LABA

PB23111715 王一凡

自测结果

```
100% tests passed, 0 tests failed out of 49

Total Test time (real) =   1.21 sec
```

实现方法

利用 C++ 实现

1. ```
   auto parse_decimal_number(char const* current, char const* end) -> char
   const* {
       if (current == end) {
           return end;
       }

       // Check for optional sign
       if (*current == '+' || *current == '-') {
           ++current;
       }

       // Parse digits
       while (current != end && std::isdigit(*current)) {
           ++current;
       }

       return current;
   }
   ```

   先判断是否为空串，然后判断正负号，之后依次判断传入的字符串是否为数字，最后输出第一个非数字的字符

2. ```
   auto parse_string_literal(char const* current, char const* end) -> char
   const* {
       while (current != end) {
           if (*current == '"') {
               return ++current;
           }
           if (*current == '\n') {
               return current;
           }
           ++current;
       }
       return end;
   }
   ```

   依次判断字符串的各个字符，如果有双引号，那么输出第一个双引号后的字符，如果有换行，那么输出换行符，如果都没有，则输出 `end` 指向的字符

3.
```
auto Parser::parse_instruction() -> Instruction {
    Instruction instr;

    // Check if the current token is of type `Label`. If it is, we add the
label to `instr` and move
    // to the next token.
    if (current_token().kind() == Token::Label) {
        instr.set_label(current_token());
        next_token();
    }

    while (current_token().kind() == Token::EOL)
    {
        next_token();
    }

    // Now `current_token()` points to the opcode. If the first token was a
label, `opcode_token`
    // points to the second token, otherwise it points to the first token.
    Token const& opcode_token = current_token();

    // Check whether `current_token` represents a valid opcode or pseudo-
instruction. If it does
    // not, emit a diagnostic message and return an unknown instruction. You
can obtain an unknown
    // instruction by returning `{}`.
    if (opcode_token.kind() != Token::Opcode && opcode_token.kind() !=
Token::Pseudo) {
        emit_opcode_diag_at_current_token();
        return {};
    }

    // Add the opcode to `instr`.
    instr.set_opcode(opcode_token);

    // Move to the next token to continue parsing.
    next_token();

    // Now we need to parse the operand list. The operand list is a sequence
of tokens separated by
    // `Token::Comma`.
    return parse_operand_list(std::move(instr));
}
```

对于一条指令，先检测它的第一个 `token` 的类型，如果是 `Label`，就进行标记。然后跳过可能存在的换行，检测下一个 `token` 的类型，如果既不是 `Opcode` 也不是 `Pesudo`，那么弹出一个报错信息，否则就设置对应的指令标签。最后设置指令的 `Operand`

4.
```
auto string_to_integer(std::string const& content, bool* ok) -> std::int16_t
{
    try {
        std::size_t pos = 0;
        int base = 10;
        if (content[0] == '#') {
            pos = 1;
```

```
        } else if (content[0] == 'x') {
            base = 16;
            pos = 1;
        } else if (content[0] == 'b') {
            base = 2;
            pos = 1;
        }

        long long value = std::stoll(content.substr(pos), nullptr, base);

        if (value > std::numeric_limits<std::uint16_t>::max()
            || value < std::numeric_limits<std::int16_t>::min()) {
            *ok = false;
            return 0;
        }

        *ok = true;
        return static_cast<std::int16_t>(value);
    } catch (...) {
        *ok = false;
        return 0;
    }
}
```

判断立即数前的标识符，确定立即数的进制，然后利用 `std::stoll` 函数将字符串转换为数字，之后进行数字范围检查，最后将数字转换为 `int16_t` 的类型输出

5.
```
auto Instruction::immediate_range() const -> std::pair<std::int16_t,
std::int16_t> {
    // clang-format off
    switch (opcode_) {
    case TRAP:
        // trapvect8
        return { static_cast<std::int16_t>(0), static_cast<std::int16_t>(255)
};

    case ORIG: case FILL: case BLKW:
        // 16-bit integer
        return {
            std::numeric_limits<std::int16_t>::min(),
            std::numeric_limits<std::int16_t>::max(),
        };

    case ADD: case AND:
        // 5-bit signed integer
        return { static_cast<std::int16_t>(-16), static_cast<std::int16_t>
(15) };

    case LD: case LDI: case LEA: case ST: case STI:
    case BR: case BRn: case BRz: case BRp: case BRzp: case BRnp: case BRnz:
case BRnzp:
        // 9-bit signed integer
        return { static_cast<std::int16_t>(-256), static_cast<std::int16_t>
(255) };

    case LDR: case STR:
```

```
        // 6-bit signed integer
        return { static_cast<std::int16_t>(-32), static_cast<std::int16_t>
(31) };

    case JSR:
        // 11-bit signed integer
        return { static_cast<std::int16_t>(-1024), static_cast<std::int16_t>
(1023) };

    default:
        return {};
    }
    // clang-format on
}
```

