Проектиране на вградени системи
Лабораторно упражнение №7

Алгоритъм за отделяне на ръбове и граници на обекти от цифрово изображение с многопроцесорна система върху чип.

==========

1. Стартира се развойната среда Eclipse Indigo, модифицирана за работа с Daedalus програмите, от горния панел на Desktop-а.

2. Създава се нов проект File → New → Other → Espam Designer → Espam Designer Project и се натиска Next.

3. Задава се име на проекта “sobel” и се оказва пътя до библиотеките, които ще бъдат използвани за синтеза. Това са библиотеките в директорията:

/home/user/programs/daedalus_gui/daedalus_examples/lib/espam_designer_lib

4. Полетата Platform, Application и Map се оставят празни. Натиска се бутон Finish.

5. Ако създаването на проекта е преминало успешно, ще се отвори средата Eclipse с Daedalus перспектива.

6. Избира се току-що създадения проект от дървото с проекти, натиска се десен бутон - → Import → File System → Next → From Directory: Browse → /home/user/programs/daedalus_gui/daedalus_examples/daedalus_sobel_template/ → OK → поставя се отметка на директорията sobel_template, което трябва да постави автоматично отметка на субдиректорията sequential и всички файлове вдясно (sobel.map, sobel.pla, Makefile, run.sh) → Finish.

7. От дървото на проекта се отваря сорс файлът sobel.c от директорията sequential. Въвежда се код на програма за отделяне на ръбове и граници на обекти от цифрово изображение (edge detection), използвайки Собел (Sobel) оператора. Нека програмата да съдържа:

- функция readPixel, която да има един изходен параметър, който е указател към целочислена стойност(int), която е прочетена от файл. Нека функцията да отваря .raw файл с име car_gray.B, да чете един символ (char) и да го предава на изходния параметър. Този файл всъщност е raw изображение (450X275), съдържащо само синята компонента на JPEG картинка с автомобил.
- функция writePixel, която да приема указател към променлива от целочислен тип и да я записва във файл с име car_gray_sobel.raw.
- функция gradient, която да има 6 целочислени входа и един целочислен изход. Нека тя да изчислява градиента на даден пиксел.
- функция absVal, която да изчислява абсолютната стойност на вектора |G| = |Gx| + |Gy| (което ни позволява да не използваме корен квадратен и степенуване).
- функция threshold, която да определя дали даденият вектор е контур или заден фон. Нека тази

граница да е 32, като задният фон да се изобразява с бяло (255), а контурът – с черно (0).
- Нека всички функции от ниско ниво за запис и четене на файл да се разделят в един отделен сорс и хедър файл с имена imageIO.h и imageIO.c. Така, ако не включим тези файлове в проекта, по-късно в упражнението входните данни (изображението) ще могат да се записват през RS232 порта във външната RAM памет на XUPV5-LX110T Evaluation Platform от Xilinx.

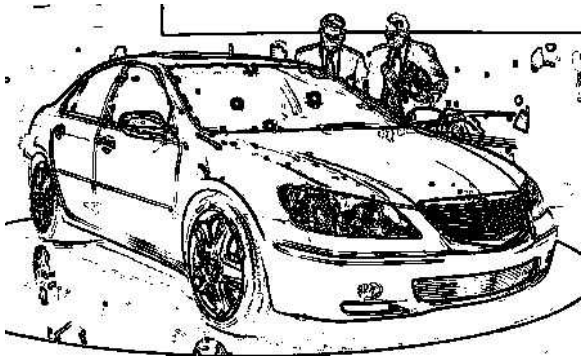
ВНИМАНИЕ: В края на лабораторното е дадена програма, която отговаря на тези Daedalus правилата и която реализира алгоритъма на работа.

Алгоритъмът на програмата да се напише в main( ), като се спазват правилата за Daedalus статичен код (Daedalus static affine nested-loop program) и се използват горе-изброените функции. Компилира се от Run → External Tools → Compile Sequential.

