Есть ли у вас вопросы?

Краткое содержание предыдущей серии

- Как в ассемблере осуществляется сложение и вычитание?
- В чем опасность умножения?
- А деления?
- Как в ассемблере осуществляется сравнение двух чисел?

Краткое содержание этой серии

- Разбор полетов
- Как же все-таки связан тип переменной и ее адрес
- Подробнее о сравнениях и флагах
- Как в ассемблере осуществляются циклы и ветвления
- Операции языка C (продолжение)

Оформление

Вызов значения локальной переменной a из стекового кадра.

0x00080124 E50B3028 STR R3,[R11,#-0x0028]

Запись значения локальной переменной iLoc в память, т.о. выделена памят объект (выделено 4 байта, т.к. тип данной переменной int).

35: iLoc *= b;

0x00080128 E51B2028 LDR R2,[R11,#-0x0028]

Вызов значения локальной переменной iLoc из стекового кадра.

0x0008012C E51B3024 LDR R3,[R11,#-0x0024]

Вызов значения локальной переменной b из стекового кадра.

0x00080130 E0030392 MUL R3,R2,R3

0x00080134 E50B3028 STR R3,[R11,#-0x0028]

36: iLoc = (iLoc -A1) | M1;

0x00080138 E51B3028 LDR R3,[R11,#-0x0028]

0x0008013C E2433043 SUB R3,R3,#0x00000043

0x00080140 E383301C ORR R3,R3,#0x0000001C

0x00080144 E50B3028 STR R3,[R11,#-0x0028]

Отчет нормального человека

// В регистр R2 загружается адрес счётчика i.

0x00080174 E3A03002 MOV R3,#0x00000002

// В регистр R3 помещается константа 0x2=2.

0x00080178 E1C230B0 STRH R3,[R2]

// Запись младшего полуслова(2 байта) R3 в память по адресу, находящемуся в регистре R2.

С помощью этих команд производится инициализация счётчика i.

0x0008017C E59F213C LDR R2,[PC,#0x013C]

// В R2 заносится адрес j.

0x00080180 E3A03F46 MOV R3,#0x00000118

// В R3 записывается константа $0x118=280_{10}$.

0x00080184 E2833003 ADD R3,R3,#0x00000003

// К значению R3 добавляется 0x3=3, в итоге там будет 283.

0x00080188 E1C230B0 STRH R3,[R2]

// Запись младшего полуслова(2 байта) R3 в память по адресу, находящемуся в регистре R2.

С помощью этих команд производится инициализация счётчика j.

0x0008018C E59F3128 LDR R3,[PC,#0x0128]

// В R3 вносится адрес i.

Отчет, от которого вытекают глаза

Вот так тоже не надо

```
#include <ADuC7026.H>
#include "bitfields.h"
void RxHandler(void);
// все глобальные переменные, которые используются и в прерывании и где-то еще
// должны быть объявлены как volatile - иначе компилятор может соптимизировать
обращения
volatile char newByte;
char *volatile new_byte1;
int main(void)
{
        new_byte1 = &newByte;
        GP4DAT |= BIT2 << 24;
        GP1CON = 0x011;     // Конфигурируем выводы P1.0 и P1.1 для UART
        // Настройка UART на 9600bps
        COMCON0 = 0x080; // Откроем доступ к делителю
        COMDIV0 = 0x088;  // Зададим величину делителя для 9600bps
        COMDIV1 = 0x000;
        COMCON0 = 0x003; // Установим формат пакета
        COMIEN0 = BIT0;             // включим прерывание по приему байта
        IRQEN |= UART_BIT;          // Разрешим прерывание от UART
        IRQ = RxHandler;        // Направляем прерывание IRQ на обработчик
        while(1);
        return 0;
}
```

Вот так тоже не надо

0x080003D8 2022 MOVS r0,#0x22

Данная строка значит, что по адресу **0x080003D8** хранится команда 2022 (MOVS r0,#0x22): положить в регистр **r0** число **22**. **S**–флаг обновления регистров состояния.

