图片包含 徽标

描述已自动生成

**设计报告**

**题目 RISCV 领域加速指令设计与仿真**

**姓名**

**学号**

**目录**

[1 模拟器的实现思路及测试结果 3](#_Toc153240061)

[1.1 模拟器的实现 3](#_Toc153240062)

[1.1.1 instforms：指令编码 3](#_Toc153240063)

[1.1.2 decode：指令译码 3](#_Toc153240064)

[1.1.3 hart：指令执行 3](#_Toc153240065)

[1.2 模拟器的测试结果 3](#_Toc153240066)

[1.2.1 原有指令的执行 3](#_Toc153240067)

[1.2.2 拓展指令的执行 3](#_Toc153240068)

[2 优化SM3加密算法 3](#_Toc153240069)

[2.1 拓展指令优化 3](#_Toc153240070)

[2.1.1 rotleft和rotright型替换 3](#_Toc153240071)

[2.1.2 reverse型替换 3](#_Toc153240072)

[2.1.3 notand型替换 5](#_Toc153240073)

[2.2 自定拓展指令 5](#_Toc153240074)

[2.2.1 指令组合的选择 5](#_Toc153240075)

[2.2.2 自定拓展指令的实现 6](#_Toc153240076)

[2.2.3 自定拓展指令替换实例 6](#_Toc153240077)

[2.3 汇编代码优化 6](#_Toc153240078)

[2.3.1 立即数合并 6](#_Toc153240079)

[2.3.2 增加寄存器 7](#_Toc153240080)

[2.3.3 a0/a1的直接赋值和使用 8](#_Toc153240081)

[2.3.4 根据物理含义优化 8](#_Toc153240082)

[2.3.5 重复运算优化 9](#_Toc153240083)

[2.4 优化结果 9](#_Toc153240084)

# 1 模拟器的实现思路及测试结果

## 1.1 模拟器的实现

### 1.1.1 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，下面是在 instforms.cpp 中实现对应的编码函数，仅需要修改bits.funct3的值：

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;

  return true;

}

### 1.1.2 decode：指令译码

InstEntry.cpp中对拓展指令进行声明，需要修改name、id和code：

{ "rotleft", InstId::rotleft, 0x4001033, top7Funct3Low7Mask,

InstType::Int,

OperandType::IntReg, OperandMode::Write, rdMask,

OperandType::IntReg, OperandMode::Read, rs1Mask,

OperandType::IntReg, OperandMode::Read, rs2Mask },

{ "rotrigt", InstId::rotrigt, 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 },

在Instld.hpp中声明拓展指令：

rotleft,

rotrigt,

reverse,

notand,

maxId = muladd

在decode.cpp中根据funct3返回对应拓展指令的getEntry：

  else if(funct7 == 2)

    {

      // cube

      if (funct3 == 0) return instTable\_.getEntry(InstId::cube);

      // rotleft

      if (funct3 == 1) return instTable\_.getEntry(InstId::rotleft);

      // rotrigt

      if (funct3 == 2) return instTable\_.getEntry(InstId::rotrigt);

      // reverse

      if (funct3 == 3) return instTable\_.getEntry(InstId::reverse);

      // notand

      if (funct3 == 4) return instTable\_.getEntry(InstId::notand);

    }

### 1.1.3 hart：指令执行

在Hart.cpp中编辑执行函数：

template <typename URV>

inline

void

Hart<URV>::execRotleft(const DecodedInst\* di)

{

  /\* INSERT YOUR CODE  HERE \*/

  uint32\_t rs1 = intRegs\_.read(di->op1());

  uint32\_t rs2 = intRegs\_.read(di->op2());

  URV v = (rs1 << rs2) | (rs1 >> (32 - rs2));

  intRegs\_.write(di->op0(), v);

}

template <typename URV>

inline

void

Hart<URV>::execRotright(const DecodedInst\* di)

{

  /\* INSERT YOUR CODE  HERE \*/

  uint32\_t rs1 = intRegs\_.read(di->op1());

  uint32\_t rs2 = intRegs\_.read(di->op2());

  URV v = (rs1 >> rs2) | (rs1 << (32 - rs2));

  intRegs\_.write(di->op0(), v);

