Лабораторная работа №3	M3136	2022
ISA	Исаев Максим Викторович	

1. Цель работы.

знакомство с архитектурой набора команд RISC-V.

2. Инструментарий и требования к работе.

Для написания программы использовался язык C++.

Компилятор	g++ 12.2.0 MinGW-W64 x86_64-msvcrt-posix-seh
Система сборки	CMake 3.24.3 Generator: MinGW Makefiles

3. Описание системы кодирования команд RISC-V.

Расширения.

В RISC-V есть 32, 64 и 128 битные варианты. В нашем задании рассматривается только 32 битный.

Базовой набор RV32I содержит 32 битные целочисленные операции. Помимо служебных инструкций в него входят минимальное количество арифметических и битовых операций, операции с памятью и переходы (условные и безусловные). Этот набор инструкций поддерживается по умолчанию в том числе и в рамках этого задания.

В расширении RV32-M добавляются инструкции для целочисленного умножения и деления. Они также поддерживаются написанным дизассемблером.

Расширения F, D и Q добавляют операции с числами с плавающей точкой одинарной, двойной и четверной точности соответственно.

Так же существует расширение E всего с 16 регистрами.

Ещё есть интересные расширения Zfinx, Zdinx, Zhinx и Zhinxmin позволяющие производить вычисления с плавающей точкой в целочисленных регистрах.

Регистры.

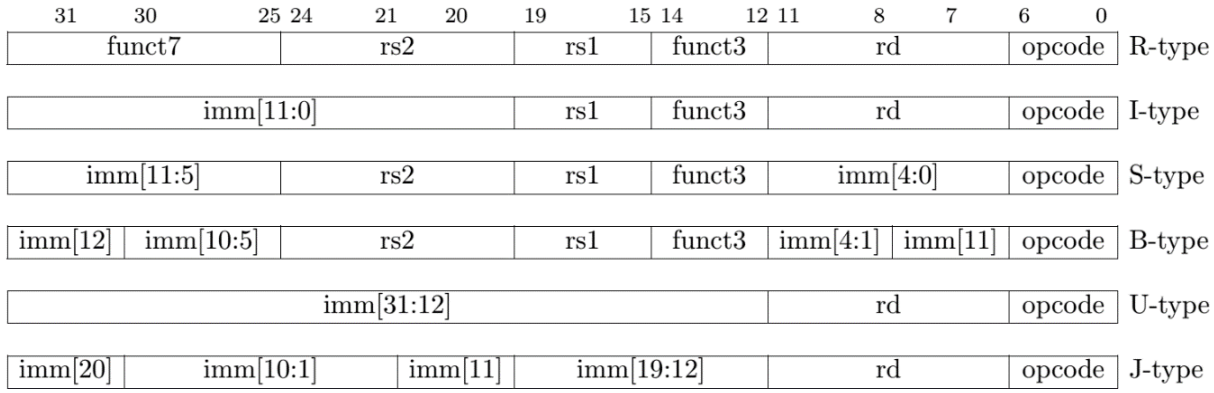
В базовом наборе используются 32 регистра, x0 – нулевой регистр, и 31 целочисленный регистр. Из регистра x0 всегда считывается значение 0,

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

чтобы в него не писали. Благодаря этому можно сократить количество инструкций, заменив их на инструкции с участием x0. Так же у регистров есть свои имена и использование по умолчанию.

Инструкции.

В RISC-V все инструкции имеют фиксированный размер 4 байта. Для их кодирования используется только модель little-endian.



Инструкции разбиты на 6 типов, представленных на схеме выше.

Отмечу, что все offset’ы являются относительными, т. е. показывают расстояние, на которое нужно сместиться относительно адреса текущей инструкции. Благодаря этому можно уменьшить размер инструкций, т. к. не приходится хранить адреса целиком. Так же все offset’ы являются знаковыми и хранятся в дополнении до двух.

R-type это инструкции, которые берут на вход два регистра (rs1 и rs2) и результат тоже записывают в регистр (rd). В основном это такие действия как арифметические операции (сложение, умножение) и различные побитовые операции. В нашем наборе инструкций все инструкции R типа имеют одинаковый opcode. Для идентификации используются funct3 и funct7. А вот add, xor и slt можно отличить по funtc3. Но add, sub и mul, к примеру, имеют идентичные opcode и funtc3, их можно различить только по funct7.

I-type это инструкции принимающие на вход регистр и константу и записывающие результат в регистр. Внутри их можно разделить на те, которые работают с константой как с числом (например сложение), и на те, которые интерпретируют его как сдвиг относительно адреса инструкции (например lb – load byte). Для идентификации используется opcode и funct3. Так же среди них 3 необычные инструкции – побитовые сдвиги (slli, srli и srai). Они в immediate хранят величину сдвига shamt (shift amount) и подобие funct7 из R типа, позволяющего отличить srli и srai.

S-type это инструкции записывающие данные из регистров в память (store). Они интерпретируют immediate как offset. В rs2 хранятся данные, которые нужно записать. В rs1 хранится адрес, от которого откладывается offset.

B-type это branch инструкции, условные переходы. Каждая инструкция содержит условие, которое должно выполниться для rs1 и rs2, для осуществления перехода. Immediate это offset относительно адреса инструкции.

U-type это инструкции работающие с 20 битным immediate как со, старшими битами 32 битного значения. LUI (load upper immediate) позволяет загрузить это значение в регистр, AUIPC (add upper immediate to program counter) увеличить program counter.

J-type это инструкции (инструкция) позволяющие сделать безусловный переход (jump). Immediate представляет из себя 20 битный offset относительно адреса инструкции.

Магия перемешивания битов.

Странное на первый взгляд перемешивание битов в immediate в B и J типах, на самом деле происходит по вполне понятным мотивам. Если внимательно посмотреть на позиции битов 1–4 в immediate типов S и B, то видно, что они находятся на одних и тех же местах в инструкции, с 8 по 11. То же самое можно увидеть для битов 12–19 в U и J, битов 5–10 в S и B, 5–11 в I и S. Это всё позволяет уменьшить количество логики для разбора команд и упростить принятие решений при разборе.

4. Описание структуры файла ELF.

ELF файл начинается с ELF заголовка. Он содержит информацию для идентификации, информацию о версии, разрядности. Так же он содержит указатели на другие части файла, такие как массивы Program Header и Section Header, индекс таблицы строк Section Header. Помимо этого, он содержит пустые байты, которые служат для выравнивания последующих, ведь в elf формате есть соглашение, что все поля размера n должны начинаться на адресах кратных n.

