

# Project1: RISCV-Simulator

3190102060 黄嘉欣 信工 1903 班

## 一、实验目的

- ① 构建 RISC-V 指令集仿真器应用，使用 RISC-V 工具链编译得到二进制文件，通过仿真器模拟 RISC-V 程序的运行；
- ② 根据 RV32I 指令集，完成仿真器编码、译码与执行过程中的部分内容，得到一个功能完整的指令集仿真器；
- ③ 在实验过程中了解 RISC-V 仿真器的结构及其运行过程，加深对 RISC-V 指令结构及其实现的认识。

## 二、代码及实现思路

### ① instforms.cpp: 指令编码

RISC-V 指令集仿真器在 `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，与 RV32I 指令集相对应。各编码函数的具体实现在 `instforms.cpp` 中完成，其输入参数是寄存器操作数（和立即数操作数）。对各种指令及指令类型，需要根据 RISC-V 编码规则为每个结构体内容赋值，以完成编码。各结构体定义汇总如下：

表 2.1 结构体定义汇总

指令类型	指令	结构体
I-format	LB	<pre>struct {     unsigned opcode : 7;     unsigned rd      : 5;     unsigned funct3  : 3;     unsigned rs1     : 5;     int      imm      : 12; } fields;</pre>
	SLLI	<pre>struct {     unsigned opcode : 7;     unsigned rd      : 5;     unsigned funct3  : 3;     unsigned rs1     : 5;     unsigned shamt   : 5;     unsigned top7    : 7; } fields2;</pre>

B-format	BEQ	<pre> struct {     unsigned opcode : 7;     unsigned imm11 : 1;     unsigned imm4_1 : 4;     unsigned funct3 : 3;     unsigned rs1 : 5;     unsigned rs2 : 5;     unsigned imm10_5 : 6;     int imm12 : 1; } bits; </pre>
S-format	SB	<pre> struct {     unsigned opcode : 7;     unsigned imm4_0 : 5;     unsigned funct3 : 3;     unsigned rs1 : 5;     unsigned rs2 : 5;     int imm11_5 : 7; } bits; </pre>
U-format	LUI	<pre> struct {     unsigned opcode : 7;     unsigned rd : 5;     int imm : 20; } bits; </pre>
J-format	JAL	<pre> struct {     unsigned opcode : 7;     unsigned rd : 5;     unsigned imm19_12 : 8;     unsigned imm11 : 1;     unsigned imm10_1 : 10;     int imm20 : 1; } bits; </pre>

可以发现，各个结构体中将 32 位指令分为了操作码、寄存器操作数、功能码、立即数等多个部分，与 RV32I 指令集的规定格式一一对应。当结构体中某个立即数的类型为 `int` 时，意味着它包含着输入立即数的正负性，需要注意赋值时的截断问题。在此基础上，可以编写以下六条指令的编码函数为：

#### 1) LB 指令：

```

bool
IFormInst::encodeLb(unsigned rdv, unsigned rs1v, int offset)
{
    if (rdv > 31 or rs1v > 31)

```

```
    return false; // 寄存器最大索引为 31

    if (offset >= (1 << 11) or -offset < -(1 << 11))
        return false; // 立即数长度为 12 位

    fields.opcode = 0x03; // LB 的 opcode 为 3
    fields.rd = rdv & 0x1f; // 只取低 5 位, 即小于 32
    fields.funct3 = 0; // LB 的 funct3 为 0
    fields.rs1 = rs1v & 0x1f; // 同 rd

    // offset value checked in preceding code.
    #pragma GCC diagnostic push // 屏蔽局部代码警告
    #pragma GCC diagnostic ignored "-Wconversion"
    fields.imm = offset;
    #pragma GCC diagnostic pop

    return true;
}
```

实现思路: 根据 I-format 指令类型中的 LB 指令结构体定义, 寄存器操作数为 5 位(最大值为 31), 偏移立即数为 12 位。由于 LB 类型的偏移立即数为符号数, 故 offset 范围应在  $[-2^{11}, 2^{11}-1]$  之间。对输入参数, 可以先行判断其是否超出理论范围, 将不合要求者剔除。由 RV32I 指令集规则, 可以确定 LB 指令的操作码为 3, 功能码 funct3 为 0; 由于寄存器索引最大为 31, 故对输入的寄存器参数, 只能取其低 5 位并赋值到结构体的寄存器操作数中。除此之外, fields.imm 为符号数且与 offset 的长度不同, 可能会导致赋值时的截断, 从而引起编译器警告。因为已经对 offset 的范围进行过判断限制, 截断后取值不会发生错误, 故可将此警告屏蔽, 同时确定 fields.imm 的值。

