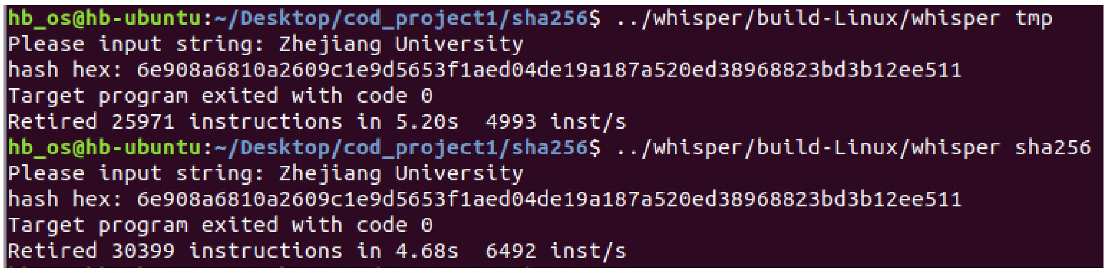


```
.insn r 0x33,3,3,a4,a5,x0
```

注意到，对 MAJ, EP0, EP1, SIG0, SIG1 进行的封装会减少程序的可读性，而且可能不具有迁移性，倘若其他程序用不到这种指令，那么拓展的指令就毫无用处了；但优点在于，如果能够用上，那么汇编代码会大量的减少

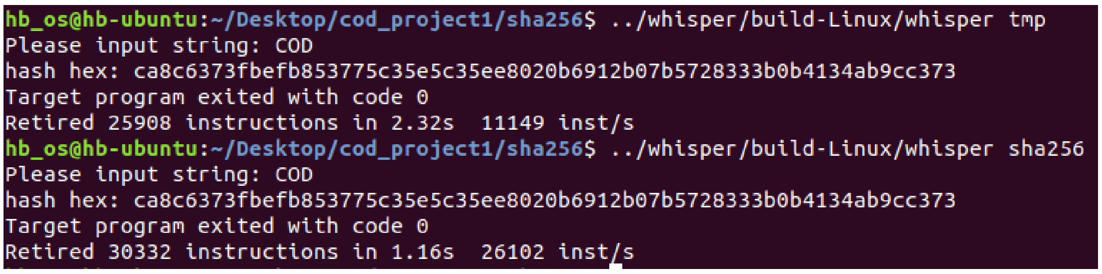
3. 优化结果

优化截图

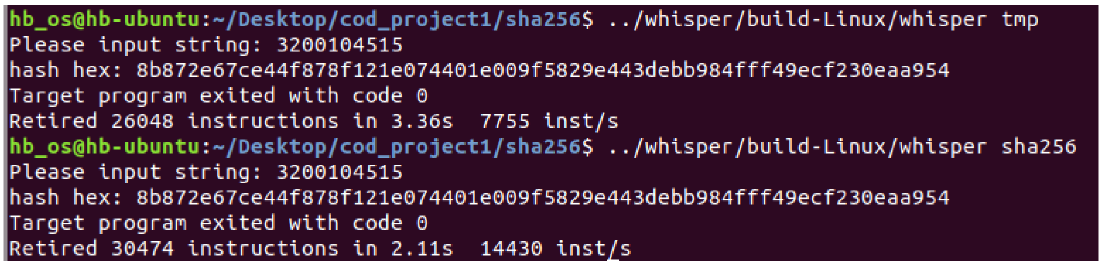


```
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper tmp
Please input string: Zhejiang University
hash hex: 6e908a6810a2609c1e9d5653f1aed04de19a187a520ed38968823bd3b12ee511
Target program exited with code 0
Retired 25971 instructions in 5.20s  4993 inst/s
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper sha256
Please input string: Zhejiang University
hash hex: 6e908a6810a2609c1e9d5653f1aed04de19a187a520ed38968823bd3b12ee511
Target program exited with code 0
Retired 30399 instructions in 4.68s  6492 inst/s
```

上面的结果为优化结果，下面的为原始结果，比较 hash hex 发现程序正确



```
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper tmp
Please input string: COD
hash hex: ca8c6373fbefb853775c35e5c35ee8020b6912b07b5728333b0b4134ab9cc373
Target program exited with code 0
Retired 25908 instructions in 2.32s  11149 inst/s
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper sha256
Please input string: COD
hash hex: ca8c6373fbefb853775c35e5c35ee8020b6912b07b5728333b0b4134ab9cc373
Target program exited with code 0
Retired 30332 instructions in 1.16s  26102 inst/s
```



```
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper tmp
Please input string: 3200104515
hash hex: 8b872e67ce44f878f121e074401e009f5829e443debb984fff49ecf230eaa954
Target program exited with code 0
Retired 26048 instructions in 3.36s  7755 inst/s
hb_os@hb-ubuntu:~/Desktop/cod_project1/sha256$ ../whisper/build-Linux/whisper sha256
Please input string: 3200104515
hash hex: 8b872e67ce44f878f121e074401e009f5829e443debb984fff49ecf230eaa954
Target program exited with code 0
Retired 30474 instructions in 2.11s  14430 inst/s
```

三、 实验收获和心得

本次实验的代码量较小，只需要按着文档一步步进行即可，在优化过程中也注意到了寄存器的复用问题，实现了在使用拓展指令之外的进一步优化，但是在配置工具链的过程中遇到了许多困难，比如 git 仓库在海外，配置 riscv32 的工具链时下载速度较慢，最后采用了直接下载编译好的工具链的方式编译文件，另外还有一开始使用了 riscv32-unknown-linux-gnu-gcc 的工具链，导致即使代码正确，可执行文件也无法正常运行的情况，最后改成了 riscv32-unknown-elf-gcc 才解决。