**Отчет №4.**

*13. Суть алгоритма сортировки.*

Изначально я хотела использовать алгоритм сортировки бинарным деревом, но поняла, что это не рационально, так как в отчете нужно разбирать работу со стеком, а не с кучей. Поэтому я использую менее эффективный по времени и более простой алгоритм - «Гномья» сортировка (или обменами), не требующий вложенных циклов.

Её суть в том, что мы шагаем по массиву слева направо и сравниваем 2 соседних элемента. Если предыдущий меньше текущего, то делаем шаг вперед, иначе меняем элементы местами и делаем шаг назад. При этом если мы шагали назад, то в следующий раз мы не будем снова проверять все пройденные элементы, а вернемся на ту позицию, где остановилось движение вперед. Сортировка закончена, когда мы дошли до последнего элемента массива.

Сложность сортировки по времени

|  |  |
| --- | --- |
| Лучшая | O(n) |
| Средняя | O(n2) |
| Худшая | O(n2) |

*14.* *Напишите функцию, которая возвращает элемент, который был бы на j-й позиции в отсортированном варианте массива.*

*Объясните, как работает эта функция; как достигается временная сложность О(n). Поясните, какой случай является "худшим".*



Функция getOrderStatistic вызывает другую функцию quickselect, которая производит непосредственное нахождение элемента. Параметрами этой функции являются: сам массив, номер первого элемента и номер последнего, номер элемента по порядку, который мы хотим найти.

Если первый элемент совпал с последним, значит, это единственный элемент в массиве, и мы возвращаем его. Если нет, тогда случайным образом мы находим «опорный» элемент из выбранного промежутка (то есть сначала из всего массива) и раскидываем элементы слева от опорного – меньшие, справа - большие.



В функции partition параметрами являются массив, границы промежутка и номер опорного элемента. В ней происходит проход по всему промежутку слева направо и проверяется, меньше / равен ли элемент опорному. Если да, то элемент отправляется в левую часть (меньшие элементы кладутся в начало друг за другом слева направо). В quickselect возвращается новая позиция опорного элемента. Если она совпала с тем номером, что мы искали, то такой элемент массива возвращается в getOrderStatistic. Если он меньше нужного, то уменьшаем промежуток справа, если больше, то слева, и снова проводим выбор опорного элемента и перемещение меньших элементов налево.

Выгода этого алгоритма в том, что каждый раз выбирая новый промежуток, не нужно заново проходить по всему массиву, нужно перемещать элементы только в пределах промежутка, в нем всегда находится искомый элемент. Временная сложность сильно зависит от выбираемого опорного элемента. Худший вариант временной сложности получается, когда, например, нужно найти наибольший элемент в отсортированном массиве, а опорный элемент выбран наименьший. Тогда она равна O(n2). В среднем же достигается временная сложность O(n) таким образом: опорный элемент разбивает массив в среднем на примерно 2 равные части, то есть каждая следующая итерация проводится с половиной данных предыдущего шага. Тогда временная сложность:

Например, поиск 1-го по порядку элемента (отсчет начинается с нуля):

{20, 15, 3, 21, 4, 15, 1, 78, 2, 35, 76, 45, 11}

1. Случайным образом выбираем опорный элемент №4, равный 4. Промежуток от 0 до 12. Сравнивая каждый с опорным, получаем:

{ 3, 1, 2, 4, 11, 15, 15, 78, 20, 35, 76, 45, 21}

Опорный элемент переместился на позицию 3, что меньше искомого j=1, поэтому продолжаем поиск. Новая правая границ будет (позиция опорного элемента – 1) = 2

1. Теперь промежуток от 0 до 2. Номер нового опорного элемента из промежутка равен 0:

{ 3, 1, 2}

Сравнивая, получаем:

{ 2, 1, 3}

Опорный элемент переместился на позицию 2, что меньше искомого j=1, поэтому продолжаем поиск, уменьшив правую границу на 1.

