Лабораторная работа №3	Б10	2022
ISA	Хорохорин Андрей Сергеевич	

Цель работы

Знакомство с архитектурой набора команд RISC-V

Инструментарий

1. clang version 14.0.6
2. VIM - Vi IMproved 9.0 (2022 Jun 28, compiled Nov 19 2022 14:37:14)
3. GNU Make 4.3
4. XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022/Arch Linux)

Описание системы кодирования команд RISC-V

Общая структура ISA

Стандарт RISC-V декларирует 4 минимальных ISA для 32, 64 и даже 128 битных систем, а также множество ISA-расширений для них. RISC-V может быть расширена и при помощи сторонних, не входящих в стандарт RISC-V. Это достигается некоторой избыточностью кодирования команд и резервированием некоторых кодов команд для сторонних команд-расширений.
Стандарт написан таким образом, чтобы RISC-V можно было реализовать не только в виде процессора, поддерживающего эту ISA, но и как программный эмулятор. Любую сущность поддерживающую описанный набор инструкций называют платформой RISC-V. Вне зависимости от платформы логически есть только одно адресное пространство, которое может как дизъюнктно объединять несколько присутствующих физических, так и объединять имея некоторое не пустое пересечение адресных пространств.
Базовая ISA предполагает 32 битное кодирование для всех команд, что оправдывает название архитектуры, но стандарт также позволят использовать кодирование с переменной длинной инструкции, но необходимо, чтобы длина каждой была кратна 16 битам.
Отдельно стоит отметить, что некоторые стандарты уже утверждены и не могут быть изменены в будущем, как например разрабатываемая нами RV32I, RV32M, в то же время есть стандарты, которые могут быть изменены в будущем.
Для примера устройства базовой ISA, к которой уже в дальнейшем будут накладываться улучшения, возьмём RV32I. Остальные базовые ISA отличаются главным образом количеством и размеров регистров.

Устройство базовой ISA на примере RV32I

В RV32I в пользование программиста предоставляется 32 регистра, каждый из которых размера 32 бита. Один из регистров имеет особенность: его содержимое всегда равно нулю, даже после записи в него. Помимо этого, есть ещё один регистр pc, основной целью которого является хранения места в памяти текущей исполняемой инструкции. Каждая из команд кодируется в двоичном виде. Для облегчения декодирования все команды разбиты на 6 типов, которые приведены в таблице ниже.