## 2) SLLI 指令:

```
bool
IFormInst::encodeSlli(unsigned rd, unsigned rs1, unsigned shamt)
{
    if (rd > 31 or rs1 > 31)
        return false; // 寄存器最大索引为 31

    if (shamt > 31)
        return false; // 偏移量长度为 5 位, 且只能为正数
```

```
fields2.opcode = 0x13; // SLLI 的 opcode 为 13
fields2.rd = rd & 0x1f; // 只取低 5 位, 即小于 32
fields2.funct3 = 0x01; // SLLI 的 funct3 为 1
fields2.rs1 = rs1 & 0x1f; // 同 rd
fields2.shamt = shamt & 0x1f; // 只取低 5 位, 非符号数
fields2.top7 = 0x00; // SLLI 的高七位为 0

return true;
}
```

实现思路: 与 LB 指令同理, 在对结构体内容赋值之前, 可以对输入参数进行范围判断, 剔除不合要求的调用。由于 SLLI 指令的立即数偏移只有 5bit, 且为非符号数, 故其应不大于 31。根据 RV32I 指令集, 可以确定 SLLI 指令的操作码为 13, 功能码 funct3 为 1, 高 7 位码为 0; 寄存器操作数、立即数偏移都可由输入参数确定, 注意其位数要求, 取对应输入的低 5 位即可。

### 3) BEQ 指令:

```
bool
BFormInst::encodeBeq(unsigned rs1v, unsigned rs2v, int imm)
{
    if (rs1v > 31 or rs2v > 31)
        return false; // 寄存器最大索引为 31

    if (imm & 0x01)
        return false; // 立即数最低位必须为 0

    if (imm >= (1 << 12) or -imm < -(1 << 12))
        return false; // 立即数长度为 13 位

    bits.opcode = 0x63; // 0b'1100011
    bits.imm11 = (imm >> 11) & 1; // 取立即数的第 11 位(记最低位为第 0 位)
    bits.imm4_1 = (imm >> 1) & 0x0f; // 取立即数的第 1 到第 4 位
    bits.funct3 = 0; // BEQ 的 funct3 为 0
    bits.rs1 = rs1v & 0x1f;
    bits.rs2 = rs2v & 0x1f;
    bits.imm10_5 = (imm >> 5) & 0x3f; // imm[10:5]
    bits.imm12 = 0;
    if ((imm >> 12) & 0x01) // imm[12] = 1, 负数
        bits.imm12 = -1;

    return true;
}
```

```
}
```

实现思路：由指令集规则，**B-format** 指令类型的立即数被划分为多个部分分布在指令当中，尽管如此，对应的结构体内容赋值只需要对输入立即数进行移位、做与运算即可完成。需要注意的是，虽然输入的立即数长度为 13 位，但其最低位 `imm[0]` 始终为 0，故可作为前提判断，剔除不合规则的调用；偏移立即数 `imm` 可为负数，故其初步的范围应在  $[-2^{12}, 2^{12}-1]$  之间；除此之外，由于结构体最高位 `bits.imm12` 长度为 1，且是 `int` 类型，当输入立即数最高位为 1，即为负数时，应为 `bits.imm12` 赋值 -1。剩余结构体内容（操作码、功能码等）与 **LB** 指令和 **SLLI** 指令类似，由 **RV32I** 指令集及输入参数即可确定，故不再赘述。

#### 4) SB 指令:

```
bool
SFormInst::encodeSb(unsigned rs1v, unsigned rs2v, int imm)
{
    if (rs1v > 31 or rs2v > 31)
        return false; // 寄存器最大索引为 31

    if (imm >= (1 << 11) or -imm < -(1 << 11))
        return false; // 立即数长度为 12 位

    bits.opcode = 0x23; // 0b'0100011
    bits.imm4_0 = imm & 0x1f; // 取 imm 的低 5 位
    bits.funct3 = 0; // SB 的 funct3 为 0
    bits.rs1 = rs1v & 0x1f;
    bits.rs2 = rs2v & 0x1f;

    // Imm value checked in preceding code.
    #pragma GCC diagnostic push // 屏蔽局部代码警告
    #pragma GCC diagnostic ignored "-Wconversion"
    bits.imm11_5 = (imm >> 5) & 0x7f; // 取 imm[11:5], 符号数
    #pragma GCC diagnostic pop

    return true;
}
```

实现思路：与前述指令的实现类似，虽然 **S-format** 指令类型的结构体定义有所不同，但 **SB** 指令的参数范围、结构体内容的确定方法都与 **LB**、**BEQ** 等指令相同，

故不再赘述。需要注意的是, `bits.imm11_5` 包含了偏移立即数的符号, 且长度与 `imm` 不同, 但由于已对参数大小进行过限制, 故可以将编译器警告屏蔽。

#### 5) LUI 指令:

```
bool
UFormInst::encodeLui(unsigned rdv, int immed)
{
    if (rdv > 31)
        return false; // 寄存器最大索引为 31

    if (immed >= (1 << 19) or -immed < -(1 << 19))
        return false; // 立即数长度为 20 位

    bits.opcode = 0x37; // 0b'0110111
    bits.rd = rdv & 0x1f;

    // Imm value checked in preceding code.
    #pragma GCC diagnostic push
    #pragma GCC diagnostic ignored "-Wconversion"
    bits.imm = immed & 0xfffff; // imm[31:12], 符号数
    #pragma GCC diagnostic pop

    return true;
}
```