根据类型给出对应的操作数范围

6.
```
void Assembler::assign_addresses() {
    std::uint16_t address = get_instructions().front().get_opcode() ==
Instruction::ORIG
        ? get_instructions().front().get_operand(0).immediate_value()
        : 0;
    for (Instruction& instr : get_instructions()) {
        instr.set_address(address);

        switch (instr.get_opcode()) {
        case Instruction::BLKW:
            address += instr.get_operand(0).regular_decimal();
            break;
        case Instruction::STRINGZ:
            address += instr.get_operand(0).string_literal().size() + 1;
            break;
        default:
            address += 1;
            break;
        }
    }
}
```

先检测第一条指令的类型，如果是 `ORIG`，那么将起始地址设为对应数值。然后依次遍历各条指令，如果是 `BLKW` 指令，那么将下一条指令的地址增加对应数值，如果是 `STRINGZ` 指令，那么下一条指令的地址增加对应字符串长度再加 1，其余指令均是下一条指令的地址增加 1

7.
```
auto Assembler::scan_label() -> bool {
    for (Instruction const& instr : get_instructions()) {
        if (instr.has_label()) {
            if (!add_label(instr.get_label(), instr.get_address())) {
                emit_label_redefinition_diag(instr);
                return false;
            }
        }
    }
    return true;
}
```

将 `Label` 转换成对应的地址值，如果发现有有多个标签则抛出错误信息

8.
```
auto Assembler::translate_opcode(Instruction::Opcode opcode) -> std::uint16_t
{
    // clang-format off
    switch (opcode) {
    case Instruction::ADD:
        return 1;  // 0001
    case Instruction::AND:
        return 5;  // 0101
    case Instruction::BRn:  case Instruction::BRz:  case Instruction::BRp:
 case Instruction::BR:
    case Instruction::BRzp: case Instruction::BRnp: case Instruction::BRnz:
case Instruction::BRnzp:
        return 0;  // 0000
    case Instruction::JMP:
        return 12; // 1100
    case Instruction::JSR:
        return 4;  // 0100
    case Instruction::JSRR:
        return 4;  // 0100
    case Instruction::LD:
        return 2;  // 0010
    case Instruction::LDI:
        return 10; // 1010
    case Instruction::LDR:
        return 6;  // 0110
    case Instruction::LEA:
        return 14; // 1110
    case Instruction::NOT:
        return 9;  // 1001
    case Instruction::RET:
        return 12; // 1100
    case Instruction::RTI:
        return 8;  // 1000
    case Instruction::ST:
        return 3;  // 0011
    case Instruction::STI:
        return 11; // 1011
    case Instruction::STR:
        return 7;  // 0111
    case Instruction::TRAP: case Instruction::GETC:  case Instruction::OUT:
case Instruction::PUTS:
    case Instruction::IN:   case Instruction::PUTSP: case Instruction::HALT:
        return 15;  // 1111
    default:
        return 13;  // 1101
    }
    // clang-format on
}
```

操作码转换

9.
```
auto Assembler::translate_register(Operand const& reg_operand, unsigned
position) -> std::uint16_t {
    std::uint16_t reg_id = reg_operand.register_id();
    return reg_id << position;
}
```

寄存器操作数转换

10.
```
auto Assembler::translate_immediate(Operand const& imm_operand, unsigned
bits) -> std::uint16_t {
    std::int16_t value = imm_operand.immediate_value();
    std::uint16_t mask = (1 << bits) - 1;
    return static_cast<std::uint16_t>(value) & mask;
}
```

立即数转换

11.
```
auto Assembler::translate_label(  //
    Instruction const& instr,
    std::size_t operand_idx,
    unsigned bits
) const -> std::uint16_t {
    auto const& label_operand = instr.get_operand(operand_idx);
    auto const& label = label_operand.label();
    auto const instr_address = instr.get_address();

    auto const iter = symbol_table_.find(label);
    if (iter == symbol_table_.end()) {
        emit_label_not_found_diag(label_operand, instr);
        return static_cast<std::uint16_t>(-1);
    }

    auto const label_address = iter->second;
    auto const offset = static_cast<std::int16_t>(label_address -
instr_address - 1);

    auto const max_offset = (1 << (bits - 1)) - 1;
    auto const min_offset = -(1 << (bits - 1));

    if (offset < min_offset || offset > max_offset) {
        emit_label_offset_out_of_range_diag(label_operand, instr, offset);
        return static_cast<std::uint16_t>(-1);
    }

    return static_cast<std::uint16_t>(offset & ((1 << bits) - 1));
}
```

