**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

**отчет**

**по лабораторной работе №5**

**по дисциплине «Организация ЭВМ и систем»**

**Тема: Изучение режимов адресации в ассемблере RISC-V.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Снигирёв А.А |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Молодцев Д.А |

Санкт-Петербург

2024

**Цель работы**

1. Разработка программы преобразования данных для приобретения практических навыков программирования на языке ассемблера.

2. Закрепление знаний по режимам адресации в процессоре RISC-V.

**Задание:**

Требуется написать программу, которая использует разные режимы адресации для вычислений по массиву данных в памяти. Результатом выполнения вашей программы будет измененный массив в памяти.

В качестве исходных данных дается стартовый адрес в памяти для хранения массива, количество элементов в массиве и формула для требуемых вычислений (Вычисления включают изменения каждого элемента массива в зависимости от условия).

При автоматической проверке вашей программы исходные данные располагаются в регистрах следующим образом:

- a1 - адрес памяти, где расположен массив

- a2 - количество элементов в массиве

Считайте, что массив уже инициализирован и заполнен данными.

Ваша программа должна иметь следующую структуру:

.globl solution

solution:

# при старте данной метки ваша программа должна выполнить

# необходимые вычисления и изменить элементы массива согласно ветке условия и формуле в ней

ret

Доступ к массиву (чтение, изменение) должен выполняться из памяти.

Формула для вычислений будет выведена ниже (arr[i] - элемент массива, считаем что arr[-1] == 0):

ЕСЛИ ((arr[7] ^ arr[6] - arr[7]) > 798)

ТО (arr[i] = arr[i - 1] | 49)

ИНАЧЕ (arr[i] = arr[i] - 4)

Моделируемые вычисления (формула, входные данные, результаты) должны выводиться в консоль.

**Основные теоретические положения:**

1. Описание состава используемых регистров, базового набора команд и набора псевдокоманд процессора RISC-V.

2. Краткие сведения по режимам адресации в ассемблере RISC-V.

*Регистровая адресация*

При регистровой адресации регистры используются для всех операндов-источников и операндов-назначений

*Непосредственная адресация*

При непосредственной адресации в качестве операндов наряду с регистрами используют константы (непосредственные операнды).

addi rd,rs1,12 # rd = rs1 + 12

andi rd,rs1,-8 # rd = rs1 & 0xFF8

Чтобы использовать константы большего размера, следует использовать инструкцию непосредственной записи в старшие разряды lui (load upper immediate), за которой следует инструкция непосредственного сложения addi Инструкция lui загружает 20-битное значение сразу в 20 старших битов и помещает нули в младшие биты:

lui s2, 0xABCDE # s2 = 0xABCDE000

addi s2, s2, 0x123 # s2 = 0xABCDE123

При использовании многоразрядных непосредственных операндов, если указанный в addi 12-битный непосредственный операнд отрицательный, старшая часть постоянного значения в lui должна быть увеличена на единицу.

*Базовая адресация*

Инструкции для доступа в память, такие как загрузка слова(чтение памяти) (lw) и сохранение слова(запись в память) (sw), используют базовую адресацию. Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре rs1 и 12-битного смещения с расширенным знаком, являющегося непосредственным операндом. Операции загрузки (lw) – это инструкции типа I, а операции сохранения (sw) – инструкции типа S.

lw rd, 36(rs1) # rd = M[rs1+imm][0:31]

Поле rs1 указывает на регистр, содержащий базовый адрес, а поле rd указывает на регистр-назначение. Поле imm, хранящее непосредственный операнд, содержит 12-битное смещение, равное 36. В результате регистр rd содержит значение из ячейки памяти rs1+36

sw rs2, 8(rs1) # M[rs1+imm][0:31] = rs2[0:31]

Инструкция сохранения слова sw демонстрирует запись значения из регистра rs2 в слово памяти, расположенное по адресу rs1+8

*Адресация относительно счетчика команд*

Инструкции условного перехода, или ветвления, используют адресацию относительно счетчика команд для определения нового значения счетчика команд в том случае, если нужно осуществить переход. Смещение со знаком прибавляется к счетчику команд (PC) для определения нового значения PC, поэтому тот адрес, куда будет осуществлен переход, называют адресом относительно счетчика команд.