实现思路: 显然, 根据指令集规则, U-format 指令类型的结构体定义与其他指令类型相比较为简单, 操作码由指令集给出, 而寄存器操作数和立即数可由输入参数确定。由于在 RISC-V 中, LUI 取的是一个 32 位数的高 20 位, 即输入的 20 位立即数 `imm` 对应于 `imm[31:20]`, 故在对 `bits.imm` 赋值时取 `imm` 的低 20 位即可, 同时可将编译器报警信息屏蔽, 理由同 LB 指令, 不再赘述。

#### 6) JAL 指令:

```
bool
JFormInst::encodeJal(uint32_t rdv, int offset)
{
    if (rdv > 31)
        return false; // 寄存器最大索引为 31

    if (offset & 0x01)
```

```
    return false; // 立即数最低位必须为 0

    if (offset >= (1 << 20) or -offset < -(1 << 20))
        return false; // 立即数长度为 21 位

    bits.opcode = 0x6f ; // 0b'1101111
    bits.rd = rdv & 0x1f;
    bits.imm19_12 = (offset >> 12) & 0xff; // 取 imm[19:12]
    bits.imm11 = (offset >> 11) & 1; // imm[11]
    bits.imm10_1 = (offset >> 1) & 0x3ff; // imm[10:1]
    bits.imm20 = 0;
    if ((offset >> 20) & 0x01) // imm[20] = 1, 负数
        bits.imm20 = -1;

    return true;
}
```

实现思路: 与 BEQ 指令类似, 由于 JAL 指令中的偏移立即数最低位必为 0, 故可将其作为一个输入的前提判断。在结构体立即数的赋值方面, 只需对输入的立即数进行相应的移位、与运算即可完成。但与 LUI 等指令不同的是, 结构体立即数最高位 `bits.imm20` 为 `int` 类型, 且只有一位, 若直接由偏移立即数移位赋值, 只会等于 0 或 1, 从而导致错误发生。因此, 我们需要对输入的偏移立即数的正负进行判断, 并据此对 `bits.imm20` 赋值。

## ② decode.cpp: 指令译码

由 RV32I 指令集, 每个操作码空间都对应了一种类型的指令, 根据 `funct3` 等功能码的不同, 又可将这些指令细分为单独的指令条目。通过指令编码, 我们已经完成了结构体中内容的赋值; 而在指令译码部分, 我们将会根据前述工作对各指令的操作数 `op` 及其指令条目进行确定。对一般的指令, 不妨记为 `inst rd, rs1, rs2`, 有 `op0=rd, op1=rs1, op2=rs2`; 而对一些特殊的指令, 需要先将其格式映射为三个 (或两个) 操作数, 确定好对应关系后再对 `op` 进行赋值。除此之外, 根据指令集规定, 若单个操作码空间对应有多条指令, 则可以通过 `funct3`、`funct7` 等功能码对其进行区分译码。综上, 可以完成如下译码函数:

### 1) 15:

```
15: // 00101 U-form
    // opcode: 0010111, Instruction: AUIPC
```

```
{
    UFormInst uform(inst);
    op0 = uform.bits.rd;
    op1 = uform.immed(); // bits.imm << 12, 20 位扩充到 32 位
    return instTable_.getEntry(InstId::auipc);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 根据操作码 `opcode=0010111`, 可以查表得到对应指令为 `AUIPC`。由于其为 `U-format` 指令类型, 与 `LUI` 类似, 操作数只含有一个寄存器索引和一个立即数, 因此只需令 `op0=rd`, `op1=imm` 即可。通过指令编码函数, 我们已经将输入参数值保存到了结构体 `bits` 中, 可以直接读取赋值。需要注意的是, 考虑到 `instforms.hpp` 文件中定义了函数 `immed()` 用于将分散的各部分立即数合并为一个整体, 在此可以直接利用它得到 `op1`。完成赋值后, 由于 `U-format` 类型不包含功能码, 故可以直接返回对应条目。

## 2) 18:

```
18: // 01000 S-form
// opcode: 0100011, Instructions: SB/SH/SW
{
    // store rs2, rs1, offset
    SFormInst sform(inst);
    op0 = sform.bits.rs2;
    op1 = sform.bits.rs1;
    op2 = sform.immed(); // bits.imm11_5 << 5 | bits.imm4_0
    unsigned funct3 = sform.bits.funct3;
    if (funct3 == 0) return instTable_.getEntry(InstId::sb);
    if (funct3 == 1) return instTable_.getEntry(InstId::sh);
    if (funct3 == 2) return instTable_.getEntry(InstId::sw);
    if (funct3 == 3 and isRv64())
        return instTable_.getEntry(InstId::sd);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 与 15 想法类似, 查阅 `RV32I` 指令集可得, 当 `opcode` 为 `0100011` 时, 操作码空间对应三条指令 `sb/sh/sw`, 但通过功能码 `funct3` 的不同取值即可将其区分(`funct3` 可由结构体中获得)。需要注意的是, `store` 指令 “`store rs2, offset(rs1)`” 需先映射为 “`store rs2, rs1, offset`” 形式才能得到正确的