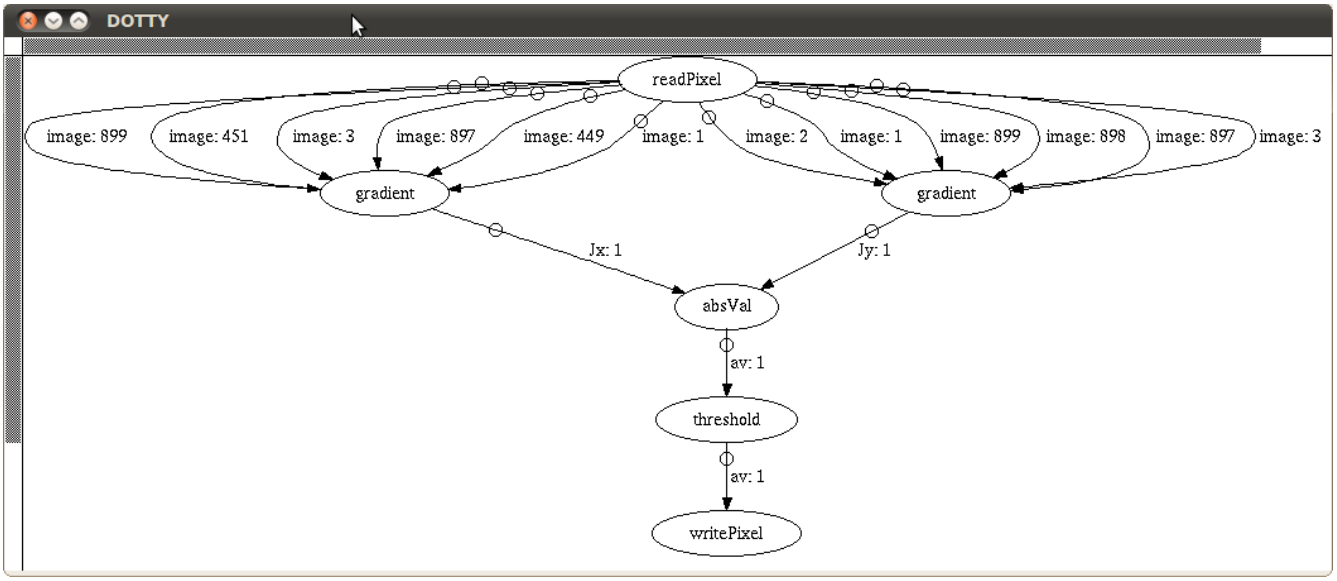
8. Стартира се последователната програма. За целта се избира Run → External Tools → Run Sequential. Ако алгоритъмът е правилен, ще се отвори файла car_gray_sobel.jpg и ще се изобрази картинка с бял фон и контурите на обектите (**фиг. 1 а**).

9. Избира се Run → External Tools → Generate KPN (no reuse).

10. Резултатите от генерирания паралелен код могат да бъдат визуализирани. Чрез Run → External Tools → Visualize KPN се изобразява т.нар. Kahn Process Network, който в случая е показан на **фиг. 1 б)**. Вижда се, че процесната мрежа отговаря логически на програмата, следователно синтеза може да продължи.



