Глава 2. Загрузка и сложение

Цель

Научиться использовать арифметические, логические и управляющие инструкции языка ассемблера RISC-V. Закрепить понимание представления знаковых и беззнаковых чисел, битовых сдвигов, условных переходов и реализации вычислительных логик с промежуточными результатами.

Теоретическая часть

Во второй главе рассматриваются ключевые арифметические инструкции (add, sub, mul, div, rem), логические операции (and, or, xor), а также сдвиги (slli, srai) и условные переходы (beq, bne, blt, bge).

Важно правильно инициализировать регистры, обрабатывать отрицательные числа через дополнительный код (например, neg), а также использовать безусловный переход jal и инструкции загрузки (li, lui, mv).

Битовые сдвиги позволяют выполнять быстрое умножение или деление на числа, равные степени двойки, условные переходы управляют логикой вычислений, а команды rem и div обеспечивают работу с остатком и целой частью от деления.

Задание лабораторной работы

Напишите программу, которая выполняет следующие действия:

1. **Загрузка и вычисление**

   Используя инструкции la и lw, загрузите x и y в регистры t0 и t1, вычислите сумму, умножьте её на 2, прибавьте 4 и поместите итог в регистр a0.

2. **Вывод результата на экран**

   Чаще всего будет значительно проще выполнять отладку программы, используя вывод арифметических операций в консоль. Вы всё ещё можете проверять результат программы непосредственно в регистрах, однако будет полезно попробовать и данный способ вывода информации. Для вывода числа в консоль необходимо разбить его на разряды и перевести их в формат ascii.

Опираясь на код ниже, попробуйте выполнить вывод результата вашей программы в консоль.

```
        # Пример: вывод числа в консоль
    .data
msg:    .ascii  "Result: "    # префикс для более аккуратного
вывода
num:    .word   64            # пример числа
buf:    .space  2             # буфер под две ASCII-цифры
nl:     .ascii  "\n"          # перевод строки

    .text
    .globl _start
    .align 2
_start:
    # -- загрузка числа
    la    t0, num
    lw    t0, 0(t0)           # t0 ← 64

    # -- разбиение числа на разряды
    li    t1, 10
    div   t2, t0, t1          # t2 = 64 / 10 = 6
    rem   t3, t0, t1          # t3 = 64 % 10 = 4

    # -- преобразование в ASCII
    addi  t2, t2, '0'         # '6'
    addi  t3, t3, '0'         # '4'

    # -- посимвольная запись в буфер buf[0..1]
    la    t4, buf
    sb    t2, 0(t4)           # buf[0] = '6'
    sb    t3, 1(t4)           # buf[1] = '4'
```

```
    # -- вывод "Result: "
    li    a0, 1             # stdout
    la    a1, msg
    li    a2, 8             # длина строки "Result: " составляет
8 символов
    li    a7, 64            # sys_write
    ecall

    # -- вывод всех элементов буфера
    li    a0, 1
    mv    a1, t4            # адрес buf
    li    a2, 2
    li    a7, 64
    ecall

    # -- перевод строки
    li    a0, 1
    la    a1, nl
    li    a2, 1
    li    a7, 64
    ecall

    # -- завершение
    li    a0, 0
    li    a7, 93            # sys_exit
    ecall
```

Пример реализации

```
        .data
x:      .word 10            # данные: x = 10
y:      .word 20            #         y = 20
msg:    .ascii "Result: "   # резервируем строку "result" для
вывода в консоль
buf:    .space 2            # буфер под две ASCII-цифры
результата
nl:     .ascii "\n"         # перевод строки

    .text
    .globl _start
    .align 2
_start:
    # 1.1 Загрузка x и y из памяти
    la    t0, x             # t0 ← адрес x
    lw    t0, 0(t0)         # t0 ← x
    la    t1, y             # t1 ← адрес y
    lw    t1, 0(t1)         # t1 ← y

    # 1.2 Вычисление выражения E = (x + y) * 2 + 4
    add   t2, t0, t1        # t2 = x + y
    slli  t2, t2, 1         # t2 = t2 * 2
    addi  t2, t2, 4         # t2 = t2 + 4

    # Разбиваем результат на десятки и единицы (допускаем только
E меньше 100)
    li    t3, 10            # делитель = 10
    div   t4, t2, t3        # t4 = t2 / 10  → цифра в разряде
десятков
    rem   t5, t2, t3        # t5 = t2 % 10  → цифра в разряде
единиц
```

```
    addi  t4, t4, '0'       # переводим цифры в ASCII
    addi  t5, t5, '0'

    # Записываем их в буфер buf[0..1]
    la    t6, buf           # t6 ← адрес buf
    addi  t6, t6, 2         # t6 ← buf + 2 (конец буфера)
    addi  t6, t6, -1        # t6 ← buf + 1
    sb    t5, 0(t6)         # buf[1] = разряд единиц
    addi  t6, t6, -1        # t6 ← buf + 0
    sb    t4, 0(t6)         # buf[0] = разряд десятков

    # Пояснения:
    #  - для вывода строк и буферов используем sys_write (номер
64)
    #  - в a0 -- дескриптор потока (1 = stdout)
    #  - в a1 -- адрес данных, в a2 -- длина, в a7 -- номер
системного вызова
    #  - инструкция ecall запускает системный вызов

    # 3.1 Вывод префикса "Result: ", хранимого в переменной msg
    li    a0, 1             # a0 = 1 → stdout
    la    a1, msg           # a1 = адрес строки "Result: "
    li    a2, 8             # a2 = 8 байт
    li    a7, 64            # a7 = sys_write
    ecall                   # печать префикса

    # 3.2 Вывод цифр результата из buf
    li    a0, 1
    mv    a1, t6            # t6 сейчас = buf
    li    a2, 2             # длина = 2 байта
    li    a7, 64
    ecall                   # печать цифр
```

```
        # 3.3 Перевод строки
        li    a0, 1
        la    a1, nl
        li    a2, 1
        li    a7, 64
        ecall                   # печать "\n"

        li    a0, 0             # код возврата (0 - код успеха)
        li    a7, 93            # sys_exit
        ecall                   # выход
```

Методы оценки

Пункт	Баллы
Вычисление выражения из условия и сохранение результата	8
Перевод результата в ASCII и вывод в консоль	2

Теоретические вопросы

1. Почему в RISC-V для доступа к данным из памяти всегда используют пару la + lw, а не одну инструкцию?

RISC-V—это «load/store» архитектура: все арифметические и логические операции работают только над регистрами, а инструкции загрузки (lw, lb) и сохранения (sw, sb) — только с памятью.

2. Как представляются отрицательные числа в RISC-V и почему при загрузке знакового 32-бита важно знать о «двоичном дополнительном коде» (two's complement)?

В RISC-V (как и в большинстве современных архитектур) целые хранятся в формате two's complement. В этом формате бит-знака — старший бит слова — автоматически расширяется при арифметике и загрузках со знаковым расширением. Отрицательное число –N–N–N кодируется как «инверсия битов числа NNN плюс единица». Это позволяет тем же инструкциям add, sub корректно вычислять со знаковыми значениями.

5. Как умножить число на 2 с помощью логического сдвига?

Для этого используется логический сдвиг влево на единицу:

```
slli rd, rs, 1   # rd = rs << 1
```