Постановка задачи

Формулировка задания

Выделить определенную вариантом задания функциональность в подпрограмму, организованную в соответствии с ABI (https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md), разработать использующую ее тестовую программу. Адрес обрабатываемого массива данных и другие значения передавать через параметры подпрограммы в соответствии с ABI. Тестовая программа должна состоять из инициализирующего кода, кода завершения, подпрограммы `main` и тестируемой подпрограммы.

Формулировка варианта задания

Перестановка местами элементов массива с четными и нечетными индексами.

Программа для RISC-V

```
.text
start:
.globl start
    lw a3, array_length  # array_length

    la a6, array         # array_ptr

    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:

finish:
    li a0, 10
    ecall

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

Решение задачи

Простейшая программа

Простейшая программа на C состоит из единственной функции `main()`, не имеющей параметров и возвращающей значение типа `int` – код завершения:

```
int main( void ) {
    return 0;
}
```

Приведенная программа может быть транслирована в следующий код на языке ассемблера RISC-V:

```
.text
start:
.globl start
    call main

finish:
    addi a1, a0, 0      # a1 = a0
    addi a0, zero, 17   # a0 = 17
    ecall               # выход с кодом завершения

.text
main:
    addi a0, zero, 0    # a0 = 0
    jalr zero, ra, 0
```

Псевдоинструкция `call` соответствует следующей паре инструкций:

```
auipc ra, %pcrel_hi(main)
jalr ra, ra, %pcrel_lo(main)
```

Исполненные одна за другой, эти инструкции обеспечивают безусловный переход (**jump**) на метку `main` с сохранением адреса следующей за `jalr` инструкции в регистре `ra` (синоним `x1`). Разберемся детально, как это происходит.

При исполнении инструкции `auipc` в целевой регистр (в данном случае, `ra`) помещается сумма текущего значения архитектурного регистра `pc`, то есть адреса самой инструкции `auipc`, и (расширенного знаком в случае RV64 и RV128) значения (`imm` << 12), где `imm` – 20-разрядный непосредственный операнд инструкции. Далее при исполнении инструкции `jalr` к значению регистра-операнда (в данном случае, `ra`) прибавляется расширенное знаком значение 12-разрядного непосредственного операнда инструкции, и полученное значение[1] служит целевым адресом. Таким образом, пара инструкций `auipc+jalr` позволяет выполнить переход на любую инструкцию, размещенную в пределах ~$\pm 2^{31}$ байт относительно первой инструкции пары.

В качестве регистра-результата инструкции `jalr` указан регистр `ra`, поэтому при выполнении перехода в `ra` будет записано значение `pc`+4, где `pc` – текущее значение архитектурного регистра

[1] Это не совсем так, предварительно у полученной суммы обнуляется младший разряд.

pc, то есть адрес самой инструкции jalr. Учитывая, что jalr кодируется 4 байтами, в ra будет записан адрес *следующей за* jalr инструкции.

Поскольку auipc является первой инструкцией программы, в симуляторе VSim она будет размещена по адресу 0x10008. Далее несложно подсчитать, что инструкция jalr будет размещена по адресу 0x1000c, следующая за ней инструкция addi – по адресу 0x10010 (этот адрес будет соответствовать метке finish), а метке main будет соответствовать адрес 0x1001c. Таким образом, безусловный переход на main обеспечивается следующей парой инструкций:

```
auipc ra, 0       # ra = pc + (0 << 12) = 0x10008
jalr ra, ra, 20   # target = ra + 20 = 0x10008 + 0x14 = 0x1001c
```

при этом в ra записывается значение 0x10010.

Ожидаемо, ассемблер RISC-V предоставляет средства, позволяющие отказаться от «ручного» подсчета адресов – механизм %pcrel_hi/%pcrel_lo. В первом приближении[2], значениями %pcrel_hi и %pcrel_lo являются константы, обеспечивающие формирование правильных адресов в парах инструкций auipc+jalr, auipc+addi, auipc+l<*x*>, auipc+s<*x*> (здесь l<*x*>, s<*x*> обозначены инструкции обращения к памяти). Кроме того, определены псевдоинструкции, соответствующие этим парам: call (а также tail), и уже знакомые нам la[3], l<*x*>, s<*x*>.

Справедливость приведенных рассуждений подтверждает пошаговое исполнение программы в симуляторе:

