

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Организация ЭВМ и систем»

Тема: Знакомство с рабочей средой эмулятора Ripes для работы с процессором RISC-V. Базовый ISA, система команд, состав регистров. Разработка и выполнение простой программы на ассемблере RISC-V.

Студент гр. 3382

Копасова К. А.

Преподаватель

Куршев Е. О.

Санкт-Петербург

2024

Цель работы: Изучить пример простейшей программы на ассемблере RISC-V, познакомиться с назначением основных регистров и изучить базовые арифметические операции.

Задание:

Напишите программу на ассемблере, которая вычисляет результат математического выражения в соответствии с вариантом. Убедитесь в корректности работы программы через автоматизированную систему.

Начальные данные на момент старта программы будут расположены в регистрах a2, a3, a4 соответственно. Результат выражения должен быть сохранен в регистр a0.

Весь код программы должен располагаться в метке solution. Программа должна заканчивать работу вызовом ret.

Шаблон программы для подготовки решения:

```
.globl solution
solution:
    # a0 = result
    ret
```

Ваше условие будет выведено ниже:

$$(a2 * (a4 \& a3)) \& (((-a2) | (a3 + (a3 + (a2 \& (-a2)))))$$

Ваш seed = 4745065824

Теоретический материал

1. Инструкция по работе с эмулятором Ripes:

Ripes - это графический симулятор процессора и редактор ассемблерного кода, созданный для архитектуры набора команд RISC-V. Он подходит для изучения того, как код на уровне ассемблера выполняется на различных микроархитектурах.

2. Описание состава используемых регистров и базового набора команд процессора RISC-V:

Регистры:

В RISC-V имеется 32 регистра общего назначения (x0-x31). Они используются для хранения данных и промежуточных результатов.

x0: регистр с фиксированным значением 0 (zero). Он не может быть изменен и используется для 0 в программах.

x1 - x31: используются для хранения данных и временных значений.

Более подробно про каждый регистр:

x1 (ra): регистр возврата (return addres).

x2 (sp): указатель стека (stack pointer).

x3 (gp): глобальный указатель (global pointer).

x4 (tp): указатель потока (thread pointer).

x5 - x7 (t0 - t2): временные регистры (temporary register).

x8 (s0/fp): сохраненный регистр/указатель кадра стека (saved register/frame pointer).

x9 - x15 (s1 - s7): сохраненные регистры (saved register).

x16 - x31 (a0 - a7): регистры аргументов и возврата значений.

Основной набор команд RISC-V:

Инструкции по передачи данных:

addi (add immediate) - добавление значения с непосредственным значением.

lui (load upper immediate) - загрузка верхних 20 битов непосредственного значения в регистр.

jal (jump and link) - переход и сохранение адреса возврата.

Инструкции арифметики и логики:

Add, sub, mul, div - инструкции сложения, вычитания, умножения и деления соответственно.

And, or, xor - логические инструкции И, ИЛИ и исключающее ИЛИ соответственно.

Sll, srl, sra - инструкции сдвига влево, вправо и арифметического сдвига вправо соответственно.

3. Краткие сведения по ассемблеру RISC-V:

Ассемблер RISC-V — это низкоуровневый язык программирования, предназначенный для работы с процессорами, использующими архитектуру RISC-V (Reduced Instruction Set Computing). Эта архитектура имеет открытый исходный код, что делает её доступной для разработки и исследования. Ассемблер RISC-V предоставляет прямой доступ к инструкциям и ресурсам процессора, позволяя эффективно управлять его возможностями.

1) Инструкции RISC-V:

RISC-V использует фиксированную длину инструкций в 32 бита (для базового набора команд), что облегчает их обработку процессором.

Архитектура поддерживает различные наборы инструкций для арифметических операций, управления потоком и работы с памятью.

2) Типы инструкций: В RISC-V ассемблере есть несколько типов инструкций, которые различаются по форматам и назначению:

а) R-type (для арифметических и логических операций):

Формат: opcode rd, rs1, rs2 — регистр rd получает результат операции между регистрами rs1 и rs2.

Пример: add x3, x1, x2 (сложение значений в регистрах x1 и x2, результат в x3).

б) I-type (для операций с немедленными значениями):

Формат: opcode rd, rs1, imm — регистр rd получает результат операции с немедленным значением imm и значением в регистре rs1.

Пример: addi x3, x1, 10 (сложение значения в регистре x1 с числом 10, результат в x3).

в) S-type (для инструкций работы с памятью):

Формат: opcode rs1, rs2, imm — сохраняет данные в память или загружает их в регистры.

Пример: sw x3, 0(x1) (сохранить значение из регистра x3 по адресу, определенному в x1).

г) B-type (для условных переходов):

Формат: opcode rs1, rs2, imm — условные переходы в зависимости от значений регистров.

Пример: beq x1, x2, label (если $x1 == x2$, перейти к метке label).

д) U-type (для загрузки и установки верхней части 32-битного значения):

Пример: lui x3, 0x12345 (загрузить в верхнюю часть регистра x3 значение 0x12345).

е) J-type (для безусловных переходов):

Пример: jal x1, label (переход к метке label и сохранение адреса возврата в x1).