}

template <typename URV>

inline

void

Hart<URV>::execReverse(const DecodedInst\* di)

{

  /\* INSERT YOUR CODE  HERE \*/

  URV rs1 = intRegs\_.read(di->op1());

  URV rs2 = intRegs\_.read(di->op2());

  URV v =  (rs1 >> (24-rs2\*8)) & 0x000000ff;

  intRegs\_.write(di->op0(), v);

}

template <typename URV>

inline

void

Hart<URV>::execNotand(const DecodedInst\* di)

{

  /\* INSERT YOUR CODE  HERE \*/

  URV rs1 = intRegs\_.read(di->op1());

  URV rs2 = intRegs\_.read(di->op2());

  URV v =  ~(rs1) & rs2;

  intRegs\_.write(di->op0(), v);

}

在Hart.cpp中添加拓展指令的label：

&&rotleft,

&&rotrigt,

&&reverse,

&&notand,

在Hart.cpp中添加拓展指令的跳转执行：

 rotleft:

  execRotleft(di);

  return;

 rotrigt:

  execRotright(di);

  return;

 reverse:

  execReverse(di);

  return;

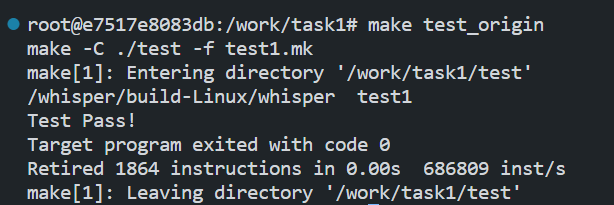
 notand:

  execNotand(di);

  return;

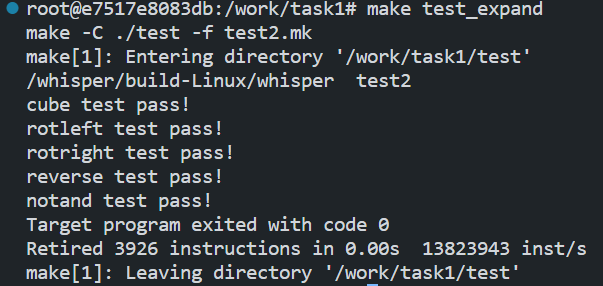
## 1.2 模拟器的测试结果

### 1.2.1 原有指令的执行



可见测试通过。

### 1.2.2 拓展指令的执行



可见测试通过。

最后对task1键入make build指令，完成whisper编译。

# 2 优化SM3加密算法

## 2.1 拓展指令优化

### 2.1.1 rotleft和rotright型替换

在sm3.c中预定义了函数ROTL和ROTR：

#define ROTL(x,n) ((x<<n)|(x>>(32-n)))

#define ROTR(x,n) ((x>>n)|(x<<(32-n)))

发现与指令rotleft、rotright契合，再去汇编文件中查找，替换方式如下：

# slli  a2,a5,7

# srli  a4,a5,25

# or    a4,a4,a2

addi a2, x0, 7

.insn r 0x33, 1,2, a4, a5, a2 # a4= ROTL(a,7)

### 2.1.2 reverse型替换

reverse指令的实际运算是：R[rd] = (R[rs1] >> (24 – R[rs2] \* 8)) & 0x000000ff。在代码中并没有字节翻转的语句，但是sm3.c中的

str[len++] = high >> 24;

str[len++] = high >> 16;

str[len++] = high >> 8;

str[len++] = high;

str[len++] = low >> 24;

str[len++] = low >> 16;

str[len++] = low >> 8;

str[len++] = low;