```
>>> locals
trivial.s
start [text] @ 0x00010008
finish [text] @ 0x00010010
main [text] @ 0x0001001c
>>> breakpoint 0x10008
>>> c
>>> s
FROM: trivial.s
PC [0x00010008] CODE:0x00000097    auipc ra, 0 » auipc x1, 0
>>> printx ra
x1 (ra) [0x00010008] {= 65544}
>>> s
FROM: trivial.s
PC [0x0001000c] CODE:0x014080e7    jalr ra, ra, 20 » jalr x1, x1, 20
>>> printx ra
x1 (ra) [0x00010010] {= 65552}
>>> printx pc
PC [0x0001001c]
```

[2] Фактически, корректные значения подставляются позже – на этапе компоновки программы. Этот вопрос будет рассмотрен в следующем разделе курса.

[3] Псевдоинструкция la может транслироваться в пару auipc+addi или lui+addi, в зависимости от параметров командной строки ассемблера и использованных директив.

Итак, в результате «исполнения» псевдоинструкции `call` управление будет передано на метку `main`, а в регистр `ra` будет записан адрес следующей за `call` инструкции. Телу функции `main()` в приведенной программе соответствуют две инструкции, следующие за меткой `main`. Первая инструкция обеспечивает запись значения 0 – возвращаемого функцией значения - в регистр `a0`, в соответствии с требованиями ABI (подраздел “Integer Calling Convention”). Вторая инструкция обеспечивает безусловный переход на адрес, находящийся в регистре `ra` (поскольку регистром-результатом служит `zero`, значение `pc+4` «теряется»). Поскольку в момент исполнения этой инструкции в регистре `ra` записан адрес инструкции, следующей за `call`, следующей будет исполнена именно она.

```
>>> s
FROM: trivial.s
PC [0x0001001c] CODE:0x00000513    addi a0, zero, 0 » addi x10, x0, 0
>>> printx ra
x1 (ra) [0x00010010] {= 65552}
>>> s
FROM: trivial.s
PC [0x00010020] CODE:0x00008067    jalr zero, ra, 0 » jalr x0, x1, 0
>>> printx pc
PC [0x00010010]
>>> FROM: trivial.s
PC [0x00010010] CODE:0x00050593    addi a1, a0, 0 » addi x11, x10, 0
```

Таким образом, псевдоинструкция `call` обеспечивает «передачу управления» (**call**) подпрограмме `main`, а последняя инструкция подпрограммы – «возврат управления» (**return**) в точку вызова, при этом возвращаемое подпрограммой значение находится в регистре `a0`.

Следующие инструкции обеспечивают завершение работы программы с формированием кода завершения. Кодом завершения является значение, возвращаемое подпрограммой `main`.

Следует отметить, что в данном случае использование двух инструкций для вызова `main` неоправданно: как показывают наши вычисления, разность целевого адреса и значения `pc` может быть закодирована 20-разрядным непосредственным операндом инструкции `jal`:

```
jal ra, main  # или просто jal main
```

Однако в общем случае 20-разрядный непосредственный операнд обеспечивает возможность перехода в пределах $\sim\pm2^{19}$ байт, что соответствует $\sim\pm2^{17}$ инструкций (до $\sim\pm2^{18}$ при использовании расширения “C”). В общем случае, это слишком строгое ограничение даже для программ умеренного размера. С другой стороны, в программах часто осуществляется переход между «рядом расположенными» подпрограммами, и экономия одной инструкции в таких случаях желательна. Средства разработки RISC-V обеспечивают указанную оптимизацию *во время компоновки* (**link-time**) программы, поэтому мы вернемся к этому вопросу в следующем разделе курса.

Использование нескольких исходных файлов

Очевидно, далее нам потребуется модифицировать подпрограмму `main`, реализовав в ней функциональность тестовой программы. В то же время, код, обеспечивающий вызов `main` и завершение работы, может использоваться «как есть» в самых разных программах. Учитывая это, мы разобьем текст программы на 2 файла: `setup.s` и `main.s`.

