Постановка задачи

Формулировка задания

Разработать программу на языке ассемблера RISC-V, реализующую определенную вариантом задания функциональность, отладить программу в симуляторе VSim или Jupiter (https://github.com/andrescv/Jupiter)[1]. Массив (массивы) данных и другие параметры (преобразуемое число, длина массива, параметр статистики и пр.) располагаются в памяти по фиксированным адресам.

Формулировка варианта задания

Перестановка местами элементов массива с четными и нечетными индексами.

Уточнение постановки задачи

Что делать, если массив содержит нечетное число элементов? Поскольку в постановке задачи про это ничего не сказано, определим поведение программы самостоятельно: будем игнорировать последний элемент.

Используется симулятор VSim v2.0.2.

Пример

Пусть в памяти машины в смежных 4-байтных *словах* (**word**) размещены следующие значения: 0, 1, 2, 3, 4, 5, 6. После выполнения программы в тех же словах должны располагаться числа: 1, 0, 3, 2, 5, 4, 6 (переставлены 0-й и 1-й элементы, 2-й и 3-й, 4-й и 5-й).

Решение задачи

Решать задачу будем по шагам.

Шаг 1. Простейшая программа

Простейшая программа состоит из инструкций, выполнение которых приводит к останову симулятора.

Система команд RISC-V не имеет инструкции останова, аналогичной приказу “Z” машины EDSAC, однако предусматривает инструкцию `ecall` для обращения программы к среде поддержки выполнения (обычно, операционной системе). Симулятор VSim использует механизм `ecall` для реализации операций отладочной печати чисел, символов и строк, файлового ввода-вывода и пр., в том числе для останова. Инструкция `ecall` не имеет операндов, а требуемая операция определяется значением *регистра* (**register**) процессора `x10` в момент выполнения инструкции `ecall`. В ассемблере RISC-V для регистров `x0`-`x31` определены синонимы в соответствии с соглашениями, установленными ABI, так, синонимом “x10” является “a0” (см. https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md, подраздел «General Registers»).

Простейшая программа для выполнения в симуляторе VSim имеет следующий вид:

```
# Простейшая программа
```

[1] Дистрибутивы симулятора: https://github.com/andrescv/Jupiter/releases.

```
.text
start:
.globl start
    li a0, 10   # x10 = 10
    ecall       # ecall при значении x10 = 10 => останов симулятора
```

Первая строка программы является однострочным *комментарием* (**comment**), комментарии открываются символом "#" и продолжаются до конца строки.

Далее в тексте программы приведена *директива* (**directive**) `.text` – указание ассемблеру размещать последующие инструкции в секции кода ("text"). В вычислительной системе, имитируемой симулятором VSim, содержимое этой секции размещается в памяти, начиная с адреса 0x00010008.

В следующей строке определяется *метка* (**label**) "start". Метки обозначают определенные «точки» программы, их использование позволяет поручить подсчет адресов ассемблеру (ср. с программированием для EDSAC). Поскольку в секции кода еще ничего не размещено, метке `start` соответствует начальный адрес секции кода – 0x00010008.

Директива `.globl` указывает, что данный символ является экспортируемым, этот вопрос будет подробно обсуждаться позже, пока следует только принять тот факт, что для метки, начиная с которой предполагается выполнение программы, должна быть указана директива `.globl`.

Далее следует *псевдоинструкция* (**pseudoinstruction**) загрузки константы 10 в регистр a0 (т.е. x10, см. выше). Псевдоинструкция *транслируется* ассемблером в последовательность *инструкций* (**instruction**) системы команд RISC-V, обеспечивающую выполнение требуемого действия, в данном случае (поскольку абсолютное значение константы 10 мало), указанной псевдоинструкции соответствует инструкция "`addi x10, x0, 10`".

Наконец, в последней строке программы записана *инструкция* "`ecall`".

Шаг 1.5. Исполнение простейшей программы в симуляторе

Симулятор VSim является "**assemble-and-go**" системой, то есть программа поступает в симулятор в форме исходного текста на языке ассемблера, симулятор осуществляет ассемблирование, загрузку и исполнение программы.

Текст программы необходимо сохранить в текстовом файле, например, "`first.s`". Для запуска симулятора следует ввести следующую команду в командной строке оболочки:

```
java -jar V-Sim-2.0.2.jar -start start -debug first.s
```

Опция `-start` командной строки симулятора определяет метку, с которой начинается исполнение программы. Опция `-debug` приводит к запуску симулятора в режиме отладчика. Управление симулятором осуществляется командами, вводимыми с консоли (после *приглашения* (**prompt**) ">>>"). Работа симулятора прекращается по команде "`quit`", перечень доступных команд отображается по команде "`help`".

Команда `locals` позволяет просмотреть перечень символов и их значений, параметром команды является имя файла, содержащего исходный код:

```
>>> locals
```

```
first.s
start [text] @ 0x00010008
```

Метка “start” является символом, относится к секции “text” и имеет значение 0x00010008.

Исполнение программы состоит в последовательном выполнении инструкций, адрес текущей инструкции содержится в регистре PC (**program counter**):

```
>>> printx pc
PC  [0x00010000]
```

Можно видеть, что исполнение программы начинается с адреса 0x10000, а не 0x10008. В байтах памяти с адресами 0x10000-0x10007 автоматически размещается пара инструкций `auipc+jalr`, формируемых симулятором, эти инструкции обеспечивают переход к метке, указанной в параметром командной строки “-start”.

Команда `printx` позволяет также выводить содержимое регистров общего назначения, параметром команды является название или синоним регистра.

```
>>> printx a0
x10 [0x00000000] (a0) {= 0}
```

Для отображения содержимого памяти используется команда `memory`; содержимое памяти отображается «строками» по 16 (4×4) байт, в параметрах команды задается начальный адрес и количество «строк»:

```
>>> memory 0x00010008 1
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x00010008] 0x00a00513 0x00000073 0x00000000 0x00000000
```

Как было указано выше, псевдоинструкция “`li a0, 10`”, приведенная в тексте программы, транслируется в инструкцию “`addi x10, x0, 10`”, которая кодируется следующим образом:

$$\underbrace{000000001010}_{\substack{imm[11:0]\\10}}\ \underbrace{00000}_{\substack{rs1\\x0}}\ \underbrace{000}_{\substack{funct3\\ADDI}}\ \underbrace{01010}_{\substack{rd\\x10}}\ \underbrace{0010011}_{\substack{opcode\\I-type}}$$

Разобьем этот 32-разрядный код на байты и запишем значения октетов в 16-й системе счисления:

$$\underbrace{00000000}_{00}\ \underbrace{10100000}_{a0}\ \underbrace{00000101}_{05}\ \underbrace{00010011}_{13}$$

Из приведенного выше вывода команды memory можно видеть, что указанный код действительно содержится в байтах памяти с адресами 0x00010008-0x0001000b.