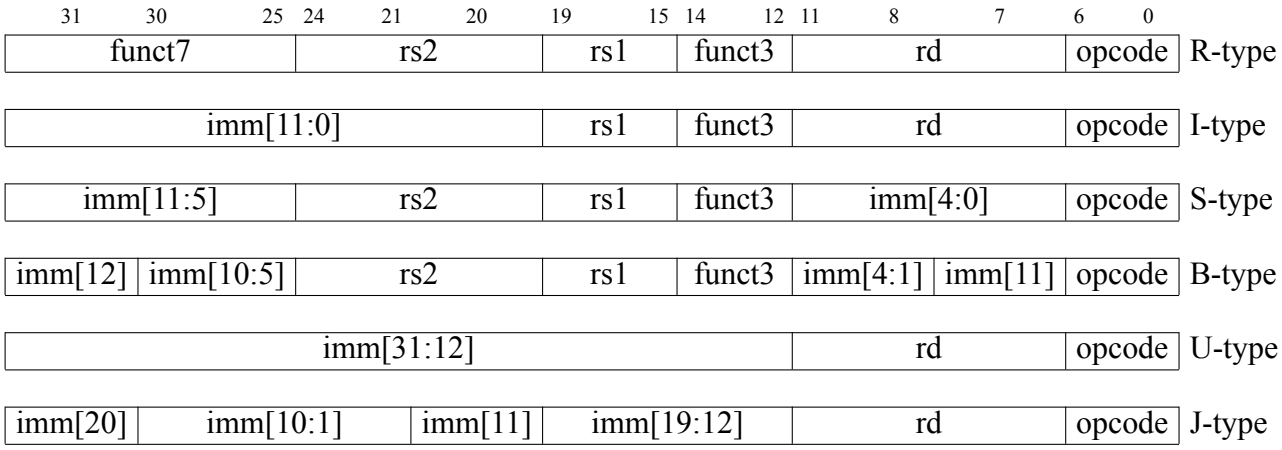

Рис. 1: Виды инструкций RISC-V

Заметим, что типы U,J и B,S почти совпадают. Они существуют для кодирования команд, где данные imm точно делится на 2, позволяет передать один дополнительный старший бит. А такая странная адресация в B, J сделана из-за того, что аппаратно легче записать один бит, чем записывать один бит и делать битовый сдвиг всего imm, чтобы кодировать в imm[12:6], imm[5:1].
В таблице выше за rd обозначен регистр куда сохранять результат команды, а за rs — из какого регистра считывать данные. Из такой системы кодирования следует, что данная архитектура относится к типу регистр-регистр. Помимо этого заметим, что команды построены так, чтобы rd и rs всегда были на одном и том же месте, для того, чтобы облегчить их декодирование.
Чтобы уже сейчас разобраться с типом нашей ISA скажу, что наша ISA является Load-Store, что значит, что каждая операция либо производит какое-либо вычисление на ALU, либо работает с памятью, но ни в коем виде не делает это одновременно.

Целочисленные арифметические инструкции

Инструкции управления исполнения

Инструкции для работы с памятью

Fence

Инструкции-подсказки исполнения

Инструкции-подсказки исполнения

Расширение RV32M

Описание структуры файла ELF

ELF(Executable and Linkable Format) — бинарный формат файла предназначенный как для хранения как целых исполняемых файлов с машинным кодом и дополнительной информацией для его запуска, так и для хранения отдельных его частей, которые предварительно должны быть слинкованы, чтобы получить полноценный исполняемый файл.
Сам ELF файл состоит из 4 частей:

- **Заголовок файла.** Хранит в себе метаданные о файле и предполагаемом исполнителе, а также информацию о расположении заголовков программ и заголовков секций.

- **Заголовки программ.** Служат для описания процесса выделения памяти до запуска программы. Может выставлять некоторые флаги, описывающие уровен доступа к тому или иному участку памяти.

- **Секции.** Служат для хранения произвольных данных, начиная от кода программы и таблицы символов, заканчивая инициализированными переменными. Каждая из секций может иметь какую-либо структуру или не имеет её вовсе. Это зависит от самой секции и прописано в документации. Порядок секций внутри ELF файла не задан. Есть как обязательные секции, без которых программа не запустится, так и опциональные.

- **Заголовки секций.** Задают расположение непосредственно секций внутри ELF файла, а также их тип, для того, чтобы не определять его по ходу.

Для данной работы необходимо работать с двумя видами секций и заголовком файла, поэтому опишем их подробнее.

- **Заголовок файла**
  Всегда находится в самом начале файла и имеет следующие поля, которые идут именно в перечисленном ниже порядке.

1. Первые 4 байта всегда хранят значение `7f 45 4c 46`, которые служат сигналом того, что данный файл следует считать ELF файлом.
2. Следующий байт хранит информацию о разрядности архитектуры, где значения 1 и 2 означают 32 и 64 битные архитектуры соответственно.
3. Следующий байт аналогично предыдущему может иметь значение либо 1 либо 2, означающие тип кодирования little endian и big endian соответственно.
4. Следующий байт содержит версию стандарта файла ELF, но на данный момент есть только одна версия. Поэтому значение этого байта всегда будет равно 1.
5. Следующие 2 байт указывают на тип ABI(Application Binary Interface) и его версию целевой операционной системы.
6. Следующие 7 байт зарезервированы под дальнейшее расширение стандарта и не используются в данный момент.
7. 2 байта по адресу `0x10` указывают на тип объектного файла ELF. Это поле необходимо, так как требования к наличию различных секций у исполняемого файла и, например, динамической библиотеки различные.
8. 2 байта по адресу `0x12` указывают архитектуру набора инструкций. RISC-V соответствует код `0xf3`.
9. 4 байта по адресу `0x14` указывает на версию ELF и равно 1 для актуальной версии ELF.
10. 4 байта по адресу `0x18` задаёт точку входа в программу, то есть адрес первой исполняемой инструкции.
11. 4 байта по адресу `0x1C` задаёт адрес начала таблицы заголовков программ.
12. 4 байта по адресу `0x20` задаёт адрес начала таблицы заголовков секций.
13. 4 байта по адресу `0x24` оставлены для использования в нуждах архитектуры и их содержимое зависит от неё.
14. 2 байта по адресу `0x28` хранят суммарный размера заголовка файла. В 32 битном случае он равен 52.
15. 2 байта по адресу `0x2A` хранят размер одной записи в таблице заголовков программ.
16. 2 байта по адресу `0x2C` хранят количество записей в таблицу заголовков программ.
17. 2 байта по адресу `0x2E` хранят размер одной записи в таблице заголовков секций.
18. 2 байта по адресу `0x30` хранят количество записей в таблицу заголовков секций.
19. 2 байта по адресу `0x32` хранят индекс секции с именами секций в таблице заголовков секций.