```
# setup.s
.text
start:
.globl start
    call main

finish:
    mv a1, a0  # a1 = a0
    li a0, 17  # a0 = 17
    ecall      # выход с кодом завершения
```

Попутно мы использовали псевдоинструкции `mv` и `li`, предоставив ассемблеру возможность выбрать подходящие последовательности инструкций для копирования значения регистра ("mv" - от "**move**") и загрузки в регистр константы.

```
# main.s
.text
main:
.globl main
    li a0, 0  # a0 = 0
    ret       # jalr zero, ra, 0
```

В "main.s" также использована псевдоинструкция `li`, а также новая псевдоинструкция `ret`, эквивалент которой указан в комментарии.

Следует обратить внимание на директиву `.globl main`. При разбиении текста программы на несколько файлов удобно (а при разработке реальных программ – жизненно необходимо) иметь возможность использовать в разных файлах одинаковые метки, то есть метки должны быть «локальными». С другой стороны, очевидно, на *некоторые* метки, определенные в одном файле, необходимо ссылаться из других файлов. Противоречие разрешается следующим образом: все метки по умолчанию являются локальными, директива `.globl` делает метку глобальной; при использовании метки ассемблер выполняет поиск среди меток, определенных в файле, если метка не найдена, предполагается, что она является глобальной. Поскольку симулятор VSim является assemble-and-go системой, он реализует как функции ассемблера, так и функции *компоновщика* (*редактора связей*, **linker**)[4], в частности, разрешение (**resolution**) глобальных символов: если из текста "main.s" удалить директиву `.globl`, загрузка программы в симулятор окончится с ошибкой.

Для запуска симулятора в командной строке необходимо указать названия всех файлов, содержащих исходный текст программы:

```
java -jar V-Sim-2.0.2.jar -start start –debug setup.s main.s
```

Следует обратить внимание на использование опции командной строки "-start". По умолчанию предполагается, что исполнение программы начинается с метки "main", поскольку теперь такая метка определена и является глобальной, запуск симулятора без указания в качестве точки входа (**entry point**) метки `start` приведет к неожиданным результатам.

[4] А также *загрузчика* (**loader**) и *отладчика* (**debugger**)

Задание нескольких исходных файлов при запуске симулятора примерно соответствует их конкатенации (с учетом сказанного выше). Содержимое секций “text” (“data” и др.) из разных файлов объединяется аналогично тому, как это было в случае одного файла. Локальные символы, отображаемые командой `locals`, группируются по исходным файлам. В выводе команды `s` указывается название файла, содержащего выполненную инструкцию (см. строки “FROM:” выше).

Каркас подпрограммы

Непосредственно из формулировки задания можно видеть, что разрабатываемая подпрограмма принимает следующие параметры: адрес обрабатываемого массива (то есть адрес его 0-го элемента) и его длину. Также легко понять, что подпрограмма не имеет возвращаемого значения – она вызывается не для расчета некоторого значения, а ради «*побочного эффекта*» (**side effect**) – изменения содержимого памяти. Остается еще определить *тип* элемента, поскольку в задании речь идет о рефакторинге уже имеющегося кода, мы будем использовать тип `unsigned`.

Определившись с перечнем параметров, возвращаемых значений и их типами, легко записать прототип (**prototype**) разрабатываемой подпрограммы на языке C:

```
void swap_pairs( /*inout*/ unsigned *array, size_t array_length );
```

Как уже обсуждалось ранее, тип первого параметра (`unsigned *` - указатель на `unsigned`) заключает в себе следующую информацию: во-первых, значением параметра является адрес, во-вторых, это адрес числа типа `unsigned`. Блочный комментарий `/*inout*/` не имеет какого-либо значения *для компилятора* (это комментарий!) – он предназначен для читателя, и информирует его о том, что содержимое памяти, на которую указывает `array`, используется (“in”) и модифицируется (“out”) подпрограммой. Заметим попутно, что и названия, выбранные для подпрограммы и ее параметров не важны для компилятора (пока они не совпадают с ключевыми словами, не дублируются и т.п.), но важны для читателя.

Тип второго параметра – *некоторый* целочисленный беззнаковый тип, разрядность которого позволяет представить размер любого объекта, в том числе *составного* (**aggregate**). В данном случае, в подпрограмму передается не *размер* массива в памяти, а его длина – число элементов[5], но наиболее подходящим для данного параметра типом все равно является `size_t`. Конкретный базовый тип, соответствующий `size_t`, определен в заголовочном файле “stddef.h”, поэтому для корректной обработки приведенного прототипа компилятором файл “stddef.h” должен быть *включен* (явно, с помощью директивы `#include`, или неявно – в результате включения других файлов или с помощью параметров командной строки компилятора) в исходный текст.

В RISC-V, как и во многих других архитектурах, адрес размещается в регистре общего назначения (вспомним инструкции `l<x>`, `s<x>`, `jalr`). Из этого следует также, что разрядность регистра общего назначения в точности подходит для кодирования размера объекта, то есть тип `size_t` имеет разрядность регистра. Таким образом, в соответствии с ABI, при вызове подпрограммы ее первый параметр будет находиться в регистре `a0`, второй – в регистре `a1`. Как мы уже видели, адрес возврата будет находиться в регистре `ra`. Таким образом, “каркас” разрабатываемой подпрограммы запишется следующим образом:

[5] Следует иметь в виду, что в некоторых архитектурах (со словной адресацией памяти) длина массива элементов типа `unsigned` может совпадать с его размером.

```
# swap_pairs.s
.text
swap_pairs:
.globl swap_pairs
    # в a0 – адрес 0-го элемента массива чисел типа unsigned
    # в a1 – длина массива

    # TODO: перестановка элементов массива

    ret       # jalr zero, ra, 0
```

Мы поместили текст подпрограммы в отдельный файл, и чтобы к ней можно было обратиться из других файлов, объявили символ `swap_pairs` экспортируемым.

Следует особо отметить, что *статическая типизация* (**static typing**) – это концепция высокоуровневых языков программирования. На уровне ассемблера (и на уровне системы команд) информация о типе содержимого регистра или некоторой области памяти в явном виде отсутствует. В приведенном тексте только из комментариев можно понять, что при входе в подпрограмму разряды регистра `a0` должны интерпретироваться, как *адрес*, а разряды регистра `a1` – как целое *без знака*.

Тестовая программа

Теперь вернемся к исходному файлу “main.s”, и запишем тестовую программу:

```
# main.s
.text
main:
.globl main
    la a0, array          # }
    lw a1, array_length   # } swap_pairs( array, array_length );
    call swap_pairs       # }

    li a0, 0  # }
    ret       # } return 0;

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

Следует особо отметить, что в приведенном тексте не содержится какой-либо информации о том, сколько параметров имеет вызываемая подпрограмма `swap_pairs`, и каковы типы этих параметров; это означает, что ассемблер не имеет какой-либо возможности предотвратить ошибки, связанные с неправильным вызовом подпрограммы. Например, в приведенном коде мы могли по ошибке поместить адрес массива в регистр `a1`, а его длину – в регистр `a0`, забыть установить значение одного из регистров и пр., эти ошибки можно было бы обнаружить только в результате тщательного анализа и/или тестирования программы.

Приведенный код не требует каких-либо дополнительных комментариев, с тем исключением, что в нем имеется ошибка (**bug**)!

Ошибка станет очевидной, если вспомнить, во что транслируются псевдоинструкции `call` и `ret`:

```
main:
    …
    # call swap_pairs
    auipc ra, %pcrel_hi(swap_pairs)
    jalr ra, ra, %pcrel_lo(swap_pairs)
    …
    # ret
    jalr zero, ra, 0
```

При входе в `main` адрес возврата находится в регистре `ra`, и возврат из подпрограммы осуществляется переходом на адрес, содержащийся в этом регистре, с помощью инструкции `jalr`. Однако прежде, чем это произойдет, значение `ra` изменяется в результате «выполнения» псевдоинструкции `call`: в `ra` будет записан адрес возврата для вызываемой подпрограммы `swap_pairs`, то есть адрес следующей за `call` (псевдо)инструкции, в данном случае - инструкции возврата из подпрограммы `main`. Таким образом, результатом выполнения инструкции возврата, соответствующе псевдоинструкции `ret`, будет переход на эту же инструкцию – программа зациклится!