Команда использует непосредственную адресацию.

0x080003DA 4947 LDR r1,[pc,#284] ; @0x080004F8

Данная строка значит, что по адресу **0x080003DA** хранится команда 4947 (LDR r1,[pc,#284]): положить в регистр **r1** то, что находится по адресу **pc+284**, после точки с запятой написано результат сложение, что бы нам было проще.

Эта команда использует косвенно-регистровую адресацию.

0x080003DC 8008 STRH r0,[r1,#0x00]

Данная строка значит, что по адресу **0x080003DC** хранится команда 8008 (STRH r0,[r1,#0x00]): положить по адресу **r1** со смещением **+0** то, что находится в регистре **r0**.

Постфикс **B**, **H** или его **отсутствие**, указывает на длину значения, которое необходимо сохранить:
B – byte, байт;
H – halfword, полуслово, 2 байта;
Без постфикса – полное слово, 4 байта.

Команда использует косвенно-регистровую адресацию.

1.4.2 Найдите в дизассемблере короткую команду с непосредственной адресацией. Разберите двоичное представление этой команды, найдите в ее теле непосредственно задаваемый операнд.
Возьмем первую команду из пункту 1.4.1:
0x080003D8 2022 MOVS r0,#0x22

Разберем двоичное представление команды 2022:

0010 0000 0010 0010
0010 0 – показывает что это команда MOV
000 – показывает номер регистра куда будет записано значение
0010 0010 – число записываемое в регист (000), **непосредственно задаваемый операнд.**

Вот так тоже не надо

Длинная команда, с выделенным операндом:
F44F3010 (операнд не выделяется)
1111 0100 0100 1111 0011 0000 0001 0000
Двоичное представление:
1111 0100 0100 1111 0011 0000 0001 0000
Желтым цветом выделена команда mov.

Отступы

```
a=-130; //  целое отрицательное число в пределах байта
a=2.44; // положительное число с плавающей точкой
    r=8; // r - unsigned char
    r=0x212;
    r= 0x00150;
    r=0x00151;
    r=-130;
    r=2.44;
        c=8; // c - short int
        c=0x212;
        c= 0x00150;
        c=0x00151;
        c=-130;
        c=2.44;
            d=8; // d - unsigned short int
            d=0x212;
            d= 0x00150;
            d=0x00151;
            d=-130;
            d=2.44;
                e=8; // e - int
                e=0x212;
                e= 0x00150;
                e=0x00151;
                e=-130;
                e=2.44;
                    f=8; // f - unsigned int
                    f=0x212;
                    f= 0x00150;
                    f=0x00151;
                    f=-130;
                    f=2.44;
                        j=8; // j - long long int
                        j=0x212;
                        j= 0x00150;
                        j=0x00151;
                        j=-130;
                        j=2.44;
                            k=8; // k - unsigned long long int
                            k=0x212;
                            k= 0x00150;
                            k=0x00151;
                            k=-130;
                            k=2.44;
                                int i;
                                for (i=1; i<10000; i++)
                                x[i]=-i+17;
                                for (i=1; i<100; i++)
                                y[i]=i+32;
```

Отступы

```
20      a=200;
21      b=200;
22      c=200;
23      d=200;
24      e=200;
25      f=200;
26      t=200;
27      z=200;
28
29  a=0xAAA;
30  b=0xAAA;
31  c=0xAAA;
32  d=0xAAA;
33  e=0xAAA;
34  f=0xAAA;
35  t=0xAAA;
36  z=0xAAA;
37
38      a=0x640000;
39      b=0x640000;
40      c=0x640000;
41      d=0x640000;
42      e=0x640000;
43      f=0x640000;
44      t=0x640000;
45      z=0x640000;
46
47  a=0x1234567;
48  b=0x1234567;
49  c=0x1234567;
50  d=0x1234567;
51  e=0x1234567;
52  f=0x1234567;
53  t=0x1234567;
54  z=0x1234567;
55
56      a=-50;
57      b=-50;
58      c=-50;
59      d=-50;
60      e=-50;
61      f=-50;
62      t=-50;
63      z=-50;
64
65  a=0.25;
66  b=0.25;
67  c=0.25;
68  d=0.25;
69  e=0.25;
70  f=0.25;
71  t=0.25;
```