- **.text** Секция непосредственно хранящая код программы. В стандарте ELF ничего не декларируется о его структуре.
- **.symtab** Таблица с метками, необходимая для линковки, а именно для замены ссылок, оставленных на этапе компиляции. Представляет собой некоторое количество идущих подряд записей, каждая из которых имеет следующий вид:
    1. **st_name** — хранит индекс внутри таблицы **.strtab**, в которой по этому индексу написано имя данной метки.
    2. **st_value** — хранит адрес того объекта, на который указывает данная метка.
    3. **st_size** — хранит размер объекта, на который ссылается метка. Может быть равен 0, что значит что объект либо не имеет размера, либо его размер не известен.
    4. **st_info** — хранит тип объекта и связанные с ним атрибуты, на который указывает метка. Типичные объекты, на которые ссылается метка: структура данных, функция, файл или даже секция ELF файла. Атрибуты указывают на видимость данной метки при линковке текущего файла с другими. Тип и атрибуты специфицируется согласно стандарту ELF. Объект может не иметь ни типа ни аттрибутов.
    5. **st_other** — используется для указания области видимости данной метки.
    6. **st_shndx** — указывает секцию в которой используется данная ссылка. Помимо обычных значений может встретиться два следующих значения, определённых в стандарте ELF
        - `SHN_ABS(0xfff1)` — данная метка является глобальной.
        - `SHN_UNDEF(0)` — информация отсутствует.

Описание работы написанного кода

Результат работы написанной программы

Список источников

- Спецификация RISC-V
- Статья на википедии про адресацию в машинных языках
- Краткое описание структуры ELF файла
- Спецификация ELF файла

Листинг кода

Listing 1: **../src/elfheader.cpp**

```cpp
#include "elfheader.h"
#include "elfsectiontable.h"

#include <exception>

Header parse_header(std::istream &ss) {

    const unsigned int expected_magic = 0x464C457F;
    const unsigned char expected_arch = 1;
    const unsigned char expected_endian = 1;
    const unsigned char SysV_ABI_code = 0x00;
    const unsigned short RiscV_ISA_code = 0x00F3;
    const unsigned short header_expected_size = 52;

    Header header;

    ss.read((char*)&header, sizeof(header));
    uint32_t magic = *(uint32_t*)header.e_ident;
    if (magic != expected_magic) {
        throw std::invalid_argument("Invalid ELF magic bits");
    }
    if (header.e_ident[5] != expected_arch) {
        throw std::invalid_argument("Unsupported architecture. 32bit only");
    }
    if (header.e_ident[6] != expected_endian) {
        throw std::invalid_argument("Big endian is no supported");
    }
    if (header.e_ident[7] != SysV_ABI_code) {
        throw std::invalid_argument("Unsupported ABI. System V only");
    }
    if (header.e_machine != RiscV_ISA_code) {
        throw std::invalid_argument("Unsupported ISA. RISC-V only");
    }
    if (header.e_ehsize != header_expected_size) {
        throw std::invalid_argument("Illegal elf header size");
    }
    if (header.e_shentsize != sizeof(SectionTableEntry)) {
        throw std::invalid_argument("Unexpected section header table entry size");
    }
    return header;
};

```

Listing 2: **../src/elfsectiontable.cpp**

```cpp
#include "elfsectiontable.h"
#include "typedefs.h"

#include <vector>
#include <istream>

std::vector<SectionTableEntry> parse_section_table (
     std::istream &ss,
     Elf32_Addr addr,
     unsigned int entries
 ) {
    ss.seekg(addr);
    std::vector<SectionTableEntry> res(entries);
    for (SectionTableEntry& i : res) {
        ss.read((char*)&i, sizeof(SectionTableEntry));
    }
    return res;
}

std::string get_section_name (
        std::istream &ss,
        const SectionTableEntry& shstrEntry,
        const SectionTableEntry& section)
{
    ss.seekg(shstrEntry.sh_offset + section.sh_name);
    char res[MAX_NAME];
    ss.getline(res, MAX_NAME, '\0');
    return (std::string) res;
};
```