Решение указанной проблемы состоит в следующем: исходное значение `ra` следует сохранить перед псевдоинструкцией `call`, и восстановить перед псевдоинструкцией `ret`. Значение регистра можно сохранить либо в другом регистре, либо в памяти. Как известно, регистры делятся на две группы: рабочие (**temporary**, **caller-saved**) и сохраняемые (**callee-saved**). Значение `ra` нельзя сохранить в рабочем регистре, так как значение этого регистра может быть изменено вызываемой подпрограммой[6], а чтобы сохранить значение `ra` в сохраняемом регистре, значение самого этого регистра необходимо сохранить и восстановить перед возвратом из `main`[7]. Таким образом, значение `ra` следует сохранить в памяти, и естественно использовать для этого стек.

```
main:
.globl main
    addi sp, sp, -16  # выделение памяти в стеке
    sw ra, 12(sp)     # сохранение ra

    #                  >-----------> увеличение адреса
    # структура кадра: |XXXX|XXXX|XXXX|=ra=| …
    #                   ^sp            ^(sp + 12)

    …

    lw ra, 12(sp)     # восстановление ra
    addi sp, sp, 16   # освобождение памяти в стеке
```

[6] Следует отметить, что на данном этапе мы еще не приступили к разработке подпрограммы, и, действительно, не можем определить, какие рабочие регистры она будет использовать.

[7] Текущая версия «установочного кода» (исполняемого до и после `main`) не использует сохраняемые регистры, однако они могут начать использоваться в следующих версиях, в этом случае изменение значения сохраняемого регистра в `main` приведет к ошибке, которую будет очень трудно обнаружить.

```
    ret
```

В случае 32-разрядной версии RISC-V для сохранения значения `ra` в стеке требуется только 4 байта, однако ABI RISC-V требует выравнивания указателя стека на границу 128 разрядов (16 байт), следовательно, величина изменения указателя стека должна быть кратна 16. Кроме того, в RISC-V (как и в большинстве архитектур) стек *растет вниз* (**grows downwards**), то есть *выделению памяти в стеке* (**stack allocation**) соответствует *уменьшение* значения указателя стека. Отметим, что начальное значение `sp` устанавливается симулятором.

В ABI RISC-V регистр `sp` является сохраняемым, то есть при возврате из подпрограммы он должен иметь исходное значение. Поскольку для выделения памяти в стеке значение `sp` уменьшается (в данном случае на 16), перед возвратом из подпрограммы достаточно увеличить `sp` на ту же величину.

Подпрограмма

Тело подпрограммы легко *выделить* (**extract**) из имеющейся программы, однако в отличие от программы, где используемые регистры могли выбираться произвольно, в подпрограмме разумно использовать регистр `a0` для указателя на массив (вместо `a6`) и регистр `a1` для длины массива (вместо `a3`). С учетом этих изменений, подпрограмма приобретает следующий вид:

```
# swap_pairs.s
.text
swap_pairs:
.globl swap_pairs
    # в a0 – адрес 0-го элемента массива чисел типа unsigned
    # в a1 – длина массива

    srli a7, a1, 1        # pair_count
    slli a7, a7, 3        # a7 = pair_count << 3 = ( pair_count * 2 ) * 4
    add a7, a0, a7        # end_ptr

    beq a0, a7, loop_exit  # if( array_ptr == end_ptr ) goto loop_exit
loop:
    lw t1, 0(a0)          # t1 = * array_ptr
    lw t0, 4(a0)          # t0 = * ( array_ptr + 1 )
    sw t1, 4(a0)          # * ( array_ptr + 1 ) = t1
    sw t0, 0(a0)          # * array_ptr = t0

    addi a0, a0, 8        # array_ptr += 2
    bne a0, a7, loop      # if( array_ptr == end_ptr ) goto loop
loop_exit:

    ret
```

Следует отметить, что в нашей программе не использовались регистры, зарезервированные ABI для специального применения (регистры `x1`-`x4`), и сохраняемые регистры (`x8`-`x9`, `x18`-`x27`). Если бы это было не так, при выделении подпрограммы потребовалось бы внести дополнительные изменения в исходный текст: использовать вместо `x1`-`x4` другие регистры и либо использовать вместо сохраняемых регистров рабочие, либо записывать значения используемых сохраняемых регистров в стек и восстанавливать их перед возвратом из подпрограммы.