Инструкции перехода по условию (beq, bne, blt, bge, bltu, bgeu) типа B и jal (переход и связывание) типа J используют для смещения 13- и 21-битные константы со знаком соответственно.

Инструкция jal может быть использована как для вызова функций, так и для простого безусловного перехода. В RISC-V используется соглашение, что адрес возврата должен быть сохранён в регистре адреса возврата ra ( x1).

Инструкция jal не имеет достаточного места для кодирования полного 32-битного адреса. Это означает, что вы не можете сделать переход куда-либо в коде, если ваша программа больше максимального значения смещения.

**Выполнение работы**

1. Исходное задание было приведено ранее.

2. Для красоты и наглядности были определены несколько строк и: написаны несколько функций для вывода стартового и обработанного массива.

3. Процедура *main.*  Загружает в регистры *a1, a2* адрес начала массива и его размер соответственно. Далее выводит вспомогательные сообщения вызовом соответствующих процедур. После этого вызывает процедуру *solution* для обработки и выводит измененный массив. Завершает выполнение.

4. Процедура *solution.* Загружает в регистры все нужные счетчики и значение *treshold*.Далее запускает цикл, в котором сначала высчитывается адрес текущего элемента массива, затем актуальное значение для условия. Поскольку массив с каждой итерацией изменяется, нужно каждый раз расчитывать это значение заново.   
Дальше в зависимости от значения происходит переход на метку *second* или работа с логикой метки *first.*

*First:*

Из регистра a3, в котором лежит текущее значение элемента, вычитает 4. Совершает переход на метку *update*.

*Second:*

На момент входа в метку в регистре t4 лежит значение предыдущего элемента. В регистр a3 сохраняется операция «логическое ИЛИ» содержимого t3 и числа 49. Переходит на *update*

*update:*

Командой *sw* загружает в массив содержимое регистра *a3*. Помещает в регистр *t4* значение *a3*, обновляя его. Увеличивает счетчик *t0*. Проверяет, что счетчик меньше размера массива и если это не так, переходит на метку *finish*, которая применяет *ret*. Иначе сдвигает указатель на текущий элемент на 4 байта и переходит на следующую итерацию. Исходный код программы см. в **ПРИЛОЖЕНИИ А .** Тестирование см. в **ПРИЛОЖЕНИИ Б**.

**Вывод**

В результате выполнения лабораторной работы были изучены способы адресации в ассемблере risc-v. Также были получены навыки в их использовании в ходе написания программы, выполняющей изменение массива в соответствии с условием.

**ПРИЛОЖЕНИЕ А**

**Код программы**

.data

array: .word 100,201,130,456,50,60,90,80,45,43 # Массив данных (10 элементов)

size: .word 10 # Размер массива

help1: .string "start array: " # Строка, которая будет выведена перед печатью массива

help2: .string "finish array: " # Строка, которая будет выведена после обработки массива

newline: .string "\n"

.text

.globl main

# Главная функция

main:

la a1, array # Загружаем адрес массива в a1 (начало массива)

lw a2, size # Загружаем размер массива в a2

la a0, help1 # Загружаем строку "start array: " в a0

call print\_string # Выводим строку с помощью print\_string

jal print\_arr # Вызываем функцию для печати массива

la a0, help2 # Загружаем строку "finish array: " в a0

call print\_string # Выводим строку с помощью print\_string

call solution # Вызываем функцию обработки массива (solution)

call print\_arr # Печатаем изменённый массив

li a7, 10 # Загружаем код системного вызова для завершения программы

ecall # Выполняем системный вызов (завершаем программу)

.globl solution

solution:

li t0, 0 # Инициализация счётчика индекса массива (t0 = 0)

li t4, 0 # Инициализация t4 (array[-1] = 0, начальное значение)

li t5, 0 # Инициализация смещения указателя в массиве (t5 = 0)

li s4, 798 # Инициализация порога (threshold = 798)

start:

add t6, a1, t5 # t6 = адрес array[i] (считаем адрес текущего элемента массива)

lw a3, 0(t6) # Загружаем значение array[i] в a3

lw s2, 24(a1) # Загружаем значение array[6] в s2 (24 байта от начала массива)

lw s3, 28(a1) # Загружаем значение array[7] в s3 (28 байт от начала массива)

sub s5, s2, s3 # s5 = array[6] - array[7]

xor s5, s3, s5 # s5 = array[7] ^ (array[6] - array[7]) (производим XOR)

bgt s5, s4, second # Если s5 > 798, переходим к ветке second (если условие истинно)

