浙江大学

《计算机组成与设计》课程报告

题目 RISC-V 领域加速指令设计与仿真

姓名 _____

学号 _____

专业 电子科学与技术

教学班 周四下午第 6、7、8 节

RISC-V 领域加速指令设计与仿真

1.实验简介与目的

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。

2.任务 1：仿真器拓展

2.1. 任务要求

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]
```

图 1 待拓展 R-Type 指令

2.2.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，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。

根据案例示意，只需修改 func3 字段的值即可实现以上拓展指令的编码函数。按要求修改 instforms.cpp 文件，修改的部分在 Code1 中列出。

此外，需要在对应头文件 instforms.hpp 中添加函数声明，此步已事先完成，不需要更改。

```
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;
}
```

Code 1 instforms.cpp **修改内容**

2.3.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。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。

根据样例，发现因为都是 R 型指令，所以修改 InstEntry.cpp 时只需要改变样例的 name，id 和 code，其余和 cube 保持一致即可。InstEntry.cpp 添加的内容如 Code2 所示。

```
// InstEntry.cpp 修改内容

     { "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 },

// InstId.hpp 修改内容
// 在枚举中增加对应指令，并且修改 maxId 为 notand
     cube,
     rotleft,
     rotright,
     reverse,
     notand,

     maxId = notand,
```

Code 2 InstEntry.cpp 和 InstId.hpp 修改内容

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，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，如 Code3 所示。

```
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);
    // xoradd
    if (funct3 == 5) return instTable_.getEntry(InstId::xoradd);
  }
```

Code 3 decode.cpp **修改内容**

2.4. 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。从 cube 指令的译码过程样例可知，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，完成其他四条拓展指令的执行过程。

在完成执行函数后，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），添加其他拓展指令的 label 和相应的跳转执行。

Hart.cpp 中添加的内容如 Code4 所示。

```
//第 1004 行
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)
{
  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)
{
  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)
{
  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)
{
  URV rs1 = intRegs_.read(di->op1());
  URV rs2 = intRegs_.read(di->op2());
  URV v = ~(rs1)&rs2;
  intRegs_.write(di->op0(), v);
}

//第 6032 行
     &&cube,
     &&rotleft,
     &&rotright,
     &&reverse,
     &&notand,
     &&xoradd,

//第 6201 行
  cube:
  execCube(di);
  return;

  rotleft:
  execRotleft(di);
  return;

  rotright:
  execRotright(di);
  return;

  reverse:
  execReverse(di);
  return;

  notand:
  execNotand(di);
```

Code 4 Hart.cpp 修改内容

2.5.测试

```
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  8317796 inst/s
make[1]: Leaving directory '/work/task1/test'
```

图 2 测试结果

利用 makefile 文件编译上述文件，运行 make test_expand，终端输出如图 2，可见拓展指令执行正确。

3. 任务 2：优化 SM3 加密算法

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。

3.1.优化过程和思路

首先注意到，循环左移 rotleft 和循环右移 rotright 可以相互转换。对于 32 位寄存器中的数据，循环左移 N（N<32）位等价于循环右移 32-N 位。因此两种操作的优化思路相同。

拓展前，实现循环左移的 RISC-V 汇编指令（除去 lw 等数据加载指令）可写为：

```
slli    a3,a5,15
srli    a5,a5,17
or      a5,a5,a3
```

这三条指令表示将寄存器 a5 中的值循环左移 15 位，结果存在寄存器 a5 中。

同样，循环右移可写为：

```
srli    a3,a5,20
slli    a5,a5,12
or      a5,a5,a3
```

这三条指令表示将寄存器 a5 中的值循环右移 20 位，结果存在寄存器 a5 中。

经过优化后，上述循环左移 15 位指令可以写为：

```
addi a3, x0, 15
.insn r 0x33,1,2,a5,a5,a3
```

循环右移 20 位指令可写为：

```
addi a3, x0, 12    #12=32-20
.insn r 0x33,1,2,a5,a5,a3
```

.insn r 表示 R 型指令；0x33 为 opcode 字段；1，2 为拓展指令 rotleft 的 func3 和 func7 字段，在图 1 中有说明；a5 为目的寄存器；a5、a3 为源寄存器。按照这种方式，将 sm3.s 中所有循

环左移/右移代码进行优化。经过优化后，单次循环左移指令减少了一条指令，但是由于宏定义和循环中会多次使用循环左移，且单次循环中循环左移的操作也较为频繁，故实际运行时会有更显著的效果。

拓展前，原文件中实现 notand 的汇编代码如下：

