



Проектиране на вградени автомобилни електронни системи

Лабораторно упражнение №20

Работа с Xilinx Vivado и Vitis. Работа с кросдебъгер GDB и сървърно приложение XSCT. Дебъгване на вградени системи през Интернет.

=====

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.

2. Стартирайте две копия на Cutesom (или gtkterm) и отворете портовете, отговарящи на printf канала на дебъгера и на Uartlite. 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. Сървърното приложение XSCT има за цел да уеднакви протоколите за комуникация на няколко дебъг интерфейса и да предостави дебъг достъп на потребителя до неговата система посредством стандартен TCP/IP терминал или софтуерния дебъгер GDB.

В терминала, отворен в миналата точка (№1), стартирайте сървърното приложение XSCT:

```
xsct
connect
```

а в друг терминал (№2) се свържете към него отново с XSCT, но в режим на клиент. За целта изпълнете командите:

```
xsct
connect -url TCP:127.0.0.1:3121
```

Програмирайте битстриймът на FPGA и фърмуерните файлове на двата процесора от клиентския терминал (№2) със следните команди:

```
targets
targets 4
fpga
```

```
/home/user/workspaces/xilinx_workspace/20_3/boot/bitstream.bit
targets 2
dow /home/user/workspaces/xilinx_workspace/20_3/boot/fsbl.elf
con
dow /home/user/workspaces/xilinx_workspace/20_3/boot/cortex.elf
con
targets 6
dow /home/user/workspaces/xilinx_workspace/20_3/debug/main.elf
con
```

Ако всичко е минало успешно, в терминалът на MicroBlaze трябва да се видят съобщенията от последната точка на миналото упражнение.

4. Изпълнете програмата стъпка по стъпка. За целта трябва първо да спрете и рестартирате изпълнението на програмата с командите:

```
stop
rst
```

и след това да напишете командата няколко пъти:

```
nxt 1
```

която ще ви изписва следната информация:

```
xsct% nxt 1
Info: MicroBlaze #0 (target 6) Stopped at 0x50 (Step)
XIntc_RegisterFastHandler() at xintc_l.c: 540
540:      if (CurrentIER & Mask) {
xsct% nxt 1
Info: MicroBlaze #0 (target 6) Stopped at 0x58 (Step)
408:      CfgPtr = Xintc_LookupConfig(InterruptId/32);
```

Това е start-up кода на MicroBlaze. Ако MicroBlaze започне да изпълнява код без дебъгерна информация (обектов файл, който не е компилиран с -g аргумента), XSCT ще изпише съобщение за грешка:

```
Info: MicroBlaze #0 (target 6) Stopped at 0x23c (Step)
_crtinit() at crtinit.S: 91
91:  couldn't open  "/usr/src/debug/libmblebspmfpd-libgloss/4.1.0-r0/newlib-
4.1.0/libgloss/microblaze/crtinit.S": no such file or directory
```

което е очакван резултат.

След няколко стъпки можете да пуснете отново програмата да се изпълнява до край с командата:

```
con
```

5. С команди на XSCT може да се четат и записват регистри на микроконтролера. Това са командите:

`mrd [опции] [адрес] [n]` – чете *n* брой думи от *адрес* в паметта

`mwr [опции] [адрес] [стойности] [n]` – записва *n* *стойности* с начален *адрес* в паметта

Прочетете първите 32 32-битови думи от началото на BRAM паметта, реализираща локалната памет на микропроцесора MicroBlaze (0x0000.0000):

```
mrd 0x00000000 32
```

6. Пуснете светодиода от предишното лабораторно да свети без да използвате микропроцесорът MicroBlaze. За целта първо спрете захранването на платката, след това го пуснете и инициализиращите команди до пускането на FSBL. След това изберете MicroBlaze с команда `targets 6`. На този етап хардуерът на PL частта ще работи, но MicroBlaze няма да има зареден фърмуерен файл (.elf). С `mrd` и `mwr` командите модифицирайте съответните регистри:

```
//GPIO_TRI, Set all as outputs
```

```
0x40000004 0x00000000
```

```
//GPIO_DATA, Set output 3 to logic 1, clear others
```

```
0x40000000 0x00000008
```

7. Програмирайте фърмуерите на ARM Cortex A9 и MicroBlaze. Затворете терминал 2 на XSCT. В Линукс терминал влезте в `debug` директорията на проекта ви от миналото упражнение и стартирайте софтуерната дебъг програма GDB [1], която е клиентско приложение:

```
mb-gdb
```