Необходимо отметить, что в выводе команды `memory` четверка смежных байтов памяти группируется в одно 32-разрядное целое без знака, и полученное значение выводится в 16-й записи; в соответствии с принятым в архитектуре RISC-V порядком байтов *от младшего к старшему* (**little-endian**), первый по порядку возрастания адресов байт определяет младшие 8 разрядов результирующего числа, четвертый байт – старшие 8 разрядов. Таким образом, байт по адресу 0x00010008 имеет значение 0x13, байт по адресу 0x00010009 - 0x05 и т.д.

Для выполнения программы следует ввести команду `c` (`continue`), при этом программа будет исполнена до конца и симулятор завершит работу.

Для выполнения программы по шагам вместо команды `c` следует использовать команду `s` (`step`):

```
>>> printx a0
x10 (a0) [0x00000000] {= 0}
>>> s
FROM: start
PC [0x00010000] CODE:0x00000317    call start » auipc x6, 0
>>> s
FROM: start
PC [0x00010004] CODE:0x008300e7    call start » jalr x1, x6, 8
>>> s
FROM: first.s
PC [0x00010008] CODE:0x00a00513    li a0, 10 » addi x10, x0, 10
>>> x10 (a0) [0x0000000a] {= 10}
>>> FROM: first.s
PC [0x0001000c] CODE:0x00000073    ecall » ecall x0, x0, 0

vsim: exit(0)
```

Из приведенной сессии работы с симулятором видно, что первыми исполняются инструкции `auipc` и `jalr` (см. выше). Далее исполняется инструкция, находящаяся по адресу, соответствующему метке `start`, указанной в параметрах командной строки симулятора, как точка входа; *в данном случае* метке `start` соответствует адрес первой инструкции программы (0x10008). После выполнения первой инструкции программы (`addi`) регистр `a0` получил ожидаемое значение.

Шаг 2. Определение тестовых данных

Добавим к имеющейся программе следующие строки:

```
.data
array_length:
    .word 11
array:
    .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Директива `.data` указывает ассемблеру размещать последующие слова в секции (изменяемых) данных. В вычислительной системе, имитируемой симулятором VSim, содержимое секций данных размещается в памяти, начиная с адреса 0x10000000.

Директива `.word` указывает ассемблеру сформировать последовательность 32-разрядных значений (машинных *слов*), соответствующих указанным константам, разбить их на байты и разместить полученные значения в текущей секции, в данном случае – в секции данных.

Слово данных по адресу (поскольку мы имеем дело с little-endian архитектурой, адресом слова является адрес его *младшего* байта), соответствующему метке `array_length`, содержит число элементов обрабатываемого массива. Слова данных, начиная с адреса соответствующего метке `array`, содержит элементы обрабатываемого массива.

```
>>> locals
second.s
array [data] @ 0x10000004
```

```
start [text] @ 0x00010008
array_length [data] @ 0x10000000
>>> memory 0x10000000 1
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000000] 0x0000000b 0x00000000 0x00000001 0x00000002
>>> memory 0x10000004 3
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000004] 0x00000000 0x00000001 0x00000002 0x00000003
[0x10000014] 0x00000004 0x00000005 0x00000006 0x00000007
[0x10000024] 0x00000008 0x00000009 0x0000000a 0x00000000
```

Из вывода команд `locals` и `memory` можно видеть, что при загрузке программы содержимое секции ”data” действительно находится в памяти, начиная с адреса 0x10000000. В исходном тексте программы содержимое секций кода и данных может перемежаться, например, имеющуюся к настоящему времени программу можно было бы записать так:

```
.data  # секция данных
array_length:
    .word 11

.text  # секция кода
start:
.globl start
    li a0, 10  # x10 = 10
    ecall      # ecall при значении x10 = 10 => останов симулятора

.data  # секция данных
array:
    .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Содержимое секции данных при этом «собирается» из нескольких фрагментов:

```
>>> locals
second.s
array [data] @ 0x10000004
start [text] @ 0x00010000
array_length [data] @ 0x10000000
>>> memory 0x10000 1
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x00010000] 0x00a00513 0x00000073 0x00000000 0x00000000
>>> memory 0x10000000 3
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000000] 0x0000000b 0x00000000 0x00000001 0x00000002
[0x10000010] 0x00000003 0x00000004 0x00000005 0x00000006
[0x10000020] 0x00000007 0x00000008 0x00000009 0x0000000a
```

Поскольку значение длины массива является константой – не изменяется во время работы программы, правильно перенести его в секцию “rodata” (**read-only data**):

```
.rodata  # секция неизменяемых данных
array_length:
    .word 11
```

Шаг 3. Рабочий цикл программы

Рабочий цикл программы соответствует следующему фрагменту на языке C:

```
for( size_t i = 1; i < length_of_the_array; i += 2 ) {
    // здесь следует переставить местами элементы
    // [i - 1] и [i] массива
}
```

Здесь `size_t` – это целочисленный беззнаковый тип с разрядностью, достаточной для представления размера любого объекта. Например, при компиляции программы для 32-разрядной машины `size_t` обычно имеет разрядность 32, тогда как при компиляции *той же* программы для 64-разрядной машины - разрядность 64.

