САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

«ISA. АССЕМБЛЕР. ДИЗАССЕМБЛЕР»

Выполнил: Белоус Данила Павлович

Номер ИСУ: 334927

студ. гр. M3139

Санкт-Петербург

2021

Цель работы: знакомство с архитектурой набора команд RISC-V.

Теоретическая часть

RISC-V — ISA (instruction set architecture, набор команд для общения с процессором), разработанный в 2010 году в университете Беркли, США. Его целями были открытость, бесплатность, удобство в академическом использовании и возможность реального применения.

RISC-V имеет модульную структуру: есть набор базовых команд (RV32I/64I), их количество очень ограничено — несколько арифметических команд (сложение, битовые сдвиги, and/or/xor, команды для работы с памятью и ещё системные команды). Дополнительные модули реализуют также распространённые команды, но уже не такие необходимые, как в базовом наборе — например, умножение и деление реализованы в модуле M, single-precision числа с плавающей точкой представлены в модуле F, команды для виртуализации подключаются с модулем H и т.д. Также есть набор сжатых команд, описанных в модуле C, в котором наиболее часто используемые команды закодированы меньшим количеством бит (например, не 32 битами, а 16) и которые позволяют уменьшить размер кода в среднем на 20-30%[1].

Как вообще работают команды? В итоговом файле, который процессор уже может брать и исполнять, записаны только нули и единицы. А ISA говорит, какие последовательности нулей и единиц каким командам соответствуют.

[1] Документация RISC-V, глава 16

В этой работе нам нужно было взять бинарный файл, достать оттуда секцию с кодом программы .text (структура файла будет описана ниже), и дизассемблировать эту бинарную последовательность обратно в команды, а также декодировать секцию .symtab, содержащую метки программы.

Как дизассемблировать набор бинарных команд? В основном смотреть в документации, какие команды во что переводятся. Чтобы понять, в каком формате эти команды выводить — также читаем документацию. Инструкции продуманы достаточно хорошо, чтобы не возникало неоднозначностей декодирования, но в первую очередь они разработаны для простоты восприятия этих команд процессором, а не для комфортного декодирования, поэтому зачастую приходится повозиться.

Все инструкции состоят из нескольких модулей, некоторые из которых могут отсутствовать: уникальный код команды (обязателен), регистр назначения rd (опционально), регистры-источники rs1, rs2 (их может быть от нуля до двух), число-константа immediate (опционально).

Наличие тех или иных частей зависит от типа команды. Например, команда ADD берёт значение из двух регистров и записывает в третий, поэтому в этой команде не будет содержаться константа. А команда ADDI прибавляет константу к значению одного регистра-источника и записывает в регистр-назначение, поэтому в этой команде будут части rs1, rd, immediate. То, в какой части команды что записано — описано в документации.

В качестве примера разберём процесс декодирования команды ADDI, которая добавляет константу immediate к значению, сохранённому в

регистре rs1, и записывает результат в регистр rd. Эта команда представлена в базовом наборе RV32I, поэтому она содержит 32 бита. Эти биты распределяются по блокам следующим образом:

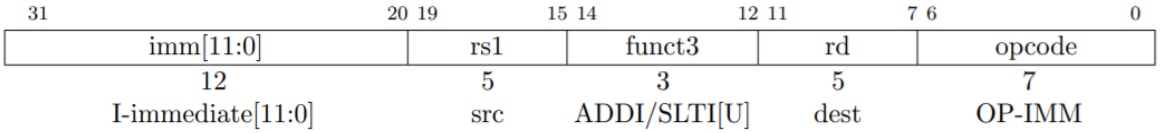

Рисунок 1 — Устройство команды ADDI.

Рассмотрим какую-то конкретную команду ADDI: 0xc3418513. В двоичной записи она будет выглядеть как $11000011010000011000010100010011_2$. opcode, то есть код операции, по которому мы определяем, что перед нами именно ADDI, а не какая-то другая команда, тоже можно взять из документации[2]. Он равен 0010011_2, что, не удивительно, совпадает с окончанием кода нашей команды. Далее 5 бит уходят под кодирование регистра — $01010_2 = 10_{10}$. Это значит, что регистр назначения. Ищем в таблице регистров (рисунок 2) имя регистра с нужным индексом (10) — получаем a0. Как можно заметить, у регистров есть два имени, но по заданию мы используем вариант ABI. Аналогично декодируем регистр-источник — gp. Для декодирования константы собираем её по частям из инструкции. Например, в данной команде с 20 по 31 биты записаны биты с 0 по 11 биты константы. Итого получается команда addi a0, gp, -972. Отрицательные числа получаются, потому что константы sign-extended (последний бит копируется во все значения старших битов в 32-битном типе).

[2] Документация RISC-V, глава 25

Register Name	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	t0	Temporary/Alternate Link Register
x6-7	t1-t2	Temporary Register
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-11	a0-a1	Function Argument/Return Value Registers
x12-17	a2-a7	Function Argument Registers
x18-27	s2-s11	Saved Registers
x28-31	t3-t6	Temporary Registers

Рисунок 2 — Описание регистров.