Массив Program Header’ов состоит из заголовков, каждый из которых содержит информацию для операционной системы, необходимую для запуска программы. Там указано количество памяти, виртуальный адрес внутри этой памяти, куда нужно загрузить, ту или иную часть файла.

После этого идут сами данные, однако информация разъясняющая, где что находится содержится в массиве Section Header’ов по адресу, на который указывает указатель из Elf Header’a. Хоть данное расположение и кажется неудобным при разборе и дизассемблировании файла, но это сделано так, потому что при исполнении файла нам не важна вся это информация и при таком расположении файл можно читать последовательно, ничего не пропуская.

Массив Section Header’ов содержит описания секции, которые позволяют определить по какому адресу лежит, как называется и что содержит та или иная секция. Название хранится в виде индекса начала строки в shStrTab. Первый элемент всегда является нулевым, он играет роль контрольного значения (Sentinel value), а также в случае, если количество секций больше или равно SHN_LORESERVE (0xff00), то их количество хранится в этом первом элементе (см. источники про SHT_NULL).

Все названия, как секций, так и меток (например функций) хранятся в таблицах строк. Названия секций хранятся в shStrTab – таблице строк

Section Header’ов, а имена меток в strTab – просто таблице строк. Для адресации по ним нужные элементы хранят ссылку на начало нужной строки. Строки представляют из себя c_style null terminated string, т. е. хранится не длина строки, а в конец каждой строки дописывается символ ‘\0’, которой означает конец строки. Так же первый индекс этих таблиц является этим нулевым символом (терминатором), что позволяет в случае отсутствия имени у какой-либо структуры ссылаться на 0 индекс, что вернёт пустую строку. Так, например делает SHT_NULL.

В секции .text хранятся непосредственно сами инструкции.

Так же есть таблица символов .symtab, в которой хранятся записи о метках. Каждая запись содержит тип метки, её видимость, индекс, виртуальный адрес, а также ссылку на название в таблице строк.

5. Описание работы написанного кода.

Программе в качестве аргументов командной строки передаются пути входного и выходного файлов. Для удобства я взял оригинальный typedef (см. источники) и продублировал все структуры файла были в виде классов. Это очень упростило разбор файла, т. к. расположение в памяти стало 1 к 1. Так же это позволяет локализовать логику валидации считываемых данных. Для разбора входного файла класс ElfParser создаёт необходимые структуры, а те заполняют свои поля из входного потока. Благодаря этому легче контролировать в какой части файла сейчас находишься т. к. не нужно высчитывать сдвиги, по сколько мы просто последовательно читаем из файла нужно количество байт.

Для решения проблемы с тем, что Section Header’ы находятся в конце файла, всё что находится между считывается в буфер, из которого мы потом считываем информацию для заполнения остальных структур.

Т.к порядку разбора и формат вывода разных типов инструкций значительно отличаются, то тут имеет место иерархия разных типов объектов, имеющих общего родителя, инкапсулирующих логику разбора и отображения. Благодаря свойству кодирования RISC-V, что по opcode можно однозначно определить тип инструкции, можно реализовать определение типа при помощи ассоциативного контейнера. Нужный тип объекта создаётся фабричным методом.

6. Результат работы написанной программы.

```
.text
00010074   <main>:
   10074:    ff010113     addi   sp, sp, -16
   10078:    00112623     sw     ra, 12(sp)
   1007c:    030000ef     jal    ra, 000100ac <mmul>
   10080:    00c12083     lw     ra, 12(sp)
   10084:    00000513     addi   a0, zero, 0
   10088:    01010113     addi   sp, sp, 16
   1008c:    00008067     jalr   zero, 0(ra)
   10090:    00000013     addi   zero, zero, 0
   10094:    00100137     lui    sp, 0x00100
   10098:    fddff0ef     jal    ra, 00010074 <main>
   1009c:    00050593     addi   a1, a0, 0
   100a0:    00a00893     addi   a7, zero, 10
   100a4:    0ff0000f     unknown_instruction
   100a8:    00000073     ecall
000100ac   <mmul>:
   100ac:    00011f37     lui    t5, 0x00011
   100b0:    124f0513     addi   a0, t5, 292
   100b4:    65450513     addi   a0, a0, 1620
   100b8:    124f0f13     addi   t5, t5, 292
   100bc:    e4018293     addi   t0, gp, -448
   100c0:    fd018f93     addi   t6, gp, -48
   100c4:    02800e93     addi   t4, zero, 40
000100c8   <L2>:
   100c8:    fec50e13     addi   t3, a0, -20
   100cc:    000f0313     addi   t1, t5, 0
   100d0:    000f8893     addi   a7, t6, 0
   100d4:    00000813     addi   a6, zero, 0
000100d8   <L1>:
   100d8:    00088693     addi   a3, a7, 0
   100dc:    000e0793     addi   a5, t3, 0
```

```
   100e0:    00000613     addi   a2, zero, 0

000100e4   <L0>:
   100e4:    00078703     lb     a4, 0(a5)
   100e8:    00069583     lh     a1, 0(a3)
   100ec:    00178793     addi   a5, a5, 1
   100f0:    02868693     addi   a3, a3, 40
   100f4:    02b70733     mul    a4, a4, a1
   100f8:    00e60633     add    a2, a2, a4
   100fc:    fea794e3     bne    a5, a0, 000100e4 <L0>
   10100:    00c32023     sw     a2, 0(t1)
   10104:    00280813     addi   a6, a6, 2
   10108:    00430313     addi   t1, t1, 4
   1010c:    00288893     addi   a7, a7, 2
   10110:    fdd814e3     bne    a6, t4, 000100d8 <L1>
   10114:    050f0f13     addi   t5, t5, 80
   10118:    01478513     addi   a0, a5, 20
   1011c:    fa5f16e3     bne    t5, t0, 000100c8 <L2>
   10120:    00008067     jalr   zero, 0(ra)


.symtab
Symbol Value              Size Type    Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE  LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION LOCAL    DEFAULT       1 .text
[   2] 0x11124               0 SECTION LOCAL    DEFAULT       2 .bss
[   3] 0x0                   0 SECTION LOCAL    DEFAULT       3 .comment
[   4] 0x0                   0 SECTION LOCAL    DEFAULT       4
.riscv.attributes
[   5] 0x0                   0 FILE    LOCAL    DEFAULT     ABS test.c
[   6] 0x11924               0 NOTYPE  GLOBAL   DEFAULT     ABS
__global_pointer$
[   7] 0x118F4             800 OBJECT  GLOBAL   DEFAULT       2 b
[   8] 0x11124               0 NOTYPE  GLOBAL   DEFAULT       1
__SDATA_BEGIN__
[   9] 0x100AC             120 FUNC    GLOBAL   DEFAULT       1 mmul
[  10] 0x0                   0 NOTYPE  GLOBAL   DEFAULT   UNDEF _start
```

```
      [  11] 0x11124            1600 OBJECT   GLOBAL   DEFAULT       2 c
      [  12] 0x11C14               0 NOTYPE   GLOBAL   DEFAULT       2
__BSS_END__
      [  13] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       2
__bss_start
      [  14] 0x10074              28 FUNC     GLOBAL   DEFAULT       1 main
      [  15] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1
__DATA_BEGIN__
      [  16] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1 _edata
      [  17] 0x11C14               0 NOTYPE   GLOBAL   DEFAULT       2 _end
      [  18] 0x11764             400 OBJECT   GLOBAL   DEFAULT       2 a
```