1. Промежуток от 0 до 1. Номер опорного элемента 1.

{ 2, 1}

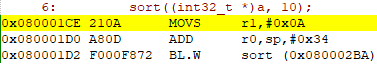
Получаем:

{1, 2}

Опорный элемент переместился на позицию 0, что больше искомого j=1, поэтому продолжаем поиск. Увеличиваем левую границу промежутка на 1.

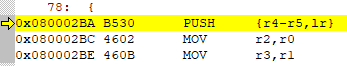
1. Промежуток от 1 до 1, то есть, он состоит из одного элемента, который мы и возвращаем. Результат равен 2.
2. *Разберите дизассемблерный листинг вашей программы, обращая основное внимание на следующие моменты:*

*а) Передача управления в функцию.*





BL.W производит переход на адрес первой инструкции функции *sort* 0x080002BA (W – использование 32-битной команды), при этом в LR сохраняется адрес следующей инструкции после BL, чтобы вернуться на нее после выполнения функции, управление передается в sort.





Затем происходит сохранение контекста в стек (см. пункт 2.)

*б) Возврат управления из функции.*

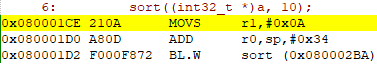


При выходе из функции sort контекст восстанавливается, т.е. сохраненные значения из стека помещаются в регистры (см. пункт 2). Когда восстанавливается адрес возврата в регистр LR, происходит перемещение обратно в main и управление передается ей.

*в) Передача параметров и возврат результата.*

В main параметров нет, из нее результат возвращается return 0; При этом 0 помещается в регистр r0.

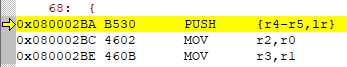
Для функции sort:





В функции main происходит вызов функции сортировки массива sort с параметрами: указатель на массив функции и его размер.

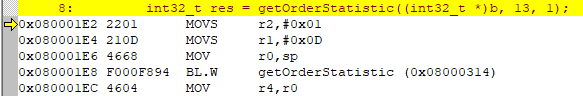
В r1 помещается размер массива (10), а в r0 адрес sp+0x34 =0x20004524, то есть адрес *а[0].*



Чтобы стать локальными переменными, из r0 и r1 параметры помещаются в r2 и r3 соответственно.

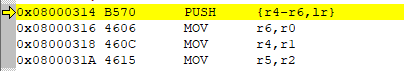
Так как функция sort имеет тип void, она ничего не возвращает.

Для функции getOrderStatistic:



Параметры функции – массив, количество элементов в нем, номер искомой j-й порядковой статистики.

В r2 помещается j = 1, в r1 количество элементов массива =13, в r0 адрес текущей вершины стека, по которому находится первый элемент массива b. BL.W – переход в функцию getOrderStatistic с сохранением адреса возврата.



После сохранения контекста происходит преобразование параметров в локальные переменные: из регистров r0, r1, r2 данные помещаются в r6, r4, r5 соответственно.





В функции результат поиска порядковой статистики записался в регистр r0. После восстановления контекста при передаче управления в функцию main, чтобы результат был присвоен переменной res, из r0 значение перемещается в r4 и становится значением локальной переменной.

В getMedian передача параметров и возврат результата происходят аналогично.

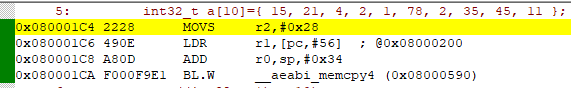
*г) Сохранение и восстановление контекста.*

Чтобы не потерять данные, используемые в функции main (например), при входе в другую функцию нужно сохранить контекст. Сохранение и восстановление производит вызываемая функция, а не main. См. ответ на пункт 2 д)).

