Лабораторно упражнение №2

Работа с кроскомпилатор GCC и кросасемблер AS за микроконтролери. Дисасемблиране на обектов код с Objdump.

1. Да се допълни top-level Makefile-а от лабораторно упражнение 1, така че компилирането и асемблирането на програмата да става на отделни етапи. Променете само target-а за main.c файла. За целта се използват следните команди [1]:

- За преобразуване на C сорс файл в Асемблерен сорс файл:

```
arm-none-eabi-gcc -S main.c -o main.s
```

- За преобразуване на Асемблерен сорс файл в обектов файл с относителни (фиктивни) адреси:

```
arm-none-eabi-as main.s -o main.o
```

- За преобразуване на обектов файл с относителни адреси в обектов файл с абсолютни адреси (линкване):

```
arm-none-eabi-gcc main.o led.o uart.o startup.o printf.o -o main.axf
```

- За премахване на служебна за toolchain-а информация от линкнатия файл:

```
arm-none-eabi-objcopy -O binary main.axf main.bin
```

Разгледайте полученият текстови файл main.s.

2. За да разгледате двоичния файл `main.o` ще трябва да използвате специална програма, която да го преобразува в текстови файл с дисасемблер. В GCC toolchain-а това е програмата Objdump:

```
arm-none-eabi-objdump -S -D main.o > main.lst
```

Направете отделен target за тази команда. Отворете текстовия файл main.lst и обърнете внимание на адресите при извикването на всяка една функция от main( ).

3. Добавете командата size като нов target, за да показвате размера на двойчния файл .axf:

```
arm-none-eabi-size main.axf
```

което ще ви върне следния отговор:

```
   text        data         bss         dec         hex    filename
  11280          52           0       11332        2c44    ./debug/main.axf
```

Тук text е размера на потребителската програма и променливи инициализирани като const. Този сегмент се помества /обикновено/ във Flash паметта. Сегментът data показва колко SRAM памет ще заемат глобалните инициализирани и статичните инициализирани променливи. Сегментът bss показва колко SRAM ще заемат глобалните неинициализирани и статични неинициализирани променливи.
ВНИМАНИЕ: байтовете, които ще се запишат във Flash паметта на контролера са:

FLASH total = text + data

защото инициализираните променливи се пазят във Flash, а при стартирането на програмата ви ще се копират в SRAM.

Полето dec показва сумата на всички предишни полета:

dec = text + data + bss

а полето hex показва полето dec в шестнадесетичен вид.

4. Добавете командния аргумент за оптимизация на кода '-On', където n е числото 0, 1, 2, 3 или буквата s. Числото нула означава изключена оптимизация, а 1 – 3 означава включена оптимизация с баланс между бързодействие/размер на програмата. По-голямо число означава по-агресивна оптимизация. Буквата s означава оптимизация за размер на програмата.
Сравнете получените размери на програмата. Проверете работоспособността на програмата като я програмирате в микроконтролера след всяка компилация.
ВНИМАНИЕ: за да се види ефекта от оптимизацията, аргументът '-On' трябва да се включи при компилацията на всички сорс файлове.

5. Изключете оптимизациите и добавете командния аргумент '-g' за включване на дебъг информация в .axf файла. Компилирайте програмата и разгледайте Асемблерния листинг main.lst. Преименувайте main.lst на main_orig.lst. Включете -Os оптимизациите. Компилирайте и сравнете main.lst с main_orig.lst.

6. Компилаторът GCC позволява от командния ред да се дефинират макроси, които да бъдат предадени към сорс кода на програмата [2]. Това става посредством аргумент '-D'.
Променете програмата в main.c, така че в зависимост от един макрос, нека се

казва PRINTF_MESSAGE, да се изписват три различни съобщения по UART интерфейса. Променете target-а за main.s в top-level Makefile-а, така че да използва аргумент -D. Тествайте и трите варианта на програмата. Примерно използване на макроси е:

```
arm-none-eabi-gcc -DPRINTF_MESSAGE=2 main.c -o main.axf
```

7. Компилирайте програмата с числа с плаваща запетая от директория **02_7**. За целта използвайте аргументите на кросасемблера -mfloat-abi=hard -mfpu=fpv5-d16 (подайте ги на компилатора gcc, той ще ги прехвърли на асемблера). Разгледайте дисасемблерния файл main.lst и потърсете инструкциите за FPU модула. Отбележете си размера на програмата.

ВНИМАНИЕ: всички обектови файлове трябва да бъдат компилирани с FPU поддръжка, иначе линкерът ще даде грешка. Крайната команда за линкване (т.е. target-а за main.axf) също трябва да използва въпросните аргументи.

Заредете програмата във Flash. Ако тя работи, би трябвало в терминала да се види резултатът от изчисленията и светодиодът LD3 да започне да мига.

Тя, обаче, няма да работи – FPU-то изрично трябва да бъде включено от потребителския код, не е достатъчно само компилаторът да генерира FPU инструкциите.
За да пуснете FPU модула трябва да вдигнете в единица в съответния регистър CPACR от контролния блок на ARM (с име SCB) двойките битове CP10 и CP11. Това може да стане по следния начин [3]:

```
#define SCB_CPACR_CP10_CP11_EN       0xF00000

volatile uint32_t *scb_cpacr = (volatile uint32_t *)0xE000ED88;

*scb_cpacr |= SCB_CPACR_CP10_CP11_EN; //Enable ARM's FPU
```

Проверете числата принтирани по UART-а с числата дадени в коментарите. Ако всичко е минало успешно, двете редици трябва да са еднакви.

