Отчёт по 3 работе.

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

*Сравнение :*

*26: for (j=0;j<i;j++){if (arr[j]==1){k++;}}*

0x08000230 F1BC0F01 CMP r12,#0x01 – сравнить r12 с числом 1 и обновить флаг состояния

*Арифметические операции :*

43: int ololo = tt+2;

0x08000222 1CA6 ADDS r6,r4,#2 - пример сложения : в r6 записать r4+2

*Битовые операции :* 42: int tt= cnrrr<<2;

43:

0x08000220 009D LSLS r5,r3,#2 – специально созданная паразитная строчка, созданная с целью идентифицировать то, что в ассемблере есть сдвиг … Можно было проще, но я выбрал этот вариант:

45: for (int i = 0;i<32;i++){if ( (n>>i) == 1){j = i+1;break;}}

0x0800022A FA21F700 LSR r7,r1,r0 – сдвигаем r1 на r0 и записываем в r7

*Что ещё … умножение:*

*219: rib = n1 \* n2;*

*0x08000BC6 F50D60AF ADD r0,SP,#0x578 – плюсуемк sp 578*

*0x08000BCA C803 LDM r0,{r0-r1} -по адресу вычесленному ранее загружаем регистры*

*0x08000BCC FBA06208 UMULL r6,r2,r0,r8 – перемножаем (согласно мануалу r0 r8 ) и сохраняем в r6 r2 по 32 бита в каждый*

*0x08000BD0 FB012108 MLA r1,r1,r8,r2*

*0x08000BD4 FB001709 MLA r7,r0,r9,r1 -как и предыдущая команда умножает но при этом добавляя 3-е значение регистра (что?)*

код на С выгладит при этом так : *rib = n1 \* n2;* (поэтому я и решил не искать соответствие строк н ассемблере и непосредственно но С (а просто копировать место, в котором эта строка написана на С в дизассемблерном листинге, ведь описываемая строка уже написана в начале ))

*Переход в ф-ию через BL.W по адресу расположения начала этой функции* :

0x0800029A F7FFFF8E BL.W getMedian (0x080001BA)

*Интересная команда, которая ничего не делает но при этом увеличивает счётчик:*

111: }

0x08000418 BF00 NOP

*Команда сравнения с нулём :*

0x08000466 B108 CBZ r0,0x0800046C

*Команда деления:*

0x08000448 FB95F8F4 SDIV r8,r5,r4

**Смените в функции eval тип всех переменных на int64\_t. Назовите эту функцию eval\_64 (заготовка для нее уже есть в файле eval.c):**

**int64\_t eval\_64( const char \* formula );**

**Посмотрите на дизассемблерный листинг функции eval\_64; разберите основные отличия по сравнению с оригинальной функцией.**

Команды занимают больше регистров :

было

for (int i = 0;i<strlen(formula);i++){str[i]= formula[i];}

0x0800043E 2400 MOVS r4,#0x00

стало

for (int64\_t i = 0;i<strlen(formula);i++){str[i]= formula[i];}

0x080006C6 2000 MOVS r0,#0x00

0x080006C8 4607 MOV r7,r0

0x080006CA 4680 MOV r8,r0

Также немного изменились и сами команды, к примеру

Инициализация присвоения:

8: int Olo= eval("2+2"); - 64 битный eval

0x080001DA A005 ADR r0,{pc}+0x18 ; @0x080001F0

0x080001DC F000F8A7 BL.W eval (0x0800032E)

0x080001E0 4604 MOV r4,r0

9: Olo= eval64n("2+2"); - просто eval

0x080001E2 A003 ADR r0,{pc}+0x10 ; @0x080001F0 – в 64 варианте добавляется число 18

0x080001E4 F001F943 BL.W eval64n (0x0800146E)

0x080001E8 4604 MOV r4,r0

Сложение :

609: rib = n1 + n2; - обычный

0x08001A2E EB080406 ADD r4,r8,r6 – складываем r8 и r6 и записываем в r4 результат

208: rib = n1 + n2; - 64 битный

0x08000B74 F8DD1578 LDR r1,[sp,#0x578]

0x08000B78 F8DD057C LDR r0,[sp,#0x57C] – загружаем значения регистров по данным адресам в скобках

0x08000B7C EB110608 ADDS r6,r1,r8 – ранее в r8 было записано значение переведённого в инт64 числа n1 (а в r9 - n2) и складываем это с r1 (обновляя флаг состояния (если не ошибаюсь))