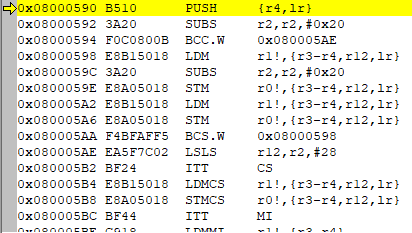
*д) Выделение памяти под локальные переменные, массивы и структуры (далее "локальные объекты").*

Локальные переменные сохраняются в регистры, память для них в стеке не выделяется.

Массив помещается в стек как локальный объект один раз в функции main, затем происходит обращение к этим ячейкам из других функций. Выделение памяти в main для массива:

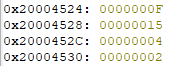


Здесь возникла трудность с тем, что помещается в r2. Предполагаю, что 40 – это количество байт, выделяемых под массив, и впоследствии проверяется, когда все элементы будут помещены и нужно будет выйти из встроенной функции. В r1 загружается адрес константы 0x000F. В r0 помещается адрес, в котором будет находиться последний элемент массива. Затем происходит переход на следующую функцию с сохранением адреса возврата в LR.



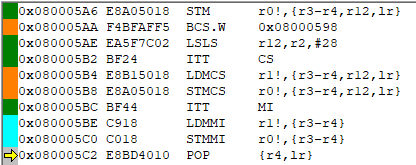
Это дизассемблерный листинг функции \_\_aeabi\_memcpy4

LDM загружает по порядку значения элементов массива в регистры r3 = 15, r4 = 21, r12 = 4 и LR = 2, в r1 находится адрес константы 1.



STM помещает в стек первые 4 элемента массива из регистров.

LDM помещает в регистры следующие 4 элемента массива, STM загружает их в память.



LDMI и STMI так же загружают в регистры и помещают в память последние 2 элемента.

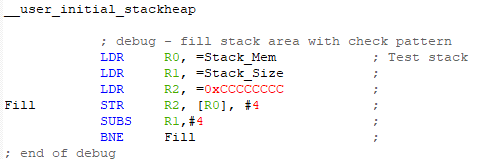
*е) Организацию отдельных стековых кадров для каждого экземпляра функции.*

См. пункт 2. При каждом вызове функций структура стековых кадров не меняется.

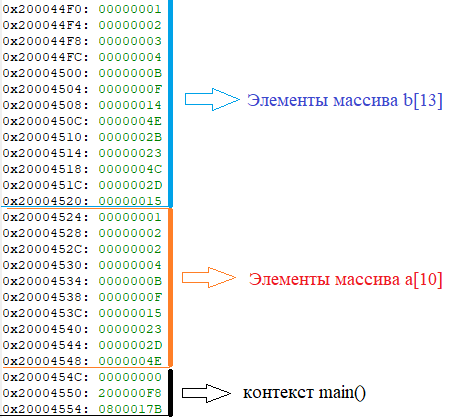
1. *Для каждой функции (в том числе, для функции main) изобразите структуру стекового кадра (в виде таблицы), отметьте на ней назначение и содержимое элементов стекового кадра. Для этого воспользуйтесь окном memory.  
   Так как каждый элемент стека имеет размер 4 байта, при этом удобно использовать режим просмотра памяти unsigned int и выровнять ширину окна memory так, чтобы память выглядела как столбик чисел.*

*а) Покажите, где в стековом кадре функции расположены локальные объекты и что это за объекты.*

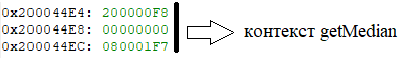
/\* Чтобы было удобно видеть, какая память выделена под стек, и сразу обнаружить его переполнение, на этапе компиляции все ячейки заполняются 0xCCCCCCCC вот таким образом:

\*/

Стековый кадр для функции main:



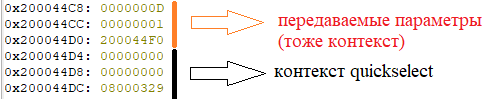
Стековый кадр функции getMedian (имеется в виду контекст, сохраненный для main):