实际可以通过reverse实现。在汇编中进行如下替换：

# lw    a5,-28(s0)

# srli  a3,a5,24

# lw    a5,-40(s0)

# addi  a4,a5,1

# sw    a4,-40(s0)

# mv    a4,a5

# lw    a5,-36(s0)

# add   a5,a5,a4

# andi  a4,a3,0xff

# sb    a4,0(a5)

# lw    a5,-28(s0)

# srli  a3,a5,16

# lw    a5,-40(s0)

# addi  a4,a5,1

# sw    a4,-40(s0)

# mv    a4,a5

# lw    a5,-36(s0)

# add   a5,a5,a4

# andi  a4,a3,0xff

# sb    a4,0(a5)

# lw    a5,-28(s0)

# srli  a3,a5,8

# lw    a5,-40(s0)

# addi  a4,a5,1

# sw    a4,-40(s0)

# mv    a4,a5

# lw    a5,-36(s0)

# add   a5,a5,a4

# andi  a4,a3,0xff

# sb    a4,0(a5)

# lw    a5,-40(s0)

# addi  a4,a5,1

# sw    a4,-40(s0)

# mv    a4,a5

# lw    a5,-36(s0)

# add   a5,a5,a4

# lw    a4,-28(s0)

# andi  a4,a4,0xff

# sb    a4,0(a5)

# lw    a5,-40(s0)

mv  a6,x0

lw  a3,-28(s0)

.insn r 0x33,3,2,a7,a3,a6

addi    a5,a5,1

sb  a7,0(a5)

addi    a6,a6,1

# lw    a3,-28(s0)

.insn r 0x33,3,2,a7,a3,a6

addi    a5,a5,1

sb  a7,0(a5)

# lw    a3,-28(s0)

addi    a6,a6,1

.insn r 0x33,3,2,a7,a3,a6

addi    a5,a5,1

sb  a7,0(a5)

# lw    a3,-28(s0)

addi    a6,a6,1

.insn r 0x33,3,2,a7,a3,a6

addi    a5,a5,1

addi    a4,a4,8

sw  a4,-40(s0)

sb  a7,0(a5)

代码仅展现low部分，high部分的替换同理。需要注意的是，由于加入了用于计数的寄存器，需要以四行代码（sm3.c）作为整体进行替换。还有一些其他的优化细节（如sw和lw指令的节省）在[2.3](#_2.3_汇编代码优化)节阐释。

### 2.1.3 notand型替换

在sm3.c中，找到预定义函数GG2：

#define GG2(a,b,c) (((a)&(b))|((~a)&(c)))

用到了与非运算，对应到汇编文件，为：

# not   a3,a5

lw  a3,-548(s0)

# and   a5,a3,a5

.insn r 0x33, 4, 2, a5, a7, a3

## 2.2 自定拓展指令

### 2.2.1 指令组合的选择

自定拓展指令的选择有以下几点考量：

* 该指令是否可以用两个寄存器运算，这是能够将指令合并的先决条件；
* 单次优化的性价比，即一组指令在程序中运算的次数，亦即该组指令是否在循环中；
* 优化的重复性，即该组指令是否在程序中反复写及。

综合以上考量，选择slli、add指令组合：

slli    a5,a5,2

add a4,a4,a5

### 2.2.2 自定拓展指令的实现

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0000010 | rs2 | rs1 | 101 | rd | 0110011 | muladd |

muladd rd, rs1, rs2: R[rd] = R[rs1] \* 4 + R[rs2]

执行函数如下：

template <typename URV>

inline

void

Hart<URV>::execMuladd(const DecodedInst\* di)

{

  URV rs1 = intRegs\_.read(di->op1());

  URV rs2 = intRegs\_.read(di->op2());

  URV v =  (rs1 << 2) + rs2;

  intRegs\_.write(di->op0(), v);

}

其他添加拓展指令的步骤不再赘述。

### 2.2.3 自定拓展指令替换实例

# slli  a5,a5,2

# add   a0,a4,a5

.insn r 0x33, 5, 2, a0, a5, a4

## 2.3 汇编代码优化

本模块在不添加拓展指令的前提下，为汇编代码的优化提供思路。

### 2.3.1 立即数合并

代码中常有立即数可以合并的指令。如

（1）

li  a1,2043428864

addi    a1,a1,1305

可以写成

li a2, 2043430169

（2）

slli    a5,a5,1

addi    a4,s0,-540

slli    a5,a5,2

add    a5,a4,a5

可以写成

slli    a5,a5,3

addi    a4,s0,-540

# slli  a5,a5,2

add a5,a4,a5

### 2.3.2 增加寄存器

大部分代码并没有利用好a6、a7的寄存器资源，导致某个经常要读取的memory内容产生很多非必要lw指令。以.L3为例

.L3:

    lw  a5,-20(s0)

    lw  a4,-36(s0)

    add a5,a4,a5

    lbu a5,0(a5)

    mv  a1,a5

    lui a5,%hi(.LC1)

    addi    a0,a5,%lo(.LC1)

    call    printf

    lw  a5,-20(s0)

    addi    a5,a5,1

    sw  a5,-20(s0)