操作数取值:  $op0=rs2$ ,  $op1=rs1$ ,  $op2=offset$ ; 若  $funct3$  取值不合规定, 则应当返回  $InstId::illegal$ 。

3) 113:

```
113: // 01101 U-form
// opcode: 0110111, Instruction: LUI
{
    // lui rd, imm
    UFormInst uform(inst);
    op0 = uform.bits.rd;
    op1 = uform.immed(); // bits.imm << 12, 20 位扩充到 32 位
    return instTable_.getEntry(InstId::lui);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 查表可得, 当  $opcode=0110111$  时, 对应的指令为 LUI。由于与 AUIPC 指令同属于 U-format 类型, LUI 与 AUIPC 的指令格式高度相似, 两者的译码思路与过程基本相同, 故不再赘述。

4) 124:

```
124: // 11000 B-form
// opcode: 1100011, Instructions: BEQ/BNE/BLT/BGE/BLTU/BGEU
{
    BFormInst bform(inst);
    op0 = bform.bits.rs1;
    op1 = bform.bits.rs2;
    op2 = bform.immed(); // 将各部分立即数合并为 13 位立即数
    unsigned funct3 = bform.bits.funct3;
    if (funct3 == 0) return instTable_.getEntry(InstId::beq);
    if (funct3 == 1) return instTable_.getEntry(InstId::bne);
    if (funct3 == 4) return instTable_.getEntry(InstId::blt);
    if (funct3 == 5) return instTable_.getEntry(InstId::bge);
    if (funct3 == 6) return instTable_.getEntry(InstId::bltu);
    if (funct3 == 7) return instTable_.getEntry(InstId::bgeu);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 与 18 类似, 虽然操作空间对应了多条指令条目, 但每条指令都拥有独特的  $funct3$  功能码, 可用于判断区分。除此之外, 由于此操作空间内的各指

令写法相同,以 BEQ 指令为例,其格式为“beq rs1,rs2,offset”,故 op0=rs1, op1=rs2, op2 等于合并后的立即数。

5) 125:

```
125: // 11001 I-form
// opcode: 1100111, Instruction: JALR
{
    // JALR rd, rs1, offset
    IFormInst iform(inst);
    op0 = iform.fields.rd;
    op1 = iform.fields.rs1;
    op2 = iform.immed(); // fields.imm
    unsigned funct3 = iform.fields.funct3;
    if (funct3 == 0) return instTable_.getEntry(InstId::jalr);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 查表可知,当 opcode=1100111 时,对应的指令为 JALR。在进行操作数赋值前,需先将指令“JALR, rd, (offset)rs1”映射为“JALR, rd, rs1, offset”,以获得正确的操作数对应关系。与此同时,虽然此操作空间只含有一条指令条目,但由指令集可知, JALR 包含功能码 funct3,译码时仍需对其进行判断,以排除非法输入。

6) 127:

```
127: // 11011 J-form
// opcode: 1101111, Instruction: JAL
{
    // JAL rd, offset
    JFormInst jform(inst);
    op0 = jform.bits.rd;
    op1 = jform.immed(); // 将各部分立即数合并为 21 位立即数
    return instTable_.getEntry(InstId::jal);
}
return instTable_.getEntry(InstId::illegal);
```

实现思路: 与 JALR 指令不同,由于属于 J-format 类型, JAL 不包含功能码,故可以直接返回对应指令条目;除此之外,由于指令中只包含两个操作数 rd 和

offset, 在进行译码赋值时只需对应给出 op0 和 op1 即可。

7) l12:

```
l12: // 01100 R-form
// opcode: 0110011, Instructions: ADD/SUB/SLL/SLT/SLTU
//                               /XOR/SRL/SRA/OR/AND
{
    RFormInst rform(inst);
    op0 = rform.bits.rd;
    op1 = rform.bits.rs1;
    op2 = rform.bits.rs2;
    unsigned funct3 = rform.bits.funct3;
    unsigned funct7 = rform.bits.funct7;
    if (funct7 == 0)
    {
        if (funct3 == 0) return instTable_.getEntry(InstId::add);
        if (funct3 == 1) return instTable_.getEntry(InstId::sll);
        if (funct3 == 2) return instTable_.getEntry(InstId::slt);
        if (funct3 == 3) return instTable_.getEntry(InstId::sltu);
        if (funct3 == 4) return instTable_.getEntry(InstId::xor_);
        if (funct3 == 5) return instTable_.getEntry(InstId::srl);
        if (funct3 == 6) return instTable_.getEntry(InstId::or_);
        if (funct3 == 7) return instTable_.getEntry(InstId::and_);
    }
    .....
}
return instTable_.getEntry(InstId::illegal);
```

实现思路：通过查阅指令集可得，当 opcode=0110011 时，对应的指令条目有 ADD/SUB 等算术、逻辑操作，通过功能码 funct3 和 funct7 可将其区分。在代码缺失部分，funct7=0，此时包含的指令条目有 ADD/SLL/SLT 等（如代码中所示），利用 funct3 进行判断译码即可返回各指令对应的 InstEntry。一般而言，R-format 类型的指令为“inst rd,rs1,rs2”，故 op0=rd,op1=rs1,op2=rs2。

### ③ Hart.cpp: 指令执行

在指令的译码过程中，我们对操作数 op 完成了赋值，从而可以在指令执行时对 op 进行调用，完成寄存器、立即数操作。通过查阅 Hart.hpp、IntRegs.hpp、DecodedInst.hpp 等文件，可以了解到 URV、SRV、pc\_、intRegs\_.read()、

op0()等定义和函数的含义,进而帮助我们补全如下指令的执行过程:

1) AND 指令:

```
template <typename URV>
void
Hart<URV>::execAnd(const DecodedInst* di)
{
    // 无符号寄存器类型, 结果视为非符号数
    URV v = intRegs_.read(di->op1()) & intRegs_.read(di->op2());
    intRegs_.write(di->op0(), v);
}
```

实现思路:由译码过程可知,在非立即数与操作中,op0=rd,op1=rs1,op2=rs2,故可利用 intRegs\_.read()函数从寄存器 regs\_[rs1]和 regs\_[rs2]中取值,进行与运算后,再利用 intRegs\_.write()将结果写回到寄存器 regs\_[rd]中。由于在程序中,寄存器堆 regs\_默认为无符号 URV 类型,即通过 read()返回得到的值为无符号,故按位与后得到的结果也需视为非符号数,采用 URV 类型。

2) ANDI 指令:

```
template <typename URV>
inline
void
Hart<URV>::execAndi(const DecodedInst* di)
{
    // 有符号寄存器类型
    SRV imm = di->op2As<SRV>(); // 第三个操作对象为有符号立即数
    SRV v = intRegs_.read(di->op1()) & imm; // 将结果视为符号数
    intRegs_.write(di->op0(), v);
}
```

实现思路:同理,在 and 操作的基础上,andi 将 rs2 改为了立即数而非寄存器索引,故需先利用 op2As()函数将立即数 imm 读出。需要注意的是,通过此函数读出的立即数为 int32\_t 类型,是有符号数,故需采用 SRV 有符号寄存器。将寄存器 regs\_[rs1]中内容与立即数 imm 按位与后,得到的结果仍为有符号数,可用 write()将其写入到寄存器 regs\_[rd]中。

ps:op2As()与 intRegs\_.read(di->op2())区别:op2As()直接将操作数 op2\_以 32 位有符号数返回;intRegs\_.read(di->op2())是从寄存器 regs\_[op2\_]

中读取内容。

### 3) SLT 指令:

```
template <typename URV>
void
Hart<URV>::execSlt(const DecodedInst* di)
// 比较指令, [rd] = ([rs2] > [rs1] ? 1 : 0)
// [rs2], [rs1]都是符号数
{
    SRV v1 = intRegs_.read(di->op1()); // v1 = [rs1]
    SRV v2 = intRegs_.read(di->op2()); // v2 = [rs2]
    URV v = v2 > v1 ? 1 : 0;
    intRegs_.write(di->op0(), v);
}
```

实现思路: 根据译码过程, 在有符号比较指令中,  $op0=rd$ ,  $op1=rs1$ ,  $op2=rs2$ 。

由于 `slt` 指令将 `regs_[rs2]` 和 `regs_[rs1]` 中的值视为符号数, 取值过程需利用 `intRegs_.read()` 函数实现, 并将 `v1` 和 `v2` 的类型设为 `SRV`。比较得到的结果非 1 即 0, 是非符号数, 故采用 `URV` 寄存器类型。

### 4) SLTU 指令:

```
template <typename URV>
void
Hart<URV>::execSltu(const DecodedInst* di)
// 比较指令, [rd] = ([rs2] > [rs1] ? 1 : 0)
// [rs2], [rs1]都是非符号数
{
    URV v1 = intRegs_.read(di->op1()); // v1 = [rs1]
    URV v2 = intRegs_.read(di->op2()); // v2 = [rs2]
    URV v = v2 > v1 ? 1 : 0;
    intRegs_.write(di->op0(), v);
}
```

实现思路: 与 `slt` 指令类似, `sltu` 将寄存器中内容视为非符号数, 故只需在 `slt` 执行代码的基础上将 `v1` 和 `v2` 的类型修改为 `URV` 即可, 其余过程不变。

### 5) BNE 指令:

```
template <typename URV>
```

```
inline
void
Hart<URV>::execBne(const DecodedInst* di)
{
    URV v1 = intRegs_.read(di->op0()), v2 = intRegs_.read(di->op1());
    if (v1 == v2) // 若[rs1] = [rs2], 则不跳转
        return;
    setPc(currPc_ + di->op2As<SRV>()); // 否则 pc = currPc + offset
    lastBranchTaken_ = true; // 记录跳转
}
```

实现思路: 由译码过程可知, 对 `bne` 指令, 有 `op0=rs1`, `op1=rs2`, `op2=offset`。因为只需要比较两寄存器中值是否一致, `v1` 和 `v2` 采用无符号 `URV` 即可。当 `v1` 等于 `v2` 时, 不满足跳转条件, 故不执行操作; 否则需利用 `setPc()` 函数将下条指令地址调整为当前指令地址(`currPc_`)+地址偏移(`di->op2As<SRV>()`), 以完成跳转。需要注意的是, 由于地址偏移直接以操作数保存, 在读取时必须使用 `op2As()` 函数。

ps: `pc_` 与 `currPc_` 的区别: `pc_` 在指令执行前更新, 保存的是下条指令的地址; `currPc_` 保存的则是当前执行指令的地址。

#### 6) BLTU 指令:

```
template <typename URV>
void
Hart<URV>::execBltu(const DecodedInst* di)
{
    // 将寄存器中的值视为非符号数
    URV v1 = intRegs_.read(di->op0()), v2 = intRegs_.read(di->op1());
    if (v1 < v2)
    {
        setPc(currPc_ + di->op2As<SRV>());
        lastBranchTaken_ = true; // 记录跳转
    }
}
```

实现思路: 与 `bne` 指令类似, `bltu` 的跳转条件为 `regs_[rs1]<regs_[rs2]`, 当此不等式满足时, 即调用 `setPc()` 更改 `pc_` 所指向的下条指令地址, 并将此次跳转记录。需要注意的是, `bltu` 指令是非符号操作, 故从寄存器中取值时需采

用 URV 类型。

7) BGE 指令:

```
template <typename URV>
void
Hart<URV>::execBge(const DecodedInst* di)
{
    // 将寄存器中的值视为符号数
    SRV v1 = intRegs_.read(di->op0()), v2 = intRegs_.read(di->op1());
    if (v1 >= v2)
    {
        setPc(currPc_ + di->op2As<SRV>());
        lastBranchTaken_ = true; // 记录跳转
    }
}
```

实现思路: 与 bltu 指令同理, 当寄存器中内容满足 `regs_[rs1]>=regs_[rs2]` 时, 跳转发生。由于在进行比较时, bge 指令将比较的两数视为符号数, 故 v1 和 v2 的类型需设置为 SRV, 以确保正确跳转。

8) BGEU 指令:

```
template <typename URV>
void
Hart<URV>::execBgeu(const DecodedInst* di)
{
    // 将寄存器中的值视为非符号数
    URV v1 = intRegs_.read(di->op0()), v2 = intRegs_.read(di->op1());
    if (v1 >= v2)
    {
        setPc(currPc_ + di->op2As<SRV>());
        lastBranchTaken_ = true; // 记录跳转
    }
}
```

实现思路: 在 bge 指令的基础上, bgeu 将比较的两数视为非符号数, 故只需在 bge 执行代码的基础上将 v1 和 v2 的类型设置为 URV 即可, 其余过程不变。

9) JAL 指令:

```
template <typename URV>
void
Hart<URV>::execJal(const DecodedInst* di)
{
    // pc_在指令执行前更新
    intRegs_.write(di->op0(), pc_); // [rd]=下条指令地址
    setPc(currPc_ + di->op1As<SRV>()); // 偏移量为符号数
    lastBranchTaken_ = true; // 记录跳转
}
```

实现思路：根据指令功能，当执行“jal rd, offset”指令时，需要将下条指令地址保存在 regs\_[rd] 中，并跳转到 currPc\_+offset 处。由译码过程可知，op0=rd, op1=offset。考虑到 pc\_ 在指令执行前更新，当程序执行到 jal 语句时，pc\_ 中已保存有下条指令的地址，故可直接利用 intRegs\_.write() 函数将其赋值到 regs\_[rd] 中，再执行 setPc() 函数，使 pc\_=currPc\_+offset，完成跳转。由于 offset 直接保存在操作数 op1 中，且为符号数，读取过程可以采用 op1As() 函数来完成。

#### 10) JALR 指令：