Listing 3: **../src/elfsymbtable.cpp**

```
#include "elfsymbtable.h"
#include "elfsectiontable.h"
#include "typedefs.h"

#include <istream>
#include <vector>

std::vector<SymTableEntry> parse_symtable(
        std::istream &ss,
        Elf32_Addr offset,
        unsigned int entries) {
    ss.seekg(offset);
    std::vector<SymTableEntry> res(entries);
    for (SymTableEntry &i : res) {
        ss.read((char*) &i, sizeof(SymTableEntry));
    }
    return res;
}

```

Listing 4: **../src/main.cpp**

```
#include "elfheader.h"
#include "elfsectiontable.h"
#include "elfsymbtable.h"
#include "typedefs.h"
#include "output.h"

#include <iostream>
#include <algorithm>
#include <fstream>
#include <vector>
#include <string>
#include <iterator>

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

    if (argc != 2) {
        std::cout << "Usage: rv3 <elf_input_file_name> <output_file_name>\n";
        return 1;
    }

    std::ifstream fin((std::string) argv[1], std::ios::binary);

    Header header = parse_header(fin);
    std::vector<SectionTableEntry> sectionTable =
        parse_section_table(fin, header.e_shoff, header.e_shnum);

    SectionTableEntry &shstrHeader = sectionTable[header.e_shstrndx];

    auto find_tab_by_name = [&] (std::string name) {
        return std::find_if(
            sectionTable.begin(),
            sectionTable.end(),
            [&] (SectionTableEntry& el) {
                return get_section_name(fin, shstrHeader, el) == name;
            }
        );
    };

    std::vector<SectionTableEntry>::iterator symtabHeader = find_tab_by_name(".symtab");

    std::vector<SectionTableEntry>::iterator strtabHeader = find_tab_by_name(".strtab");

    std::vector<SymTableEntry> symtab =
        parse_symtable(fin, symtabHeader->sh_offset,
                symtabHeader->sh_size / symtabHeader->sh_entsize);
```

```
    std::string symbols(strtabHeader->sh_size, 0);
    fin >> symbols;

    print_symtable(symtab, symbols);

    // --
    std::cout << header.e_shnum << ' ' << header.e_shstrndx << '\n';
    for (auto i : sectionTable) {
        std::cout << get_section_name(fin, shstrHeader, i) << ' ';
    }
    // --
    return 0;
}
```

Listing 5: **../src/output.cpp**

```
#include ”output.h”
#include ”elfsymbtable.h”
#include ”typedefs.h”

#include <stdexcept>
#include <vector>
#include <string>
#include <tuple>
#include <algorithm>
#include <exception>

std::string format_bind(unsigned char bind) {
    if (bind == STB_LOCAL)
        return ”LOCAL”;
    else if (bind == STB_GLOBAL)
        return ”GLOBAL”;
    else if (bind == STB_WEAK)
        return ”WEAK”;
    else if (bind == STB_LOPROC)
        return ”LOPROC”;
    else if (bind == STB_HIPROC)
        return ”HIPROC”;
    throw std::invalid_argument(”Undefined bind value”);
}

std::string format_type(unsigned char type) {
    if (type == STT_NOTYPE)
        return ”NOTYPE”;
    else if (type == STT_OBJECT)
        return ”OBJECT”;
    else if (type == STT_FUNC)
        return ”FUNC”;
    else if (type == STT_SECTION)
        return ”SECTION”;
    else if (type == STT_FILE)
        return ”FILE”;
    else if (type == STT_LOPROC)
        return ”LOPROC”;
    else if (type == STT_HIPROC)
        return ”HIPROC”;
    throw std::invalid_argument(”Undefined type value”);
}

std::tuple<std::string,std::string> format_info(const unsigned char& info) {
    unsigned char bind = (info>>4);
    unsigned char type = (info&0xf);
    return {format_bind(bind), format_type(type)};
}


std::string format_index(const Elf32_Half& idx) {
    if (idx == SHN_ABS)
        return ”ABS”;
    else if (idx == SHN_UNDEF)
        return ”UNDEF”;
    else
        return std::to_string(idx);
}