7. Список источников.

Структура ELF файла.

Про магию перемешивания битов.

Полный список кодов для Elf Header.

Typedef типов используемых в стуктурах.

Про SHT_NULL.

8. Листинг кода.

ElfParser.cpp

```
#include "ElfParser.h"

ElfParser::ElfParser(std::ifstream& f) : file(f) {}

void ElfParser::parse() {

    // ELF HEADER
```

```
elfHeader.fill(file);
// PROGREMM HEADERS
programHeaders = new ProgrammHeader[elfHeader.phnum];
for (int i = 0; i < elfHeader.phnum; i++) {
    programHeaders[i].fill(file);
}
// SECTION HEADERS
bufferOffset = elfHeader.phoff + elfHeader.phnum * elfHeader.phentsize;
const int bufferSize = elfHeader.shoff - bufferOffset;
const auto buff = new char[bufferSize];
for (int i = 0; i < bufferSize; i++) {
    file.read(&buff[i], sizeof(char));
}
sectionHeaders = new SectionHeader[elfHeader.shnum];
for (int i = 0; i < elfHeader.shnum; i++) {
    sectionHeaders[i].fill(file);
    if (sectionHeaders[i].type == STR_TAB) {
        if (i == elfHeader.shstrndx) {
            shStrTabAddress = sectionHeaders[i].offset;
            shStrTabSize = sectionHeaders[i].size;
        } else {
            strTabAddress = sectionHeaders[i].offset;
            strTabSize = sectionHeaders[i].size;
        }
    } else if (sectionHeaders[i].type == SYM_TAB) {
        symTabAddress = sectionHeaders[i].offset;
        symTabEntrySize = sectionHeaders[i].entsize;
```

```
                symTabEntriesCount = sectionHeaders[i].size / symTabEntrySize;
// by default ent size is 0x10

            }

        }



        fillStrTab(buff);

        fillShStrTab(buff);

        // SYMBOL TABLE

        symTableEntries = new SymTabEntry[symTabEntriesCount];

        std::stringstream bufferStream;

        bufferStream.write(buff + symTabAddress - bufferOffset, bufferSize -
(symTabAddress - bufferOffset));

        bufferStream.seekg(0);

        for (int i = 0; i < symTabEntriesCount; i++) {

            symTableEntries[i].fill(bufferStream);

            if (symTableEntries[i].info % 0b00010000 == STT::FUNC) {

                labels[symTableEntries[i].value]                                =
getStringFromStrTab(symTableEntries[i].name);

            }

        }

        // INSTRUCTIONS

        for (int i = 1; i < elfHeader.shnum; i++) {

            if (getStringFromShStrTab(sectionHeaders[i].name) == ".text") {

                textAddress = sectionHeaders[i].offset;

                textVirtualAddress = sectionHeaders[i].addr;

                textSize = sectionHeaders[i].size;

                SectionHeader::validateTextSize(textSize);

                break;        }

        }
```

```
            int labelsCounter = 0;

            for (uint32_t curAddress = 0; curAddress < textSize; curAddress += 4) {

                Instruction*                        newInstr                        =
InstructionFabric::createInstruction(*reinterpret_cast<uint32_t*>(&buff[curAddress
]));

                newInstr->setAddress(textVirtualAddress + curAddress);

                if (newInstr->needLabel()) {

                    uint32_t address = newInstr->getImmAddr();

                    if (labels.count(address) <= 0) {

                        labels[address] = "L" + std::to_string(labelsCounter++);

                    }

                }

                instructions.push_back(newInstr);

            }

            for (auto& inst : instructions) {

                if (inst->needLabel()) {

                    uint32_t address = inst->getImmAddr();

                    inst->setLabel(labels[address]);

                }

            }

            delete[] buff;

        }

        void ElfParser::printDotText(std::ostream& out) {

            out << ".text\n";

            int curAddress = textVirtualAddress;

            for (int i = 0; i < instructions.size(); i++, curAddress += 4) {

                if (labels.count(curAddress) > 0) {

                    constexpr int buffSize = 128;
```

```
                char buff[buffSize];

                snprintf(buff,  buffSize,  "%08x       <%s>:\n",  curAddress,
labels[curAddress].c_str());

                out << buff;

            }

            instructions.at(i)->toString(out);

        }



    void ElfParser::printSymtab(std::ostream& out) const {

        out << ".symtab\n"

            << "Symbol Value                  Size Type   Bind   Vis
Index Name\n";

        for (int i = 0; i < symTabEntriesCount; i++) {

            SymTabEntry curEntry = symTableEntries[i];



            std::string name;

            if (curEntry.info % 0b00010000 == STT::SECTION) {

                name                                                          =
getStringFromShStrTab(sectionHeaders[curEntry.shndx].name);

            } else {

                name = getStringFromStrTab(curEntry.name);

            }



            constexpr int buffSize = 128;

            char buff[buffSize];

            snprintf(buff, buffSize, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s",
i, curEntry.value, curEntry.size,

                     toStringSTT(static_cast<STT>(curEntry.info               %
0b10000)).c_str(),

                     toStringSTB(static_cast<STB>(curEntry.info >> 4)).c_str(),
```

```
                    toStringSTV(static_cast<STV>(curEntry.other)).c_str(),
                    toStringSHN(static_cast<SHN>(curEntry.shndx)).c_str(),
                    name.c_str());
            out << buff << "\n";
        }
    }
    void ElfParser::fillStrTab(const char* buff) {
        strTab = new char[strTabSize];
        for (int j = 0; j < strTabSize; j++) {
            strTab[j] = buff[strTabAddress + j - bufferOffset];
        }
    }

    void ElfParser::fillShStrTab(const char* buff) {
        shStrTab = new char[shStrTabSize];
        for (int j = 0; j < shStrTabSize; j++) {
            shStrTab[j] = buff[shStrTabAddress + j - bufferOffset];
        }
    }

    std::string ElfParser::getStringFromStrTab(const uint32_t offset) const {
        if (offset > strTabSize) {
            throw std::runtime_error("strTab index '" + std::to_string(offset) +
"' is out of bound for size '" + std::to_string(strTabSize) + "'");
        }

        std::stringstream ss;
        int charsRead = 0;
```

```
            while (strTab[offset + charsRead] != '\0') {

                ss << strTab[offset + charsRead];

                ++charsRead;

            }



            return ss.str();

        }



        std::string ElfParser::getStringFromShStrTab(const uint32_t offset) const {

            if (offset > shStrTabSize) {

                throw std::runtime_error("shStrTab index '" + std::to_string(offset)
+ "' is out of bound for size '" + std::to_string(shStrTabSize) + "'");

            }



            std::stringstream ss;

            int charsRead = 0;

            while (shStrTab[offset + charsRead] != '\0') {

                ss << shStrTab[offset + charsRead];

                ++charsRead;

            }



            return ss.str();

        }



        ElfParser::~ElfParser() {

            for (const auto& instruction : instructions) {

                delete instruction;

            }
```

```
    delete[] programHeaders;
    delete[] sectionHeaders;
    delete[] symTableEntries;
    delete[] strTab;
    delete[] shStrTab;
}
```

