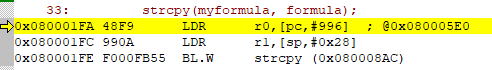
**Отчет №3**

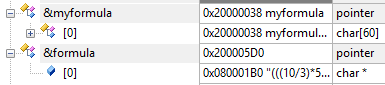
**Часть первая – арифметика**

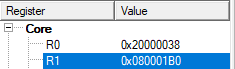
*1.3. Разберите дизассемблерный листинг функции eval, уделите основное внимание ассемблерным инструкциям, с помощью которых реализованы арифметические действия, битовые операции, сравнения и логические операторы языка С.*



Для примера, разберем некоторые команды ассемблера для теста, где присутствуют все арифметические действия.





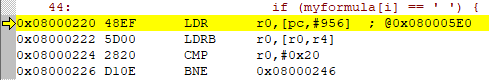


В r0 загружается адрес указателя на массив myformula, в r1 адрес первого элемента formula в стеке.

Затем командой strcpy происходит копирование formula в myformula. BL.W – переход на встроенную функцию по адресу 0x080008AC с запоминанием адреса возврата в регистре r14. Копирование нужно для того, чтобы исходную формулу можно было изменять и таким образом упрощать дальнейшие действия.

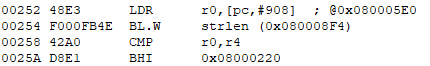


В r4 кладется значение переменной i=0.



Здесь происходит удаление пробелов в формуле, если это необходимо. В r0 загружается сначала адрес первого элемента массива myformula командой LDR, затем происходит обращение к нужному элементу массива, то есть в r0 помещается r0 + i (адрес нулевого + номер нужного элемента = 0x00000038 + 0). Команда CMP сравнивает текущий элемент с пробелом (0x20 – пробел в кодировке ASCII). В данном случае ‘(‘ ≠ ‘ ‘, поэтому выполняется BNE – переход на следующую команду, если условие не выполнилось.



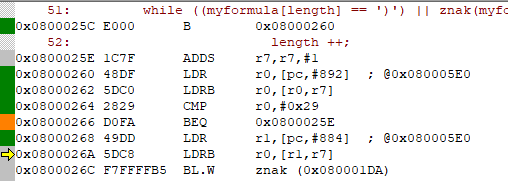


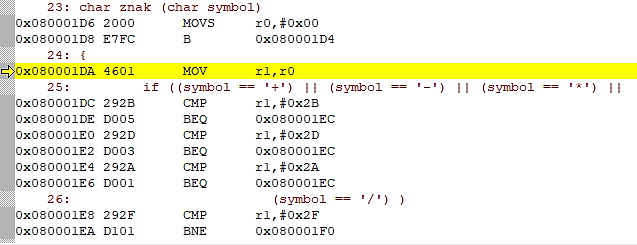
i=i+1;

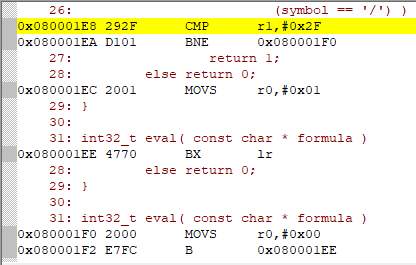
BL.W – переход на встроенную функцию с сохранением текущего адреса команды в регистре. После ее выполнения в r0 находится длина формулы. CMP – сравнение, больше ли длина формулы, чем i. Если да, то по команде BHI происходит переход внутрь цикла.

В этой формуле пробелов нет, поэтому условие никогда не выполнится, просто произойдет выход из цикла.

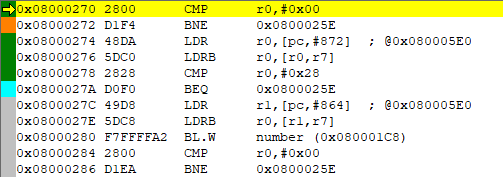
51: while ((myformula[length] == ')') || znak(myformula[length]) || (myformula[length] == '(') || (number(myformula[length])))