Така ще се стартира нова дебъг сесия. Свържете GDB с XSCT с помощта на командата:

```
target remote 127.0.0.1:3002
```

където всеки порт отговаря за различна архитектура:

*порт 3000 е за Arm

*порт 3001 е за AARCH64

*порт 3002 е за MicroBlaze [2]

*порт 3003 е за MicroBlaze64 [3]

Вместо `remote` може да се напише `extended` аргументът, който оказва, че ще се дебъгва многоядрен микропроцесор. В настоящото лабораторно такъв е ARM

Cortex A9. Тогава може да се използват командите `add-infe`, `infe`, `attach` и `detach`, с които се оказва кое ядро се дебъгва в момента.

ВНИМАНИЕ! GDB дебъг сървърът се пуска автоматично от XSCT след въвеждане на командите `xsct` и `connect`. Програмата `xsct` съдържа драйвер за JTAG интерфейса на Zybo и тя е отговорна за конвертирането на GDB команди в команди за хардуерния дебъг модул на MicroBlaze. Ако XSCT не е стартиран, GDB няма да може да осъществи връзка (това е софтуерен дебъгер от високо ниво).

Заредете програмата ви за изчисляване на RMS стойност посредством GDB командата:

```
load main.elf
```

Заредете символната и дебъгерната информация от `.elf` файла в GDB, за да можете да дебъгвате със съответните команди:

```
file main.axf
```

Клиентската програма GDB позволява да се предават команди директно на сървърната програма посредством командата `monitor`. Например следните команди са команди на XSCT сървъра:

```
monitor ps  
monitor info  
monitor help
```

За съжаление, към днешна дата това са единствените команди, които може да се предадат към XSCT.

Поставете точка на прекъсване в началото на `main()` функцията ви посредством (действието се нарича `set breakpoint`):

```
break main (или само b main)
```

Аналогично можете да сложите точки и в други места на програмата ви:

```
break led_set  
break led_clear
```

Точка на прекъсване може да се сложи на произволен ред така:

```
break <номер на ред>
```

`break <име на файл : номер на ред>`

Списък с всички активни точки може да се покаже с командата:

`info break`

Премахване на всички точки или само на някои от тях може да стане с :

`delete breakpoints`

`delete breakpoint <номер на точката от списъка "info break">`

Пуснете програмата да се изпълнява, до безкрай или докато не срещне точка на прекъсване (действието се нарича `resume`):

`continue` (или съкратено `c`)

Ако всичко е минало успешно в терминала ви ще се изпише:

```
(gdb) c
Continuing.
```

```
Breakpoint 3, led_clear () at led.c:46
46          XGpio_DiscreteWrite(&output, 1, 0x00);
```

Сега можете да изпълнявате програмата стъпка по стъпка, т.е. ред по ред във вашата C програма посредством командата:

`step` (или съкратено `s`)

След всяка стъпка дебъгерът ще ви показва информация за програмния брояч (не забравяйте, че програмният брояч нараства в зависимост от размера на изпълняваните инструкции, както и от размера на конвейера на съответния микропроцесор). С тази команда, ако срещнете функция в сорс кода ви ще влезете в нея да я дебъгвате (действието се нарича `step into`).

Възможно е стъпките да са на ниво Асемблер, тогава трябва да се използва:

`stepi`

Ако е излишно дебъгването на всяка срещната функция, може да се използва командата:

`next` (или съкратено `n`)

която срещайки функция, ще изпълни кода ѝ и ще предаде контрол обратно на

програмиста на излизане от функцията (действието се нарича step over). Ако сте влезли в много голяма функция и искате да излезете на ниво функцията, която я извикала първоначално, използвайте командата (действието се нарича step out):

```
finish
```

Ако е необходимо, паметта на микроконтролера може да се прочита с командата:

```
x <адрес>
```

Разгледайте първите две думи в началото на LMB (Local Memory Bus, реализирана в BRAM) паметта, както направихте в предходните точки:

```
x 0x00000000  
x 0x00000004
```

Докато е спряно изпълнението на програмата ви, разгледайте променливите ѝ с помощта на командата:

```
r <име на променлива>  
r <име на файл :: име на променлива>  
r <име на функция :: име на променлива>
```

Командата r може да приема и форматиращи спецификатори (подобно на printf) като се сложи наклонена черта и буква (виж таблицата на следващата страница):

```
r/<форматиращ спецификатор> <име на променлива>
```