В *неструктурированном* (**spaghetti code**) виде, с использованием операторов перехода `goto`, этот цикл может быть записан следующим образом:

```
    size_t i = 1;
loop:
    if( !( i < length_of_the_array ) )
        goto loop_exit;

    // здесь следует переставить местами элементы
    // [i - 1] и [i] массива

    i += 2;
    goto loop;
loop_exit:
```

Значение автоматической переменной `i` будет размещено в одном из регистров процессора, мы будем использовать регистр `a2`. Число элементов обрабатываемого массива мы также разместим в регистре (`a3`), перед началом выполнения цикла это значение будет загружаться из памяти по адресу, соответствующему метке `array_length`.

```
.text
start:
.globl start
    la a3, array_length  #}
    lw a3, 0(a3)         #} a3 = <длина массива>

    li a2, 1             # a2 = 1
loop:
    bgeu a2, a3, loop_exit    # if( a2 >= a3 ) goto loop_exit

    # здесь следует переставить местами элементы
    # [i - 1] и [i] массива, где i - значение регистра a2

    addi a2, a2, 2       # a2 += 2
    jal zero, loop       # goto loop
loop_exit:

    li a0, 10  # x10 = 10
```

```
    li a1, 0    # x11 = 0
    ecall       # ecall при значении x10 = 10 => останов симулятора

.rodata
array_length:
    .word 11
.data
array:
    .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Псевдоинструкция `la` обеспечивает установку регистра `a3` в значение *адреса*, соответствующего метке `array_length`. Псевдоинструкция будет транслирована в последовательность инструкций (пару `auipc+addi` или `lui+addi`), при выполнении которых, как мы установили выше, в регистр `a3` будет помещено значение 0x10000000. Инструкция `lw` обеспечивает загрузку из памяти слова данных с заданным адресом. Адрес загружаемого слова данных записан в виде "`0(a3)`", то есть, адресом загружаемого слова является сумма 0 + <значение регистра a3> = 0 + 0x10000000 = 0x10000000. Полученное из памяти значение помещается в регистр `a3` (регистр назначения команды `lw`). Суммируя сказанное, в результате выполнения первых 3 инструкций программы в регистр `a3` будет записано число 11.

Для завершения цикла используется команда *условного перехода* (**branch**) `bgeu`. Содержимое каждого регистра-операнда (`a2` и `a3`) *интерпретируется*, как целое число без знака. Если при этом <значение a2> ≥ <значение a3> (то есть, <значение i> ≥ <длина массива>), исполнение программы продолжается с адреса, соответствующего метке `loop_exit`, в противном случае продолжается исполнение инструкций, следующих за `bgeu` – инструкций, образующих «тело» цикла.

Инструкция `jal` обеспечивает *безусловный переход* (**jump**) на начало цикла. При этом адрес следующей по порядку инструкции записывается в регистр `zero` (синоним `x0`), то есть теряется, так как в архитектуре RISC-V регистр `x0` всегда имеет значение 0.

Проследить выполнение программы можно по шагам, используя команду `s`, но в данном случае это уже затруднительно, поскольку число выполняемых инструкций велико. Вместо отладки по шагам установим точку останова на первой инструкции цикла (`bgeu`):

```
>>> locals
third.s
array [data] @ 0x10000008
loop [text] @ 0x00010018
loop_exit [text] @ 0x00010024
start [text] @ 0x00010008
array_length [rodata] @ 0x10000000
>>> breakpoint 0x10018
>>> c
>>> printx pc
PC [0x00010018]
>>> printx a2
x12 (a2) [0x00000001] {= 1}
>>> printx a3
x13 (a3) [0x0000000b] {= 11}
```

```
>>> s
FROM: third.s
PC [0x00010018] CODE:0x00d67663 bgeu a2, a3, loop_exit » bgeu x12, x13, 12
>>> s
FROM: third.s
PC [0x0001001c] CODE:0x00260613 addi a2, a2, 2 » addi x12, x12, 2
```

Команда `breakpoint` устанавливает точку останова (**breakpoint**) на указанном адресе (адрес 0x10018 соответствует метке `loop`, то есть инструкции `bgeu`). Теперь выполнение программы может быть продолжено (команда `c`, см. выше), однако когда выполнение дойдет до точки останова, то есть наступит очередь выполнять инструкцию по адресу, указанному в параметре команды `breakpoint`, исполнение программы будет приостановлено. В этот момент можно проанализировать содержимое регистров (`printx`), продолжить выполнение программы по шагам (`s`), изучить содержимое памяти (`memory`) или возобновить выполнение программы (`c`).

Можно установить несколько точек останова. Для вывода списка установленных точек останова используется команда `list`, для удаления точки останова – `delete`:

```
>>> list
Breakpoints:

    0x00010018
>>> delete 0x10018
>>> list
vsim: no breakpoints yet
```

Отметим, что инструкции `bgeu` и `jal` используют относительную адресацию (**relative branch/jump**). Поскольку каждая *инструкция* кодируется 4 байтами (расширение “C” системы команд не используется), и инструкции размещаются, начиная с адреса 0x10008, несложно подсчитать, что инструкция `bgeu` будет размещаться в байтах с адресами 0x10018-0x1001b, инструкция `jal` – в байтах 0x10020-0x10023, а меткам `loop` и `loop_exit` будут соответствовать адреса 0x10018 и 0x10024 соответственно. В системе команд RISC-V значение (архитектурного) регистра PC равно адресу *текущей* инструкции, следовательно, при выполнении инструкции `bgeu`, если условие перехода выполняется, значение PC должно измениться с 0x10018 на 0x10024, что эквивалентно действию PC←PC+12, поэтому непосредственный операнд инструкции `bgeu` равен 12 (что можно видеть в приведенном выше выводе симулятора после выполнения данной инструкции). Аналогично, при выполнении инструкции `jal` должно значение PC должно измениться с 0x10020 на 0x10018, что эквивалентно действию PC←PC+(-8):

```
>>> breakpoint 0x10020
>>> c
>>> s
FROM: third.s
PC [0x00010020] CODE:0xff9ff06f    jal zero, loop ? jal x0, -8
```

Шаг 4. Тело рабочего цикла программы

Теперь остается только реализовать функциональность, обозначенную комментарием в тексте имеющейся программы: выполнить перестановку смежных элементов массива. В языке C тело рабочего цикла можно записать следующим образом:

```
const unsigned t = array[ i - 1 ];
array[ i - 1 ] = array[ i ];
array[ i ] = t;
```

В архитектуре RISC-V роль t может играть один из регистров процессора, точно так же, как роль переменной цикла `i` играет регистр `a2`. Тогда первая строка приведенного фрагмента кода реализуется уже знакомой нам инструкцией `lw`, а последняя строка – аналогичной инструкцией `sw`, выполняющей запись содержимого регистра процессора в память. Вторая строка приведенного фрагмента – копирование слова данных в памяти, в RISC-V (и в большинстве других архитектур) требует двух инструкций: загрузки слова данных в регистр (`lw`) и записи содержимого регистра в память (`sw`).

Как мы уже видели в случае `lw`, адрес слова данных в памяти, к которому производится обращение, должен находиться в одном из регистров процессора. Перед выполнением цикла установим значение регистра `a4`, равное адресу 0-го элемента массива:

```
…
la a4, array
…
```

Теперь в теле цикла в регистре `a4` находится адрес 0-го элемента массива, а в регистре a2 – значение i в текущей итерации цикла. Поскольку каждый элемент массива занимает 4 байта памяти, адреса требуемых элементов могут быть вычислены следующим образом:

<адрес элемента i> = <значение a4> + 4 × <значение a2>
<адрес элемента i-1> = <адрес элемента i> - 4

Поскольку умножение на 4 эквивалентно сдвигу на 2 разряда влево, следующая последовательность инструкций обеспечивает вычисление требуемых адресов в регистрах `a5` и `a6`:

```
…
slli a5, a2, 2  # a5 = a2 << 2 = a2 * 4
add a5, a4, a5  # a5 = a4 + a5 = a4 + a2 * 4
addi a6, a5, -4  # a6 = a5 + (-4) = a5 - 4
…
```

Теперь перестановка элементов массива реализуется следующей последовательностью инструкций:

```
…
lw t1, 0(a6)  # t1 = array[i-1]
lw t0, 0(a5)  # t0 = array[i]
sw t1, 0(a5)  # array[i] = t1
sw t0, 0(a6)  # array[i-1] = t0
…
```

здесь `t0`, `t1` – синонимы регистров `x5`, `x6`.

Запишем программу целиком:

```
.text
```

```
start:
.globl start
    la a3, array_length  #}
    lw a3, 0(a3)         #} a3 = <длина массива>

    la a4, array         # a4 = <адрес 0-го элемента массива>

    li a2, 1             # a2 = 1
loop:
    bgeu a2, a3, loop_exit    # if( a2 >= a3 ) goto loop_exit

    slli a5, a2, 2       # a5 = a2 << 2 = a2 * 4
    add a5, a4, a5       # a5 = a4 + a5 = a4 + a2 * 4
    addi a6, a5, -4      # a6 = a5 + (-4) = a5 - 4

    lw t1, 0(a6)         # t1 = array[i-1]
    lw t0, 0(a5)         # t0 = array[i]
    sw t1, 0(a5)         # array[i] = t1
    sw t0, 0(a6)         # array[i-1] = t0

    addi a2, a2, 2       # a2 += 2
    jal zero, loop       # goto loop
loop_exit:

finish:
    li a0, 10  # x10 = 10
    li a1, 0   # x11 = 0
    ecall      # ecall при значении x10 = 10 => останов симулятора

.rodata
array_length:
    .word 11
.data
array:
    .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Для анализа результата выполнения программы в симуляторе необходимо остановить выполнение программы до исполнения инструкции `ecall`, завершающей работу симулятора. Очевидно, точки останова удобно устанавливать на адресах, соответствующих некоторым меткам, определенным в тексте программы. В данном случае можно было бы использовать метку `loop_exit`, однако *по смыслу* (ср. программы на шаге 1 и шаге 3) эта метка определяет точку выхода из рабочего цикла, а не точку входа в последовательность инструкций, завершающих работу симулятора. С учетом этого, в текст программы была добавлена метка `finish`.

Проверим работу программы в симуляторе:

```
>>> locals
array [data] @ 0x10000008
loop [text] @ 0x00010020
loop_exit [text] @ 0x00010048
start [text] @ 0x00010008
```

```
finish [text] @ 0x00010048
array_length [rodata] @ 0x10000000
>>> printx pc
PC  [0x00010000]
>>> memory 0x10000008 3
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000008] 0x00000000 0x00000001 0x00000002 0x00000003
[0x10000018] 0x00000004 0x00000005 0x00000006 0x00000007
[0x10000028] 0x00000008 0x00000009 0x0000000a 0x00000000
>>> breakpoint 0x10048
>>> c
>>> printx pc
PC  [0x00010048]
>>> memory 0x10000008 3
              Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000008] 0x00000001 0x00000000 0x00000003 0x00000002
[0x10000018] 0x00000005 0x00000004 0x00000007 0x00000006
[0x10000028] 0x00000009 0x00000008 0x0000000a 0x00000000
>>> c
vsim: exit(0)
```

Приведенная программа примерно соответствует следующей программе на языке C:

```
#include <stddef.h>

static unsigned array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
static const size_t array_length = sizeof( array ) / sizeof( array[ 0 ] );

int main( void ) {
    for( size_t i = 1; i < array_length; i += 2 ) {
        const unsigned t = array[ i - 1 ];
        array[ i - 1 ] = array[ i ];
        array[ i ] = t;
    }

    return 0;
}
```

Первая строка обеспечивает *включение* (**include**) в программу содержимого файла “stddef.h” (входящего в состав пакета разработки), в котором определен используемый в программе тип данных `size_t`.

Ключевые слова `static` в определениях `array` и `array_length` указывают на то, что эти символы являются локальными, то есть для них компилятор не должен формировать (аналоги) директивы `.globl`.

Ключевые слова `const` в определениях `array_length` и `t` означают, что значения этих переменных устанавливаются один раз и далее не модифицируются. Это предотвращает ошибки в программе, например, попытки модифицировать `array_length`, *а также* предоставляет компилятору дополнительную информацию о *семантике* программы, которая может использоваться для ее оптимизации.

Определение `array_length` обеспечивает согласованность фактической длины массива `array` и значения `array_length`. В тексте программы на ассемблере при изменении длины массива `array` необходимо соответствующим образом модифицировать значение слова с меткой `array_length`, в приведенной программе правильное значение `array_length` рассчитывается автоматически компилятором[2]. Значением `sizeof(array)` является размер, в числе *адресуемых ячеек* памяти (*в случае* RISC-V – в байтах), массива `array`. Значением `sizeof(array[0])` является размер одного элемента массива (все элементы массива имеют одинаковый размер). Учитывая это, значением приведенного выражения действительно является число элементов массива – его длина.

Следует отметить, что в приведенном тексте на языке C отдельно стоящий идентификатор `array_length`, например, в выражении "`i < array_length`", обозначает *значение*, а в тексте на ассемблере – адрес слова памяти, содержащего это значение[3].

Исполнение программы на языке C состоит в исполнении *функции* (**function**) `main`, которая в простейшем случае не имеет параметров (пусто, **void**) и *возвращает* (**return**) целое число (int – сокр. от **integer**) – *код завершения* (**termination status**); код 0 означает «нормальное» завершение программы. Обычно код завершения программы передается внешнему программному компоненту (родительскому процессу, отладчику), и позволяет ему анализировать результат выполнения программы или причину ее останова.