Отступы

```
// 1.5
a=e; // присвоение одной переменной зн
char *u;
u=&a;// создание указателя и присвоени
bool result1 = false;
int8_t w11 = -1;
uint8_t w21 = 1;
if( w11 < w21 )
{
result1 = true;
}
else
{
result1 = false;
}

bool result2 = false;
int32_t w12 = -1;
uint32_t w22 = 1;
if( w12 < w22 )
{
result2 = true;
}
else
{
result2 = false;
}


return 0;
```

Отступы

```
uA=100; uA=314; uA=4864;    uA=768; uA=4378;    uA=-126;    uA=1.14E2;
uB=100; uB=314; uB=4864;    uB=768; uB=4378;    uB=-126;    uB=1.14E2;
uC=100; uC=314; uC=4864;    uC=768; uC=4378;    uC=-126;    uC=1.14E2;
uD=100; uD=314; uD=4864;    uD=768; uD=4378;    uD=-126;    uD=1.14E2;
```

Отступы

```
int main(void)
{

uint8A=230; uint8A=1200;uint8A=0x25000;
    uint8A=0x7654321;uint8A=-65;uint8A=50.51;
uint16B=230; uint16B=0x111;uint16B=0x25000;
    uint16B=0x7654321;uint16B=-65;uint16B=50.51;
uint32C=230; uint32C=1200;uint32C=0x25000;
    uint32C=0x7654321;uint32C=-65;uint32C=50.51;
    uint64D=230; uint64D=1200;uint64D=0x25000;
    uint64D=0x7654321;uint64D=-65;uint64D=50.51;
    int8E=230; int8E=1200;int8E=0x25000;
    int8E=0x7654321;int8E=-65;int8E=50.51;
    int16F=230; int16F=1200;int16F=0x25000;
    int16F=0x7654321;int16F=-65;int16F=50.51;
    int32G=230; int32G=1200;int32G=0x25000;
    int32G=0x7654321;
    int32G=-64;
    int32G=50.51;
    int64H=230; int64H=1200;int64H=0x25000;
    int64H=0x7654321;
    int64H=-1117;
    int64H=50.51;
    uint16B=uint8A;
```

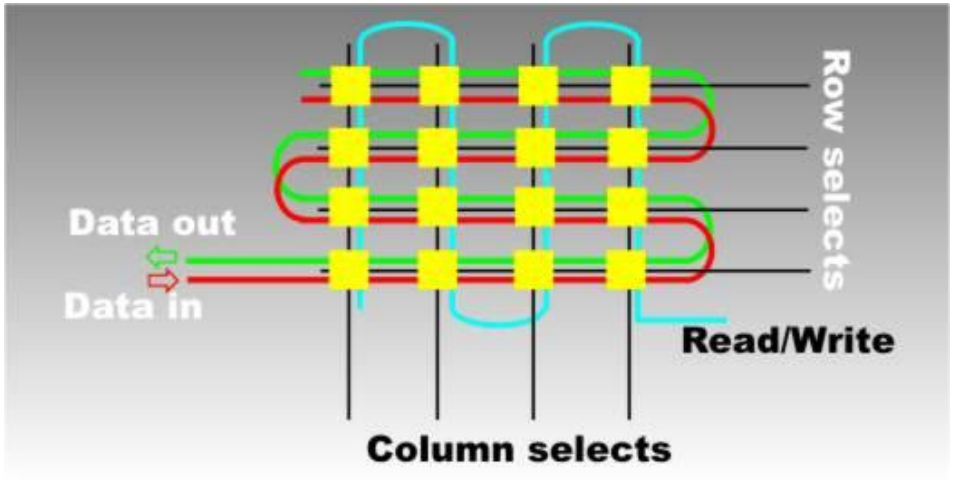
Как же связан тип переменной и ее адрес?