```
template <typename URV>
void
Hart<URV>::execJalr(const DecodedInst* di)
{
    // pc_在指令执行前更新
    intRegs_.write(di->op0(), pc_); // [rd]=下条指令地址
    // pc_ = [rs1] + offset
    setPc(intRegs_.read(di->op1()) + di->op2As<SRV>());
    lastBranchTaken_ = true; // 记录跳转
}
```

实现思路：与 JAL 指令类似，在执行指令“jalr rd, (offset)rs1”时，需要将下条指令地址保存在 regs\_[rd] 中，并跳转到 regs\_[rs1]+offset 处，两条指令只在跳转终点存在差异。由译码过程可知，op0=rd, op1=rs1, op2=offset，故只需要在前述 JAL 执行代码的基础上将指令跳转函数 setPc() 的参数修改为 regs\_[rs1]+offset，即 intRegs\_.read(di->op1())+di->op2As<SRV>() 即可。



## 11) AUIPC 指令:

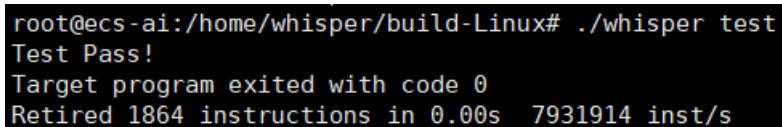
```
template <typename URV>
void
Hart<URV>::execAuipc(const DecodedInst* di)
// add upper immediate to PC, 将立即数中的高 20 位加进 PC
// 再将结果写到寄存器 rd 中
{
    intRegs_.write(di->op0(), currPc_ + di->op1As<SRV>());
}
```

实现思路: 由指令功能, “auipc rd, imm” 将有符号立即数 imm 左移 12 位后加进 pc 当中, 再将结果写进寄存器 regs\_[rd]。在译码过程中, 我们完成了立即数的左移操作, 并有 op0=rd, op1=imm, 因此, 在编写执行代码时, 只需要将操作数 op1\_中的值以 32 位符号数的形式取出, 与当前指令地址相加后写入寄存器 regs\_[rd]。利用 intRegs\_.write()函数可将上述过程一步实现, 注意不要将 pc\_和 currPc\_混淆即可。

## 三、测试结果

## ① 测试程序 test

将修改好的源文件进行编译、安装后, 可得到应用程序 whisper。将 whisper 与 test 移动到同一目录后执行 “./whisper test” 命令, 得到测试结果如图:



```
root@ecs-ai:/home/whisper/build-Linux# ./whisper test
Test Pass!
Target program exited with code 0
Retired 1864 instructions in 0.00s 7931914 inst/s
```

终端打印出 “Test Pass!”, 总共运行指令 1864 条, 说明仿真器正确执行了测试程序中的指令, 仿真器代码设计正确。

## ② 测试程序 mytest

为了对仿真器进行自测试, 使用 C 语言编写一简单的测试文件 mytest.c, 其内容如下:

```
#include<stdio.h>
#include<math.h>

int main(int argc, char* argv[])
```

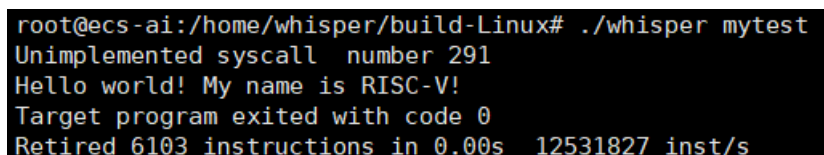
```
{
    int a=1, b=10;
    char c[20]="Hello world! ", d[20]="RISC-V!";
    if (a+2==sqrt(b-1)){
        printf("%s",c);
        printf("My name is %s\n",d);
    }
    else {
        printf("Sorry! Please have another try!");
    }
    return 0;
}
```

显然,若程序正确运行,则会在终端打印出“Hello world! My name is RISC-V!”按照教程安装好 RISC-V 交叉编译器后,使用 Makefile 对测试文件进行编译,其内容如下:

```
TOOL_CHAIN = /opt/riscv/bin
CROSS_COMPILE = $(TOOL_CHAIN)/riscv32-unknown-linux-gnu-gcc

mytest: mytest.c
    $(CROSS_COMPILE) -static -O3 -o $@ $<
clean:
    rm -f mytest
```

在 mytest.c 与 Makefile 目录下执行命令“make mytest”,等待编译完成,即得到测试程序 mytest,将其与应用程序 whisper 移动到同一目录后执行“./whisper mytest”命令,可得测试结果如图:



```
root@ecs-ai:/home/whisper/build-Linux# ./whisper mytest
Unimplemented syscall number 291
Hello world! My name is RISC-V!
Target program exited with code 0
Retired 6103 instructions in 0.00s 12531827 inst/s
```

可见,测试结果与预测正确结果一致,说明仿真器代码正确。

#### 四、心得体会

此次实验,我们利用 C++对 RISC-V 仿真器的部分内容进行了完善,并编写了自测试程序对仿真器功能进行了验证,与预期吻合良好。

总的来说,此次项目并不算特别复杂,但我们对 RISC-V 指令的掌握提出了很高的要求。因为有示例代码的存在,通过不断阅读、类比,我们可以逐渐掌

握各部分代码的写法与用意,发现要点,从而对其进行补充完善。不论是编码部分对结构体内容的对应赋值,还是译码部分操作数 **op** 与指令中寄存器和立即数的对应关系,抑或是执行部分有符号与无符号类型的选取、常用函数的用法及区分,都是对我们的基础知识掌握度和学习能力的一次考验。

当然,在实验的过程当中,我遇到过一些困难,但其中让我印象最为深刻的,还是对自己编写的程序进行测试一步。在初次使用 **whisper** 进行自测试时,程序产生 **Error** 报错,表示存在 64 条连续的非法指令。当尝试在编译指令中添加 “**-march=rv32i**” 指定指令集时,汇编器出现了 **ABI** 无法使用的错误。受此报错“启发”,我查阅了源码的 **readme** 文件,以为是出于自己错误安装了 64 位的工具链导致程序不兼容,于是多次重装了 32 位工具链,但问题仍未得到解决。由于已通过了程序 **test** 测试,我认为错误可能出现在自己编写的测试文件 **mytest.c** 或编译过程中,遂在 **whisper** 中使用 “**run**” 指令查看自测试程序的运行过程,在确认译码结果等正确的前提下,发现指令从第 28 条开始全为 “**illegal**”,修改 **main()** 函数的主要内容,得到的运行代码基本不变,如图:

```

whisper> run
#1 0 000102f0 202d r 01 000102f2 c.jal . + 0x2a
#2 0 0001031a 00002197 r 03 0001231a auipc x3, 0x2
#3 0 0001031e 4e618193 r 03 00012800 addi x3, x3, 0x4e6
#4 0 00010322 8082 r 00 00000000 c.jr x1
#5 0 000102f2 87aa r 0f 00000000 c.mv x15, x10
#6 0 000102f4 00000517 r 0a 000102f4 auipc x10, 0x0
#7 0 000102f8 09650513 r 0a 0001038a addi x10, x10, 0x96
#8 0 000102fc 4582 r 0b 00000001 c.lwsp x11, 0x0
#9 0 000102fe 0050 r 0c ffffff64 c.addi4spn x12, 0x1
#10 0 00010300 ff017113 r 02 ffffff60 andi x2, x2, -0x10
#11 0 00010304 00000697 r 0d 00010304 auipc x13, 0x0
#12 0 00010308 0a268693 r 0d 000103a6 addi x13, x13, 0xa2
#13 0 0001030c 00000717 r 0e 0001030c auipc x14, 0x0
#14 0 00010310 0f270713 r 0e 000103fe addi x14, x14, 0xf2
#15 0 00010314 880a r 10 ffffff60 c.mv x16, x2
#16 0 00010316 37e9 r 01 00010318 c.jal . - 0x36
#17 0 000102e0 00002e17 r 1c 000122e0 auipc x28, 0x2
#18 0 000102e4 d2ce2e03 r 1c 000102b0 lw x28, -0x2d4(x28)
#19 0 000102e8 000e0367 r 06 000102ec jalr x6, 0x0(x28)
#20 0 000102b0 00002397 r 07 000122b0 auipc x7, 0x2
#21 0 000102b4 41c30333 r 06 0000003c sub x6, x6, x28
#22 0 000102b8 d503ae03 r 1c ffffff6f lw x28, -0x2b0(x7)
#23 0 000102bc fd430313 r 06 00000010 addi x6, x6, -0x2c
#24 0 000102c0 d5038293 r 05 00012000 addi x5, x7, -0x2b0
#25 0 000102c4 00235313 r 06 00000004 srli x6, x6, 0x2
#26 0 000102c8 0042a283 r 05 00000000 lw x5, 0x4(x5)
#27 0 000102cc 000e0067 r 00 00000000 jalr x0, 0x0(x28)
#28 0 ffffffff 0000 c 0300 00003800 illegal +
#28 0 ffffffff 0000 c 0341 ffffffff illegal +
#28 0 ffffffff 0000 c 0342 00000002 illegal +
#28 0 ffffffff 0000 c 0343 00000000 illegal +
#29 0 00000000 0000 c 0300 00003800 illegal +
#29 0 00000000 0000 c 0341 00000000 illegal +
#29 0 00000000 0000 c 0342 00000002 illegal +
#29 0 00000000 0000 c 0343 00000000 illegal +
#30 0 00000000 0000 c 0300 00003800 illegal +
#30 0 00000000 0000 c 0341 00000000 illegal +
  
```

机器码

指令地址

对寄存器的修改

译码结果

基于此,我基本确认了问题是由于错误编译导致。在对源文件进行检查后,我发现其第 13 行的分支语句存在语法错误,未用分号结尾。同时,由于最初编译是使用工具链中的 **gcc** 从 **.c** 文件逐步转换,过程中得到的 **.s**、**.o** 文件等可能存在

错误, 将其改为一步实现后, 终端终于能够正确打印输出。虽然整个调试纠错过程花费了较多的时间, 但通过这个问题, 我锻炼了自己的调试能力, 也认识到工程中一个小小的错误可能引起的巨大影响, 更认识到了细心和对知识熟练掌握的重要性, 受益匪浅。

综上, 通过此次实验, 我们既加深了对 **RISC-V** 指令的理解, 也对仿真器的设计思路与流程进行了学习, 虽然花费了较多的时间和精力, 但却有效锻炼了我们的问题解决能力和综合素质, 学会了许多常用工具、系统的基本使用方法。与此同时, 作为学习时不可或缺的一步, 在试错的过程中, 我们学会主动思考、融会贯通, 从发现问题到解决问题, 不断进步, 不失为很好的成长体验。