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

Студентка гр. 2384 _____ Валеева А.А.

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

$$array[0] = a + b + c$$

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

   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.

4	ЕСЛИ (arr[8] + arr[5] + arr[7] != threshold) ТО (res1 = arr[7] | arr[4]) ИНАЧЕ (res2 = arr[8] - b)	threshold -> s6 res1 -> a3 res2 -> t0

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.

```
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 определяются константы *a*, *b*, *c*, *threshold* и задаётся массив из 10 чисел 32-битной разрядности, заполненный нулями, строки для вывода константных и результирующих значений, разделитель и окончание строки. Программа выводит начальные значения.

1. Создана процедура *fill_array*, которая заполняет массив с помощью заданных функций, в регистре *a0* указываем адрес начала массива, в регистр *a1* заносим количество элементов. На каждой итерации проверяем текущее количество элементов в массиве, если оно меньше заданного в регистре *a1,* то продолжаем заполнять. Проверка выполняется через инструкцию перехода *blt* (перейти, если меньше) и дополнительную инструкцию *fill_continuation*.
2. Процедура *print_array* выводит массив в консоль, разделяя элементы сепаратором. В регистре *a0* должен быть указан адрес массива, в регистре *a1* количество элементов массива.
3. Процедура *calc* вычисляет значения в соответствии с условием. Процедура использует базовую и регистровую адресацию, а также адресацию относительно счётчика команд. В регистр *s6* нужно загрузить значение порога *threshold*. В регистре *a0* должен храниться адрес массива. Процедура помещает в регистр a3 значение *res1*, а в t0 значение *res2*.

После подсчёта значений на экран выводится *res1* и *res2* через

сепаратор. Разработанный программный код см. в приложении А.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	threshold = 582	0, 221	ИНАЧЕ
2.	threshold = 0	202, 0	ТО
3.	threshold = -100	202, 0	ТО
4.	threshold = 99999	202, 0	ТО

Выводы

Была разработана программа, преобразующая данные. Были приобретены практические навыки программирования на языке ассемблера. Закреплены знания по режимам адресации в процессоре RISC-V.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lr_6.s

```
.equ a 22 # 2+3+8+4+0+5
.equ b 7 #strlen("Валеева")
.equ c 5 #strlen("Алина")
.equ threshold 582 # arr[8] + arr[5] + arr[7] == 582
.data


const_values: .string "My const values: a = 22, b = 7, c = 5,
threshold = "
result: .string "Answer: {r1, r2} = "
array_values: .string "This is my 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 #здесь храниться адрес на массив
li a1, 10 #а здесь кол-во элементов
call fill_array # заполним массив числами по формуле
call print_array # и напечатаем его
la a0, array
li s6, threshold
call calc #функция дл вычисления результата
```

```
# печатаем результат
li a7, 4
la a0, result
ecall
li a7, 1
mv a0, a3 #res1
ecall
call print_separator


li a7, 1
mv a0, t0 #res2
ecall
li a7, 4
la a0, endl
ecall
# выход из программы
li a7, 10
ecall


fill_array: # функция заполнения массива
mv t0, a0 # начало
li t1, 1 #кол-во


#arr[0] = a + b + c
li s0, a
addi s0, s0, b
addi s0, s0, c
sw s0, 0(t0) # записываем в t0 (массив)
addi t0, t0, 4 # смещаемся


fill_continuation:
#array[i+1] = arr[i] + a+ b -c
addi s0, s0, a
addi s0, s0, b
addi s0, s0, -c
sw s0, 0(t0) # записываем в t0
addi t1, t1, 1 # кол-во
addi t0, t0, 4 # смещаем указатель
blt t1, a1, fill_continuation # если t1<a1 (пока не достигли нужного
кол-во эл-ов) -> 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 (arr[8] + arr[5] + arr[7] != threshold), то (res1 = arr[7] |
arr[4]), иначе (res2 = arr[8] - b)
# a0: адрес массива
# s6: threshold
# a3: res1
# t0: res2
calc:


# 0 в регистрах с ответами
mv a3, zero #res1
mv t0, zero # res2


lw s1, 32(a0) # s1 = arr[8]; 8*4
lw s2, 20(a0) # s2 = arr[5] ;5*4
add s1, s1, s2 # s1 += s2 (8+5)
lw s2, 28(a0) # s2 = arr[7] ;7*4
add s1, s1, s2 # s1 += s2 ; arr[8] + arr[5] + arr[7]


bne s1, s6, res1 # если s1 != s6 то res1


#иначе (res2 = arr[8] - b)
lw t0, 32(a0) # a4 = arr[8]
addi t0, t0, -c # t0 = a4 - c
j endif
res1: #res1 = arr[7] | arr[4]
```

```
lw t1, 16(a0) # t1 = arr[4]
lw s2, 28(a0) # s2 = arr[7]
or a3, t1, s2 # a3 = t1|s2
endif:
ret
```