Шаг 4.5. Оптимизация программы

Использование непосредственного операнда инструкций `lw` и `sw` позволяет отказаться от явного вычисления адреса элемента (i-1) массива:

```
    …
    slli a5, a2, 2        # a5 = a2 << 2 = a2 * 4
    add a5, a4, a5        # a5 = a4 + a5 = a4 + a2 * 4

    lw t1, -4(a5)          # t1 = array[i-1]
    lw t0, 0(a5)          # t0 = array[i]
    sw t1, 0(a5)          # array[i] = t1
    sw t0, -4(a5)          # array[i-1] = t0
    …
```

Более сложная оптимизация состоит в изменении организации рабочего цикла. В текущей версии «накладные расходы» на организацию цикла составляют 2 инструкции (`bgeu` и `jal`) на итерацию. Следующая организация цикла позволяет уменьшить эти «накладные расходы» вдвое:

```
    …
    bgeu a2, a3, loop_exit    # if( a2 >= a3 ) goto loop_exit
loop:
    # тело цикла

    addi a2, a2, 2        # a2 += 2
```

[2] Подобные языковые средства могут предоставляться и ассемблерами.

[3] В языке C имеется определенная нерегулярность в использовании идентификаторов, свойственная, впрочем, большинству языков программирования; например, в выражении `i = i + 2` идентификатор "i" в правой части означает значение `i`, а в левой – «место», в которое следует записать полученную сумму (действительно, *значению* нельзя *присвоить значение*).

```
    bltu a2, a3, loop     # if( a2 < a3 ) goto loop
loop_exit:
    …
```

Заметим, что здесь *перед циклом* контролируется условие *завершения* цикла, а внутри цикла – условие *продолжения* цикла; в тексте программы на языке C в операторах for, while и do-while записывается именно условие продолжения цикла.

В общем случае условие продолжения цикла может быть произвольно сложным, и его дублирование (с точностью до наоборот) перед циклом может оказаться нежелательным. Следующая организация цикла является универсальной:

```
    …
    jal zero, loop_check  # goto loop_check
loop:
    # тело цикла

    addi a2, a2, 2        # a2 += 2
loop_check:
    bltu a2, a3, loop     # if( a2 < a3 ) goto loop
loop_exit:
    …
```

Шаг 5. Оптимизация: применение идеи указателей

Дальнейшая оптимизация программы позволяет отказаться от расчета адреса i-го элемента массива в теле цикла:

```
    …
    la a6, array          # a6 = <адрес 0-го элемента массива>

    li a2, 1              # a2 = 1
    bgeu a2, a3, loop_exit  # if( a2 >= a3 ) goto loop_exit
loop:
    lw t1, 0(a6)          # t1 = array[i-1]
    lw t0, 4(a6)          # t0 = array[i]
    sw t1, 4(a6)          # array[i] = t1
    sw t0, 0(a6)          # array[i-1] = t0

    addi a6, a6, 8        # a6 += 2 * 4
    addi a2, a2, 2        # a2 += 2
    bltu a2, a3, loop     # if( a2 < a3 ) goto loop
loop_exit:
    …
```

В приведенной программе регистр `a6` содержит адрес (i-1)-го элемента массива (регистр `a6` выбран потому, что он уже использовался для этой цели ранее, с равным успехом можно было бы использовать, например, регистры `a4` или `a5`). При первом входе в цикл i=1, и `a6` должен содержать адрес 0-го элемента массива; в каждой итерации значение i увеличивается на 2, и значение `a6` должно увеличиваться на значение 2×<размер элемента>=2×4=8.

Приведенная программа примерно соответствует следующей программе на языке C:

```
#include <stddef.h>

static unsigned array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
static const size_t array_length = sizeof( array ) / sizeof( array[ 0 ] );

int main( void ) {
    unsigned *array_ptr = & array[ 0 ];
    for( size_t i = 1; i < array_length; i += 2 ) {
        const unsigned t = * array_ptr;
        * array_ptr = * ( array_ptr + 1 );
        * ( array_ptr + 1 ) = t;
        array_ptr += 2;
    }

    return 0;
}
```

В приведенной программе `array_ptr` – переменная-*указатель* (**pointer**). В нашей программе (и, обычно, в результате трансляции приведенной программы компилятором C) переменная `array_ptr`, как и переменная `i`, размещена в регистре. При определении переменной (в первой строки тела функции `main`) ее значение устанавливается равным адресу 0-го элемента массива `array` (в языке C *унарная* операция "`&`" – операция взятия адреса указанного объекта). В этом случае говорят, что `array_ptr` указывает (**points to**) на 0-й элемент массива.

В теле цикла используется "обратная" к "`&`" операция разыменовывания (**dereference**) указателя "`*`". Переменная `t` инициализируется значением слова данных в памяти по адресу, который содержится в переменной `array_ptr`. В первой итерации цикла `array_ptr` содержит адрес 0-го элемента массива `array` (см. выше), следовательно, t получает значение `array[ 0 ]`.

Во второй строке тела цикла используется так называемая «адресная арифметика» (**pointer arithmetic**). Значением выражения `(array_ptr + 1)` является *адрес* слова, следующего в памяти за словом, адрес которого содержится в `array_ptr`. Аналогично, значением `(array_ptr - 1)` был бы адрес предыдущего слова, а значением `(array_ptr + 2)` – адрес слова, следующего за следующим и т.д. Например, в первой итерации цикла `array_ptr` содержит адрес 0-го элемента массива, следовательно, `(array_ptr + 1)` является адресом 1-го элемента.

Подобно тому, как отдельно стоящий идентификатор в правой части присваивания обозначает значение переменной, а в левой части – «место», в которое следует записать результат (см. примечание выше), в следующей строке тела функции `(* array_ptr)` в левой части обозначает «место», в которое следует записать значение `*( array_ptr + 1 )`. В первой итерации цикла `array_ptr` указывает на 0-й элемент массива, следовательно, в 0-й элемент будет записано значение 1-го элемента.

Следующая строка тела цикла аналогична только что рассмотренной и не требует комментариев.

Последняя строка тела цикла также является примером адресной арифметики. Поскольку выражение (x += 2) эквивалентно (x = x + 2), смысл этой строки несложно понять: если переменная `array_ptr` содержала адрес j-го элемента массива (то же самое: указатель `array_ptr` указывал на i-й элемент массива), то теперь в ней содержится адрес (j+2)-го элемента массива.