# array[i] = array[i] - 4

first:

li t1,4 # Загружаем значение 4 в t1

sub a3, a3, t1 # a3 = array[i] - 4 (уменьшаем текущее значение на 4)

j update # Переходим к обновлению индекса и смещения

# array[i] = array[i - 1]|49

second:

ori a3, t4, 49 # a3 = (array[i - 1]) | 49 (применяем операцию ИЛИ с 49 к предыдущему элементу массива)

update:

sw a3, 0(t6) # Сохраняем обновлённое значение в array[i]

mv t4, a3 # Обновляем t4 (сохраняем текущее значение array[i] для использования в следующей итерации)

addi t0, t0, 1 # Увеличиваем индекс i (t0 = t0 + 1)

bge t0, a2, finish # Если индекс i >= size, переходим к завершению программы (переход, если все элементы обработаны)

addi t5, t5, 4 # Увеличиваем смещение на 4 (для перехода к следующему элементу массива)

j start # Переходим к следующей итерации (обрабатываем следующий элемент массива)

finish:

ret # Завершаем выполнение функции solution и возвращаемся в main

# Процедура для вывода массива

print\_arr:

li t1, 0 # Индекс массива i = 0

print\_loop:

bge t1, a2, print\_end # Если i >= size, завершаем печать массива

# Загружаем arr[i] в t2

slli t3, t1, 2 # t3 = i \* 4 (смещение для доступа к элементу массива)

add t3, t3, a1 # t3 = адрес arr[i] (получаем адрес элемента массива)

lw t2, 0(t3) # t2 = arr[i] (загружаем значение элемента массива)

# Печатаем число

mv a0, t2 # Перемещаем значение в a0 (аргумент для системного вызова)

li a7, 1 # Системный вызов для вывода числа (код 1)

ecall # Выполняем системный вызов (печать числа)

# Печатаем пробел

li a0, 32 # Код символа пробела (ASCII 32)

li a7, 11 # Системный вызов для вывода символа (код 11 для вывода символа)

ecall # Выполняем системный вызов (печать пробела)

addi t1, t1, 1 # Увеличиваем индекс i (t1 = t1 + 1)

j print\_loop # Переходим к следующей итерации (печатаем следующий элемент массива)

print\_end:

# Печатаем новую строку

la a0, newline # Загружаем строку новой строки в a0

li a7, 4 # Системный вызов для вывода строки (код 4)

ecall # Выполняем системный вызов (печать новой строки)

ret # Возвращаемся из функции print\_arr

# Процедура для вывода строки

print\_string:

li a7, 4 # Системный вызов для вывода строки (код 4)

ecall # Выполняем системный вызов (печать строки)

ret # Возвращаемся из функции print\_string

# Процедура для вывода числа

print\_number:

li a7, 1 # Код системного вызова для печати числа (код 1)

ecall # Выполняем системный вызов (печать числа)

la a0, newline # Загружаем строку новой строки в a0

li a7, 4 # Системный вызов для вывода строки (код 4)

ecall # Выполняем системный вызов (печать новой строки)

ret # Возвращаемся из функции print\_number

**ПРИЛОЖЕНИЕ Б**

**ТЕСТИРОВАНИЕ**

|  |  |
| --- | --- |
| **in** | **out** |
| 100,201,130,456,50,60,90,80,45,43 | 96 197 126 452 46 56 86 76 41 39 |
| 100,201,130,456,50,60,900,80,45,43 | 49 49 49 49 49 49 49 76 41 39 |
| 1,2,3,4,5,6,7,8,9 | -3 -2 -1 0 1 2 3 4 5 |
| 1,2,3,4,5,6,810,8,9 | 49,49,49,49,49,49,49,4,5 |
| 10,10,10,10,10,10,10,10 | 6 6 6 6 6 6 6 6 |