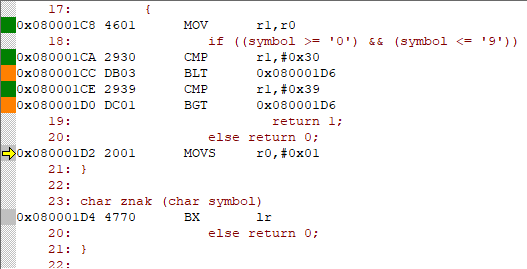
Так как условия внутри while объединяются оператором || (ИЛИ), то проверяются все условия по очереди, пока текущий элемент перестанет быть цифрой или допустимым символом. В r7 находится текущее значение length, в r0 элемент массива по номеру length. Сравнивается, является ли элемент ‘)’ (0x29 по ASCII). Если да, то командой ADDS прибавляем length+1 для проверки следующего символа в массиве. Если нет, то проверяем следующее условие – является ли элемент знаком. Для этого после загрузки в регистры переходим в функцию znak командой BL.W с сохранением адреса текущей команды.



В r1 помещается текущий элемент массива и по очереди сравнивается с каждым знаком командой CMP. Если равенство верно, то командой BEQ выполняется переход на return 1, т.е. в r0 помещается 1. Если ни одно из условий не выполняется, то в r0 помещается 0 и выполняется переход к BX. BX lr – переход по адресу в регистре R14, то есть возврат к циклу while.



Аналогично проверяются следующие 2 условия – открытая скобка (0x28) и число – переход на функцию number.

Функция number: MOV помещает в r1 адрес текущего элемента массива. Так как внутри if условия объединены && (И), то если хоть одно из условий ложно, то сразу происходит выход командой BGT(>) или BLT (<). CMP сравнивает с ‘0’ (0x30). Если меньше ‘0’, то по команде BLT переход на MOVS, который помещает в r0 0.

Если больше ‘9’, то BGT производит переход на MOVS, и в r0 помещается 0. Если оба условия выполнились, то MOVS возвращает истину 1 в r0. BX возвращает назад к текущей инструкции, адрес которой в регистре lr.

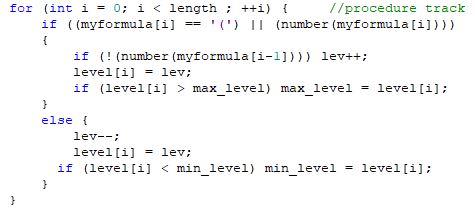


Если все условия истинны, то продолжается поиск конца формулы, и по команде BNE осуществляется переход в начало тела цикла.

Если в r0 находится 0, значит, формула закончилась, и можно выйти из цикла.

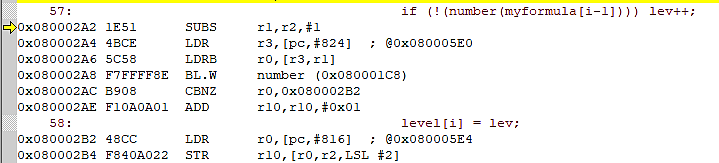


Здесь после удаления пробелов устанавливается новый конец формулы. LDR в r1 загружает адрес ячейки, указывающий на нулевой элемент массива myformula[0]. В r7 хранится длина формулы length. STRB помещает нуль-символ (который всегда находится в r0 после предыдущих действий) в ячейку массива myformula[0+length].



Здесь происходит расстановка порядка действий в формуле следующим образом: для открытых скобок и чисел к номеру действия прибавляется 1, для арифметических действий и закрытых скобок вычитается 1.

Ассемблерные команды для цикла for и условия if аналогичны выше описанным.



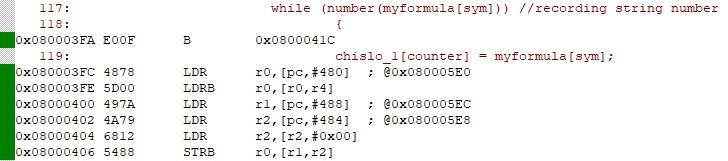
В регистре r2 находится i, в r10 находится переменная lev, команда SUBS помещает в r1 (i - 1). В r3 загружается адрес i-го элемента массива, в r0 значение (i-1) – го. Если в (i-1)-ом элементе число, то функция number возвращает в r0 1, иначе 0.

CBNZ производит переход на команду LDR, если r0 != 0, то есть если было найдено число. Если оно не найдено, то к lev прибавляется 1 командой ADD.