ElfParser.h

```
#pragma once
#include <ElfHeader.h>
#include <InstructionFabric.h>
#include <ProgrammHeader.h>
#include <SectionHeader.h>
#include <SymTabEntry.h>
#include <SymTabInfoEnum.h>
#include <fstream>
#include <sstream>
#include <vector>
class ElfParser {
    static constexpr uint8_t SYM_TAB = 2;
    static constexpr uint8_t STR_TAB = 3;
   public:
    explicit ElfParser(std::ifstream& f);
    ~ElfParser();
    void parse();
    void printDotText(std::ostream& out);
    void printSymtab(std::ostream& out) const;
```

```
private:
 std::ifstream& file;
 ElfHeader elfHeader;
 ProgrammHeader* programHeaders;
 int bufferOffset;  // offset of address in buff relative to file
 SectionHeader* sectionHeaders;
 // SYM_TAB
 uint32_t symTabAddress;
 uint32_t symTabEntrySize;
 uint32_t symTabEntriesCount;
 SymTabEntry* symTableEntries;
 // STR_TAB
 uint32_t strTabAddress;
 uint32_t strTabSize;
 void fillStrTab(const char* buff);
 char* strTab;
 // SH_STR_TAB
 uint32_t shStrTabAddress;
 uint32_t shStrTabSize;
 void fillShStrTab(const char* buff);
 char* shStrTab;

 std::string getStringFromStrTab(uint32_t offset) const;
 std::string getStringFromShStrTab(uint32_t offset) const;
 // .text
 uint32_t textAddress;
 uint32_t textVirtualAddress;
```

```
    uint32_t textSize;

    std::vector<Instruction*> instructions;

    std::unordered_map<uint32_t, std::string> labels;

    // std::unordered_map<uint32_t, int> addressLabels;

};
```

Instruction.cpp

```
#include <Instruction.h>

Instruction::Instruction(const uint32_t bits) : bits(bits){};

std::string Instruction::addressString() const {

    return toHexString(address);

}

void Instruction::toString(std::ostream& out) const {

    std::string instrStr = instructionString();

    if (label.size() > 0) {

        instrStr += " <" + label + ">";

    }

    constexpr int buffSize = 128;

    char buff[buffSize];

    snprintf(buff, buffSize, "      %05x:\t%08x\t%7s\n", address, bits,
instrStr.c_str());

    out << buff;

}

Instruction::~Instruction() = default;

void Instruction::setAddress(const uint32_t givenAddress) {

    address = givenAddress;
```

```
    }

    void Instruction::setLabel(const std::string givenLabel) {

        label = givenLabel;

    }

    uint8_t Instruction::parseOpcodeBits(uint32_t bits) {

        uint16_t opcode = 0;

        for (size_t i = 0; i < 7; i++) {

            opcode += bits & (1 << i);

        }

        return opcode;

    }

    uint8_t Instruction::parseFunct3(const uint32_t bits) {

        uint8_t funct3 = 0;

        for (size_t i = 0; i < 3; i++) {

            funct3 += isBitSet(bits, i + 12) > 0 ? (1 << i) : 0;

        }

        return funct3;

    }

    uint8_t Instruction::parseFunct7(const uint32_t bits) {

        uint8_t funct7 = 0;

        for (int i = 0; i < 7; i++) {

            funct7 += isBitSet(bits, i + 25) > 0 ? (1 << i) : 0;

        }

        return funct7;

    }

    uint8_t Instruction::parseRegIndex(const uint32_t bits, const int
startAddress) {

        uint8_t index = 0;
```

```
    for (int i = 0; i < 5; i++) {
        index += isBitSet(bits, i + startAddress) > 0 ? (1 << i) : 0;
    }
    return index;
}
std::string Instruction::parseRd(const uint32_t bits) {
    return Storage::getRegisterName(parseRegIndex(bits, 7));
}
std::string Instruction::parseRs1(const uint32_t bits) {
    return Storage::getRegisterName(parseRegIndex(bits, 15));
}

std::string Instruction::parseRs2(const uint32_t bits) {
    return Storage::getRegisterName(parseRegIndex(bits, 20));
}
bool Instruction::isBitSet(const uint32_t bits, const int index) {
    return (bits & (1 << index)) > 0;
}
```

