Лабораторная работа №3

ISA

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

Инструментарий: работа должна быть выполнена на C, C++, Python или Java. В отчёте указываем язык и компилятор/интерпретатор, на котором вы работали.

Порядок выполнения и сдачи работы:

1. Изучить систему кодирования команд RISC-V.
2. Изучить структуру elf файла.
3. Написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.
4. Оформить отчет в формате pdf.
5. Загрузить файл отчета и файлы с исходным кодом на github в выданный вам репозиторий.

Содержание отчета

1. Минититульник (таблица с ФИО и названием работы из шаблона).
2. Цель работы и инструментарий.
3. Описание системы кодирования команд RISC-V.
4. Описание структуры файла ELF.
5. Описание работы написанного кода.
6. Результат работы написанной программы на приложенном к заданию файле (дизассемблер и таблицу символов).
7. Список источников.
8. Листинг кода.

Пояснения:

В *пункте 3* опишите что из себя представляет ISA RISC-V, что она описывает, какой это вид ISA, как в ней кодируются команды и регистры. Подробнее стоит остановится на тех наборах команд, которые вам выданы по заданию (про остальные можно написать буквально 1-2 предложения, больше не нужно).

Пункт 4. Общее описание структуры ELF файлов. Подробно стоит расписать то те секции, которые вам необходимо проанализировать в ходе выполнения работы. Про другие в отчёте не нужно расписывать.

Пункт 5. Вы самостоятельно парсите файл и декодируете инструкции. Использовать готовые решения по парсингу elf файлов нельзя (пример: elf.h).

Если вы пользовались какими-то источниками информациями (спецификация, статьи и пр.), то в пункте 7 нужно оставить ссылки на эти интернет-ресурсы.

В *пункт 8* вставить результат работы программы, описывать ничего не нужно. Файл лежит у вас в репозитории в .github/workflows/*

Задание

Необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Должен поддерживаться следующий набор команд: RISC-V RV32I, RV32M. Подробнее (volume 1): https://riscv.org/technical/specifications/

Кодирование: little endian.

Обрабатывать нужно только секции .text, .symtable.

Для каждой строки кода указывается её адрес в hex формате.

Обозначение меток нужно найти в Symbol Table (.symtab). Если же название метки там не найдено, то используется следующее обозначение: L%i, например, L2, L34. Нумерация начинается с 0. Для каждой метки перед названием указывается адрес (пример ниже).

Про L метки: Когда в коде кто-то захочет перейти на определённый адрес, у которого нет метки, то тогда ставим метку L%i.

Например, посмотрим в Readme. Там есть следующая строка:

```
   100fc:  fea794e3            bne  a5,a0,100e4 <mmul+0x38>
```

В bne задан offset на адрес, для которого явно не определена метка. Значит на 100e4 назначается метка (например, L0) и вашем коде дизасм может выглядеть одним из следующих случаев (приведена часть строки, первый вариант предпочтительнее):

```
bne a5, a0, 0x100e4 <L0>
bne a5, a0, L0
bne a5, a0, 0x100e4
```

Шаблон файла дизассемблера

Файл должен состоять из двух частей: .text и .symtab, отделенных друг от друга одной пустой строкой. Сначала идет .text, затем .symtab.

Ниже приведены комментарии (строки, начинающиеся с ;) и форматы оформления.

```
; формат строк указан по правилам printf (Си)
.text
; строки оформляются в следующем формате
; с меткой: "%08x   <%s>:\n", аргументы: адрес, метка
; без метки: аргументы: адрес, полный код инструкции, инструкция, аргументы
; инструкции с 3 аргументами: "   %05x:\t%08x\t%7s\t%s, %s, %s\n"
; инструкции с 2 аргументами: "   %05x:\t%08x\t%7s\t%s, %s\n"
; load/store, jalr инструкции:  "   %05x:\t%08x\t%7s\t%s, %s(%s)\n"
; immediate (константы): dec формат
; offset (в переходах J*, B*): hex формат
; примеры (отображение немного съехало, но в общем суть должна быть понятна):
00010074     <main>:
   10074:    00000013           addi zero, zero, 0
   10078:    00100137           lui sp, 256
   100c8:    fcf42e23           sw a5, -36(s0)

; между секциями text и symtab одна пустая строка
.symtab
; заголовок таблицы
; "Symbol Value           Size Type    Bind   Vis    Index Name\n"
; строки таблицы
; "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n"
; пример (отображение немного съехало, но в общем суть должна быть понятна):
Symbol Value            Size Type     Bind      Vis       Index Name
[   0] 0x0                 0 NOTYPE   LOCAL     DEFAULT   UNDEF
[   1] 0x10074             0 SECTION  LOCAL     DEFAULT       1
[   2] 0x112F8             0 SECTION  LOCAL     DEFAULT       2
[   3] 0x0                 0 SECTION  LOCAL     DEFAULT       3
[   4] 0x0                 0 SECTION  LOCAL     DEFAULT       4
[   5] 0x0                 0 FILE     LOCAL     DEFAULT     ABS test.c
[   6] 0x11AF8             0 NOTYPE   GLOBAL    DEFAULT     ABS __global_pointer$
```

Комментарии

Fence: примеры дизассемблирования fence можно найти **здесь**, но в рамках этой работы команды fence можно не обрабатывать (в тестах их также не будет).