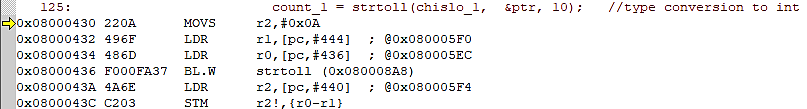
STR по адресу в r0+i\*4(байта) помещает lev.



Команда SUB выполняет вычитание: в r10 хранится lev, (lev – 1) помещается в r10.



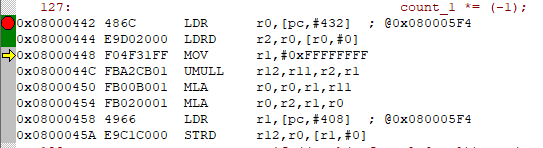
После проверки, цифра ли находится в этом элементе массива, если да, то в r0 помещается адрес элемента. В r1 загружается адрес указателя на массив chislo\_1. В r2 загружается «номер цифры» в числе, и затем STRB сохраняет элемент из адреса в r0 в r1 со сдвигом r2, то есть myformula[sym] в chislo\_1[counter].



В r2 помещается 10 (основание системы счисления). В r1 загружается адрес ptr, куда в случае обнаружения некорректного символа (не цифры) сохраняется его адрес. В r0 загружается адрес указателя на массив chislo\_1. Функция strtoll переводит строку в long long int. STM сохраняет в стеке регистры.



В r0 загружается значение minus\_flag из адреса sp+0x0C. Если 0, то происходит выход из условия if.



В r0 адрес count\_1. В r2 (младшие биты) и r0 (старшие биты) загружается значение count\_1 (суффикс D, так как count\_1 long long int). В r1 помещается -1.

UMULL – целочисленное умножение с 64-битным результатом r2\*r1 = count\_1 \* (-1), результат сохраняется в r12 (младшие биты) и r11 (старшие биты). MLA – умножение с накоплением: r0 = r0\*r1+r11 = старшие биты count\_1 \* (-1) + старшие биты после UMULL, при этом старшие биты результата обрезаются.

r0 = r2\*r1+r0 = младшие биты count\_1 \* (-1) + младшие биты после MLA

Например, в данном случае эта операция используется для count\_1 = A = 10, значит в r0 0, в r2 A

UMULL: r2\*r1=A \* FFFFFFFF = 9 FFFFFFF6

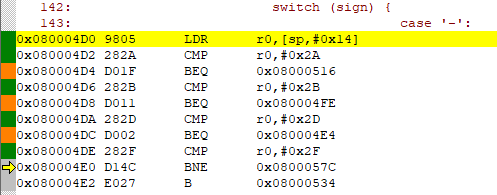
r11 r12

MLA: r0 = r0 \* r1 + r11 = 0 \* FFFF FFFF + 9 = 9

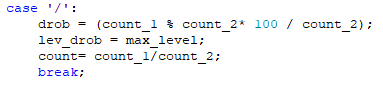
MLA: r0 = r2 \* r1 + r0 = A \* FFFF FFFF + 9 = ~~FFFF FFFF~~ FFFF FFFF

Далее LDR загружает в r1 адрес count\_1. STRD сохраняет значения из r12 (младшие биты результата) и r0 (старшие) в count\_1, получается

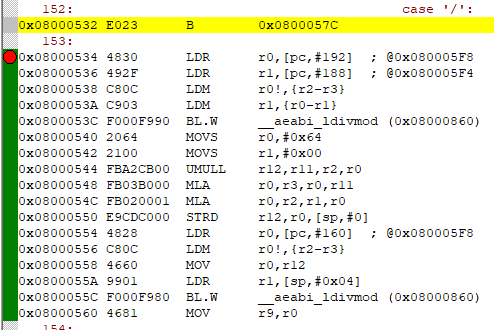
count\_1 = FFFF FFFF FFFF FFF6 = -10



В r0 загружается текущий знак (‘/’ = 2F). И командой CMP сравнивается с каждым из case – ов: 2A = ‘\*’, 2B = ‘+’, 2D = ‘-‘, 2F = ‘/’ и производится переход на соответствующий адрес команды.



При делении сначала вычисляется дробный остаток, в том случае, если он не равен нулю, в дальнейшем он хранится в переменной drob и участвует в вычислениях.