Съдържанието на регистрите на ядрото може да бъде показано с командата:

```
info registers
```

8. Дебъггерната програма GDB може да чете стекови извадки (stack frame). За да тествате тази команда, напишете във вашия main.c файл три функции, всяка от която извършва някакви изчисления и вика следващата, например:

```
main → my_func_1 → my_func_2 → my_func_3
```

Спрете дебъг сесията, компилирайте и заредете програмата, стартирайте нова дебъг сесия. Сложете точка на прекъсване в най-вътрешната функция (my_func_3) и напишете някоя от командите:

Форматиращ спецификатор	Детайли
x	Целочислена стойност в шестнадесетичен вид
d	Целочислена стойност в десетичен вид със знак
u	Целочислена стойност в десетичен вид без знак
o	Целочислена стойност в осмичен вид
t	Целочислена стойност в двоичен вид
a	Адрес на променливата в паметта + отместване от най-близкия символ
c	Символ + числен еквивалент на символа
f	Число с плаваща запетая
s	Символен низ
z	Същото като 'x', но стойността ще бъде предшествана от нули в зависимост от размера ѝ в паметта
r	Python pretty-printer

backtrace (или само bt)
backtrace full

Втората команда показва всички локални променливи, които се съхраняват в стек паметта (SRAM) в дадения момент от програмата ви.

9. Дебъгерната програма GDB може да работи в мрежи с отдалечени target-и. За да изпробвате тази възможност нека една от работните маси да бъде Xilinx hw_server сървър с демо платка, а друга маса да бъде GDB клиент с фърмуер и bitstream, който ще се дебъгва.

На сървърното PC стартирайте терминал и напишете:

```
ifconfig
```

Оттам ще разберете IP адреса на въпросното PC. Стартирайте hw_server по следния начин (в терминал, в който е бил изпълнен скриптът settings64.sh на Vitis):

hw_server

като имате предвид, че Xilinx hw_server стартира дебъг сървър, обвързан с два мрежови интерфейса:

*127.0.0.1:3002 (виртуален)

*xxxx.xxxx.xxxx.xxxx (физически)

където xxxx.xxxx.xxxx.xxxx е IP на компютърът, работещ като сървър. В настоящето лабораторно IP-тата започват с 192.168.0.xxx.

На клиентското PC стартирайте XSCT и се свържете с hw_server на сървърното PC. Програмирайте bitstream-ът и всички .elf файлове без mb.elf. Затворете клиентското XSCT. Стартирайте GDB на клиентското PC и на командата target remote напишете IP- то на сървъра на вашите колеги. Например, ако IP-то на сървъра е 192.168.1.143, то изпълнете следните команди:

```
mb-gdb
```

```
target remote 192.168.1.143:3002
```

Тествайте командите от предишните задачи.

ВНИМАНИЕ: Примерът работи в локална мрежа. На практика, ако двата компютъра са отдалечени, със свои реални IP адреси, ще е необходимо да настроите т.нар. Port Forwarding на вашия рутер, свързан към компютъра, който ще има сървърна роля (т.е. там където ще е FPGA). Фиксирайте едно IP от DHCP pool-а за вашия MAC адрес, след което с това IP и порт 3002 направете Port Forwarding. Когато трябва да се свържете с клиента GDB, укажете реалното IP (т.е. WAN адреса на рутера), а не локалното, което ви е дадено от DHCP.

10. Настройка на развойната среда Eclipse за работа с GDB и XSCT. Развойната среда с отворен код Eclipse поддържа toolchain-а GCC и е добре интегриран с него. Вземете архивния файл на Eclipse и го разархивирайте (най-новата версия може да бъде свалена от [4], от раздела “Eclipse IDE for C/C++ Developers”). Влезте в новосъздадената папка “eclipse” и натиснете два пъти върху файла “eclipse”. Укажете работна директория на Desktop-а и натиснете OK.

За да отворите вашия проект, изберете File → Import → C/C++ → Existing Code as Makefile Project → Next → Existing Code Location → Browse → укажете пътя до вашия проект от миналите задачи, в същия прозорец изберете Cross GCC, задайте примерно име на проекта в полето Project Name → Finish. Изтрийте изходните файлове с Project → Clean → отметка на Clean all projects и маха се отметката на Start a build immediately. Ако сега натиснете бутона за build-ване (иконка на чук) в таб-а Console ще ви излезе грешката:


```
make[1]: mb-gcc: Command not found
```

Това е нормално, защото в средата (environment) на операционната система може да не е указан пътя до GCC. Това може да бъде направено от Eclipse по следния начин: десен бутон на top-level директорията на току-що импортнатия ви проект → Properties → C/C++ Build → Environment → изберете променливата PATH → Edit → в края на полето Value добавете дуето и укажете пътя до GCC → OK → OK → Build. Сега не би трябвало да излязат грешки, а в конзолата ще видите това, което досега виждахте в терминала на Линукс.