Instruction.h

```
#pragma once
#include <Storage.h>
#include <fstream>
#include <iomanip>
class Instruction {
   protected:
    uint32_t bits;
```

```
    uint32_t address{};
    std::string label{};
    explicit Instruction(uint32_t bits);
    std::string addressString() const;
    virtual std::string instructionString() const = 0;
  public:
    virtual void toString(std::ostream& out) const;
    virtual ~Instruction();
    void setAddress(uint32_t givenAddress);
    void setLabel(std::string givenLabel);
    virtual bool needLabel() const {
        return false;
    }
    virtual uint32_t getImmAddr() const {
        return 0;
    }
    static uint8_t parseOpcodeBits(uint32_t bits);
    static uint8_t parseFunct3(uint32_t bits);
    static uint8_t parseFunct7(uint32_t bits);
    static uint8_t parseRegIndex(uint32_t bits, int startAddress);
    static std::string parseRd(uint32_t bits);
    static std::string parseRs1(uint32_t bits);
    static std::string parseRs2(uint32_t bits);
    static bool isBitSet(uint32_t bits, int index);
    template <typename T>
    static std::string toHexString(T number);
};
```

```
template <typename T>
std::string Instruction::toHexString(T number) {
    std::ostringstream ss;
    ss << std::setfill('0') << std::setw(sizeof(T) * 2) << std::hex << number;
    return ss.str();
}
```

InstructionFabric.h

```
#pragma once
#include <BType.h>
#include <EType.h>
#include <IAddrType.h>
#include <IType.h>
#include <JType.h>
#include <RType.h>
#include <SType.h>
#include <UType.h>
#include <UnknownType.h>
class InstructionFabric {
   public:
    static Instruction* createInstruction(const uint32_t bits) {
        const Type type = Storage::getType(
            Instruction::parseOpcodeBits(bits));
        if (type == Type::R) {
            return new RType(bits);
        } else if (type == Type::I) {
            return new IType(bits);
        } else if (type == Type::IAddr) {
```

```
            return new IAddrType(bits);
        } else if (type == Type::S) {
            return new SType(bits);
        } else if (type == Type::B) {
            return new BType(bits);
        } else if (type == Type::U) {
            return new UType(bits);
        } else if (type == Type::J) {
            return new JType(bits);
        } else if (type == Type::E) {
            return new EType(bits);
        } else {
            return new UnknownType(bits);
        }
    }
};
```

main.cpp

```
#include <ElfParser.h>
#include <InstructionFabric.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>
ElfParser* parseFile(std::ifstream& input, const char* path) {
    input.open(path, std::ios_base::binary);
    if (!input.is_open()) {
```

```
            throw std::ios_base::failure("Can`t open input file");
        }

        const auto parser = new ElfParser(input);
        parser->parse();
        return parser;
    };

    void openOutFile(std::ofstream& output, const char* path) {
        output.open(path, std::ios_base::binary);
        if (!output.is_open()) {
            throw std::ios_base::failure("Can`t open output file");
        }
    }
    int main(const int argc, char const* argv[]) {
        if (argc < 3) {
            std::cout << "2 arguments expected, " + std::to_string(argc - 1) + "
found\n";
            return 0;
        }
        try {
            std::ifstream input;
            ElfParser* parser = parseFile(input, argv[1]);
            try {
                std::ofstream output;
                openOutFile(output, argv[2]);
                parser->printDotText(output);
                output << "\n";
```

```
            parser->printSymtab(output);
        } catch (const std::ios_base::failure& e) {
            std::cout << e.what() << std::endl;
        }
        delete parser;
    } catch (std::ios_base::failure& e) {
        std::cout << e.what() << std::endl;
    } catch (std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

Storage.cpp

```
#include <Storage.h>

std::unordered_map<uint8_t, Type> Storage::typesMap = {
    {0b0110111, Type::U},
    {0b0010111, Type::U},
    {0b1101111, Type::J},
    {0b1100111, Type::IAddr},
    {0b0000011, Type::IAddr},
    {0b1100011, Type::B},
    {0b0100011, Type::S},
    {0b0010011, Type::I},
    {0b0110011, Type::R},
    {0b1110011, Type::E},
```

```
};
```

Storage.h

```
#pragma once

#include <Type.h>


#include <string>

#include <unordered_map>


class Storage {

   private:

    Storage() = delete;


    static std::unordered_map<uint8_t, Type> typesMap;


   public:

    static Type getType(const uint8_t opcode) {

        if (typesMap.count(opcode) > 0) {

            return typesMap[opcode];

        } else {

            return Type::UNKNOWN;

        }

    }


    static std::string getRegisterName(const uint8_t index) {

        if (index == 0) {
```

```
            return "zero";
        } else if (index == 1) {
            return "ra";
        } else if (index == 2) {
            return "sp";
        } else if (index == 3) {
            return "gp";
        } else if (index == 4) {
            return "tp";
        } else if (5 <= index && index <= 7) {
            return "t" + std::to_string(index - 5);
        } else if (index == 8) {
            return "s0";
        } else if (index == 9) {
            return "s1";
        } else if (10 <= index && index <= 17) {
            return "a" + std::to_string(index - 10);
        } else if (18 <= index && index <= 27) {
            return "s" + std::to_string(index - 18 + 2);
        } else if (28 <= index && index <= 31) {
            return "t" + std::to_string(index - 28 + 3);
        } else {
            return "invalid reg index: " + std::to_string(index);
        }
    }
};
```

SymTabInfoEnum.h

```
#pragma once

#include <string>


enum STT : char {

    NOTYPE = 0,

    OBJECT = 1,

    FUNC = 2,

    SECTION = 3,

    FILE_TYPE = 4,

    COMMON = 5,

    LOOS = 10,

    HIOS = 12,

    LOPROC = 13,

    HIPROC = 15,

};


inline std::string toStringSTT(const STT type) {

    if (type == NOTYPE) {

        return "NOTYPE";

    } else if (type == SECTION) {

        return "SECTION";

    } else if (type == OBJECT) {

        return "OBJECT";