Таким образом, в первой итерации цикла `array_ptr` указывает на 0-й элемент массива `array`, в следующий – на 2-й элемент, затем – на 4-й и т.д.; вообще, на входе в тело цикла `array_ptr` указывает на (i-1)-й элемент массива `array`.

Как компилятор «понимает», что выражение (i += 2) означает увеличение значения i на 2, а выражение (array_ptr += 2) - увеличение `array_ptr` на 2×<размер элемента>, и чему при этом равен <размер элемента>? Дело в том, что каждая переменная в программе имеет *тип* (**type**). Так, `i` имеет тип `size_t` – целое число без знака (см. выше), а `array_ptr` – тип указателя *на* `unsigned` (тип беззнаковых целых). Поскольку *для машины с архитектурой RISC-V* тип `unsigned` имеет разрядность 32[4], и адресуемой ячейкой памяти является (8-битный) байт, <размер элемента> = 4.

Следует отметить, что современный оптимизирующий компилятор способен сформировать код, аналогичный нашему, транслируя программу на языке C, приведенную на «Шаге 4».

Шаг 5.5. Оптимизация: исключение переменной цикла

В оптимизированной с использованием указателей программе регистр `a2` (переменная i) используется только в условии цикла, то есть для (неявного) определения числа итераций. Однако число итераций можно рассчитать заранее, как floor(array_length / 2). Это позволяет упростить логику программы:

```
…
const size_t pair_count = array_length / 2;
unsigned *array_ptr = & array[ 0 ];
for( size_t pair_index = 0; pair_index < pair_count; ++ pair_index ) {
    const unsigned t = * array_ptr;
    * array_ptr = * ( array_ptr + 1 );
    * ( array_ptr + 1 ) = t;
    array_ptr += 2;
}
…
```

Здесь используется тот факт, что при *целочисленном* делении *положительных* чисел результат округляется вниз. Деление числа без знака на 2, разумеется, эквивалентно логическому сдвигу на один разряд вправо (инструкция `srli` архитектуры RISC-V), но в программе на C не следует использовать этот факт, так как современные оптимизирующие компиляторы, несомненно, способны выполнить это преобразование самостоятельно, а выражение (array_length / 2) в данном контексте понятнее, чем (array_length >> 1)[5].

Далее, от подсчета числа итераций можно вовсе отказаться, контролируя значение указателя `array_ptr`. Действительно, несложно понять, что обработку пар смежных элементов следует продолжать до тех пор, пока значение `array_ptr` не сравняется с ((`& array[0]`) + `pair_count * 2`). Таким образом, тело функции `main` приобретает следующий вид:

```
…
const size_t pair_count = array_length / 2;
```

[4] При компиляции для других машин тип `unsigned` может иметь другое значение:16, 36, 40, 64…

[5] Как компилятор «понимает», какую инструкцию необходимо сформировать для реализации операции “>>” – `srli` (логический сдвиг вправо) или `srai` (арифметический сдвиг вправо)? Требуемая инструкция определяется типом `array_length`!

```
    unsigned *end_ptr = ( & array[ 0 ] ) + pair_count * 2;
    for( unsigned *array_ptr = & array[ 0 ];
         array_ptr != end_ptr;
         array_ptr += 2 ) {
        const unsigned t = * array_ptr;
        * array_ptr = * ( array_ptr + 1 );
        * ( array_ptr + 1 ) = t;
    }
    …
```

Соответствующий код на языке ассемблера:

```
    …
    la a6, array          # a6 = <адрес 0-го элемента массива>

    srli a7, a3, 1        # ~ pair_count
    slli a7, a7, 3        # a7 = pair_count << 3 = ( pair_count * 2 ) * 4
    add a7, a6, a7        # ~ end_ptr

    beq a6, a7, loop_exit  # if( array_ptr == end_ptr ) goto loop_exit
loop:
    lw t1, 0(a6)          # t1 = * array_ptr
    lw t0, 4(a6)          # t0 = * ( array_ptr + 1 )
    sw t1, 4(a6)          # * ( array_ptr + 1 ) = t1
    sw t0, 0(a6)          # * array_ptr = t0

    addi a6, a6, 8        # array_ptr += 2
    bne a6, a7, loop      # if( array_ptr == end_ptr ) goto loop
loop_exit:
    …
```

Можно видеть, что в коде инициализации вместо одной инструкции, устанавливавшей значение `a2`, теперь используется три, устанавливающие значение `a7`. Однако это позволило исключить одну инструкцию из тела цикла. Поскольку при обработке массива инициализация выполняется однократно, а тело цикла – для каждой пары переставляемых элементов, данное изменение действительно оптимизирует программу, *для достаточно больших массивов*.

Приведенный выше фрагмент на языке С можно улучшить стилистически. Во-первых, поскольку изменение значения указателя `end_ptr` не предполагается, следует использовать квалификатор `const`, как это сделано в `array_length` и `t` (следует обратить внимание на положение квалификатора `const` – *после* “*”).

```
    unsigned * const end_ptr = ( & array[ 0 ] ) + pair_count * 2;
```

Во-вторых, в данном случае указатель `end_ptr` не должен использоваться для доступа к памяти. Действительно, в случае массива нечетной длины `end_ptr` указывает на последний элемент массива, и доступ к нему просто не требуется. В случае же массива четной длины `end_ptr` и вовсе указывает на слово данных, *следующее* за последним элементом массива, и что находится в этой области памяти, вообще говоря, неизвестно. Таким образом, указатель `end_ptr` должен использоваться просто как некоторое значение адреса, в языке С эту роль играет указатель на `void`:

```
void * const end_ptr = ( & array[ 0 ] ) + pair_count * 2;
```

Можно сказать, что тип `void *` сохраняет для компилятора информацию о том, что значением переменной является адрес, но теряет <размер элемента> (см. выше). Тогда понятно, что выражения вида `(* end_ptr)`, `(end_ptr + 1)` и др. не имеют смысла, а значит, их использование приводит к ошибке времени компиляции.

Дополнительная информация

Директивы .byte, .half; инструкции lb[u], sb, lh[u], sh

Аналогично рассмотренной директиве `.word` в программе на языке ассемблера RISC-V могут использоваться следующие директивы (список не является избыточным, см. документацию):

`.byte` – для определения последовательности 8-разрядных целых чисел (байтов);
`.half` – для определения последовательности 16-разрядных целых чисел (полуслов, **halfword**).