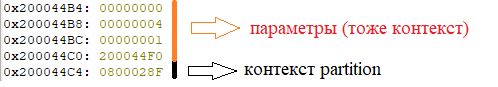
Стековый кадр функции getOrderStatistic (имеется в виду контекст, сохраненный для main):



Стековый кадр функции quickselect (имеется в виду контекст, сохраненный для getOrderStatistic), параметры являются локальными объектами:



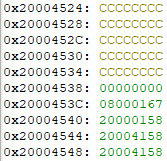
Стековый кадр функции partition (имеется в виду контекст, сохраненный для quickselect), параметры являются локальными объектами:



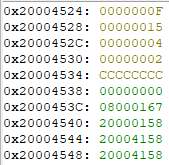
Конкретно про то, какие локальные объекты сохраняются, см. пункт д), так как все они являются контекстом. Те локальные переменные, которые не сохраняются в контексте, хранятся в регистрах.

*б) Покажите, как происходит выделение памяти для локальных объектов в стековом кадре.*

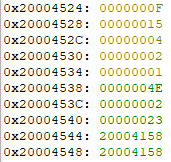
В функции main для массива a[10] (соответственно пункту 1 д)):



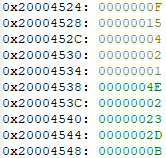
До помещения элементов массива в стек соответственная часть стекового кадра выглядит так.



После выполнения STM в стек помещены первые 4 элемента.



После следующей STM в стек помещен еще 4 элемента.

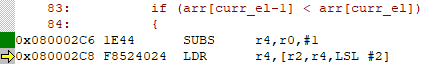


STMMI помещает последние 4 элемента.

*в) Покажите, как происходит освобождение памяти, выделенной под локальные объекты.*

При выходе из функции значения переменных остаются в стеке (или в регистрах), но они перестают принадлежать каким-либо переменным, и в любой момент могут быть заменены новыми. При этом меняется положение указателя на вершину стека – он сдвигается к объектам, работа с которыми продолжится.

*г) Как происходит обращение к локальным объектам в стековом кадре?*

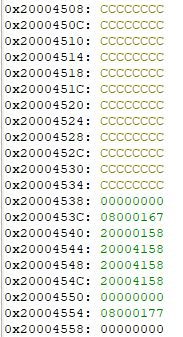
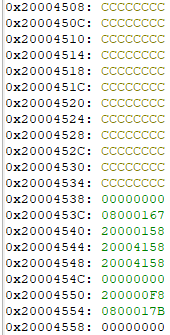


Здесь происходит обращение к нулевому элементу массива. SUBS помещает в r4 номер элемента (0), в r2 хранится адрес 0-го элемента массива. LDR загружает в r4 значение нулевого элемента: из адреса (r2+r4\*4). Умножаем на 4, так как все элементы стека по 4 байта.

*д) В каком месте стекового кадра будет расположен сохраняемый контекст вызова? В каком порядке будут сохранены регистры, в каком порядке они будут восстановлены? Какие команды будут для этого использованы?*

Контекст сохраняется в начале стекового кадра (в меньшие адреса). При записи нескольких регистров в стек регистры с меньшим номером записываются в меньшие адреса, LR всегда записывается первым из них (в самом “низу” стека).

При входе в функцию main стековый кадр выглядит так (до и после выполнения 0x080001C0 B530 PUSH {r4-r5, lr}):

До  После 

SP

SP

Порядок записи в стек при сохранении контекста:

1. Указатель стека уменьшается на 4 (т.к. размер любого регистра 32 бита), становится 0x 20004554.

2. Записывается значение регистра LR (самый старший по номеру).

3. Указатель стека уменьшается на 4, 0x20004550.

4. Записывается значение регистра R5

5. Указатель стека уменьшается на 4, 0x2000454С.

