|  |  |  |
| --- | --- | --- |
| Лабораторная работа №3 | M3138 | 2023 |
| ISA | Попович Виталий Сергеевич | |
|

**Цель работы**

Знакомство с архитектурой набора команд RISC-V.

**Инструментарий и требования к работе**

Языка программирования – C++

Компилятор - [x86\_64-12.2.0-release-posix-seh-msvcrt-rt\_v10-rev2.7z](https://www.virustotal.com/gui/file/3c495fad25a092d3dd555b8932821171461c2bbd1f882ff55d703c3ee5f95273)

**Описание RISC-V**

В данной работе мы используем 32 битный RISC-V(также существуют 64 и 128 битные).

Обычный набор RV32I содержит 32 битные целочисленные операции. Также входят: служебные инструкции, мин. кол-во арифметических и битовых операций(также операции с памятью и переходы, безусловные так и условные)

Существует расширение RV32-М, где добавляются инструкции для умножения и деления(всё целочисленное).

Также существуют ещё некоторые расширения:

F – операции чисел с плавающей точкой(одинарная точность)

D - операции чисел с плавающей точкой(двойная точность)

Q - операции чисел с плавающей точкой(четверная точность

**Регистры RISC-V**

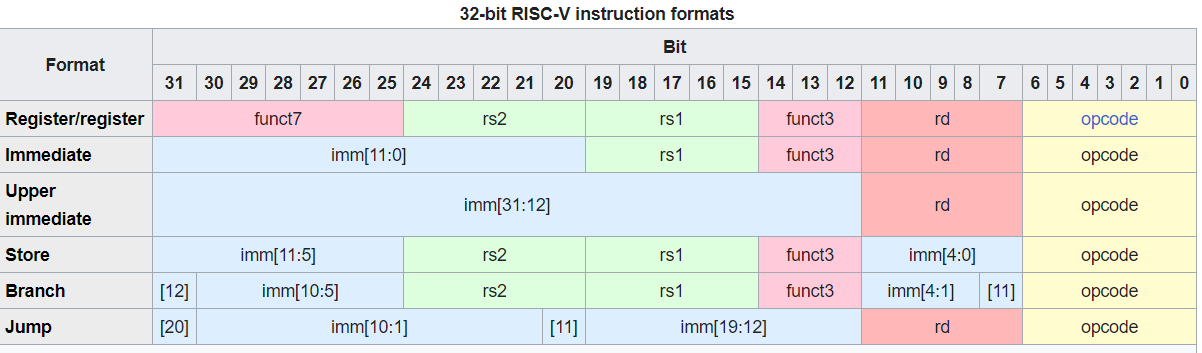
Стандартный набор содержит 32 регистра. Описание каждого из них можно увидеть ниже(см. Рисунок 1)



**Рисунок – 1**

**Инструкции RISC-V**

Каждая инструкция в RISC-V состоит из размера ровно 4 байта(см. Рисунок 2)



**Рисунок – 2**

Все инструкции можно разбить на 6 типов, каждый из которых мы рассмотрим:

**Register** – на вход подаётся два регистра(rs1, rs2), результат же записывается в один регистр rd. Используется для арифметических и побитовых операций(Стоит заметить, что в данном наборе инструкций все Register инструкции имеют одинаковый opcode)

**Immediate** – на вход подаётся один регистр(rs1) и некоторая константа, результат же записывается в один регистр rd(константа может использоваться как число, так и как значение сдвига)

**Upper immediate** – аналогично Immediate, но работает как со старшими битами 32-ух битных значений.

**Store** – на вход подаётся два регистра rs1(адрес, относительно которого offset) и rs2(данные для записи).Используется для записи из данных из регистров прямо в память(как раз они использует immediate в качестве offset)

**Branch** – на вход подаётся два регистра rs1 и rs2.Используется для условного(некоторого условия между rs1 и rs2) перехода(также использует immediate в качестве offset)

**Jump** – используется для безусловного перехода(также использует immediate в качестве offset)

**Описание структуры файла ELF**

Каждый такой файл начинается с ELF заголовка, который содержит необходимую информацию о версии, разрядности(а также для идентификации), указатели на части файла(массивы Program Header и Section Header). Ещё заголовок хранит байты(свободные), которые используются для “подгонки” следующих(так как существует соглашение о том, что все поля размера n начиаются с адресов кратных n)