Адрес переменной кратен ее размеру в байтах!

Это называется «выравнивание» - alignment

Зачем нужно выравнивание?

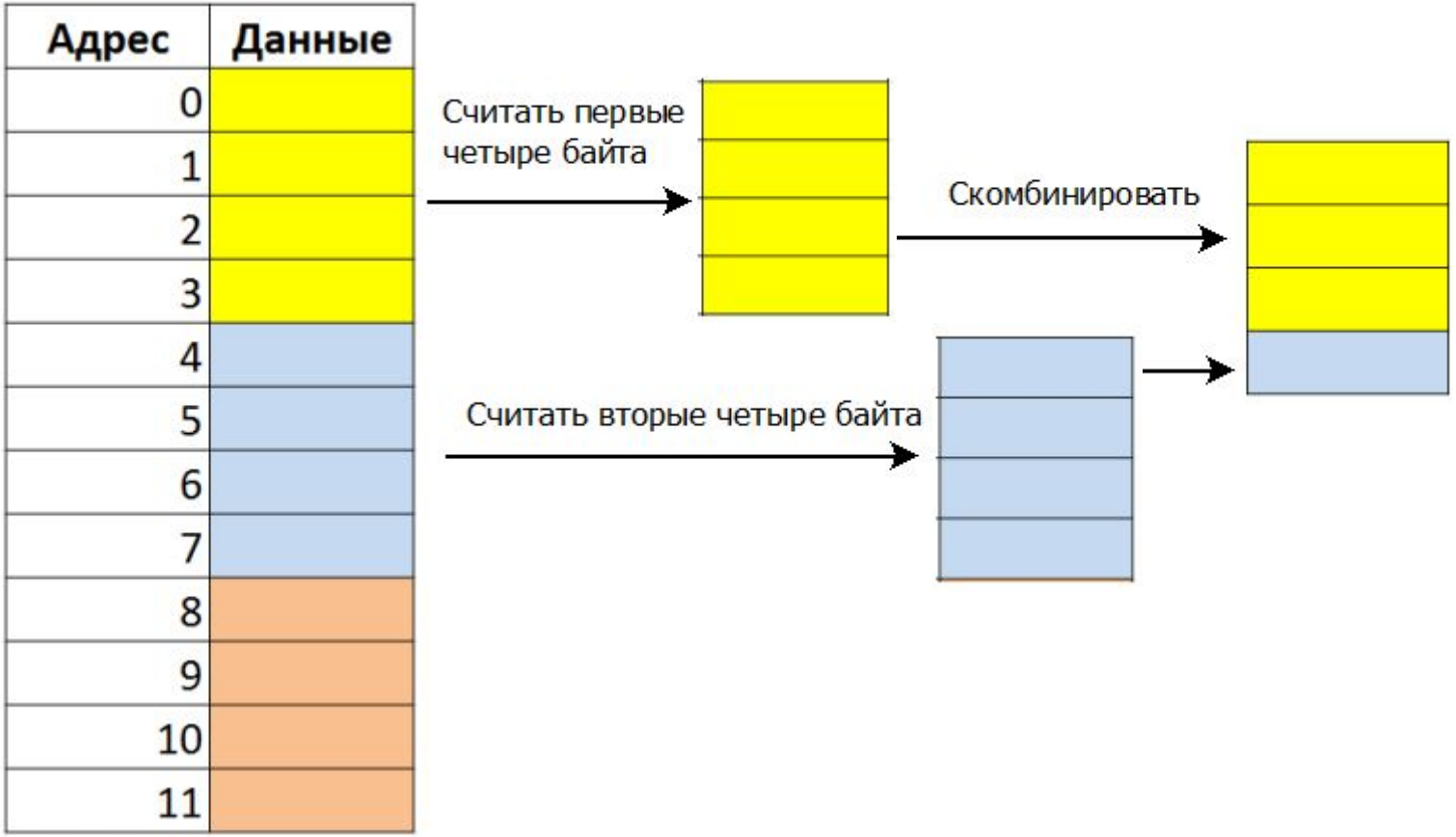
Вспоминаем, как устроена память



- Если хочется читать, например, по 4 байта за такт, то в памяти одну «ячейку» делают размером в 4 байта.
- Поэтому из памяти ВСЕГДА будет читаться по 4 байта за такт, даже если 2 из них не нужны
- Но как отсюда вытекает необходимость выравнивания?