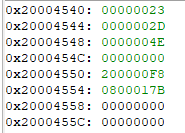
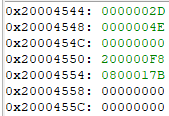
6. Записывается значение регистра R4.

|  |  |
| --- | --- |
| R4 | 0x00000000 |
| R5 | 0x200000F8  Стековый кадр функции main |
| LR (R14 – адрес возврата из main) | 0x0800017B |
| Контекст вызова main | |

Восстановление контекста:

При чтении из стека используется команда POP, порядок действий обратный в отличие от PUSH – сначала значение из ячейки помещается в регистр, затем адрес SP увеличивается на 4.

Стековый кадр до и после выполнения POP {r4-r5,pc}

До*:* После:

SP

SP

Порядок записи в регистры при восстановлении контекста:

1. Запишем элемент по адресу SP в регистр R4.

2. Увеличим указатель на 4: 0x20004550.

3. Запишем элемент из нового SP уже вершины в R5

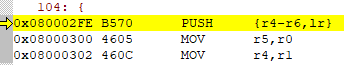
4. Увеличим указатель на 4: 0x20004554

5. Запишем элемент из SP в регистр PC (R15), и сразу перейдем на команду, стоящую после вызова main.

6. Увеличим указатель на 4: 0x20004558

При входе в функцию GetMedian контекст сохраняется так:



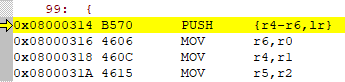


В r4 лежит значение локальной переменной из main res, в r5 адрес переменной med, в r6 0. Эти значения загружаются в стек.

В r1 лежит 13 – размер массива, записывается в r4 как локальная переменная. В r0 лежит адрес первого элемента массива, который становится локальным и теперь лежит в стеке, этот адрес записывается в r4.

|  |  |
| --- | --- |
| R4 (res) | 0x00000002 |
| R5 (адрес med) | 0x200000F8  Стековый кадр функции getMedian |
| R6 | 0x00000000 |
| LR (R14 – адрес возврата из getMedian) | 0x080001F7 |
| Контекст вызова getMedian | |

При входе в функцию getOrderStatistic контекст сохраняется так:



В r5 лежит адрес локальной переменной из main med, в r4 значение res, в r6 0, в lr адрес, на который нужно вернуться после выполнения функции. Эти значения загружаются в стек командой PUSH.

В r1 лежит 13 – размер массива, записывается в r4 как локальная переменная.

В r0 лежит адрес первого элемента массива b, который становится локальным и теперь лежит в стеке, этот адрес записывается в r6. В r2 лежит j – номер порядковой статистики, который мы ищем. Он помещается в r5.

|  |  |
| --- | --- |
| R4 (res) | 0x00000000 |
| R5 (адрес med) | 0x200000F8  Стековый кадр функции getOrderStatistic |
| R6 | 0 |
| LR (R14 – адрес возврата из getOrderStatistic) | 0x080001ED |
| Контекст вызова getOrderStatistic | |

Восстановление контекста аналогично описанному выше, функцией POP.

При входе в функцию quickselect контекст сохраняется аналогично:



|  |  |
| --- | --- |
| R4 (size) | 0x0000000D |
| R5 (j) | 0x00000001 |
| R6 (адрес arr) | 0x200044F0  Стековый кадр функции quickselect |
| R7 (res) | 0x00000000 |
| R8 | 0x00000000 |
| LR (R14 – адрес возврата) | 0x08000329 |
| Контекст вызова quickselect | |

*е) Покажите, как стек расположен в памяти; в какую сторону он растет.*

Стек – массив, вершина которого хранится в SP и меняется при помещении в стек новых данных. По умолчанию стек растет в сторону кучи, в сторону уменьшения адресов.

|  |  |  |
| --- | --- | --- |
| Статическая область памяти | Куча | ←Стек |

*ж) Определите, сколько вложенных вызовов функций происходит в вашей программе? Схематично изобразите последовательность стековых кадров для момента самого глубокого вложенного вызова.*

Вызов программы→ main→getOrderStatistic→quickselect→partition

Происходит 4 вложенных вызова.