Например,

```
byte_seq: .byte 0xEF, 0xBE, 0xAD, 0xDE, 0xBE, 0xBA, 0xFE, 0xCA
```

Следует обратить внимание на особенности отображения содержимого памяти в выводе команды `memory` симулятора VSim:

```
>>> memory 0x10000000 1
               Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000000] 0xdeadbeef 0xcafebabe 0x00000000 0x00000000
```

Как объяснялось выше, при отображении содержимого памяти последовательность из 4 байтов группируется в 4-байтное слово в порядке от младшего байта к старшему, после чего это слово отображается в 16-м представлении. Например, из расположенных в памяти один за другим байтов 0xEF, 0xBE, 0xAD, 0xDE будет сформировано значение $0x\underbrace{DE}_{[3]}\underbrace{AD}_{[2]}\underbrace{BE}_{[1]}\underbrace{EF}_{[0]}$, которое и будет напечатано в выводе команды.

Та же последовательность байтов в памяти может быть определена другими способами[6]:

```
half_seq: .half 0xBEEF, 0xDEAD, 0xBABE, 0xCAFE
word_seq: .word 0xDEADBEEF, 0xCAFEBABE
```

Загрузку индивидуальных байтов из памяти в регистры процессора обеспечивают инструкции `lb` и `lbu`, аналогичные рассмотренной выше инструкции `lw`. Содержимое байта – 8 двоичных разрядов – может *интерпретироваться* как 8-разрядное целое со знаком или без знака. При нахождении в регистре это целое размещается в 32-разрядной сетке, при этом, естественным образом, разряды целого записываются в младшие 8 разрядов регистра, а старшие 24 разряда заполняются либо знаком – значением старшего разряда 8-разрядного целого, либо нулями. Инструкция `lb` соответствует интерпретации загружаемого байта, как целого со знаком, инструкция `lbu` – как целого без знака.

[6] А также `.quad 0xCAFEBABEDEADBEEF`, но эта директива в настоящее время не поддерживается симулятором VSim

```
la a4, byte_seq  # a4 = & byte_seq[ 0 ]
lb t0, 3(a4)     # t0 = (int) byte_seq[ 3 ] = 0xFFFFFFDE
lbu t1, 3(a4)    # t1 = (unsigned) byte_seq[ 3 ] = 0x000000DE
```

Запись индивидуальных байтов осуществляется инструкцией `sb`, при этом младшие 8 разрядов регистра-операнда записываются в память по указанному адресу:

```
sb t0, 3(a4)     # byte_seq[ 3 ] = (char) t0 = 0xDE
```

Аналогично, для загрузки и записи полуслов используются инструкции `lh`, `lhu` и `sh`.

Директива .string

Директива `.string` может использоваться для определения последовательности байтов, соответствующих символам строки:

```
.rodata
str: .string "hello"
```

Особенностью этой директивы является добавление к последовательности байтов одного байта со значением 0 (т.н. **null-terminator**). Приведенная директива эквивалентна следующей (используются ASCII-коды символов “h”, “e” и т.д.):

```
str: .byte 104, 101, 108, 108, 111, 0  # нулевой байт в конце!
```

В строке могут использоваться *экранированные последовательности* (**escape sequence**) для кодирования специальных символов, например:

`\n` – символ перевода строки
`\t` – символ (горизонтальной) табуляции

```
strnl: .string "hello\n"  # ~ .byte 104, 101, 108, 108, 111, 10, 0
                          #                new line character ^
```

Для чтения и записи индивидуальных символов строки используются инструкции `lb[u]` и `sb`.

Директива .zero

Для определения последовательности нулевых байтов (например, для определения выходного буфера, в котором будет формироваться вывод программы) может использоваться директива `.zero`, параметром директивы является размер буфера:

```
target_buf: .zero 16
```

эквивалентно

```
target_buf: .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Отметим, что инициализированные нулями буферы значительного размера принято размещать не в секции `.data`, а в секции `.bss`:

```
.bss
big_buf: .zero 1024
```

Исторически, название директивы является акронимом от “Block Started by Symbol”, альтернативная расшифровка - “Better Save Space”[7].

Псевдоинструкции l<*x*>, s<*x*>

Выше мы использовали следующую последовательность (псевдо)инструкций для загрузки длины массива в регистр процессора:

```
la a3, array_length  #}
lw a3, 0(a3)         #} a3 = <длина массива>
```

Как обсуждалось выше, псевдоинструкция `la` обеспечивает формирование в `a3` адреса, соответствующего метке `array_length`, и транслируется в пару инструкций `auipc+addi`; инструкция `auipc` обеспечивает «грубое» формирование адреса, а `addi` – коррекцию полученного значения прибавлением 12-разрядного непосредственного операнда к результату `auipc`. Можно заметить, что результирующая последовательность инструкций является избыточной, поскольку коррекцию адреса прибавлением 12-разрядного непосредственного операнда позволяет выполнить сама инструкция `lw`.

Это рассуждение справедливо также для других инструкций загрузки (`lb`[`u`], `lh`[`u`] и т.д.) и записи (`sw` и др). Учитывая это, ассемблер RISC-V определят набор псевдоинструкций `lw`, `sw`, `lb` и т.д., соответствующих парам инструкций `auipc+l`<*x*>и `auipc+s`<*x*>. Таким образом, приведенную выше последовательность (псевдо)инструкций лучше заменить следующей псевдоинструкцией:

```
lw a3, array_length  # a3 = <длина массива>
```

Целочисленное умножение и деление

Симулятор VSim поддерживает инструкции (стандартное расширение “M”) целочисленного умножения и деления. Для всех инструкций используется формат R (то есть операнды операции хранятся в двух регистрах процессора, результат записывается в регистр процессора, используемые регистры явно указываются в команде).

Следующая последовательность инструкций осуществляет умножение двух 32-разрядных чисел без знака, находящихся в регистрах `a2` и `a3`, с формированием 64-разрядного результата в паре регистров `a4` (младшие разряды) и `a5` (старшие разряды):

```
.text
    mul a4, a2, a3    # a4 = младшие разряды (a2 * a3)
    mulhu a5, a2, a3  # a5 = старшие разряды (a2 * a3)
```

Умножение чисел со знаком осуществляется аналогично:

```
.text
    mul a4, a2, a3    # a4 = младшие разряды (a2 * a3)
    mulh a5, a2, a3   # a5 = старшие разряды (a2 * a3)
```

Частное и остаток от деления двух чисел формируются инструкциями `div`*[`u`]* и `rem`*[`u`]*:

---

[7] Аналогично тому, как использование директивы `.zero` позволяет уменьшить размер исходного текста программы (сравните директиву `.zero` с директивой `.byte` для определения `big_buf`), секция “bss” позволяет уменьшить объем *объектного файла* (**object file**), получаемого в результате ассемблирования исходного текста; этот вопрос подробно обсуждается в следующем разделе курса.