Массив Program Header-ов состоит из заголовков, содержащие информацию для операционной системы(информация для запуска, такая как: кол-во памяти, виртуальный адрес памяти, место, куда загрузить)

Далее располагаются сами данные(информация о них находится в массиве Section Header-ов, адрес, которых находится в указателях из Elf Header)(Таким образом файл возможно читать последовательно)

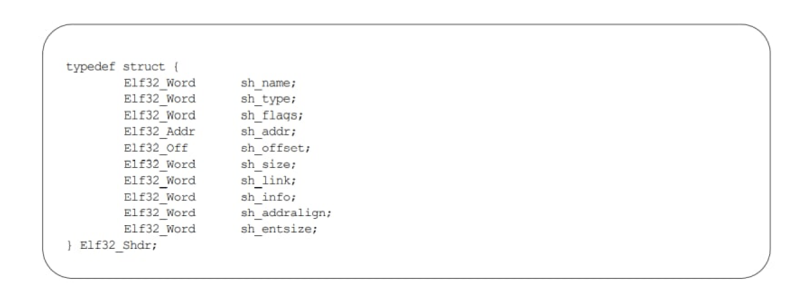
Массив Section Header’ов хранит информацию о секциях(адрес нахождения, название, содержание)

Названия секций, меток располагаются в таблицах строк.

Все названия, как секций, так и меток (например функций) хранятся в таблицах строк(Названия – в shStrTab – таблице строк Section Header’ов, Меток - в strTab – таблице строк).Адресация происходит за счёт того, что элементы содержат ссылку на начало необходимой строки.

**Основные секции ELF:**

**.text**

Хранение исполняемого кода(см. Рисунок 3)

**Рисунок – 3**

Рассмотрим Elf32\_Shdr:   
Sh\_name – имя раздела, задает индекс в .strtab

sh\_offset – смещение секции в файле

sh\_size – размер секции в файле

sh\_info – дополнительная информация о файле. Данная структура подходит не только для .text, но и для всех остальных секций.

**.strtab**

Хранение имён “символов” из .symtab

**.symtab**

Хранят “символы” – имена функций и переменных. Имена существуют для представления определенного места в фазйле или в памяти(В данном случае “символ” может занимать более 1 места)(см. Рисунок 4)



**Рисунок – 4**

Рассмотрим Elf32\_Sym:

st\_name – хранение позиции, с которой в strtab можно определить имя

st\_info – хранение тип “символа”, и его связи.

st\_shndx - хранение индекса таблица заголовка раздела.

**Описание работы написанного кода**

Пути входного и выходного файлов содержатся в аргументах командной строки.

Чтобы было проще разбирать файл, нужно добиться расположения в памяти 1 к 1.Для этого будем использовать typedef(из оригинальных источников), повторим все структуры файла в виде классов C++.

Разбором файла занимается класс “ParserOfElf”, он создаст нужные структуры, заполняющие поля из входного потока(так будет проще “ориентироваться” в какой части файла мы находимся, так как мы читаем последовательно)

Также существует одна проблема: Section Header’ы находятся в конце файла.Для решения будем действовать так: всё, что находится “между” сохраняется в некоторый буфер, из которого позже берём информацию для заполнения оставшихся структур.

Заметим, что RISC-V, имеет особенность: по opcode можно однозначно определить тип инструкции(реализуем определение типа при помощи ассоциативного контейнера)

**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**

Value Size 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

**Список источников**

1.[Создаем ELF-файл с отладочной информацией](https://habr.com/ru/post/199490/)

2.[Создание исполняемого файла ELF вручную](https://habr.com/ru/post/543622/)

3.[Введение в ELF-файлы в Linux: понимание и анализ](https://habr.com/ru/post/480642/)

4.[Устройство ELF-файлов](https://xakep.ru/2022/08/31/elf-anatomy-2/)

5.[Структуры ELF](https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblj/index.html)

6.[RISC-V](https://ru.wikipedia.org/wiki/RISC-V)

7.[Список кодов Elf Header](https://www.sco.com/developers/gabi/latest/ch4.eheader.html)

8.[Typedef типов в структурах](https://github.com/u-boot/u-boot/blob/master/include/elf.h)