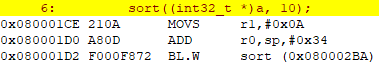
1. *Измените уровень оптимизации на Level 1 (Project -> Options for Target -> C/C++ -> выпадающее меню Optimization). Сравните дизассемблированный код с тем, что получился при отсутствии оптимизации. Как изменился дизассемблерный листинг при включении оптимизации?*

Например, разница в функции partition – нет лишнего помещения переменных в регистры при оптимизации:

|  |  |
| --- | --- |
| Level 0 | Level 1 |
| 0x08000204 B5F0 PUSH {r4-r7,lr}  0x08000206 4604 MOV r4,r0  0x08000208 460D MOV r5,r1  10: int32\_t pivot = a[pivotIndex];  11: // Move pivot to end  0x0800020A F8547023 LDR r7,[r4,r3,LSL #2]  12: int32\_t temp = a[pivotIndex];  0x0800020E F8546023 LDR r6,[r4,r3,LSL #2]  13: a[pivotIndex] = a[right];  0x08000212 F854C022 LDR r12,[r4,r2,LSL #2]  0x08000216 F844C023 STR r12,[r4,r3,LSL #2]  14: a[right] = temp;  15: // elements less than pivot will be pushed to the left of pIndex  16: // elements more than pivot will be pushed to the right of pIndex  17: // equal elements can go either way  0x0800021A F8446022 STR r6,[r4,r2,LSL #2]  18: uint32\_t pIndex = left;  19: uint32\_t i;  20: // each time we find an element less than or equal to pivot, pIndex  21: // is incremented and that element would be placed before the pivot.  0x0800021E 4628 MOV r0,r5 | 0x08000200 B570 PUSH {r4-r6,lr}    10: int32\_t pivot = a[pivotIndex];  11: // Move pivot to end    12: int32\_t temp = a[pivotIndex];  0x08000202 F8505023 LDR r5,[r0,r3,LSL #2]  13: a[pivotIndex] = a[right];  0x08000206 F8504022 LDR r4,[r0,r2,LSL #2]  0x0800020A F8404023 STR r4,[r0,r3,LSL #2]  14: a[right] = temp;  15: // elements less than pivot will be pushed to the left of pIndex  16: // elements more than pivot will be pushed to the right of pIndex  17: // equal elements can go either way    18: uint32\_t pIndex = left;  19: uint32\_t i;  20: // each time we find an element less than or equal to pivot, pIndex  21: // is incremented and that element would be placed before the pivot.  0x0800020E F8405022 STR r5,[r0,r2,LSL #2] |

1. *В чем разница между передачей параметров по указателю и по значению?*

Например, в функцию sort передается указатель на массив а и переменная size.



Значение size помещается в регистр r1 при помощи MOVS, а адрес первого элемента с командой ADD в r0.

То есть при передаче по указателю передается адрес, а по значению – значение.

1. *Как в языке С в функцию передать одномерный массив? А двухмерный? В чем разница между указателем и массивом, в каких случаях массив "распадается" (decays) в указатель?*

Передача одномерного массива в функцию:

1. void func(int \*ar);
2. void func(int arr[]);
3. void func(int arr[size]);

По сути, передавая массив, на самом деле мы передаем адрес первого элемента.

Передача двумерного массива:

Вообще, в Си нет многомерных массивов, но можно их «симулировать» как комбинации одномерных. Можно передавать двумерный массив как указатель на одномерный

void func (int \*arr)

Динамические можно передавать так: void func (int\*\* arr)

Либо как массивы массивов:

void func (int arr[] []);

При объявлении массива в памяти выделяется место под каждый из элементов массива, а при объявлении указателя нужно место только для него самого, при этом он может указывать на любую переменную/элемент/массив символов.