这里-20(s0)被load word了两次，倘若增加一个寄存器a6，就可以避免后面重复的lw了：

.L3:

    # lw    a5,-20(s0)

    lw  a6,-20(s0)

    lw  a4,-36(s0)

    add a5,a4,a6

    lbu a5,0(a5)

    mv  a1,a5

    lui a5,%hi(.LC1)

    addi    a0,a5,%lo(.LC1)

    call    printf

    # lw    a5,-20(s0)

    addi    a6,a6,1

    sw  a6,-20(s0)

须知有一些在call指令之后的lw指令看似重复，实则不然。call指令调用函数之后，memory当中的数值可能发生了改变，如果删去就会变成原先的数据，导致错误；或者寄存器在函数中经历了lw指令的刷新，因此也需要call指令之后重新赋值。

### 2.3.3 a0/a1的直接赋值和使用

a0、a1寄存器通常用于函数返回，直接运算可能导致程序不稳定的情况，但是为了进一步优化程序，可以考虑将指令的rd直接设置为a0、a1寄存器，从而跳过繁琐的mv指令。如

add a5,a4,a5

mv  a0,a5

可以写成

add a0,a4,a5

同理，当需要利用a0、a1寄存器的值时，也可以直接使用：

sw  a0,-36(s0)

sw  zero,-24(s0)

addi    a5,s0,-24

sw  a5,-20(s0)

lw  a5,-36(s0)

lbu a4,3(a5)

可以写成

sw  a0,-36(s0)

sw  zero,-24(s0)

addi    a5,s0,-24

sw  a5,-20(s0)

# lw    a5,-36(s0) # 直接用a0

lbu a4,3(a0)

### 2.3.4 根据物理含义优化

有些指令不需要复杂的条件判断。如sm3.c中的

int u = len % 64;

语句，其在汇编文件中呈现为：

lw  a4,-40(s0)

srai    a5,a4,31

srli    a5,a5,26

add a4,a4,a5

andi    a4,a4,63

然而，len在程序中不可为负，而取模最重要的只有andi指令，其他指令是对len是否为负的处理，如果len是正数，则a5是0，srli和add指令都无意义。因此作以下处理：

# srai  a5,a1,31

# srli  a5,a5,26

# add   a4,a1,a5

andi    a5,a1,63

### 2.3.5 重复运算优化

汇编代码中有一些意义不明的重复运算代码。如sm3.s中第367行至第374行：

xor a4,a4,a5 # a4= a^b^ROTL(c, 15)

# 这里再算一遍a^b^ROTL(c,15)没有意义

# lw    a3,-40(s0) # a

# lw    a5,-36(s0) # b

# xor   a3,a3,a5 # a3= a^b

# lw    a5,-32(s0)

# slli  a2,a5,15

# srli  a5,a5,17

# or    a5,a5,a2

# xor   a5,a3,a5 # a5= a^b^ROTL(c,15)

也许是由于该汇编代码是脚本自动生成的，在生成

x[5] = P1(x[0],x[1],x[2],x[3],x[4]);