Выравнивание

Как выглядит **невыровненный** доступ?



Последствия

- Выровненный доступ быстрее (особенно если есть кэш-память)
- Невыровненный доступ поддерживается не всегда (в Cortex M3 – только к 4 и 2 байтам)
- Байт всегда выровнен
- Для выравнивания требуется больше памяти?

Нет, порядок расположения переменных компилятор может менять при оптимизации

- Padding в структурах

Что такое padding?

- Порядок элементов внутри структуры менять нельзя (даже при оптимизации)
- Но элементы должны быть выровнены!
- Итог: размер структуры не всегда равен сумме размеров ее элементов
- Padding – пустые байты между элементами

Обычно есть спец. слова вроде __packed, чтобы паддинг убрать (не стандартизированные)

Арифметика в ассемблере
Какие еще есть флаги?

- C – флаг Carry (перенос)
- N – флаг Negative (отрицательный результат)
- Z – флаг Zero (результат 0)
- V – флаг oVerflow (знаковое переполнение, «неверная» смена знака)

Есть и другие, но к арифметике они не относятся

Сравнения в ассемблере

- CMP r0, r1 temp = r0 – r1, обновить регистр состояний, отбросить temp (аналогично SUBS temp, r0,r1)
- CMN r0,r1 temp = r0 + r1, обновить регистр состояний, отбросить temp (аналогично ADDS temp, r0,r1)
- TEQ r0,r1 (test equality), аналог ==, компилятором используется редко

Как работает сравнение и зачем их два?

Мы хотим сравнить два числа A и B.

Как узнать, какое из них больше с помощью арифметики?

Вычесть одно из другого!

Если C = A-B отрицательное, значит A меньше B.

Команда CMP a,b это и делает (и обновляет регистр состояний).

Знак результата показывает флаг N (1 если Negative).

А зачем CMN?

Вторым аргументом команды может быть регистр или число. Число должно лежать прямо в коде команды.

А если это число отрицательное?

Из-за дополнительного кода в нем будет очень много единиц в старших битах.

Но A-(-B) = A+B!

CMN a,b – это сравнение a – (-b), через сложение.

Если результат отрицательный, значит a < b. Флаг тот же.

Что означает флаг V?

V – от слова oVerflow означает знаковое переполнение.

Зачем он нужен?

Знак при арифметических операциях может меняться неправильно.

Например:

```
int8_t a = -128; // 1000 0000 в двоичном коде
a = a-1; // чему равно a?
```

a будет равно +127, потому что -129 не влезает в один байт.

(можно сказать, что флаг C – тоже переполнение, только беззнаковое)

Что означает флаг V?

Неверная смена знака – знаковое переполнение.

В языке C это undefined behavior.

При сравнениях это тоже может происходить (ведь сравнения – это вычитания).

Поэтому операции, которые используют результаты сравнений, проверяют и флаг V.

А какие команды используют результаты сравнений?

Где в языке С используется сравнение?

- if – else
- for, while, do-while
- switch

Следовательно, команды ассемблера, которые реализуют циклы и ветвления, используют результаты сравнений.

В основном, это команды перехода (передачи управления).

Переходы в языке C

- if – else
- for
- while
- do – while
- switch
- goto
- break, continue, return
- вызов функции

Несколько слов о goto

goto – оператор безусловного перехода:

```
... some code...
P:      // метка
... some code..
goto P; // безусловный переход к метке P
```

Этот оператор есть в огромном количестве языков программирования, но используется крайне редко.

goto

"В течение нескольких лет я знаком с точкой зрения, что качество программистов это убывающая функция от плотности операторов go to в коде, который они пишут.