Забележка: за да разберете пътя до компилатора, в Линукс терминал напишете `which mb-gcc`.

За да може Eclipse да използва графичен front-end за GDB в режим на TCP/IP комуникация първо трябва да се инсталира Java Plug-in наречен “GDB Hardware Debugging”. Първо проверете дали плъгинът не е вече инсталиран Help → About Eclipse → Installation Details → type filter text → GDB. Ако се покаже поле “C/C++ GDB Hardware Debugging”, значи плъгинът е инсталиран. Ако това поле липсва, плъгинът трябва да се инсталира изрично [5]. Това може да стане от Help → Install New Software → Work with → Изберете от падащото меню версията на Eclipse, например по-старите версии са с имена като Kepler - <http://download.eclipse.org/releases/kepler>, а по-новите са с имена [година]-[месец], когато е излязла средата → в полето “Type filter text” напишете “C/C++ GDB Hardware Debugging” (или изберете Work with: --All Available Sites-- → Mobile and device development → C/C++ GDB Hardware Debugging) и натиснете Enter → Ще се появи категория (за конкретната версия името ѝ е “Mobile and Device Development”), която трябва да отворите и да сложите тикче върху “C/C++ GDB Hardware Debugging” → натиснете Next и следвайте указанията за инсталацията. Когато инсталацията приключи, средата ще се рестартира.

За да се настрой дебъг сесия трябва да се избере Run → Debug Configurations → кликнете два пъти върху GDB Hardware Debugging → ще ви се отвори таб New Configuration, който можете да конфигурирате от десния панел.

Най-важните полета, които трябва да попълните са:

- *C/C++ Application – път до изпълнимия файл с дебъгерна информация main.elf
- *Project – директория, в която се намират сорс файловете на проекта;
- *Debugger → GDB Command – да се укаже точния префикс на GDB командата – в случая mb-gdb;
- *Use remote target – това тикче трябва да е сложено, за да укажем TCP/IP връзка;
- *JTAG Device – избира се Generic TCP/IP;

*Host name or IP address – указва се IP адреса на сървъра, т.е. на компютъра, на който върви вече пуснат Xilinx hw_server; Port number – номер на порт, на който се очакват връзки от сървъра, за hw_server видяхме, че това е 3002;
*Startup → Load Image и Load Symbols – и двете трябва да сочат към изпълнимия файл, т.е. “Use project binary” (тези радио бутони са еквивалентни на командите load и file от GDB, които тествахме по-рано);
*Set breakpoint at – пише се main, аналогично на командата breakpoint main;
*Run commands – добре е да се запишат monitor init и monitor reset halt, за да започваме винаги от ресет хендлера.

След като сме готови с всички настройки по менютата може да натиснем debug, което ще програмира BRAM паметта на FPGA през мрежата и ще изкара графичен интерфейс на GDB, с помощта на който може да се дебъгне програмата. Забележете, че сега за всяка команда си има по един бутон, с който лесно можете да дебъгвате.

*Тествайте различните бутони на графичния дебъгер. Сложете точки на прекъсване, наблюдавайте локални и глобални променливи, и т.н.

Забележка: дебъг режимът може да бъде използван и само на един компютър, но въпреки това отново дебъгването се нарича “remote”, защото протоколът си остава TCP/IP. Ако желаете да дебъгвате на един компютър, вместо IP на сървър напишете IP на виртуалния ви мрежови интерфейс 127.0.0.1 (също известен като localhost), като не забравяйте преди това да стартирате OpenOCD на същия този компютър.

Забележка: в последните години се появиха много Plugin-и за Eclipse, които са специално оптимизирани и с повече възможности за конкретни GDB сървъри, например за работа с OpenOCD. Такъв е проектът GNU MCU on Eclipse [6].