Массив может «распасться», когда он передается как параметр функции, и неизвестно, сколько элементов в нем содержится. Чтобы этого не происходило, нужно передавать еще и его размер.

1. *В чем смысл ключевого слова const? В чем разница между константным указателем и указателем на константу?*

Const контролирует неизменность какого-либо объекта.

* 1. Константный указатель:

int \*const num = &a - означает, что указатель нельзя менять (нельзя записать в num никакой другой адрес кроме &a)

* 1. Указатель на константу:

int const\* num или const int\* num – означает, что нельзя менять значение по указателю (num = &a - на что указывает num можно менять, но число менять нельзя ~~\*num = 5~~)

1. *Что такое "указатель на функцию"? Как его использовать? Зачем это может быть нужно?*

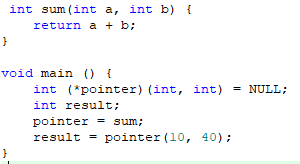
Указатель на функцию представляет собой выражение или переменную, которые используются для представления адреса функции. Адрес, присвоенный указателю, является входной точкой в функцию.

Можно использовать имя функции, как указатель, а можно создать переменную, которая будет указателем на функцию, таким образом:

*тип (\*имя\_указателя) (параметры);*

Указателю на функцию можно присвоить функцию, которая соответствует указателю по возвращаемому типу и параметрам.

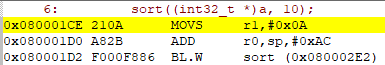
Использование указателей позволяет создавать функции, аргументами которых являются другие функции и уменьшать сложность программы.

Пример использования указателя на функцию: 

pointer - указатель на функцию, которому мы присваиваем адрес функции sum.

Вызов производится так же, как и для обычной функции.

1. *Разберите отличия в дизассемблерном листинге между функцией sort и функцией sortStruct: как сохраняется и восстанавливается контекст, как передаются параметры и как возвращается результат?*



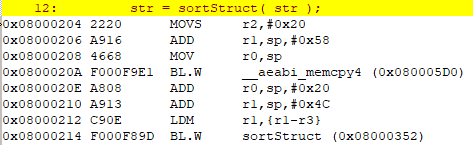
В sort параметры – указатель на массив и размер помещаются в регистры r1, r0.



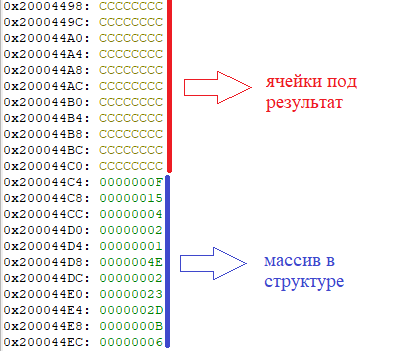
Сохранение контекста и преобразование параметров в локальные переменные (описано ранее).



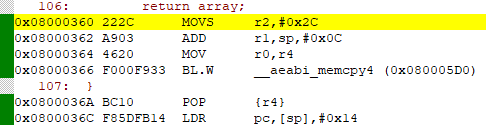
Восстановление контекста производится POP из стека в регистры.



Для sortStruct встроенная функция \_\_aeabi\_memcpy помещает в стек массив, но после этого еще первые 3 элемента массива LDM загружаются в регистры r1,r2,r3.



Сохранение контекста производится теперь двумя командами PUSH. В стеке остается выделенное место под результат функции, указатель на первую пустую ячейку кладется в r4.



Встроенная функция перед восстановлением заполняет пустые ячейки результатом. Восстановление контекста производится POP. И вместо сохранения адреса возврата в LR, он сохранен в стеке. Передача управления в main происходит с помощью извлечения этого адреса из стека и помещения в PC.