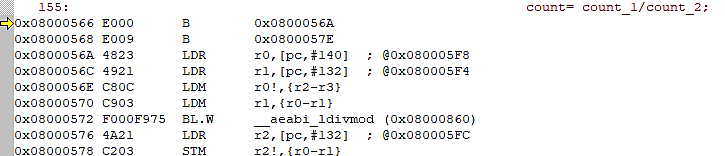


В r0 загружается адрес count\_2, в r1 адрес count\_1, затем в r2 и r3 помещается значение count\_2 = 3, а в r0 и r1 значение count\_1 = A. Вычисляется остаток от деления встроенной функцией \_\_aeabi\_ldivmod, которая возвращает частное и остаток. Остаток, равный 1, помещается в r2.

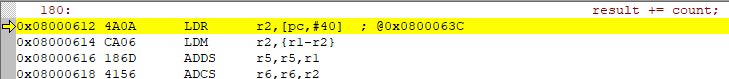
В r0 помещается 64 (10010), в r1 0, потом командами UMULL и MLA выполняется умножение остатка на 64, аналогично описанному выше алгоритму, и результат записывается в r12 и r0. Для деления снова используется \_\_aeabi\_ldivmod. И в r9 помещается результат 2116.



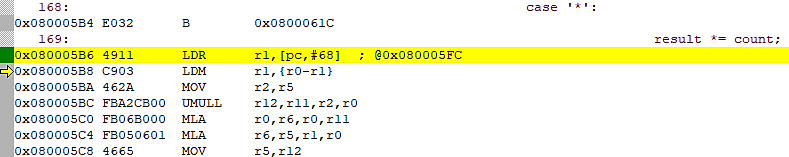
Из стека извлекается значение max\_level и помещается в r8.



Здесь выполняется непосредственное деление. После загрузки значений в регистры, снова выполняется встроенная функция, в r0 и r1 (так как 64 бита) помещается ее результат 3, в r2 остаток 1. Это деление должно быть целочисленным, поэтому остаток не используется. В r2 помещается адрес переменной count, затем STM из r0 и r1 помещает по адресу r2 значение count.



В r2 загружается адрес count, в r1 из памяти загружаются значения младших битов count = 3, в r2 значения старших. В r5 хранятся младшие биты значения result = 0, в r6 старшие, ADDS складывает младшие биты result+count и помещает в r5, то есть в младшие биты result. Суффикс S – обновление флагов, в данном случае обновляется флаг N = 0, то есть вычисленное значение больше нуля. ADCS складывает старшие биты и кладет в r6.



В r1 загружается адрес count, в r0 и r1 из памяти по адресу r1 загружается значение count = 5. В r2 помещается промежуточное значение result=3 после предыдущих вычислений. UMULL, MLA перемножают count и result, MOV помещает результат из r12 в r5, то есть в переменную result.

*1.4 Смените в функции eval тип всех переменных на int64\_t. Назовите эту функцию eval\_64 (заготовка для нее уже есть в файле eval.c). Посмотрите на дизассемблерный листинг функции eval\_64; разберите основные отличия по сравнению с оригинальной функцией.*

В функции eval все переменные, хранящие промежуточные значения, имеют размер 64 бита, поэтому ее выполнение ничем не отличается от eval\_64.

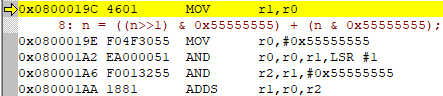
**Часть вторая - битовые операции**

*2.1. CountSetBits*



Разберем пример для данного теста 0xF = 11112.

Для нахождения единичных битов сначала используется «параллельный алгоритм», его суть в нахождении суммы битов в каждом байте числа.



В r1 помещается значение n = 0xF. В r0 помещается 0x5555 5555.

В команде AND: LSR – логический сдвиг вправо в битовом представлении r1 (0xF), производится побитовое логическое И.