11. Тествайте командата за извличане на символи от .elf файл [7]. Под символи се разбират имена на всички функции и глобални променливи в обектовия файл. С Линукс терминал влезте в директорията с main.elf файла на вашия пример и изпълнете командата:

```
mb-nm -a -l main.elf
```

което ще принтира адрес в паметта, където ще се намира символа при изпълнението на програмата, буква-спецификатор, която указва вида на символа, името на символа, така както е дадено от програмиста и накрая (най-дясна колонка) файлът, в който е деклариран символът. Таблицата по-долу показва обобщение на символните спецификатори. Главна буква указва външен (глобален) символ, докато с малка буква се указва локален символ. Ако желаете,

Символен спецификатор	Значение
A	Абсолютна стойност на символа.
B, b	Символът ще се намира в .bss региона на паметта.
C	Неинициализиран символ.
D, d	Символът ще се намира в .data региона на паметта.
G, g	Символът ще се намира в .data региона за малки обекти (глобална int променлива е малък обект, глобален масив с int е голям обект).
i	Символът се намира в специална секция от DLL. Адресът е условен и ще бъде известен едва по време на изпълнението на програмата.
I	Символът е указател към друг символ.
N	Символът е за подпомагане на дебъгването.
n	Символът е в .const data региона на паметта
p	Символът се намира в stack unwinding региона на паметта (C++ деструктори).
R, r	Същото като n.
S, s	Символът ще се намира в .bss региона за малки обекти.
T, t	Символът ще се намира в .text региона на паметта.
U	Символът е неуточнен.
u	Символът е уникален глобален символ. Линкерът ще се погрижи в цялото адресно поле да няма други символи с такова име.
V, v	Символ, деклариран като weak.
W, w	Същото като V.
?	Символът е неуточнен или специфичен за дадения обектов файл.

можете да подредите символите по адреси, а не по азбучен ред на имената им. Това става с аргумента -n, или -v, или - -numeric-sort.

12. Тествайте командата за статичен анализ на използваната памет mb-size [8]. С нейна помощ може да се определи размера на .text, .data и .bss регионите. Регионите stack и heap зависят от изпълнението на програмата и техният размер може да бъде определен приблизително със симулация. Поради сложността на проблема, GCC няма програма, която да е способна на това. Изпълнете програмата с аргумент main.elf файла на вашия проект:

```
mb-size main.elf
```

В миналото лабораторно упражнение беше споменато, че байтовете, които ще се запишат в програмната памет на MicroBlaze са:

FLASH total = text + data

По подразбиране командата използва форматът Berkeley за изобразяване на данните. По-детайлна разбивка на секциите от паметта дава форматът System V:

```
mb-size --format=sysv main.elf
```

където по-горната формула може да се разшири:

FLASH total = isr_vector + text + rodata + ARM.extab + ARM + preinit_array + init_array + fini_array + data

13. Програмата mb-objcopy копира съдържанието на един обектов файл в друг [9]. В процеса на копиране може да се приложат трансформации на данните. Именно заради това тази програма се използва често. Най-широкото ѝ използване е за преобразуване на обектови файлове с дебъгерна информация в обектови файлове без дебъгерна информация (“чисти” двоични файлове). Или накратко казано – с нейна помощ от .axf (.elf) файл се прави .bin файл.

Направете от .elf файлът един .bin, един .hex и един .rec файл като знаете аргументите на командата:

```
mb-objcopy -O [вид-изходен-файл] [име-на-входния-файл] [име-на-изходния-файл]
```

където “вид-изходен-файл” може да бъде binary, srec, ihex и др. (всички видове можете да видите с аргумента -info).

ВНИМАНИЕ: първо направете .bin файлът, след това от него направете другите два, като изрично укажете вид-входен-файл да е binary:

```
mb-objcopy -I [вид-входен-файл] -O [вид-изходен-файл] [име-на-входния-файл] [име-на-изходния-файл]
```

*

*

*

[1] <https://www.gnu.org/software/gdb/documentation>

[2] https://www.xilinx.com/htmldocs/xilinx2019_1/SDK_Doc/SDK_tasks/sdk_working_with_GDB.html

[3] <https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/GDB-GNU-Project-Debugger>

[4] <https://www.eclipse.org>

[5] <https://eclipse-embed-cdt.github.io>

[6] <https://mcuoneclipse.com/2016/04/09/freertos-thread-debugging-with-eclipse-and-openocd>

[7] <https://sourceware.org/binutils/docs/binutils/nm.html>

[8] <https://embarc.org/man-pages/binutils/size.html>

[9] <https://sourceware.org/binutils/docs/binutils/objcopy.html>

доц. д-р инж. Любомир Богданов, 2023 г.