Поговорим о структуре .elf-файлов. ELF — Executable and Linkable Format — стандартный формат двоичных файлов для исполняемых файлов, объектных кодов и библиотек. Структура ELF файлов представлена на рисунке 3. Для того, чтобы декодировать секции .text и .symtab, нужно сначала декодировать заголовок файла и таблицу, содержащую информацию о секциях. Секция .text содержит исходный код программы, а секция .symtab — метки, используемые программой.

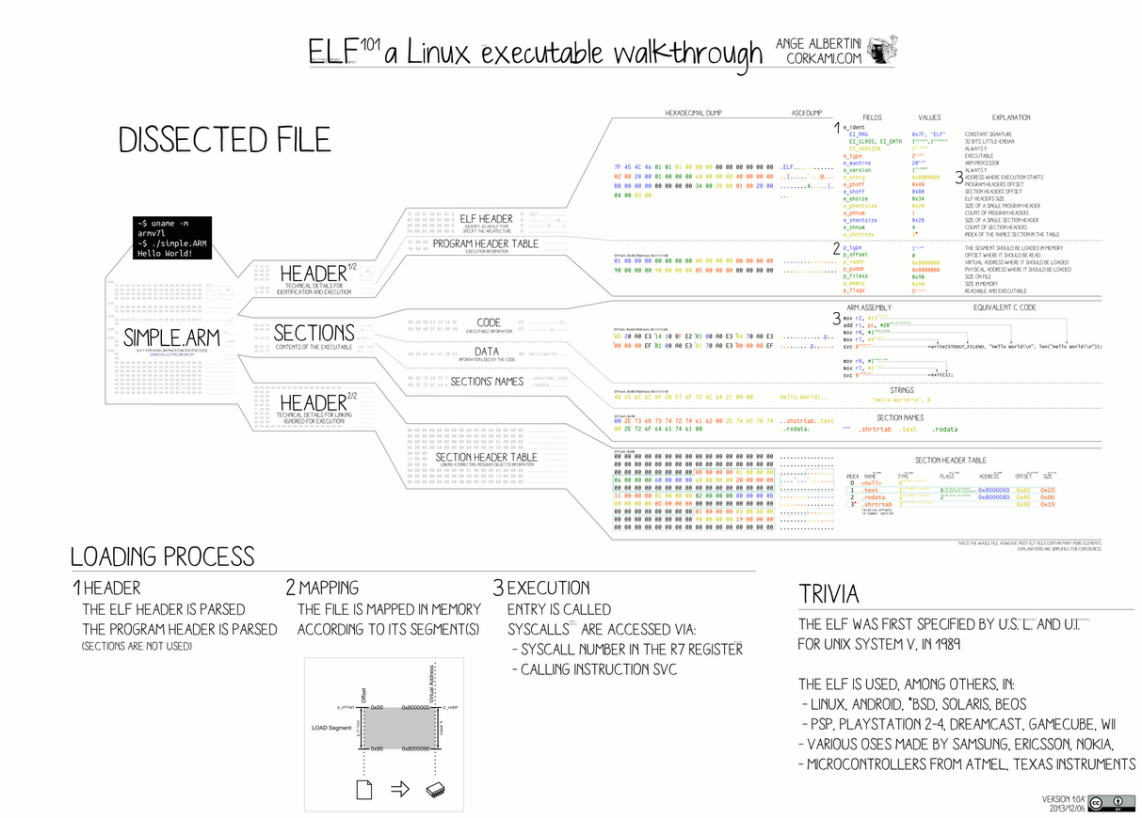

Рисунок 3 — Структура ELF-файла.

Практическая часть

Задача: декодировать .symtab и .text. Из .text нужно дизассемблировать команды из модулей RV32I, RV32M и RVC. Как декодировать и откуда брать информацию, описано выше. Здесь напишу какие-то замечания, которые касаются реализации. Для выполнения работы использовалась **Java версии 16.0.2**, но, вероятнее всего, для компиляции подойдёт какая-то более старая версия, например, Java 13.

Всю работу удобно было разделить на пакеты: elf для декодирования и структурных элементов elf файла, и riscv для дизассемблирования.

- Исходники проекта

~~По умолчанию используется форматирование то, которое было указано в ДЗ, но при использовании флагов оно отключается и выводится в том формате, в котором мне кажется более красивым.~~ В версии для гитхаба везде используется то форматирование, которое мне больше нравится.

При использовании команд типа jal соблюдается формат, указанный в документации, то есть последним аргументом выводится метка.

~~По умолчанию бросаются исключения о неизвестных командах, но при использовании в формате ДЗ4 исключения замалчиваются и вместо них в выводе появляются “unknown_command”.~~ Смотри README.md.

Здесь было дублирование README, листинг кода и пример работы программы на одном из файлов для тестирования, но они без толку занимают очень много места, поэтому в версии для гитхаба я убрал их.