    } else if (type == FUNC) {

        return "FUNC";
```

```
    } else if (type == FILE_TYPE) {
        return "FILE";
    } else {
        return "STT '" + std::to_string(type) + "'not defined yet";
    }
}

enum STB : char {
    LOCAL = 0,
    GLOBAL = 1,
    WEAK = 2,
};

inline std::string toStringSTB(const STB type) {
    if (type == LOCAL) {
        return "LOCAL";
    } else if (type == GLOBAL) {
        return "GLOBAL";
    } else if (type == WEAK) {
        return "WEAK";
    } else {
        return "STB '" + std::to_string(type) + "' not defined yet";
    }
}

enum STV : char {
    DEFAULT = 0,
```

```
    INTERNAL = 1,
    HIDDEN = 2,
    PROTECTED = 3,
};

inline std::string toStringSTV(const STV type) {
    if (type == DEFAULT) {
        return "DEFAULT";
    } else if (type == INTERNAL) {
        return "INTERNAL";
    } else if (type == HIDDEN) {
        return "HIDDEN";
    } else if (type == PROTECTED) {
        return "PROTECTED";
    } else {
        return "STV '" + std::to_string(type) + "' not defined yet";
    }
}
enum SHN : int {
    UNDEF = 0,
    LORESERVE = 0xff00,
    ABS = 0xfff1,
    SHN_COMMON = 0xfff2,
    HIRESERVE = 0xffff,
};

inline std::string toStringSHN(const SHN type) {
```

```
    if (type == UNDEF) {
        return "UNDEF";
    } else if (type == LORESERVE) {
        return "LORESERVE";
    } else if (type == ABS) {
        return "ABS";
    } else if (type == SHN_COMMON) {
        return "COMMON";
    } else if (type == HIRESERVE) {
        return "HIRESERVE";
    } else if (0xff00 <= type && type <= 0xff1f) {
        return "SPEC: " + std::to_string(type);
    } else {
        return std::to_string(type);
    }
}
```

Type.h

```
#pragma once
enum Type : char {
    R,    I,    IAddr,    S,    B,    U,    J,    UNKNOWN,    E,
};
```

AbstractStruct.h

```
#pragma once
```

```
#include "typedef.h"


class AbstractStruct {
   protected:
    template <typename T>
    static void read(T& place, const int bytes, std::istream& f) {
        f.read((char*)(&place), bytes);
    }

   public:
    virtual ~AbstractStruct() = default;
    virtual void fill(std::istream& f) = 0;
};
```

ElfHeader.h

```
#pragma once


#include "AbstractStruct.h"


class ElfHeader : AbstractStruct {
   public:
    static const int EI_NIDENT = 16;


    unsigned char name[EI_NIDENT];
    Elf32_Half type;
    Elf32_Half machine;
    Elf32_Word version;
```

```
Elf32_Addr entry;
Elf32_Off phoff;
Elf32_Off shoff;
Elf32_Word flags;
Elf32_Half ehsize;
Elf32_Half phentsize;
Elf32_Half phnum;
Elf32_Half shentsize;
Elf32_Half shnum;
Elf32_Half shstrndx;


void fill(std::istream& f) override {
    read(name, EI_NIDENT, f);
    validateName();
    read(type, sizeof(type), f);
    read(machine, sizeof(machine), f);
    validateMachine();
    read(version, sizeof(version), f);
    validateVersion();
    read(entry, sizeof(entry), f);
    read(phoff, sizeof(phoff), f);
    read(shoff, sizeof(shoff), f);
    read(flags, sizeof(flags), f);
    read(ehsize, sizeof(ehsize), f);
    read(phentsize, sizeof(phentsize), f);
    read(phnum, sizeof(phnum), f);
    read(shentsize, sizeof(shentsize), f);
```

```
            read(shnum, sizeof(shnum), f);

            read(shstrndx, sizeof(shstrndx), f);

            validateShstrndx();

        }


        void validateName() {

            // ID

            const char id[] = {0x7f, 'E', 'L', 'F'};

            for (int i = 0; i < 4; i++) {

                if (name[i] != id[i]) {

                    throw std::runtime_error("Invalid file identification");

                }

            }

            // CLASS

            if (name[4] != 1) {

                const std::string classStr = (name[4] == 2) ? "64-bit" : "Invalid
class";

                throw  std::runtime_error("Invalid  class.  Expected:  32-bit,
Found: " + classStr);

            }

            // DATA ENCODING

            if (name[5] != 1) {

                const std::string encodingStr = (name[5] == 2) ? "ELFDATA2MSB
(most significant)" : "Invalid data encoding";

                throw  std::runtime_error("Invalid  data  encoding.  Expected:
ELFDATA2LSB (least significant), Found: " + encodingStr);

            }

            // VERSION

            if (name[6] != 1) {
```

```
                    throw std::runtime_error("Invalid version. Expected: Current
version, Found: Invalid version");

                }

            }

            void validateMachine() {

                if (machine != 243) {

                    throw std::runtime_error("Invalid class. Expected: RISC-V");

                }

            }

            void validateVersion() {

                if (version != 1) {

                    throw std::runtime_error("Invalid version. Expected: Current
version");

                }

            }

            void validateShstrndx() {

                if (shstrndx >= shnum) {

                    throw std::runtime_error("Invalid shstrndx. In should be less
than shnum");

                }

            }

        };
```

ProgrammHeader.h

```
        #pragma once


        #include "AbstractStruct.h"
```

```
class ProgrammHeader : AbstractStruct {
   public:
    Elf32_Word type{};
    Elf32_Off offset{};
    Elf32_Addr vaddr{};
    Elf32_Addr paddr{};
    Elf32_Word filesz{};
    Elf32_Word memsz{};
    Elf32_Word flags{};
    Elf32_Word align{};

    void fill(std::istream& f) override {
        read(type, sizeof(type), f);
        read(offset, sizeof(offset), f);
        read(vaddr, sizeof(vaddr), f);
        read(paddr, sizeof(paddr), f);
        read(filesz, sizeof(filesz), f);
        read(memsz, sizeof(memsz), f);
        read(flags, sizeof(flags), f);
        read(align, sizeof(align), f);
    }
};
```

SectionHeader.h

```
#pragma once

#include "AbstractStruct.h"
```

```
class SectionHeader : AbstractStruct {

   public:

    Elf32_Word name;

    Elf32_Word type;

    Elf32_Word flags;

    Elf32_Addr addr;

    Elf32_Off offset;

    Elf32_Word size;

    Elf32_Word link;

    Elf32_Word info;

    Elf32_Word addralign;

    Elf32_Word entsize;


    void fill(std::istream& f) override {

        read(name, sizeof(name), f);

        read(type, sizeof(type), f);

        read(flags, sizeof(flags), f);

        read(addr, sizeof(addr), f);

        read(offset, sizeof(offset), f);

        read(size, sizeof(size), f);

        read(link, sizeof(link), f);

        read(info, sizeof(info), f);

        read(addralign, sizeof(addralign), f);

        read(entsize, sizeof(entsize), f);