标签转换

12.
```
auto Assembler::translate_regular_instruction(Instruction const& instr) const
-> std::uint16_t {
    std::uint16_t result = translate_opcode(instr.get_opcode()) << 12;

    switch (instr.get_opcode()) {
    case Instruction::ADD:
```

```
    case Instruction::AND:
        result |= translate_register(instr.get_operand(0), 9);
        result |= translate_register(instr.get_operand(1), 6);
        if (instr.get_operand(2).type() == Operand::Immediate) {
            result |= 1u << 5;
            result |= translate_immediate(instr.get_operand(2), 5);
        } else {
            result |= translate_register(instr.get_operand(2), 0);
        }
        break;

    case Instruction::BR:
        result |= 0x7 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRn:
        result |= 0x4 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRz:
        result |= 0x2 << 9;
        result |= (instr.get_operand(0).type() == Operand::Label)
            ? translate_label(instr, 0, 9)
            : translate_immediate(instr.get_operand(0), 9);
        break;
    case Instruction::BRp:
        result |= 0x1 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRzp:
        result |= 0x3 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRnp:
        result |= 0x5 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRnz:
        result |= 0x6 << 9;
        result |= translate_label(instr, 0, 9);
        break;
    case Instruction::BRnzp:
        result |= 0x7 << 9;
        result |= translate_label(instr, 0, 9);
        break;

    case Instruction::JMP:
    case Instruction::JSRR:
        result |= translate_register(instr.get_operand(0), 6);
        break;

    case Instruction::JSR:
        result |= 1u << 11;
        result |= translate_label(instr, 0, 11);
        break;
```

```
    case Instruction::LD:
    case Instruction::LDI:
    case Instruction::LEA:
        result |= translate_register(instr.get_operand(0), 9);
        result |= translate_label(instr, 1, 9);
        break;

    case Instruction::LDR:
    case Instruction::STR:
        result |= translate_register(instr.get_operand(0), 9);
        result |= translate_register(instr.get_operand(1), 6);
        result |= translate_immediate(instr.get_operand(2), 6);
        break;

    case Instruction::NOT:
        result |= translate_register(instr.get_operand(0), 9);
        result |= translate_register(instr.get_operand(1), 6);
        result |= 0x3F;
        break;

    case Instruction::RET:
        result |= 0x7 << 6;
        break;

    case Instruction::RTI:
        break;

    case Instruction::ST:
    case Instruction::STI:
        result |= translate_register(instr.get_operand(0), 9);
        result |= translate_label(instr, 1, 9);
        break;

    case Instruction::TRAP:
        result |= translate_immediate(instr.get_operand(0), 8);
        break;
    case Instruction::GETC:
        result |= 0x20;
        break;
    case Instruction::OUT:
        result |= 0x21;
        break;
    case Instruction::PUTS:
        result |= 0x22;
        break;
    case Instruction::IN:
        result |= 0x23;
        break;
    case Instruction::PUTSP:
        result |= 0x24;
        break;
    case Instruction::HALT:
        result |= 0x25;
        break;

    default:
```

```
        break;
    }

    return result;
}
```

将指令翻译为对应的二进制机器码

13.
```
void Assembler::translate_pseudo(Instruction const& instr,
std::vector<std::uint16_t>& results) {
    switch (instr.get_opcode()) {
    case Instruction::FILL:
        // `.FILL` will fill the memory location with the value of the
operand.
        results.push_back(static_cast<std::uint16_t>
(instr.get_operand(0).immediate_value()));
        break;

    case Instruction::ORIG:
        // `.ORIG` sets the starting address, no binary output needed.
        break;

    case Instruction::BLKW:
        // `.BLKW` reserves a block of memory.
        results.resize(results.size() +
instr.get_operand(0).regular_decimal(), 0);
        break;

    case Instruction::STRINGZ:
        // `.STRINGZ` stores a null-terminated string.
        for (char c : instr.get_operand(0).string_literal()) {
            results.push_back(static_cast<std::uint16_t>(c));
        }
        results.push_back(0);  // Null terminator
        break;

    case Instruction::END:
        // `.END` does not produce any binary output.
        break;

    default:
        break;
    }
}
```

翻译伪指令