```
lw      a5,-556(s0)
not     a3,a5
lw      a5,-548(s0)
and     a5,a3,a5
```

利用拓展指令 notand，上述代码可以改为：

```
lw a3,-556(s0)
lw a5,-548(s0)
.insn r 0x33,4,2,a5,a3,a5
```

其中 4 为 notan 的 func3 字段。按照这种方法，优化 sm3.s 文件中所有相同的操作。同样，虽然单次运行只减少了 1 条指令，但是由于循环体的存在，其实际运行效果会更显著。

原文件中使用反转字节（reverge）的操作基本没有，无需优化。

3.2.测试

首先测试运行原文件 sm3.s。按要求输入 Zhejiang University、COD、学号（3210101017），终端输出的结果如下：

```
Please input string: Zhejiang University
hash: 9B18BD689661642052B9832072B0B0136BEA7EE832409E4462E50008B71023F8
Target program exited with code 0
Retired 73379 instructions in 29.21s  2512 inst/s

Please input string: COD
hash: A5AA7953A93DC3CA903ED87226173F10ACAE6A0461C20EFCD5F1C2CC881E296E
Target program exited with code 0
Retired 73274 instructions in 10.41s  7037 inst/s

Please input string: 114514
hash: 1919810
Target program exited with code 0
Retired 73293 instructions in 14.38s  5097 inst/s
```

将优化后的汇编代码保存为 sm3opt.s，测试运行优化后的文件，终端输出如下：

```
Please input string: Zhejiang University
hash: 9B18BD689661642052B9832072B0B0136BEA7EE832409E4462E50008B71023F8
Target program exited with code 0
Retired 72491 instructions in 12.58s  5760 inst/s
```

```
Please input string: COD
hash: A5AA7953A93DC3CA903ED87226173F10ACAE6A0461C20EFCD5F1C2CC881E296E
Target program exited with code 0
Retired 72386 instructions in 8.92s  8113 inst/s

Please input string: 114514
hash: 1919810
Target program exited with code 0
Retired 72405 instructions in 12.94s  5594 inst/s
```

对比两次运行结果的 hash 输出，发现完全相同，故优化后的代码能够正确运行。将运行结果对比列为表 1。

表 1 运行结果对比

	Zhejiang University	COD	
优化前的指令条数	73379	73274	
优化后的指令条数	72491	72386	
差值	888	888	

从表 1 可知，优化使得运行的指令数减少了 888 条。

3.3.自设计拓展指令

设计 xoradd 指令 xoradd rd，rs1，rs2，实现的功能为 rd=rd⊕rs1+rs2。原汇编代码为：

```
xor a1,a3,a1
add a1,a4,a1
```

优化的代码为：

```
.insn r 0x33,5,2,a1,a3,a4
```

实现的步骤为：

- instforms.hpp 中增加解码函数声明 bool encodeXoradd(unsigned rd, unsigned rs1, unsigned rs2);
- instforms.cpp 中写对应函数实现

```
bool
RFormInst::encodeXoradd(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 = 5;
  bits.rs1 = rs1v & 0x1f;
```

```
  bits.rs2 = rs2v & 0x1f;
  bits.funct7 = 2;
  return true;
}
```

- InstEntry.cpp 中增加对应条目

```
  { "xoradd", InstId::xoradd, 0x4005033, top7Funct3Low7Mask,
  InstType::Int,
  OperandType::IntReg, OperandMode::Write, rdMask,
  OperandType::IntReg, OperandMode::Read, rs1Mask,
  OperandType::IntReg, OperandMode::Read, rs2Mask },
```

- InstId.hpp 增加对应指令 label 的枚举量，并修改 maxId

```
    cube,
    rotleft,
    rotright,
    reverse,
    notand,
    xoradd,

    maxId = xoradd,
```

- decode.cpp 中增加对应译码内容

```
    if (funct3 == 5) return instTable_.getEntry(InstId::xoradd);
```

- Hart.hpp 中增加对应执行函数声明