```
.text
    div a4, a2, a3    # a4 = a2 / a3 (divu для чисел без знака)
    rem a5, a2, a3    # a5 = a2 % a3 (remu для чисел без знака)
```

Вычисления с плавающей точкой

Симулятор VSim поддерживает инструкции (стандартное расширение “F”) вычислений с плавающей точкой *одинарной точности* (**single precision**), для кодирования которых используются 32-разрядные слова.

Директива `.float` используется для определения последовательности чисел:

```
.data
flt_arr: .float 1.0e-2, -0.2, 30
```

Стандартное расширение “F” добавляет к архитектуре тридцать два 32-разрядных регистра `f0-f31` (регистры с плавающей точкой) регистр состояния `fcsr` и набор дополнительных инструкций. Большинство новых инструкций оперируют над числами, содержащимися в регистрах `f0-f31`. Команда `printf` симулятора VSim обеспечивает вывод содержимого регистров с плавающей точкой.

Загрузка чисел из памяти в регистры с плавающей точкой осуществляется командами `flw` и `fsw`, аналогичными `lw` и `sw`, базовый адрес находится в одном из регистров общего назначения `x0-x31`.

```
.text
    la a4, flt_arr
    flw ft0, 0(a4)   # ft0 = flt_arr[ 0 ]
    flw ft1, 4(a4)   # ft1 = flt_arr[ 1 ]
    fsw ft0, 4(a4)   # flt_arr[ 1 ] = ft0
    fsw ft1, 0(a4)   # flt_arr[ 0 ] = ft1
```

Здесь “ft0” - синоним `f0`, “ft1” – синоним `f1` (см. https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md, подраздел «Floating-point Register Convention»).

Вычислительные инструкции записываются в формате R, например:

```
.text
    fadd.s ft2, ft0, ft1
    fsum.s ft3, ft0, ft1
    fmul.s ft4, ft0, ft1
    fdiv.s ft5, ft0, ft1
```

Для сравнения чисел с плавающей точкой используются инструкции `feq.s`, `flt.s`, `fle.s`, операндами являются регистры с плавающей точкой, результат сравнения (0 или 1) помещается в регистр общего назначения:

```
.text
    flt.s t0, ft0, ft1  # t0 = ( ft0 < ft1 ) ? 1 : 0
                        # эквивалентно: t0 = ( ft0 == ft1 )
```

Предусмотрены также инструкции преобразования целых чисел, содержащихся в регистрах общего назначения, в числа с плавающей точкой, и обратно:

```
.text
    fcvt.w.s t0, ft0    # t0 = (int) xround( ft0 )
    fcvt.s.w ft1, t0    # ft1 = (float) t0
    fcvt.wu.s t1, ft0   # t1 = (unsigned) xround( ft0 )
    fcvt.s.wu ft1, t1   # ft1 = (float) t1
```

Архитектура RISC-V предусматривает различные режимы округления при преобразовании числа с плавающей точкой в целое, но в настоящее время симулятор VSim их не поддерживает.

Наконец отметим инструкции копирования разрядной сетки между регистрами общего назначения и регистрами с плавающей точкой:

```
.text
    fmv.x.w t0, ft0
    fmv.w.x ft1, t0
```

Поясним различие между инструкциями `fcvt` и `fmv` на примере. Вещественное число 1,25 имеет следующее представление в памяти и в регистре с плавающей точкой:

$$\underbrace{0}_{sign} \underbrace{01111111}_{\substack{exponent \\ 0+127=127}} \underbrace{01000000000000000000000}_{\substack{fraction \\ |1.25|\times 2^{exponent-1}=1.25-1=0.25}} = 0\text{x}3\text{FA}00000$$

Пусть это число размещено в регистре `ft0`, опишем результат выполнения приведенных выше пар инструкций `fcvt` и `fmv`. В результате выполнения `fcvt.w.s` в целевом регистре `t0` будет сформировано значение 1 = 0x00000001. После этого в результате выполнения `fcvt.s.w` в целевом регистре `ft1` будет содержаться число с плавающей точкой (float) 1 = 1.0f (представленное в разрядной сетке кодом 0x3F800000). Напротив, в результате выполнения `fmv.x.w` в целевом регистре `t0` будет сформировано *представление* в разрядной сетке числа 1.25, то есть число 0x3FA00000. После этого в результате выполнения `fmv.w.x` в целевом регистре `ft1` будет содержаться число с плавающей точкой, имеющее представление 0x3FA00000, то есть число 1.25.

Полный перечень инструкций стандартного расширения “F” приведен в документе “The RISC-V Instruction Set Manual, Volume I: User-Level ISA”. Инструкции, поддерживаемые симулятором VSim, отображаются при указании опции командной строки `-iset`.

Код завершения программы

Для завершения программы вместо системного вызова (`ecall`) с кодом 10 (в регистре `a0`) может использоваться код 17. В этом случае в регистре `a1` указывается код завершения программы:

```
    li a0, 17
    li a1, 1  # код завершения 1 ~ exit(1)
    ecall
```

Ввод-вывод

Механизм `ecall` может использоваться для реализации ввода-вывода символов, строк, целых чисел, чисел с плавающей точкой, а также файлового ввода-вывода. Например, следующая программа напечатает строку “hello, world” с переводом строки:

```
.text
start:
.globl start
    li a0, 4  # 'print string' ecall
    la a1, hwstr
    ecall
finish:
    li a0, 10
    ecall

.rodata
hwstr: .string "hello, world\n"
```

В регистре `a1` содержится адрес первого символа строки, второй символ строки находится по адресу (<значение a1> + 1) и т.д., но откуда симулятор «знает», *сколько* символов содержится в строке? Ограничителем строки служит терминирующий символ с кодом 0, и директива `.string`, как мы видели, добавляет его в конец строки автоматически.

Полный перечень поддерживаемых системных вызовов приведен на сайте, посвященном симулятору (см. https://jupitersim.gitbook.io/jupiter/assembler/ecalls).