Проектиране на вградени автомобилни електронни системи
Лабораторно упражнение №19

Работа с Xilinx Vivado и Vitis. Кроскомпилатор GCC и кросасемблер AS за микропроцесори. Дисасемблиране на обектов код с Objdump.

==========

1. Превключете джъмпера вдясно на платката на позиция JTAG. Свържете µUSB кабел към PROG/UART USB куплунга. Включете платката от ключа ON/OFF. Включете USB-към-UART конвертор към сигнали JB1_P (T20), отговарящ на Uartlite_TxD, и JB1_N (U20), отговарящ на Uartlite_RxD. Модулът Uartlite ще бъде свързан към един MicroBlaze.

ЗАБЕЛЕЖКА: работната маса с платка Zybo Z7-10 трябва да използва куплунг JC и съответно сигналите Uartlite_TxD (V15), Uartlite_RxD (W15).

2. Стартирайте две копия на Cutecom (или gtkterm) и отворете портовете, отговарящи на printf канала на дебъгера. Стартирайте терминал с CTRL + ALT + t и изпълнете командите:

```
cd /home/user/workspaces/xilinx_workspace
source /home/user/programs/Xilinx/Vitis/2022.2/settings64.sh
```

3. Използвайте битстриймът от директория **19_3** (в него MicroBlaze е с FPU). Да се допълни top-level Makefile-а от миналото лабораторно упражнение, 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:

```
mb-gcc -S -I./led -I./uart -I./print -I./include -mlittle-endian -mxl-barrel-shift -mxl-pattern-compare -mcpu=v11.0 -mno-xl-soft-mul main.c -o ./debug/main.s
```

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

```
mb-as -mlittle-endian main.s -o main.o
```

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

```
mb-gcc -Wl,-T -Wl,lscript.ld -Wl,--no-relax -Wl,--gc-sections -L./ -lxil -mlittle-endian -mxl-barrel-shift -mxl-pattern-compare -mcpu=v11.0 -mno-xl-soft-mul ./debug/main.o ./debug/led.o ./debug/uart.o ./debug/print.o -o ./debug/main.elf
```

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

```
mb-objcopy -O binary ./debug/main.elf ./debug/main.bin
```

Разгледайте полученият текстови файл main.s.

4. 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:

```
mb-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().

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

```
mb-size main.axf
```

което ще ви върне следния отговор:

```
text    data    bss   dec   hex   filename
26076    356    3136  29568 7380  ./debug/main.elf
```

Тук text е размера на потребителската програма и променливи, инициализирани като const. Този сегмент се помества /обикновено/ във Flash паметта при микроконтролерите, а във конкретната FPGA отива във външната DRAM. Сегментът data показва колко DRAM памет ще заемат глобалните инициализирани и статичните инициализирани променливи. Сегментът bss показва колко DRAM ще заемат глобалните неинициализирани и статични неинициализирани променливи.

ВНИМАНИЕ: байтовете, които ще се запишат във Flash паметта или SD картата на демо платката Zybo са:

FLASH total = text + data

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

Полето dec показва сумата на всички предишни полета:

dec = text + data + bss

а полето hex показва полето dec в шестнадесетичен вид.

6. 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.

7. Изключете оптимизациите и добавете командния аргумент '-g' за включване на дебъг информация в .elf 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.

8. 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 PRINT_D_MESSAGE, да се изписва допълнително съобщение по UART интерфейса. Променете target-а за main.s в top-level Makefile-а, така че да използва аргумент -D. Тествайте новият вариант на програмата. Примерно използване на макроси е:

```
mb-gcc -DDEBUG_LEVEL=2 main.c -o main.elf
```

9. Компилирайте програмата с числа с плаваща запетая от директория **19_9**. За целта използвайте аргументът на кросасемблера -msoft-float (подайте го на компилатора gcc). 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.elf) също трябва да използва въпросните аргументи.

Заредете програмата във FPGA. Ако тя работи, би трябвало в терминала да се види резултатът от изчисленията и светодиодът LD3 да започне да мига. Проверете числата принтирани по UART-а с числата дадени в коментарите. Ако всичко е минало успешно, двете редици трябва да са еднакви.

10. Заместете единият аргумент -msoft-float с трите аргумента -mhard-float -mxl-float-convert -mxl-float-sqrt. Компилирайте и заредете програмата. Обърнете внимание на размера на програмата. Разгледайте отново main.lst.

11. Използвайте inline Асемблер в C програма [4] [5]. Копирайте програмата от директория **19_11**, която изчислява корен квадратен на 10 числа от масив, във вашия проект. Използвайте FPU инструкцията на MicroBlaze:

```
fsqrt 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: събиране на числа с плаваща запетая от една променлива 'a' с друга 'b', и записване на резултат в ‘c’, посредством инструкцията fadd rD, rA, rB:

```
float a = 3.1415;
float b = 3.1415;
float c;
```

```
asm("fadd %0, %1, %2\n" : "=r" (c) : "r" (a), "r"(b));
```

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 принудите 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.

12. Използвайте програма на Асемблер във вашата С програма. За целта копирайте С и асемблерния файл от директория **19_12** 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.

Xilinx са написали стандарт, който се нарича MABI (MicroBlaze Application Binary Interface) 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 r5 – r10 се използват за подаване на параметри към функция/подпрограма.
*Регистри r3 – r4 са за връщане на резултат, като не е нужно да се съхраняват на стека преди да се извика подпрограмата;
*Ако повече от 6 параметъра се подават, от седмия натам всички трябва да се подадат през стека, т.е. преди извикването на подпрограмата данните трябва да бъдат push-нати там;
*Седем и повече параметъра, както и С структури по стойност, се предават през стека (т.е. преди да се извика подпрограмата те биват push-вани/копирани в DRAM паметта).

Вземете инструкциите за числа с плаваща запетая от документа [7].

* * *

[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] “MicroBlaze Processor Reference Guide”, Embedded Development Kit, EDK11.4, UG081 (v10.3), Xilinx Inc, 2009.

доц. д-р инж. Любомир Богданов, 2024 г.