```
    void execXoradd(const DecodedInst*);
```

- Hart.cpp 中完成实际执行运算函数的定义

```
template <typename URV>
inline
void
Hart<URV>::execXoradd(const DecodedInst* di)
{
    uint32_t rd = intRegs_.read(di->op0());
    uint32_t rs1 = intRegs_.read(di->op1());
    uint32_t rs2 = intRegs_.read(di->op2());
    uint32_t v = (rd ^ rs1) + rs2;
    intRegs_.write(di->op0(), v);
}
```

- Hart.cpp 添加拓展指令的 label 以及相应的跳转执行

```
 //Expland
 &&cube,
 &&rotleft,
 &&rotright,
 &&reverse,
 &&notand,
 &&xoradd,
```

```
xoradd:
execXoradd(di);
return;
```

3.4.测试

测试运行修改后的 sm3opt.s 文件，终端输出如下：

```
Please input string: Zhejiang University
hash: 9B18BD689661642052B9832072B0B0136BEA7EE832409E4462E50008B71023F8
Target program exited with code 0
Retired 72475 instructions in 22.91s  3163 inst/s

Please input string: COD
hash: A5AA7953A93DC3CA903ED87226173F10ACAE6A0461C20EFCD5F1C2CC881E296E
Target program exited with code 0
Retired 72370 instructions in 9.85s  7344 inst/s

Please input string: 114514
hash: 1919810
Target program exited with code 0
Retired 72389 instructions in 15.59s  4643 inst/s
```

和上文对比，可得出加入 xoradd 优化指令后的文件执行所花费的指令条数减少了 904 条。

4.问题思考

本次设计的目标是实现指令条数的优化，减少所需执行的指令条数。但是这么做在实际上是否真的能提升 CPU 的运行效率呢？

观察拓展的指令，可以发现它们都需要多次 ALU 运算，都包含三个操作数，因此理论上，可以将 SM3 密码算法中所有的三操作数复杂计算，如 FF_j（X，Y，Z）、GG_j（X，Y，Z）、P_0（X）、P_1（X）等，都拓展出一条指令，原 C 程序中的部分宏，如 P2，也可以用一条指令来描述，这样进行优化后，将多条逻辑操作和算数操作简化为一条，可以大大减少指令条数。但是，由于这些指令有时需要取出目的寄存器 rd 中的值进行计算，即将目的寄存器 rd 也当作源寄存器 rs，这就导致此类的指令不能适用原来的 RISC-V CPU 的数据通路和控制逻辑。更重要的是，这些指令需要多次 ALU 运算，例如 rotleft 需要进行三次 ALU 操作（左移、右移、逻辑或），如果仍然使用原来的 ALU，它们就不能在一个周期内完成，这就导致它们不能使用流水线的方式运行，而流水线恰好是 RISC-V 架构中优化程序运行效率、运行处理大量指令的最重要方式之一。

为了解决这个问题，可以将原来的 ALU 替换成支持多次算数和逻辑运算、支持多操作数输入的更高级的 ALU 单元，同时更改控制逻辑，并在数据通路中加入相应的数据选择器。这么做可以继续使用流水线，但仍需注意的是，由于流水线的时钟周期取决于延迟最高的流

水阶段，而成倍的增加 EX 级的 ALU 运算次数会成倍增加 EX 级的延迟，如果 EX 的延迟大到足以影响整个流水线的时钟周期，这种优化是缺乏效率的。此外，这种更改会增加硬件成本，也会使得数据通路和控制逻辑变得更加复杂。

综上，如果考虑流水线技术，本次设计还需在硬件层面上进行设计和优化，在流水线技术的前提下，减少指令条数才是真正有意义的。

5.总结

本次设计实现了 RISC-V 仿真器的建立和运行，接触了虚拟环境、容器等概念，掌握了拓展 RISC-V 指令的设计方法，并以此对实际问题 SM3 密码杂凑算法进行了优化。本次设计加深了我对 RISC-V 指令集的机器码、指令类型等概念的认识，同时了解了如何调试、分析汇编程序和高级语言程序，并在此基础上通过对汇编指令的改写和优化实现程序性能的提高。