8. Заместете двата аргумента -mfloat-abi=hard -mfpu=fpv5-d16 с един -mfloat-abi=soft. Компилирайте и заредете програмата. Обърнете внимание на размера на програмата. Разгледайте отново main.lst.

9. Използвайте inline Асемблер в C програма [4] [5]. За целта върнете хардуерната обработка на числата с плаваща запетая и копирайте програмата от директория **02_9**, която изчислява корен квадратен на 10 числа от масив, във вашия проект. Използвайте FPU инструкцията на ARM Cortex:

vsqrt.f32 dst, src
Използвайте компилаторната директива:

asm(“инструкция” : лист-изходни-операнди : лист-входни-операнди : принадлежности);

където отделните полета са пояснени по-долу:

инструкция – мнемониката и операндите на инструкцията, която ще се вмъква в кода на C;
лист-изходни-операнди – списък с променливите от кода на C, които ще приемат резултата на инструкцията (това поле не е задължително);
лист-входни-операнди – списък с променливите от кода на C, които ще са източник на данните, които ще се обработват от инструкцията (това поле не е задължително);
принадлежности – указания към компилатора да не оптимизира регистрите, изброени в този списък (това поле не е задължително). Регистрите от този списък не може да са част от списъците лист-изходни-операнди и лист-входни-операнди. Ако компилаторът е използвал някои от регистрите, изброени тук, той ще включи код, с който да ги премести другаде преди да извика потрбителската asm инструкция.

Пример: копиране на данните от една променлива 'a' в друга 'b', посредством инструкцията mov dst, src (dst ←src):

```
int a = 0x7f7f3355;
int b;
asm("mov %[dst], %[src]"
        :[dst] "=r" (b)
        :[src] "r" (a)
      );
```

Специалните символи, последвани от буква и заградени в кавички имат следното значение:

Специален символ (modifier)	**Значение**
=	Операндът е само за запис (write only).
+	Операндът е за запис и четене (read-write). Да се използва само за изходни операнди.
&	Регистър, който може да бъде използван само като изходен.
Без специален символ	Операндът е само за четене (read only).

ВНИМАНИЕ: за по-сигурно използвайте ключовата дума volatile в asm директивата. Така ще кажете на компилатора задължително да включи вашия Асемблерен код в C програмата, дори при много агресивни оптимизации:

asm volatile(…);

Буква (constraint) [6]	Значение
t	32 x32-битови регистри s0 - s31 за числа с плаваща запетая.
h	Регистри r8 – r15.
G	Непосредствена константа с плаваща запетая.
H	Същата като G, но с отрицание.
I	Непосредствена целочислена константа.
J	Константа за индексна адресация (-4095 ↔ +4095).
K	Същата като I, но инвертирана (inverted).
L	Същата като I, но с отрицание (negated).
I	Същата като r.
M	Константа в диапазона 0 ↔ 32.
m	Валиден адрес от картата на паметта.
N	Константа в диапазона 0 ↔ 31.
O	Константа в диапазона -508 ↔ 508, кратна на 4.
r	Регистри r0 - r15
w	16 x64-битови регистри d0 - d15 за числа с плаваща запетая. В операндите се използва задължително %P0, %P1, %P2 и т.н.
X	Всички операнди.

10. Използвайте програма на Асемблер във вашата C програма. За целта копирайте C и асемблерния файл от директория **02_10** във вашия проект. Асемблерната програма реализира изчисление на средноквадратичната стойност на 4 числа с плаваща запетая по формулата:

$$x_{\mathrm{RMS}} = \sqrt{\frac{1}{n}\left(x_1^2 + x_2^2 + \cdots + x_n^2\right)}$$

Добавете target за асемблиране на асемблерния файл в Makefile-а ви.

ARM са написали стандарт, който се нарича ATPCS (ARM-Thumb Procedure Call Standard) и има за цел да стандартизира съвместната работа на обектови файлове, направени от сорс файлове с различни езици (конкретно – Асемблер, C и C++) [7].
GCC спазва този стандарт и ползването на регистрите от ядрото трябва да става

по следния начин:

- Регистри r0 – r3 се използват за подаване на параметри към функция/подпрограма, както и за връщане на резултат. Не е нужно да се съхраняват на стека преди да се извика подпрограмата.
- Ако повече от 4 параметъра се подават, от петия натам всички трябва да се подадат през стека, т.е. преди извикването на подпрограмата данните трябва да бъдат push-нати там.
- Регистър r0 се използва за връщан резултат.
- Аналогично за числа с плаваща запетая се използват регистрите s0, s1, s2, и s3 (d0, d1, d2, d3 за числа с плаваща запетая и двойна точност).
- Пет и повече параметъра, както и C структури по стойност, се предават през стека (т.е. преди да се извика подпрограмата те биват push-вани/копирани в RAM паметта).

Вземете инструкциите за числа с плаваща запетая от документа [8].

* * *

[1] B. Gough, “An introduction to GCC for the GNU compilers gcc and g++”, ISBN 0-9541617-9-3, Network Theory Limited, 2004.
[2] W. Hagen, “The Definitive Guide to GCC”, ISBN-13: 978-1-59059-585-5, Apress, 2006.
[3] “Cortex-M4 Technical Reference Manual”, ARM Ltd, 2010.
[4] https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm
[5] http://www.ethernut.de/en/documents/arm-inline-asm.html
[6] https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html
[7] ARM Developer Suite, v.1.2, ARM Limited, 2001.
[8] ARM Cortex-M7 Devices - Generic User Guide, ARM Limited, 2018.