    }
```

```
            static void validateTextSize(uint32_t size) {
                if (size % 4 != 0) {
                    throw    std::runtime_error("Invalid    .text    size:    "    +
std::to_string(size));
                }
            }
        };
```

SymTabEntry.h

```
        #pragma once

        #include "AbstractStruct.h"

        class SymTabEntry : AbstractStruct {
           public:
            Elf32_Word name;
            Elf32_Addr value;
            Elf32_Word size;
            unsigned char info;
            unsigned char other;
            Elf32_Half shndx;

            void fill(std::istream& f) override {
                read(name, sizeof(name), f);
                read(value, sizeof(value), f);
                read(size, sizeof(size), f);
                read(info, sizeof(info), f);
                read(other, sizeof(other), f);
```

```
        read(shndx, sizeof(shndx), f);
    }
};
```

typedef.h

```
#include <iostream>

typedef uint16_t Elf32_Half;
typedef int16_t Elf32_SHalf;
typedef uint32_t Elf32_Word;
typedef int32_t Elf32_Sword;
typedef uint64_t Elf32_Xword;
typedef int64_t Elf32_Sxword;

typedef uint32_t Elf32_Off;
typedef uint32_t Elf32_Addr;
typedef uint16_t Elf32_Section;
```

BType.cpp

```
#include "BType.h"

std::unordered_map<uint8_t, std::string> BType::mnemonics{
    {0b000, "beq"},
    {0b001, "bne"},
    {0b100, "blt"},
    {0b101, "bge"},
```

```
        {0b110, "bltu"},
        {0b111, "bgeu"},
    };
```

BType.h

```
    #include <Instruction.h>

    class BType : public Instruction {
       private:
        static std::unordered_map<uint8_t, std::string> mnemonics;

       public:
        explicit BType(uint32_t bits) : Instruction(bits) {}
        ~BType() override = default;

        bool needLabel() const override {
            return true;
        }

        uint32_t getImmAddr() const override {
            return address + getImm();
        }

       private:
        std::string instructionString() const override {
            return getMnemonic() + '\t' + parseRs1(bits) + ", " + parseRs2(bits)
+ ", " + parseImm();
        }
```

```
    int16_t getImm() const {
        int16_t imm = 0;
        imm += isBitSet(bits, 7) ? (1 << 11) : 0;

        for (size_t i = 0; i < 4; i++) {
            imm += isBitSet(bits, i + 7 + 1) ? (1 << (i + 1)) : 0;
        }

        for (size_t i = 0; i < 6; i++) {
            imm += isBitSet(bits, i + 25) ? (1 << (i + 5)) : 0;
        }

        imm -= (bits & (1 << (25 + 6))) > 0 ? (1 << 12) : 0;

        return imm;
    }

    std::string parseImm() const {
        return toHexString(address + getImm());
    }

    std::string getMnemonic() const {
        return mnemonics[parseFunct3(bits)];
    }
};
```

EType.h

```
#pragma once

#include <Instruction.h>


class EType : public Instruction {

   public:

    explicit EType(uint32_t bits) : Instruction(bits) {}

    ~EType() override = default;


   private:

    std::string instructionString() const override {

        return getMnemonic() + "\t\t";

    }


    std::string getMnemonic() const {

        if ((bits >> 20) == 0) {

            return "ecall";

        } else if ((bits >> 20) == 1) {

            return "ebreak";

        } else {

            return "unknown EType";

        }

    }

};
```

IAddrType.cpp

```
#include "IAddrType.h"


// [funct3 | opcode[6]]
std::unordered_map<uint8_t, std::string> IAddrType::mnemonics{
    {0b0001, "jalr"},
    {0b0000, "lb"},
    {0b0010, "lh"},
    {0b0100, "lw"},
    {0b1000, "lbu"},
    {0b1010, "lhu"},
};


IAddrType.h
#include <Instruction.h>


class IAddrType : public Instruction {
   private:
    static std::unordered_map<uint8_t, std::string> mnemonics;


   public:
    explicit IAddrType(uint32_t bits) : Instruction(bits) {}
    ~IAddrType() override = default;


    std::string instructionString() const override {
        return getMnemonic() + '\t' + parseRd(bits) + ", " + parseImm12() +
'(' + parseRs1(bits) + ')';
    }
```

```
    private:
        std::string parseImm12() const {
            int16_t imm12 = 0;
            for (size_t i = 0; i < 11; i++) {
                imm12 += isBitSet(bits, i + 20) > 0 ? (1 << i) : 0;
            }

            imm12 -= isBitSet(bits, 11 + 20) > 0 ? (1 << 11) : 0;

            return std::to_string(imm12);
        }

        std::string getMnemonic() const {
            uint8_t key = (parseFunct3(bits) << 1) + (isBitSet(bits, 6) ? 1 :
0);
            return mnemonics[key];
        }
    };
```

IType.cpp

```
    #include "IType.h"

    std::unordered_map<uint8_t, std::string> IType::mnemonics{
        {0b000, "addi"},
        {0b010, "slti"},
```

```
    {0b011, "sltiu"},
    {0b100, "xori"},
    {0b110, "ori"},
    {0b111, "andi"},

    // shamt [funct7 >> 5 | funct3]
    {0b0001, "slli"},
    {0b0101, "srli"},
    {0b1101, "srai"},
};
```

IType.h

```
#include <Instruction.h>

class IType : public Instruction {
   private:
    bool isShamt;
    static std::unordered_map<uint8_t, std::string> mnemonics;

   public:
    IType(uint32_t bits) : Instruction(bits) {
        uint8_t funct3 = parseFunct3(bits);
        isShamt = funct3 == 0b001 || funct3 == 0b101;
    }
    ~IType() = default;

    std::string instructionString() const override {
```

```
            return getMnemonic() + '\t' + parseRd(bits) + ", " + parseRs1(bits)
+ ", " + parseImm12();

        }



        private:

        std::string parseImm12() const {

            int16_t imm12 = 0;

            for (size_t i = 0; i < 11; i++) {

                imm12 += isBitSet(bits, i + 20) > 0 ? (1 << i) : 0;

            }



            imm12 -= isBitSet(bits, 11 + 20) > 0 ? (1 << 11) : 0;



            return std::to_string(imm12);

        }



        std::string getMnemonic() const {

            uint8_t key = parseFunct3(bits);

            if (isShamt) {

                key += isBitSet(parseFunct7(bits), 5) ? 0b1000 : 0;

            }



            return mnemonics[key];