**Листинг кода**

**Type.h**

#pragma once

enum Type : char {

R,

I,

IAddr,

S,

B,

U,

J,

UNKNOWN,

E,

};

**SymTabInfo.h**

#pragma once

#include <string>

using namespace std;

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 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 if (type == COMMON) {

return "COMMON";

} else if (type == LOOS) {

return "LOOS";

} else if (type == HIOS) {

return "HIOS";

} else if (type == LOPROC) {

return "LOPROC";

} else if (type == HIPROC) {

return "HIPROC";

} else {

return "Unnknown STT '" + to\_string(type) + "'";

}

}

enum STB : char {

LOCAL = 0,

GLOBAL = 1,

WEAK = 2,

STB\_LOOS = 10,

STB\_HIOS = 12,

STB\_LOPROC = 13,

STB\_HIPROC = 13,

};

inline string toStringSTB(const STB type) {

if (type == LOCAL) {

return "LOCAL";

} else if (type == GLOBAL) {

return "GLOBAL";

} else if (type == WEAK) {

return "WEAK";

} else if (type == STB\_LOOS) {

return "LOOS";

} else if (type == STB\_HIOS) {

return "HIOS";

} else if (type == STB\_LOPROC) {

return "LOPROC";

} else if (type == STB\_HIPROC) {

return "HIPROC";

} else {

return "Unnknown STB '" + to\_string(type) + "'";

}

}

enum STV : char {

DEFAULT = 0,

INTERNAL = 1,

HIDDEN = 2,

PROTECTED = 3,

};

inline 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 "Unnknown STV '" + to\_string(type) + "'";

}

}

enum SHN : int {

UNDEF = 0,

LORESERVE = 0xff00,

ABS = 0xfff1,

SHN\_COMMON = 0xfff2,

HIRESERVE = 0xffff,

};

inline 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: " + to\_string(type);

} else {

return to\_string(type);

}

}

**Storage.h**

#pragma once

#include <Type.h>

using namespace std;

#include <string>

#include <unordered\_map>

class Storage {

private:

Storage() = delete;

static 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 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" + to\_string(index - 5);

} else if (index == 8) {

return "s0";

} else if (index == 9) {

return "s1";

} else if (10 <= index && index <= 17) {

return "a" + to\_string(index - 10);

} else if (18 <= index && index <= 27) {

return "s" + to\_string(index - 18 + 2);

} else if (28 <= index && index <= 31) {

return "t" + to\_string(index - 28 + 3);

} else {

return "invalid reg index: " + to\_string(index);

}

}

};

**Storage.cpp**

#include <Storage.h>

using namespace 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},

};

**ParserOfElf.h**

#pragma once

#include <ElfHeader.h>

#include <InstructionFab.h>

#include <ProgrammHeader.h>

#include <SectionHeader.h>

#include <SymTabEntry.h>

#include <SymTabInfo.h>

#include <fstream>

#include <sstream>

#include <vector>

using namespace std;

class ParserOfElf {

static constexpr uint8\_t SYM\_TAB = 2;

static constexpr uint8\_t STR\_TAB = 3;

public:

explicit ParserOfElf(ifstream& f);

~ParserOfElf();

void parse();

void printDotText(ostream& out);

void printSymtab(ostream& out) const;

private:

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;

string getStringFromStrTab(uint32\_t offset) const;

string getStringFromShStrTab(uint32\_t offset) const;

// .text

uint32\_t textAddress;

uint32\_t textVirtualAddress;

uint32\_t textSize;

vector<Instruction\*> instructions;

unordered\_map<uint32\_t, string> labels;

};

**ParserOfElf.cpp**

#include "ParserOfElf.h"

using namespace std;

ParserOfElf::ParserOfElf(ifstream& f) : file(f) {}

void ParserOfElf::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];

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 = InstructionFab::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" + 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 ParserOfElf::printDotText(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 ParserOfElf::printSymtab(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];

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 ParserOfElf::fillStrTab(const char\* buff) {

strTab = new char[strTabSize];

for (int j = 0; j < strTabSize; j++) {

strTab[j] = buff[strTabAddress + j - bufferOffset];

}

}

void ParserOfElf::fillShStrTab(const char\* buff) {

shStrTab = new char[shStrTabSize];

for (int j = 0; j < shStrTabSize; j++) {

shStrTab[j] = buff[shStrTabAddress + j - bufferOffset];