Псевдонимы команд: псевдонимы команд парсить не нужно.

Вывод регистров: ABI.

Требования к работе программы

1. Аргументы программе передаются через командную строку:

`rv3 <имя_входного_elf_файла> <имя_выходного_файла>`

где rv3 – имя исполняемого файла (то есть это argv[0]).

2. Корректно выделяется и освобождается память, закрываются файлы, есть обработка ошибок: не удалось открыть файл, формат файла не поддерживается.
3. Если программе передано значение, которое не поддерживается – следует сообщить об ошибке.
4. В программе можно вызывать только стандартные библиотеки (например, <bits/stdc++.h> таковой не является и ее использование влечет за собой потерю баллов).
5. Если программа использует библиотеки, которые явно не указаны в файле с исходным кодом (например, <algorithm>), то за это также будут снижаться баллы.
6. Если во входном файле встречается команда, которая не распознается программой, то её следует выводить как unknown_instruction.

Оформление в отчёте

Исходный код

1. Никаких скринов кода – код в отчет добавляется только текстом на белом фоне.
2. Шрифт: `Consolas` (размер 10-14 на ваше усмотрение).
3. Выравнивание по левому краю.
4. Подсветка кода допустима. Текст должен быть читаемым (а не светло-серый текст, который без выделения на белом не разобрать).
5. В раздел Листинг код вставляется полностью в следующем виде:

<Название файла>

<Его содержимое>

Файлы исходных кодов разделяются новой строкой.

Например,

main.cpp

```
int main()
{
   return 0;
}
```

tmain.cpp

```
int tmain()
{
   return 666;
}
```

6. Фон белый (актуально для тех, у кого копипаста кода идет вместе с фоном темной темы из IDE).

Дизассемблер

1. Результат работы программы оформляется `Consolas` (размер 10-14 на ваше усмотрение).
2. Интервал: 1.0.
3. Выравнивание по левому краю.
4. Остальное зафиксировано в Шаблоне файла дизассемблера.

Github

В этой работе вам выданы исходный elf файл “test/test_elf”. Помимо этого там лежат файлы read_disams.txt и dump_disasm.txt, полученные в результате применения к test_elf `riscv64-unknown-elf-readelf` и `riscv64-unknown-elf-objdump` соответственно (подробнее Полезное). Там вы можете посмотреть на примерно то, что у вас должно получится (формат вывода у вас будет немного отличаться от этих). test.cc – исходный файл, из которого получен test_elf.

В этой работе у вас нет автотестов, поэтому вы должны сами следить, что залили все исходники на github и ваш код вообще работает.

Полезное

Полезность №1: входной файл следует открывать в двоичном режиме, иначе у вас будут проблемы при тестировании на Windows.

Также в этом разделе приведены программы, работа с которыми не обязательна для выполнения лабораторной, но может быть полезна для осознания лекционного материала и отладки лабораторной.

Кросс-компилятор RISC-V C и C++

Вы можете самостоятельно поэкспериментировать с использованием riscv-gnu-toolchain:

[Windows]:

- Prebuilt Windows Toolchain for RISC-V (10.1.0)

[Linux/MacOS]:

- Ubuntu – Package Search Results -- gcc-riscv64-linux-gnu
- GNU toolchain for RISC-V, including GCC (isa-spec 20191213)
- Prebuilt RISC-V GCC toolchains for x64 Linux.
- Releases · sifive/freedom-tools

Дизассемблер (без псевдоинструкций): `riscv64-unknown-elf-objdump --disassemble --target=elf32-littleriscv --architecture=riscv:rv32 --disassembler-options=no-aliases test.elf`

Дизассемблер (предыдущее + symtab): `riscv64-unknown-elf-objdump --disassemble --target=elf32-littleriscv --architecture=riscv:rv32 --disassembler-options=no-aliases --syms test.elf`

Таблица символов в формате из задания: `riscv64-unknown-elf-readelf --symbols --wide test.elf`

Компиляция: `riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O2 -x c test.s -o test.elf -static -lm -nostdlib`

Компиляция с сохранением “дизассемблера”: `riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O2 -S -x c test.s -o test.txt -static -lm -nostdlib`

Сохранить результат вывода на консоль в файл:

```
<command> > file.txt
```

Например сохранение сообщений компиляции в файл f.txt:

```
riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O2 -x c test.s -o test.elf -static -lm -nostdlib > f.txt
```

Симулятор RISC-V [Linux, MacOS]

michaeljclark/rv8: RISC-V simulator for x86-64 (github.com)

Для его работы необходимо установить riscv-gnu-toolchain.

Графический симулятор RISC-V

GitHub - mortbopet/Ripes: A graphical processor simulator and assembly editor for the RISC-V ISA

Визуальный симулятор архитектуры и редактор ассемблерного кода, созданный для RISC-V.

Вы можете загрузить туда код на C/C++ и при помощи gnu toolchain скопилировать его или сразу загрузить elf файл. Загруженный код можно исполнить. На выбор доступны разные конфигурации конвейера и кэша.

Скачать: https://github.com/mortbopet/Ripes/releases/tag/v2.2.5