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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Организация ЭВМ и систем»
Тема: Изучение режимов адресации в ассемблере RISC-V

Студент гр. 2384 _____ Кузьминых Е.М.

Преподаватель _____ Морозов С.М.

Санкт-Петербург
2023

Цель работы

Разработка программы преобразования данных для приобретения практических навыков программирования на языке ассемблера. Закрепление знаний по режимам адресации в процессоре RISC-V.

Задание

1. Для заданного набора констант *a, b, c* сформировать массив *array* из 10 элементов, в котором первый элемент – сумма a,b,c, остальные – сумма предыдущего элемента, a и b с вычетом c

   Доступ к массиву (инициализация, запись, чтение) должен выполняться из памяти.
2. Написать программу, которая с использованием 4 режимов адресации: регистрового, непосредственного, базового и относительного к счётчику команд реализует вычисление выражения, выбираемого из таблицы 1 в соответствии с номером студента в списке группы.

14	ЕСЛИ (arr[9] + arr[3] + arr[2] > threshold) ТО (res1 = arr[0] - arr[1]) ИНАЧЕ (res2 = arr[2] - c)	threshold -> a1 res1 -> t6 res2 -> s4

Основные теоретические положения

Регистровая адресация

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.

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

Программа выполняет следующие действия:

1. Инициализирует некоторые константы и строки.
2. Заполняет массив.
3. Выводит массив.
4. Вычисляет два значения в зависимости от условия.
5. Выводит результаты.

В начале программы инициализируются некоторые константы и строки. Это делается с помощью директив .equ и .string. Затем программа заполняет массив. Это делается с помощью функции fill_array. В этой функции используются регистры a0, t0, t1, s0, a, b и c. После заполнения массива программа выводит его. Это делается с помощью функции print_array. В этой функции используются регистры a0, t0, t1, a1. После вывода массива программа вычисляет два значения в зависимости от условия. Это делается с помощью функции calc. В этой функции используются регистры a0, s1, s2, s4, t6, t1, a1. Наконец, программа выводит результаты. Это делается с помощью вызовов функций ecall с различными значениями регистра a7 и регистра a0.

Результирующее значение вычисляется в функции calc, она выполняет две основные задачи в зависимости от условия. Это условие проверяет, является ли сумма трех элементов массива больше порога. Если условие выполняется, функция вычисляет значение res1 как разницу между первым и вторым элементами массива. В противном случае она вычисляет значение res2 как разницу между третьим элементом массива и константой c.

Функция начинается с инициализации двух регистров t6 и s4 нулями. Эти регистры будут использоваться для хранения результатов res1 и res2 соответственно.

Затем функция загружает значения из трех элементов массива в регистры s1 и s2. Значения arr[9], arr[3] и arr[2] складываются и результат сохраняется в регистре s1.

После этого функция проверяет, больше ли сумма s1 порога a1. Если это условие выполняется, функция переходит к метке res1. В противном случае функция продолжает выполнение следующих инструкций.

Если условие не выполняется, функция загружает значение arr[2] в регистр s4 и вычитает из него константу c. Результат сохраняется в регистре s4, который будет использован как res2.

Если условие выполняется, функция переходит к метке res1. Здесь функция загружает значения arr[0] и arr[1] в регистры t1 и s2 соответственно. Затем значение s2 делается отрицательным и добавляется к t1. Результат сохраняется в регистре t6, который будет использован как res1.

После выполнения всех инструкций функция возвращает управление вызывающей программе.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	threshold = 500	0, 88	ИНАЧЕ
2.	threshold = 0	- 28, 0	ТО
3.	threshold = -10	- 28, 0	ТО
4.	threshold = 600	0, 88	ИНАЧЕ

Выводы

Была разработана программа, вычисляющая значения для элементов массива в зависимости от констант и получающая результирующие ответы в зависимости от значений массива. Были закреплены знания по режимам адресации в процессоре RISC-V.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb6.s

```
.equ a 23 # 2+3+8+4+1+5
.equ b 9 #длина "Кузьминых"
.equ c 4 # длина "Егор"
.equ threshold 500 # arr[9] + arr[3] + arr[2] == 500
.data

const_values: .string "a = 23, b = 9, c = 4, threshold = "
result: .string "r1,r2 = "
array_values: .string "array: "
separator: .string ", "
endl: .string ".\n"
array: .word 0,0,0,0,0,0,0,0,0,0

.text
j start

print_separator:
  li a7, 4
  la a0, separator
  ecall
  ret

start:
  li a7, 4
  la a0, const_values #печать констант
  ecall

  li a7, 1
  li a0, threshold
```

```
ecall

li a7, 4
la a0, endl
ecall

la a0, array
li a1, 10

call fill_array  # заполение массива
call print_array # печать

la a0, array
li  a1, threshold
call calc #функция для результирующего значения

# печать
li a7, 4
la a0, result
ecall

li a7, 1
mv a0, t6 #res1
ecall
call print_separator

li a7, 1
mv a0, s4 #res2
ecall
li a7, 4
la a0, endl
ecall
```

```
  li a7, 10
  ecall

fill_array:  # заполняем массив
  mv t0, a0
  li t1, 1

  #arr[0] = a + b + c - для первого
  addi s0, s0, a
  addi s0, s0, b
  addi s0, s0, c
  sw s0, 0(t0)
  addi t0, t0, 4

  fill_continuation:
    #array[i+1] = arr[i] + a+ b -c - для оставшихся

    addi s0, s0, a
    addi s0, s0, b
    addi s0, s0, -c

    sw s0, 0(t0)

    addi t1, t1, 1
    addi t0, t0, 4
    blt t1, a1, fill_continuation
  ret

print_array:

  mv t0, a0
  li t1, 0

  li a7, 4
```

```
  la a0, array_values
  ecall

  print_continuation:
    li a7, 1
    lw a0, 0(t0)
    ecall

    addi t1, t1, 1
    addi t0, t0, 4

    beq a1, t1, skip

    li a7, 4
    la a0, separator
    ecall

  skip:
    blt t1, a1, print_continuation

  li a7, 4
  la a0, endl
  ecall
  ret

# если ЕСЛИ (arr[9] + arr[3] + arr[2] > threshold) ТО (res1 = arr[0]
- arr[1]) ИНАЧЕ (res2 = arr[2] - c)



calc:



  mv t6, zero #res1
  mv s4, zero # res2
```

```
lw s1, 36(a0) # s1 = arr[9]
lw s2, 12(a0) # s2 = arr[3]
add s1, s1, s2 # (arr[9]+arr[3])
lw s2, 8(a0) # s2 = arr[2]
add s1, s1, s2 # arr[9] + arr[3] + arr[2]

  bgt s1, a1, res1

#иначе res2 = arr[2] - c
lw s4, 8(a0) # s4 = arr[2]
addi s4, s4, -c # t0 = a4 - c
j endif

res1:  #res1 = arr[0] - arr[1]
      lw t1, 0(a0) # t1 = arr[0]
      lw s2, 4(a0) # s2 = arr[1]
      neg s2, s2 #arr[1] = -arr[1]
      add t6, t1, s2 # (arr[0]-arr[1])

endif:
  ret
```