Недавно я понял почему использование оператора go to имеет такой катастрофический эффект, и теперь я убежден, что оператор go to следует убрать из всех языков программирования "высокого уровня" (то есть из всех за исключением, возможно, машинного кода)...“

Эдсгер Дейкстра

goto позволяет писать «спагетти-код»

```
 5  int a = 2;
 6  int b = 2;
 7  int c = 2;
 8
 9
10  int main(void)
11  {
12
13    goto B;
14
15  A:
16
17    b = 3;
18    c++;
19
20    goto D;
21
22  B:
23
24    if( b >= 3 )
25    {
26      goto C;
27      c++;
28    }
29
30    a--;
31
```

```
32    while( a > 0 )
33    {
34      goto A;
35
36  C:
37
38      c--;
39      b--;
40      goto B;
41  D:
42      if(c < 0)
43      {
44        goto A;
45      }
46
47      goto B;
48
49    }
50
51
52    while(1);
```

Чему равны a, b, c к строке с точкой останова? Достижима ли точка останова?

Типичный сценарий использования goto

http://xkcd.com/292/

Вывод: лучше не использовать goto. Без него **всегда** можно обойтись.

Ветвление и циклы в ассемблере

- Безусловные переходы
- Условные переходы
- Другие команды с условным исполнением
- Сравнение с переходом
- Команда IT (If-then), «придающая условность»

В некоторых ассемблерах есть спец. команды для циклов (напр. loop)

Условное исполнение

Что это такое?

Это когда команда выполняется *по условию!*

Какие бывают условия?

Сочетания флагов состояния!

Условие задается постфиксом.

Список постфиксов см. cortex m3 user guide стр. 61 табл. 3-4

Условное исполнение

Примеры постфиксов и расшифровка:

LT – Less Then (если меньше, знаковое)
GE – Greater or Equal (если больше или равно, знаковое)
HI – Higher (если больше, беззнаковое)
NE – Not Equal (не равно)
MI – Minus (отрицательный результат)

Пример:

```
CMP     R3, #0x07; сравнить содержимое R3 и число 7
BGE     0x080003BA ; Если R3 >= 7 – перейти по адресу
MOVS    r0, #0x01
```

Условное исполнение

В некоторых наборах команд (например, ARMv5) почти все команды могли иметь условное исполнение.

В ARMv7 «сами по себе» условны только команды перехода. Для придания «условности» остальным командам используется команда IT. С ее помощью до 4 команд могут быть выполнены по условию:

```
CMP    R4, 5        ; сравнить R4 и число 5
ITTE   NE           ; три следующие команды – условные
ADDNE  R0, R0, R1 ; (если не равно – R0=R0+R1)
ADDNE  R2, R2, #1 ; (если не равно – R2=R2+1)
MOVEQ  R2, R3       ; (иначе – R2=R3)
```

Команды перехода

Названия в разных ассемблерах разное, суть одна и та же

- В x86 – команда jmp (от слова jump)
- В ARM – команда B (от слова branch)

В некоторых ассемблерах специальной команды нет, используется запись в регистр-счетчик команд. В ARM так делать можно, но не рекомендуется.

Команда B

- B адрес – переход по адресу (±16 Мб от текущего положения)
- BX r0– переход по адресу, который хранится в r0
- Условные переходы:
  - BLT – переход, если «меньше или равно»
  - BGE – переход, если «больше или равно» и т.д.
- BL и BLX – переход в функцию (а зачем отдельная команда?)

Ветвление

Псевдо-C	Псевдо-ASM
code 1; if(a > b) { *code 2;* } *code 3;*	*asm code 1* cmp a, b BLE code 3 // если меньше *code 2* *code 3*

Разумеется, возможны и другие варианты реализации ветвления.
Выбор за компилятором

Цикл

Псевдо-C	Псевдо-ASM
code 1; while(a > b) { *code 2;* a--; } *code 3;*	0: *asm code 1* 1: b 4 2: *asm code 2* 3: sub a,1 4: cmp a, b 5: bgt 2 // если больше 6: *code 3*