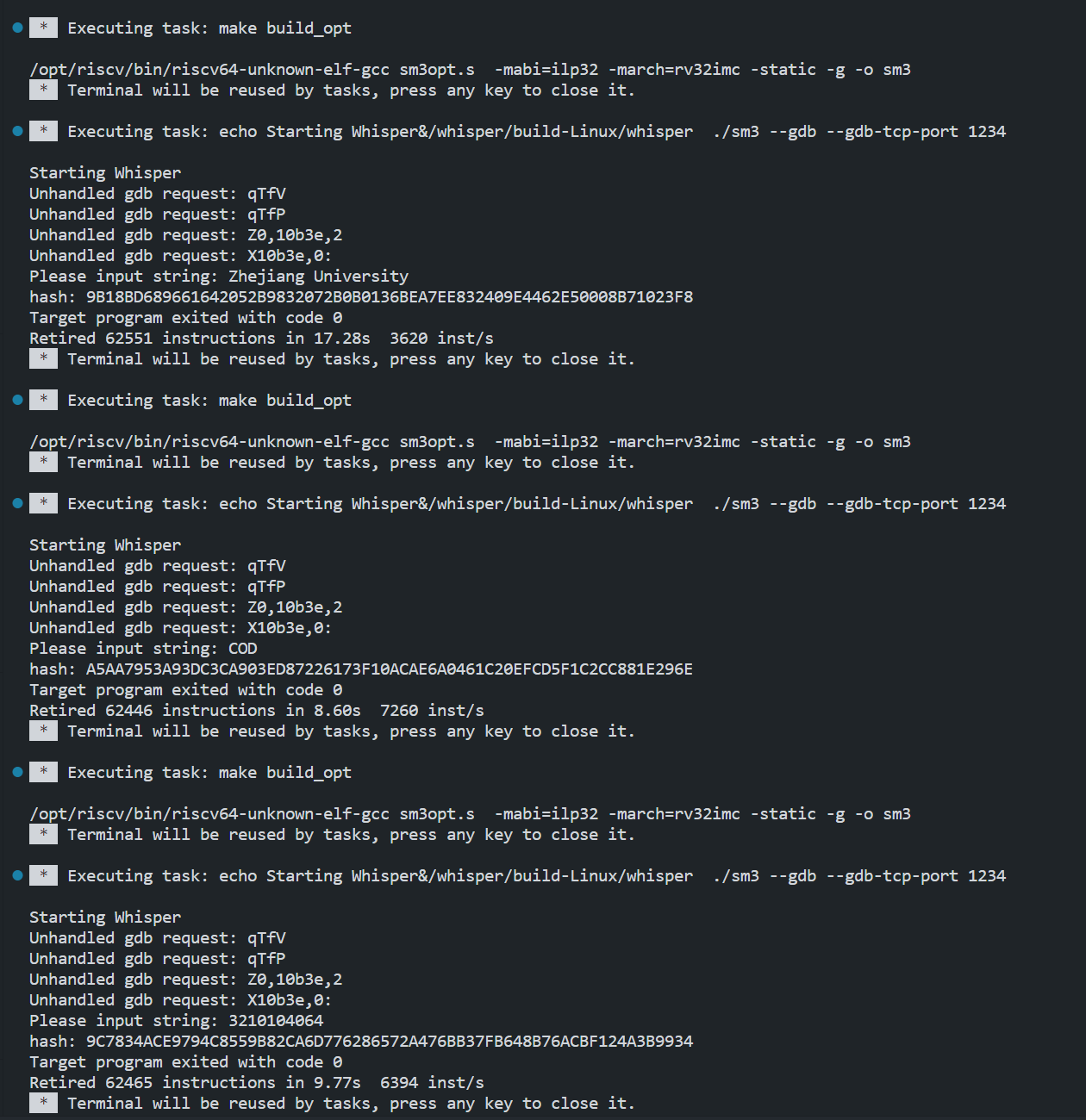
中的

P2((a)^(b)^ROTL((c),15)