3) Синтаксис: Ассемблерный код RISC-V состоит из инструкций, которые указывают процессору, какие операции выполнить. Каждая инструкция записывается на отдельной строке и состоит из:

а) ОPCODE (код операции), который определяет, какая операция будет выполнена.

б) Операнды: регистры, константы или метки.

4) Работа с памятью: В RISC-V имеется несколько инструкций для работы с памятью:

Загрузка: lw, lh, lb, lbu, lhu — для загрузки данных из памяти в регистр.

Сохранение: sw, sh, sb — для сохранения данных из регистра в память.

Управление потоком:

beq (branch if equal): условный переход, если два регистра равны.

bne (branch if not equal): условный переход, если два регистра не равны.

jal (jump and link): безусловный переход с сохранением адреса возврата.

jalr (jump and link register): безусловный переход с сохранением адреса возврата в регистре.

Ход работы

1. Выполним начальную настройку docker согласно инструкциям из раздела “Предварительная настройка среды и самостоятельная отладка решений”.

Проверка корректной установки docker:

```
ksenia@desktop-desktopovich:/mnt/d/ОргЭВМ/мои лаб работы$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
.
  (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Скачаем и запустим образ с помощью программы *docker pull*:

```
ksenia@desktop-desktopovich:/mnt/d/ОргЭВМ/мои лаб работы$ docker pull riscvcourse/workshop_risc-v:latest
```

Проверяем корректность скачивания образа с помощью команды *docker image ls*:

```
ksenia@desktop-desktopovich:/mnt/d/ОргЭВМ/мои лаб работы$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
welcome-to-docker	latest	c494bee906dd	About an hour ago	381MB
riscvcourse/workshop_risc-v	latest	c4cf7308ebd8	8 days ago	4.94GB
docker/welcome-to-docker	latest	eedaff45e3c7	12 months ago	29.5MB
hello-world	latest	305243c73457	19 months ago	24.4kB

Строка с *riscvcourse/workshop_risc-v* присутствует, значит все установлено правильно.

2. Создадим программу, которая правильно выполнит задание.

Seed = 4745065824.

Условие: $(a2 * (a4 \& a3)) \& ((-a2) | (a3 + (a3 + (a2 \& (-a2)))))$.

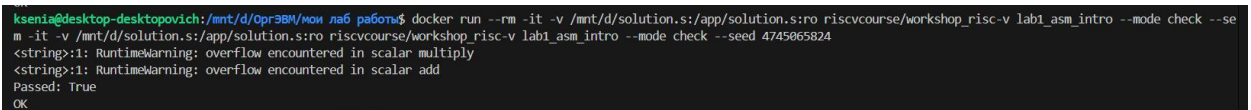
Упростим условие: $(a2 * (a4 \& a3)) \& ((-a2) | (2*a3 + (a2 \& (-a2))))$.

Создадим программу, которая вычисляет данное выражение и записывает результат в *a0*. Используем инструкции *add*, *sub*, *mul*, *sll*, *and* и *or*.

Полный код программы содержится в Приложении.

3. Проверим работоспособность программы через локально запущенный docker с помощью команды:

```
docker run --rm -it -v /mnt/d/solution.s:/app/solution.s:ro  
riscvcourse/workshop_risc-v lab1_asm_intro --mode check --sem -it -v  
/mnt/d/solution.s:/app/solution.s:ro riscvcourse/workshop_risc-v lab1_asm_intro -  
-mode check --seed 4745065824
```



```
ksenia@desktop-desktopovich:/mnt/d/Организация/лаб работы$ docker run --rm -it -v /mnt/d/solution.s:/app/solution.s:ro riscvcourse/workshop_risc-v lab1_asm_intro --mode check --se  
m -it -v /mnt/d/solution.s:/app/solution.s:ro riscvcourse/workshop_risc-v lab1_asm_intro --mode check --seed 4745065824  
<string>:1: RuntimeWarning: overflow encountered in scalar multiply  
<string>:1: RuntimeWarning: overflow encountered in scalar add  
Passed: True  
OK
```

Warning-и возникают из-за того, что в проверке написаны большие числа, которые приводят к переполнению регистров при инструкциях умножения и сложения. Но проверка пройдена верно, значит программа работает корректно.

Вывод

В ходе лабораторной работы удалось изучить пример простейшей программы на ассемблере RISC-V, познакомиться с назначением основных регистров, изучить базовые арифметические операции и создать программу, которая корректно выполняет задание в соответствии с вариантом.

Приложение

Файл solution.s:

```
.globl solution
solution:
    and a1, a4, a3    # a1 = a4 & a3
    mul a1, a1, a2    # a1 = a1 * a2 = a2 * (a4 & a3)
    sub t0, zero, a2  # t0 = -a2
    and t0, a2, t0    # t0 = a2 & t0 = a2 & (-a2)
    sll a3, a3, 1     # a3 = 2*a3
    add t0, t0, a3     # t0 = 2*a3 + (a2 & (-a2))
    sub a2, zero, a2  # a2 = -a2
    or t0, a2, t0     # t0 = a2 | t0 = (-a2) | (2*a3 + (a2 & (-a2)))
    and a0, t0, a1    # a0 = t0 & a1 = ((-a2) | (2*a3 + (a2 & (-a2)))) &
(a2 * (a4 & a3))

    ret
```