}

}

string ParserOfElf::getStringFromStrTab(const uint32\_t offset) const {

if (offset > strTabSize) {

throw runtime\_error("strTab index '" + to\_string(offset) + "' is out of bound for size '" + to\_string(strTabSize) + "'");

}

stringstream ss;

int charsRead = 0;

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

ss << strTab[offset + charsRead];

++charsRead;

}

return ss.str();

}

string ParserOfElf::getStringFromShStrTab(const uint32\_t offset) const {

if (offset > shStrTabSize) {

throw runtime\_error("shStrTab index '" + to\_string(offset) + "' is out of bound for size '" + to\_string(shStrTabSize) + "'");

}

stringstream ss;

int charsRead = 0;

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

ss << shStrTab[offset + charsRead];

++charsRead;

}

return ss.str();

}

ParserOfElf::~ParserOfElf() {

for (const auto& instruction : instructions) {

delete instruction;

}

delete[] programHeaders;

delete[] sectionHeaders;

delete[] symTableEntries;

delete[] strTab;

delete[] shStrTab;

}

**main.cpp**

#include <ParserOfElf.h>

#include <InstructionFab.h>

using namespace std;

#include <fstream>

#include <iomanip>

#include <iostream>

#include <vector>

ParserOfElf\* parseFile(ifstream& input, const char\* path) {

input.open(path, ios\_base::binary);

if (!input.is\_open()) {

throw ios\_base::failure("Can`t open input file");

}

const auto parser = new ParserOfElf(input);

parser->parse();

return parser;

};

void openOutFile(ofstream& output, const char\* path) {

output.open(path, ios\_base::binary);

if (!output.is\_open()) {

throw ios\_base::failure("Can`t open output file");

}

}

int main(const int argc, char const\* argv[]) {

if (argc < 3) {

cout << "2 arguments expected, " + to\_string(argc - 1) + " found\n";

return 0;

}

try {

ifstream input;

ParserOfElf\* parser = parseFile(input, argv[1]);

try {

ofstream output;

openOutFile(output, argv[2]);

parser->printDotText(output);

output << "\n";

parser->printSymtab(output);

} catch (const ios\_base::failure& e) {

cout << e.what() << endl;

}

delete parser;

} catch (ios\_base::failure& e) {

cout << e.what() << endl;

} catch (runtime\_error& e) {

cout << e.what() << endl;

}

return 0;

}

**InstructionFab.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 InstructionFab {

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);

}

}

};

**Instruction.h**

#pragma once

#include <Storage.h>

using namespace std;

#include <fstream>

#include <iomanip>

class Instruction {

protected:

uint32\_t bits;

uint32\_t address{};

string label{};

explicit Instruction(uint32\_t bits);

string addressString() const;

virtual string instructionString() const = 0;

public:

virtual void toString(ostream& out) const;

virtual ~Instruction();

void setAddress(uint32\_t givenAddress);

void setLabel(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 string parseRd(uint32\_t bits);

static string parseRs1(uint32\_t bits);

static string parseRs2(uint32\_t bits);

static bool isBitSet(uint32\_t bits, int index);

template <typename T>

static string toHexString(T number);

};

template <typename T>

string Instruction::toHexString(T number) {

ostringstream ss;

ss << setfill('0') << setw(sizeof(T) \* 2) << hex << number;

return ss.str();

}

**Instruction.cpp**

#include <Instruction.h>

Instruction::Instruction(const uint32\_t bits) : bits(bits){};

using namespace std;

string Instruction::addressString() const {

return toHexString(address);

}

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

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 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;

}

string Instruction::parseRd(const uint32\_t bits) {

return Storage::getRegisterName(parseRegIndex(bits, 7));

}

string Instruction::parseRs1(const uint32\_t bits) {

return Storage::getRegisterName(parseRegIndex(bits, 15));

}

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;

}

**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"

using namespace std;

unordered\_map<uint8\_t, string> BType::mnemonics{

{0b000, "beq"},

{0b001, "bne"},

{0b100, "blt"},

{0b101, "bge"},

{0b110, "bltu"},

{0b111, "bgeu"},

};

**BType.h**

#include <Instruction.h>

using namespace std;

class BType : public Instruction {

private:

static unordered\_map<uint8\_t, 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:

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;

}

string parseImm() const {

return toHexString(address + getImm());

}

string getMnemonic() const {

return mnemonics[parseFunct3(bits)];

}

};

**EType.h**

#pragma once

#include <Instruction.h>

using namespace std;

class EType : public Instruction {

public:

explicit EType(uint32\_t bits) : Instruction(bits) {}

~EType() override = default;

private:

string instructionString() const override {

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

}

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"

using namespace std;

// [funct3 | opcode[6]]

unordered\_map<uint8\_t, string> IAddrType::mnemonics{

{0b0001, "jalr"},

{0b0000, "lb"},

{0b0010, "lh"},

{0b0100, "lw"},

{0b1000, "lbu"},

{0b1010, "lhu"},

};

**IAddrType.h**

#include <Instruction.h>

using namespace std;

class IAddrType : public Instruction {

private:

static unordered\_map<uint8\_t, string> mnemonics;

public:

explicit IAddrType(uint32\_t bits) : Instruction(bits) {}

~IAddrType() override = default;

string instructionString() const override {

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

}

private:

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 to\_string(imm12);

}

string getMnemonic() const {

uint8\_t key = (parseFunct3(bits) << 1) + (isBitSet(bits, 6) ? 1 : 0);

return mnemonics[key];

}

};

**IType.cpp**

#include "IType.h"

using namespace std;

unordered\_map<uint8\_t, 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>

using namespace std;

class IType : public Instruction {

private:

bool isShamt;

static unordered\_map<uint8\_t, string> mnemonics;

public:

IType(uint32\_t bits) : Instruction(bits) {

uint8\_t funct3 = parseFunct3(bits);

isShamt = funct3 == 0b001 || funct3 == 0b101;

}

~IType() = default;

string instructionString() const override {

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

}

private:

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 to\_string(imm12);

}

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>

using namespace std;

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:

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;

}

string parseImm() const {

return toHexString(address + getImm());

}

static string getMnemonic() {

return "jal";

}

};

**RType.cpp**

#include "RType.h"

using namespace std;

unordered\_map<uint16\_t, 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>

using namespace std;

class RType : public Instruction {

private:

static unordered\_map<uint16\_t, string> mnemonics;

public:

explicit RType(uint32\_t bits) : Instruction(bits) {}

~RType() override = default;

string instructionString() const override {

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

}

private:

string getMnemonic() const {

uint16\_t key = (parseFunct7(bits) << 3) + parseFunct3(bits);

return mnemonics[key];

}

};

**SType.cpp**

#include "SType.h"

using namespace std;

unordered\_map<uint8\_t, string> SType::mnemonics{

{0b000, "sb"},

{0b001, "sh"},

{0b010, "sw"},

};

**SType.h**

#include <Instruction.h>

using namespace std;

class SType : public Instruction {

private:

static unordered\_map<uint8\_t, string> mnemonics;

public:

explicit SType(uint32\_t bits) : Instruction(bits) {}

~SType() override = default;

private:

string instructionString() const override {

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

}

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 to\_string(imm);

}

string getMnemonic() const {

return mnemonics[parseFunct3(bits)];

}

};

**UnknownType.h**

#include <Instruction.h>

using namespace std;

class UnknownType : public Instruction {

public:

explicit UnknownType(uint32\_t bits) : Instruction(bits) {}

~UnknownType() override = default;

private:

string instructionString() const override {

return "unknown\_instruction";

}

};

**UType.h**

#include <Instruction.h>

using namespace std;

class UType : public Instruction {

public:

explicit UType(uint32\_t bits) : Instruction(bits) {}

~UType() override = default;

private:

string instructionString() const override {

return getMnemonic() + '\t' + parseRd(bits) + ", 0x" + parseImm();

}

string parseImm() const {

int32\_t imm = 0;

for (size\_t i = 12; i < 32; i++) {

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

}

const string s = toHexString(imm);

return s.substr(0, s.length() - 3);

}

string getMnemonic() const {

const uint8\_t opcode = parseOpcodeBits(bits);

if (opcode == 0b0110111) {

return "lui";

} else if (opcode == 0b0010111) {

return "auipc";

} else {

throw runtime\_error("Opcode '" + to\_string(opcode) + "' doesn't match UType instruction");

}

}

};