Поделим складываемые логически числа на 16 блоков по 2 бита:

0000 … 1111 (n = 0xF)

&

0101 … 0101 (0x55555555)

0000 … 0101 в каждом блоке остались только младшие биты

0000 … 0111 (0xF>>1) старшие биты сдвинулись на младшие

&

0101 … 0101 (0x55555555)

0000 … 0101 остались старшие биты, сдвинутые на младшие

0000 … 0101

+

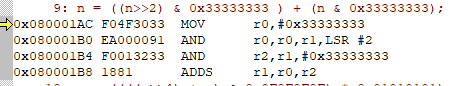
0000 … 0101

0000 … 1010 Просуммированные биты в каждом блоке

r0 = r0&(r1>>1) = 0x55555555 & (0xF>>1) = 0101 0101 0101 0101 0101 0101 0101 0101 & 0000 0000 0000 0000 0000 0000 0000 0111 = 0000….0101 = 0x5. Результат сохраняется в r0.

r2 = r1 & 0x55555555 = n & 0x55555555 = 0000….1111 & 0101 …. 0101 = 0000….0101 = 0x5.

ADDS – сложение полученных чисел с обновлением флагов, сохранение в r1 результата 0xA.



Теперь поделим числа на 8 блоков по 4 бита и найдем сумму битов в каждом из них. Арифметические операции те же, что и в предыдущей строке. Результат равен 1002 = 0x4 и помещается в r1.

AND:

0000 … 1010 (n = 0xA)

&

0011 … 0011 (0x33333333)

0000 … 0010 в каждом блоке остались только младшие биты

AND:

0000 … 0010 (0xA>>2)

&

0011 … 0011 (0x33333333)

0000 … 0010 остались старшие биты, сдвинутые на младшие

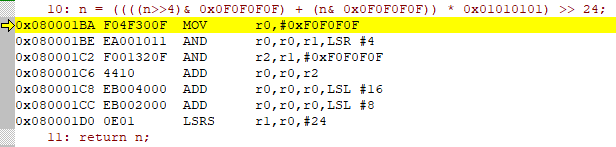
ADDS:

0000 … 0010

+

0000 … 0010

0000 … 0100 Просуммированные биты в каждом блоке



Далее найдем суммы битов в каждом из 4-х 8-битовых блоков.

AND:

0000 0000 …0000 0100 (n = 0x4)

&

00001111 …0000 1111 (0xF0F0F0F)

00000000… 0000 0100 в каждом блоке остались только младшие биты

Результат сохраняется в r0.

AND:

0000 0000 …0000 0000 (0x4>>4)

&

00001111 …0000 1111 (0xF0F0F0F)

0000 0000… 0000 0000 остались старшие биты, сдвинутые на младшие

Результат сохраняется в r2.

ADD:

0000 0000… 0000 0100

+

0000 0000… 0000 0000

0000 0000… 0000 0100 Просуммированные биты в каждом блоке

Этот результат сохраняется в регистр r0.



Теперь найдем сумму этих четырех блоков. Для этого нужно умножить на 0x01010101 = 1 0000 0001 0000 0001 0000 0001, и нужная сумма будет находиться в 3-ем бите, поэтому нужно сдвинуть на 4\*3 = 24 бита вправо.

0000 0000 0000 0000 0000 0000 0000 0100

\*

0000 0001 0000 0001 0000 0001 0000 0001

0000 0100 0000 0100 0000 0100 0000 0100

>>24\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0000 0000 0000 0000 0000 0000 0000 0100 = 0x4

В ассемблере операция умножения заменяется на 2 сложения со сдвигами влево:

ADD:

r0 = r0 + (r0<<16) = 0000 0000 0000 0000 0000 0000 0000 0100 + 0000 0000 0000 0100 0000 0000 0000 0000 = 0000 0000 0000 0100 0000 0000 0000 0100

ADD:

r0 = r0 + (r0 << 8) = 0000 0000 0000 0100 0000 0000 0000 0100 + 0000 0100 0000 0000 0000 0100 0000 0000 = 100 0000 0100 0000 0100 0000 0100

И сдвиг вправо на 24:

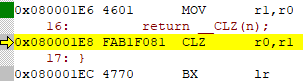
LSRS:

r1 = r0>>24 = 100 0000 0100 0000 0100 0000 0100 >> 24 = 0000 … 0100 = 0x4

*2.2. CountLeadingZeros*



Разберем функцию на примере этого теста.



В r1 помещается n = 5. Команда CLZ вычисляет ведущие нули в n и помещает их количество в регистр r0, возвращаемый в TEST\_EQUAL.