时，重复计算了一次(a)^(b)^ROTL((c)，这是完全可以避免的。

## 2.4 优化结果

对task2进行优化汇编调试，分别输入“Zhejiang University”、“COD”和学号“3210104064”，终端得到加密哈希值如下：



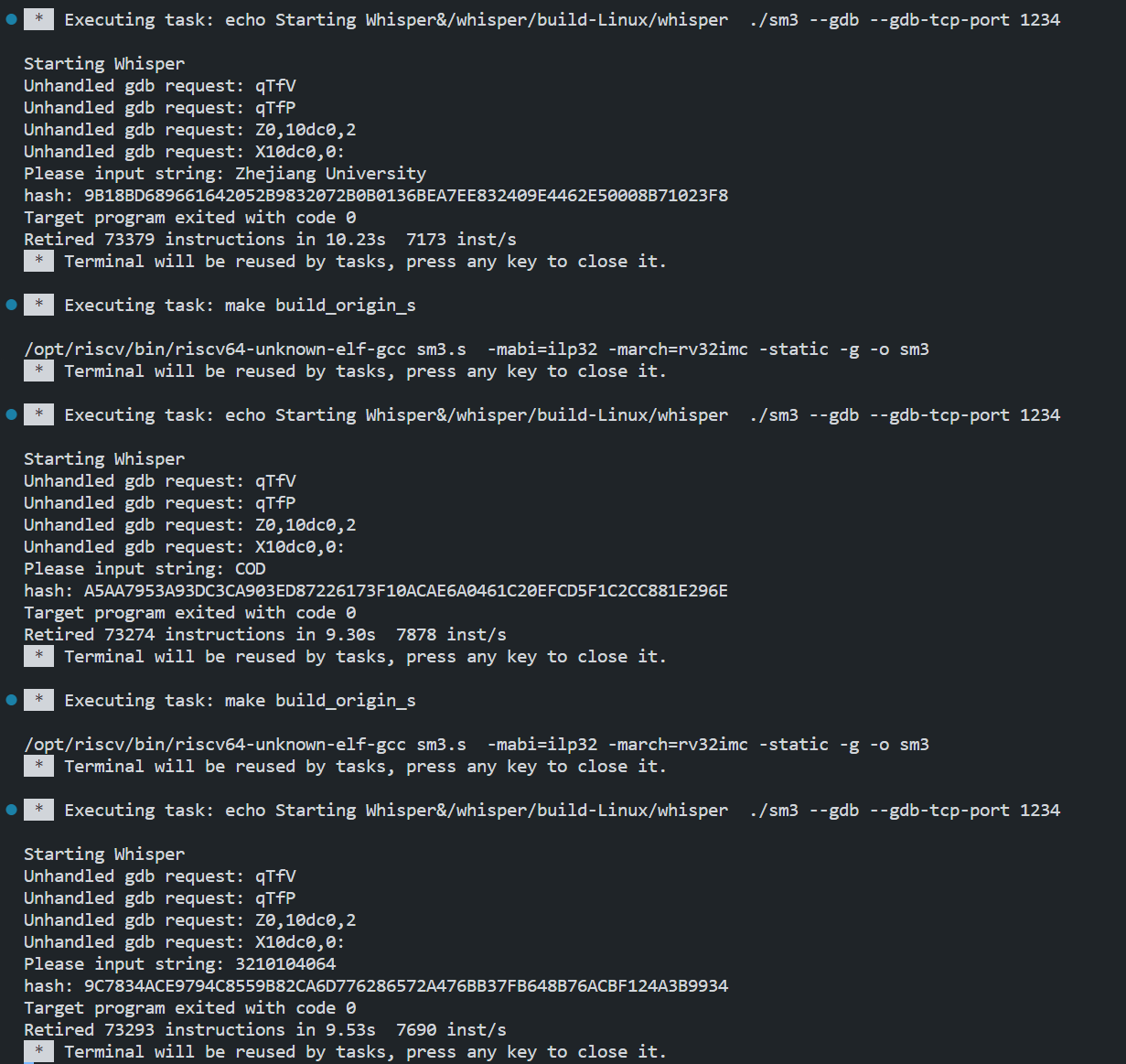
得到的哈希值分别是：

9B18BD689661642052B9832072B0B0136BEA7EE832409E4462E50008B71023F8；

A5AA7953A93DC3CA903ED87226173F10ACAE6A0461C20EFCD5F1C2CC881E296E；

9C7834ACE9794C8559B82CA6D776286572A476BB37FB648B76ACBF124A3B9934。

再对task2进行原始汇编程调试验证：



经验证，哈希值与优化得到的哈希值吻合。

|  |  |  |  |
| --- | --- | --- | --- |
| 输入字符串 | Zhejiang University | COD | 3210104064 |
| 原始指令数 | 73379 | 73274 | 73293 |
| 优化指令数 | 62551 | 62446 | 62465 |

平均减少指令10828条。