        }

    };
```

JType.h

```
#pragma once

#include <Instruction.h>


class JType : public Instruction {

   public:

    explicit JType(uint32_t bits) : Instruction(bits) {}

    ~JType() override = default;


    bool needLabel() const override {

        return true;

    }


    uint32_t getImmAddr() const override {

        return address + getImm();

    }


   private:

    std::string instructionString() const override {

        return getMnemonic() + '\t' + parseRd(bits) + ", " + parseImm();

    }


    int32_t getImm() const {

        int32_t imm = 0;


        for (size_t i = 12; i < 20; i++) {

            imm += isBitSet(bits, i) ? (1 << i) : 0;
```

```
        }

        imm += isBitSet(bits, 20) ? (1 << 11) : 0;

        for (size_t i = 0; i < 10; i++) {
            imm += isBitSet(bits, i + 21) ? (1 << (i + 1)) : 0;
        }

        imm -= isBitSet(bits, 31) ? (1 << 20) : 0;
        return imm;
    }

    std::string parseImm() const {
        return toHexString(address + getImm());
    }

    static std::string getMnemonic() {
        return "jal";
    }
};
```

RType.cpp

```
#include "RType.h"

std::unordered_map<uint16_t, std::string> RType::mnemonics{
    // RV32I
```

```
    {0b0000000000, "add"},
    {0b0100000000, "sub"},
    {0b0000000001, "sll"},
    {0b0000000010, "slt"},
    {0b0000000011, "sltu"},
    {0b0000000100, "xor"},
    {0b0000000101, "srl"},
    {0b0100000101, "sra"},
    {0b0000000110, "or"},
    {0b0000000111, "and"},

    // RV32M
    {0b0000001000, "mul"},
    {0b0000001001, "mulh"},
    {0b0000001010, "mulhsu"},
    {0b0000001011, "mulhu"},
    {0b0000001100, "div"},
    {0b0000001101, "divu"},
    {0b0000001110, "rem"},
    {0b0000001111, "remu"},
};
```

RType.h

```
#include <Instruction.h>

class RType : public Instruction {
   private:
```

```
        static std::unordered_map<uint16_t, std::string> mnemonics;


    public:
        explicit RType(uint32_t bits) : Instruction(bits) {}
        ~RType() override = default;

        std::string instructionString() const override {
            return getMnemonic() + '\t' + parseRd(bits) + ", " + parseRs1(bits)
+ ", " + parseRs2(bits);
        }

    private:
        std::string getMnemonic() const {
            uint16_t key = (parseFunct7(bits) << 3) + parseFunct3(bits);
            return mnemonics[key];
        }
};
```

SType.cpp

```
#include "SType.h"

std::unordered_map<uint8_t, std::string> SType::mnemonics{
    {0b000, "sb"},
    {0b001, "sh"},
    {0b010, "sw"},
};
```

SType.h

```
#include <Instruction.h>


class SType : public Instruction {

   private:

    static std::unordered_map<uint8_t, std::string> mnemonics;


   public:

    explicit SType(uint32_t bits) : Instruction(bits) {}

    ~SType() override = default;


   private:

    std::string instructionString() const override {

        return getMnemonic() + '\t' + parseRs2(bits) + ", " + parseImm() +
'(' + parseRs1(bits) + ')';

     }


    std::string parseImm() const {

        int16_t imm = 0;


        for (size_t i = 0; i < 5; i++) {

            imm += isBitSet(bits, i + 7) ? (1 << i) : 0;

        }


        for (size_t i = 0; i < 6; i++) {

            imm += isBitSet(bits, i + 25) ? (1 << (i + 5)) : 0;

        }
```

```
        imm -= (bits & (1 << (25 + 6))) > 0 ? (1 << 11) : 0;

        return std::to_string(imm);
    }

    std::string getMnemonic() const {
        return mnemonics[parseFunct3(bits)];
    }
};
```

UnknownType.h

```
#include <Instruction.h>

class UnknownType : public Instruction {
   public:
    explicit UnknownType(uint32_t bits) : Instruction(bits) {}
    ~UnknownType() override = default;

   private:
    std::string instructionString() const override {
        return "unknown_instruction";
    }
};
```

UType.h

```
#include <Instruction.h>


class UType : public Instruction {
   public:
    explicit UType(uint32_t bits) : Instruction(bits) {}
    ~UType() override = default;


   private:
    std::string instructionString() const override {
        return getMnemonic() + '\t' + parseRd(bits) + ", 0x" + parseImm();
    }


    std::string parseImm() const {
        int32_t imm = 0;


        for (size_t i = 12; i < 32; i++) {
            imm += isBitSet(bits, i) ? (1 << i) : 0;
        }


        const std::string s = toHexString(imm);
        return s.substr(0, s.length() - 3);
    }


    std::string getMnemonic() const {
        const uint8_t opcode = parseOpcodeBits(bits);
        if (opcode == 0b0110111) {
```

```
                return "lui";
            } else if (opcode == 0b0010111) {
                return "auipc";
            } else {
                throw std::runtime_error("Opcode '" + std::to_string(opcode) +
"' doesn't match UType instruction");
            }
        }
    };
```

CMakeListst.txt

Root

```
    cmake_minimum_required (VERSION 3.8)

    project ("RiscV-Disassembler")

    set(executableName "rv3")
    set(CMAKE_CXX_STANDARD 17)
    file (GLOB h
        "${CMAKE_CURRENT_SOURCE_DIR}/*.h"
    )
    file (GLOB cpp
        "${CMAKE_CURRENT_SOURCE_DIR}/*.cpp"
    )
    add_executable (${executableName} ${h} ${cpp})
    target_include_directories(${executableName}                          PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
    add_subdirectory(Instructions)
    add_subdirectory(Structs)
```

Types folder

```
cmake_minimum_required (VERSION 3.8)

project("RiscV-Disassembler")

file (GLOB h

    "${CMAKE_CURRENT_SOURCE_DIR}/*.h"

)

target_sources(${executableName} PRIVATE ${h})

target_include_directories(${executableName}                                PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
```

Instructions folder

```
cmake_minimum_required (VERSION 3.8)

project("RiscV-Disassembler")

file(GLOB h

    "${CMAKE_CURRENT_SOURCE_DIR}/*Type.h"

)

file(GLOB cpp

    "${CMAKE_CURRENT_SOURCE_DIR}/*Type.cpp"

)

target_sources(${executableName} PRIVATE ${h} ${cpp})

target_include_directories(${executableName}                                PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
```