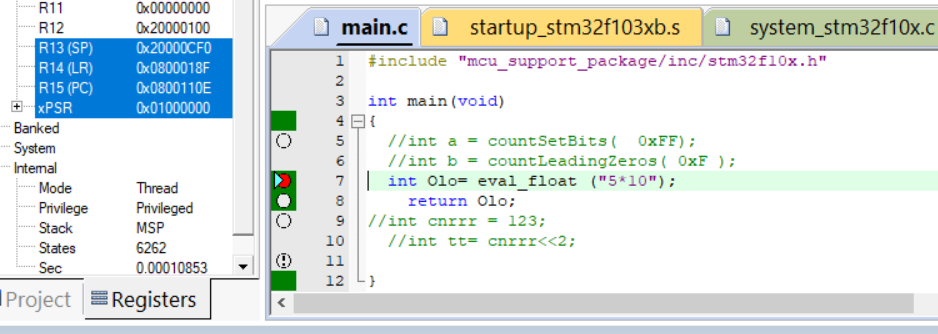
0x08000B80 EB400709 ADC r7, r0,r9 –не тоже самое, что и предыдущая строка: это сложение с переносом (при необходимости (и наличии соответствующего флага переноса)), регистры r0,r9 складываются и результат в r7 (согласно мануалу, при этом можно ещё и обновить флаги условий)

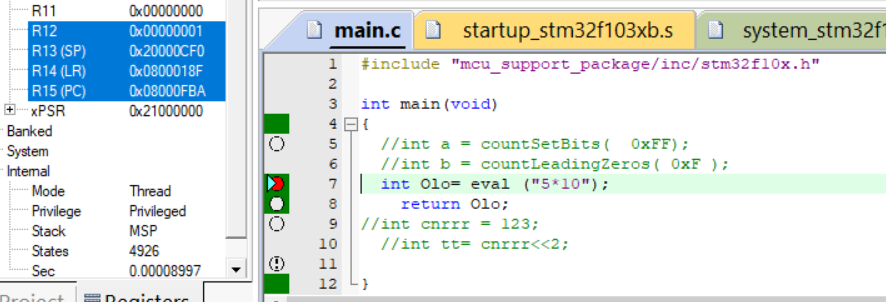
В целом, поменялось многое, но не коренным образом (инт есть инт, на сколько бы большим он не был), поэтому всё можно описать как ухищрения по принципу «разделить число на 2 меньших если оно (результат работы с ним) не влезает в 1 регистр и выполнить действия» (это моё субъективное оценочное мнение)

**Смените в функции eval тип всех переменных на float. Назовите эту функцию eval\_float (заготовка для нее уже есть в файле eval.c):**

**float eval\_float( const char \* formula )**

**С помощью счетчика тактов в симуляторе Keil оцените, насколько отличается время выполнения функции eval от времени выполнения функции eval\_float для одинаковых входных строк.**





Следует отметить, что в написанной программе переменные как правило использовались в качестве счётчиков, и их изменение на тип float незамедлительно приводило к ошибке, поэтому изменение типов переменных было произведено только в части функции, что, судя по результатам, дало значительную разницу в числе тактов.

**Разбор дисассемблерного листинга countsetbits и countleadingzeros.**

Так начинается функция dectobin – основа countsetbits

10: {

11:

0x080001B8 B530 PUSH {r4-r5,lr} – помещаем регистра r4,r5 и регистр адреса возврата lr в стек

0x080001BA 4604 MOV r4,r0

12: if ( n >= 2 )

13: {

0x080001BC 2C02 CMP r4,#0x02

0x080001BE DB05 BLT 0x080001CC - это метка условного перехода по указанному адресу (если флаги N=1 и V = 0 или наоборот)

14: DecToBin( n/2 );

Как работает цикл for (основные моменты)

26: for (j=0;j<i;j++){if (arr[j]==1){k++;}}

27:

0x080001FC BF00 NOP - ничего не делаем

0x080001FE E006 B 0x0800020E - переходим по этому адресу

0x08000200 4819 LDR r0,[pc,#100] ; @0x08000268

0x08000202 F8500024 LDR r0,[r0,r4,LSL #2] – тут не совсем понял, что но точно происходит сдвиг на 2 и обновляются флаги (помимо обычного LDR ), предположу, что в r0 записываем результат сдвига r4 на 2 позиции влево .

0x08000206 2801 CMP r0,#0x01 – сравниваем значение в регистре с 1 и ставим флаги (в зависимости от результата сравнения)

0x08000208 D100 BNE 0x0800020C – переходим по адресу если установлены флаги

0x0800020A 1C6D ADDS r5,r5,#1 - увеличиваем значение переменной на 1

0x0800020C 1C64 ADDS r4,r4,#1 – увеличиваем счётчик

0x0800020E 4817 LDR r0,[pc,#92] ; @0x0800026C

0x08000210 6800 LDR r0,[r0,#0x00]

0x08000212 4284 CMP r4,r0 – смотрим, не закончен ли цикл (и ставим флаг N при выполнении условий)

0x08000214 DBF4 BLT 0x08000200 – и если цикл, судя по флагам, не закончен, то идём сюда 0x08000200 и повторяем

В общем то других новых команд, не описанных ранее, в программе не обнаружилось.

P.S. Я удивлён, что алгоритм countSetBits, скопипащенный мной с одного из форумов корректно работает, но это факт . Я разобрал принцип его работы (не полностью) и он основан на делении числа на 2… и записи значения результата остатка от деления числа на 2 в массив … в общем, в итоге мы получаем массив символов 0 и 1 кодирующих 2-чное число, считаем 1 и выводим результат.

Можно было бы и со сдвигами (и это бы было проще с точки зрения объяснения), но я решил поступить как истинный программист.