```

```
std::string format_vis(const unsigned char& st_other) {
    int vis = (st_other&0x7);
    if (vis == STV_DEFAULT)
        return "DEFAULT";
    else if (vis == STV_INTERNAL)
        return "INTERNAL";
    else if (vis == STV_HIDDEN)
        return "HIDDEN";
    else if (vis == STV_EXPORTED)
        return "EXPORTED";
    else if (vis == STV_SINGLETON)
        return "SINGLETON";
    else if (vis == STV_ELIMINATE)
        return "ELIMINATE";
    else if (vis == STV_NUM)
        return "NUM";
    throw std::invalid_argument("Undefined visibility value");
}

void print_symtable(const std::vector<SymTableEntry>& symtable, std::string symbols) {
    for (size_t i = 0; i < symtable.size(); ++i) {
        const SymTableEntry &el = symtable[i];
        auto [bind, type] = format_info(el.st_info);
        std::string index(format_index(el.st_shndx));
        std::string name(
                        symbols.begin() + el.st_name,
                        find(symbols.begin() + el.st_name, symbols.end(), '\0')
                        );
        std::string vis(format_vis(el.st_other));
        printf("[%4zu] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n",
               i, el.st_value, el.st_size, type.c_str(), bind.c_str(),
               vis.c_str(), index.c_str(), name.c_str());
    }
}

```

Listing 6: **../include/elfheader.h**

```
#ifndef ELFHEADER_GUARDS
#define ELFHEADER_GUARDS

#include "typedefs.h"

#include <istream>

struct Header {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
};

Header parse_header(std::istream &ss);

#endif
```

Listing 7: **../include/elfsectiontable.h**

```
#ifndef ELFSECTIONTABLE_GUARDS
#define ELFSECTIONTABLE_GUARDS

#include "typedefs.h"
```

```

#include <vector>
#include <istream>

struct SectionTableEntry {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
    Elf32_Off     sh_offset;
    Elf32_Word    sh_size;
    Elf32_Word    sh_link;
    Elf32_Word    sh_info;
    Elf32_Word    sh_addralign;
    Elf32_Word    sh_entsize;
};


std::vector<SectionTableEntry> parse_section_table
    (std::istream &ss, Elf32_Addr addr, unsigned int entries);

std::string get_section_name (
        std::istream &ss,
        const SectionTableEntry& shstrEntry,
        const SectionTableEntry& section);

#endif
```

Listing 8: **../include/elfsymbtable.h**

```
#ifndef ELFSYMBTABLE_GUARDS
#define ELFSYMBTABLE_GUARDS

#include "typedefs.h"

#include <vector>
#include <istream>

struct SymTableEntry {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;    /* bind, type: ELF_32_ST_... */
    unsigned char st_other;
    Elf32_Half    st_shndx;
};

std::vector<SymTableEntry> parse_symtable(
        std::istream&,
        Elf32_Addr offset,
        unsigned int entries);


#endif
```

Listing 9: **../include/main.h**

```
// Test Header
```

Listing 10: **../include/output.h**

```
#ifndef OUTPUT_GUARDS
#define OUTPUT_GUARDS

#include "typedefs.h"
#include "elfsymbtable.h"

#include <vector>
```

```
#include <string>

void print_symtable(const std::vector<SymTableEntry>&, std::string symbols);

#endif
```

Listing 11: **../include/typedefs.h**

```
#ifndef TYPEDEFS_GUARD
#define TYPEDEFS_GUARD

typedef unsigned int   Elf32_Addr;
typedef unsigned short Elf32_Half;
typedef unsigned int   Elf32_Off;
typedef int            Elf32_Sword;
typedef unsigned int   Elf32_Word;

const unsigned short EI_NIDENT = 16;

const unsigned int MAX_NAME = 256;

const unsigned short SHN_ABS = 0xfff1;

const unsigned short SHN_UNDEF = 0;

#define STB_LOCAL      0
#define STB_GLOBAL     1
#define STB_WEAK       2
#define STB_NUM        3

#define STB_LOPROC    13
#define STB_HIPROC    15

#define STT_NOTYPE     0
#define STT_OBJECT     1
#define STT_FUNC       2
#define STT_SECTION    3
#define STT_FILE       4
#define STT_COMMON     5
#define STT_TLS        6
#define STT_NUM        7

#define STT_LOPROC    13
#define STT_HIPROC    15


#define STV_DEFAULT    0
#define STV_INTERNAL   1
#define STV_HIDDEN     2
#define STV_PROTECTED  3
#define STV_EXPORTED   4
#define STV_SINGLETON  5
#define STV_ELIMINATE  6

#define STV_NUM        7


#endif
```