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

ОТЧЕТ

по лабораторной работе №6

по дисциплине «Организация ЭВМ и систем»

Тема: Изучение режимов адресации в ассемблере RISC-V.

Студент гр. 3384 Рудаков А.Л.

Преподаватель Ковалев А.Д.

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.

Для заданного набора констант a = 20, b = 7, c = 9 сформировать массив array из 10 элементов, в котором

arr[0] = a+b+c

array[i+1] = arr[i] + a + b - c

2. Написать программу, которая с использованием 4 режимов адресации: регистрового, непосредственного, базового и относительного к счетчику команд реализует вычисление выражения, выбираемого из таблицы 1 в соответствии с номером студента в списке группы.

Вычисляемое выражение:

ЕСЛИ (arr[6] + arr[7] + arr[5] < threshold) ТО (res1 = arr[8] & arr[9]) ИНАЧЕ (res2 = arr[1] & c)

Основные теоретические положения.

При регистровой адресации регистры используются для всех операндов-источников и операндов-назначений (иными словами – для всех операндов и результата). Все инструкции типа R используют именно такой режим адресации.

```
add rd,rs1,rs2          # rd = rs1 + rs2
```

Непосредственная адресация

При непосредственной адресации в качестве операндов наряду с регистрами используют константы (непосредственные операнды). Этот режим адресации используют некоторые инструкции типа I, такие как сложение с 12-битной константой (addi) и логическая операция andi.

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. Помните, что знак addi расширяет 12-битное непосредственное значение, поэтому отрицательное непосредственное значение будет содержать все единицы в своих старших 20 битах. Поскольку в дополнительном коде все единицы означают число –1, добавление числа, у которого все разряды установлены в 1, к старшим разрядам непосредственного операнда приводит к вычитанию 1 из этого числа. Пример иллюстрирует ситуацию, когда мы хотим в s2 получить постоянное значение 0xFEEDA987:

lui s2, 0xFEEDB # s2 = 0xFEEDB000 (число, которое нужно записать в старшие 20 разрядов (0xFEEDA), предварительно увеличено на 1)

addi s2, s2, −1657 # s2 = 0xFEEDA987 (0x987 – это 12-битное представление числа -1657)(0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987)

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. Самые старшие значимые биты смещения располагаются в 12- и 20-битных полях инструкций типа B и J. Наименьший значащий бит смещения всегда равен 0, поэтому он отсутствует в инструкции.

```
beq  rs1,rs2,imm    # if(rs1 == rs2) PC += imm
jal      rd,imm             #rd = PC+4; PC += imm
```

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. Но если адрес перехода хранится в регистре, вы можете сделать переход на любой адрес (инструкция jalr типа I).

jalr rd, imm (rs1) # rd = PC + 4, PC = rs1 + imm

Большая разница состоит в том, что переход JALR не происходит относительно PC. Вместо этого он происходит относительно rs1

Инструкция auipc типа U (сложить старшие разряды константы смещения с PC) также использует адресацию относительно счетчика команд.

auipc rd,imm # rd = PC + (imm << 12)

auipc s3, 0xABCDE # s3 = PC + 0xABCDE000

Выполнение работы.

В начале программы описан сегмент *.data*, в котором хранятся данные *threshold* и пустой массив *array.*

Далее в сегменте *.text* начинается работа в процедуре *main*. Вначале инициализируются регистры под массив и значения *t0, t1, t2, t3, t4* данными *array, 0, 20, 7, 9* соответственно. После этого в регистре *t5* рассчитывается *array[0]* и сохраняется при помощи *sw* и переходит к циклу.

Цикл выполняется пока, итератор меньше *10*. Внутри происходит загрузка предыдущего значения массива в *t5*, далее вычисляется текущее значение и сохраняется при помощи *sw*.

После этого идет сравнение *array[5]+array[6]+array[7]* с *threshold*, если сумма меньше, то вычисляется *res2* и записывается в регистр *a3*. В противном случае вычисляется *res1* и записывается в регистр *t6*.

После этого программа завершается.

Вывод.

В ходе выполнения лабораторной работы была разработана программа, выполняющая действия с разными режимами адресации на ассемблере RISC-V.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ.

```
      .data
        threshold: .word 10
        array: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 # Инициализируем массив
нулями

      .text
      .globl main

      main:
        # Инициализация массива
        la t0, array
        li t1, 0
        li t2, 20 # a
        li t3, 7 # b
        li t4, 9 # c

        # arr[0] = a + b + c
        add t5, t2, t3 # a + b
        add t5, t5, t4 # a + b + c
        sw t5, 0(t0)

        # Заполнение оставшихся элементов массива
        li t1, 1 # i = 1
        li s0, 10 # s0 = 10 (для цикла)

        # Загрузка адреса array[1]
        la t6, array

        # Цикл по массиву
        loop:
          beq t1, s0, end_loop
```

```
          # Загружаем значение array[i-1]
          lw t5, 0(t6)

          # Вычисление array[i-1] + a + b - c
          add t5, t5, t2 # array[i-1] + a
          add t5, t5, t3 # array[i-1] + a + b
          sub t5, t5, t4 # array[i-1] + a + b - c

          # Сохраняем результат в массив
          addi t6, t6, 4
          sw t5, 0(t6)


          addi t1, t1, 1 # i++
          j loop
        end_loop:

        # Загрузка порога
        lw s1, threshold

        # Вычисление выражения
        la t0, array # t0 = base address of array

        li t1, 24 # offset = 6
        add t1, t0, t1 # t1 = address of arr[6]
        lw t2, 0(t1) # t2 = arr[6]

        li t1, 28 # offset = 7
        add t1, t0, t1 # t1 = address of arr[7]
        lw t3, 0(t1) # t3 = arr[7]

        li t1, 20 # offset = 5
        add t1, t0, t1 # t1 = address of arr[5]
        lw t4, 0(t1) # t4 = arr[5]

        # Сложение элементов
        add t5, t2, t3 # t5 = arr[6] + arr[7]
        add t5, t5, t4 # t5 = arr[6] + arr[7] + arr[5]

        # Сравнение с порогом
        blt t5, s1, if_condition_true # ЕСЛИ (arr[6] + arr[7] + arr[5] <
threshold)
        # ИНАЧЕ
        li t1, 4 # offset = 1
        add t1, t0, t1 # t1 = address of arr[1]
        lw t6, 0(t1) # t6 = arr[1]

        lw t5, c # t5 = c
```

```
  # Логическое И
  and a3, t6, t5 # res2 = arr[1] & c
  j end_if

if_condition_true:
  li t1, 32 # offset = 8
  add t0, t0, t1 # t0 = address of arr[8]
  lw t6, 0(t0) # t6 = arr[8]

  li t1, 36 # offset = 9
  add t0, t0, t1 # t0 = address of arr[9]
  lw t5, 0(t0) # t5 = arr[9]

  # Логическое И
  and t6, t6, t5 # res1 = arr[8] & arr[9]

end_if:

  # Остановка программы
  li a7, 10
  ecall
```