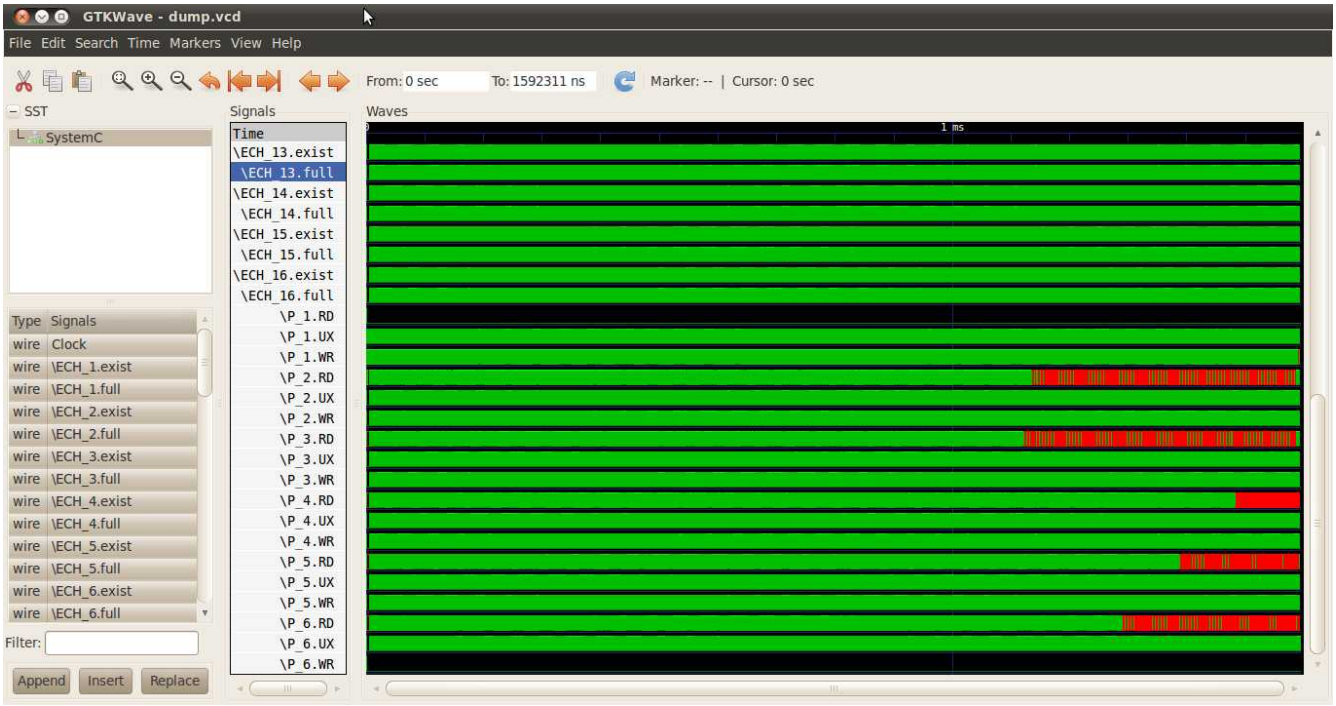
Фиг. 1 а) – Обработено изображение.



Фиг. 1 б) – Процесна мрежа на програмата.

11. Извършва се SystemC Untimed симулация. Това става, като се избере Run → External Tools → Generate SystemC Untimed и след това Run → External Tools → Simulate SystemC Untimed. Да се обърне внимание на статистиката, принтирана в терминала. Резултатите от изпълнението на паралелната програма трябва да са същите като резултатите от изпълнението на последователната програма.
Ако паралелната програма работи правилно, полученото изображение трябва да е същото като от стартирането на sequential кода.

12. Извършва се SystemC Timed симулация. Това става, като се избере Run → External Tools → Generate SystemC Timed и след това Run → External Tools → Simulate SystemC Timed. Да се обърне внимание на статистиката, принтирана в терминала. Резултатите от симулацията могат да се визуализират, като се избере Run → External Tools → Visualize Timed Simulation. Автоматично се отваря програмата GTKWave. Избира се SystemC от SST дървото, вляво на екрана. Появяват се всички налични процеси. Натиска се първият обект и се задържа бутона shift (**фиг.2**). С мишката се избират всички останали обекти до края на прозореца. Натиска се Append, след което Zoom Fit иконката:



Фиг. 2 - Резултати от timed симулацията, показани в GTKWave.

13. Проверява се горе вдясно на екрана дали настоящата перспектива е Daedalus. Ако името е изписано със стрелки <<Daedalus>>, то тази перспектива се затваря и се отваря само с името Daedalus. Избира се Project → Properties → Espam Project Settings → в полето Application се указва sobel.kpn от top-level директорията чрез browse → OK.

ВНИМАНИЕ: ако GUI дава Java грешки, уверете се, че сте стартирали Daedalus Eclipse с Java 6!

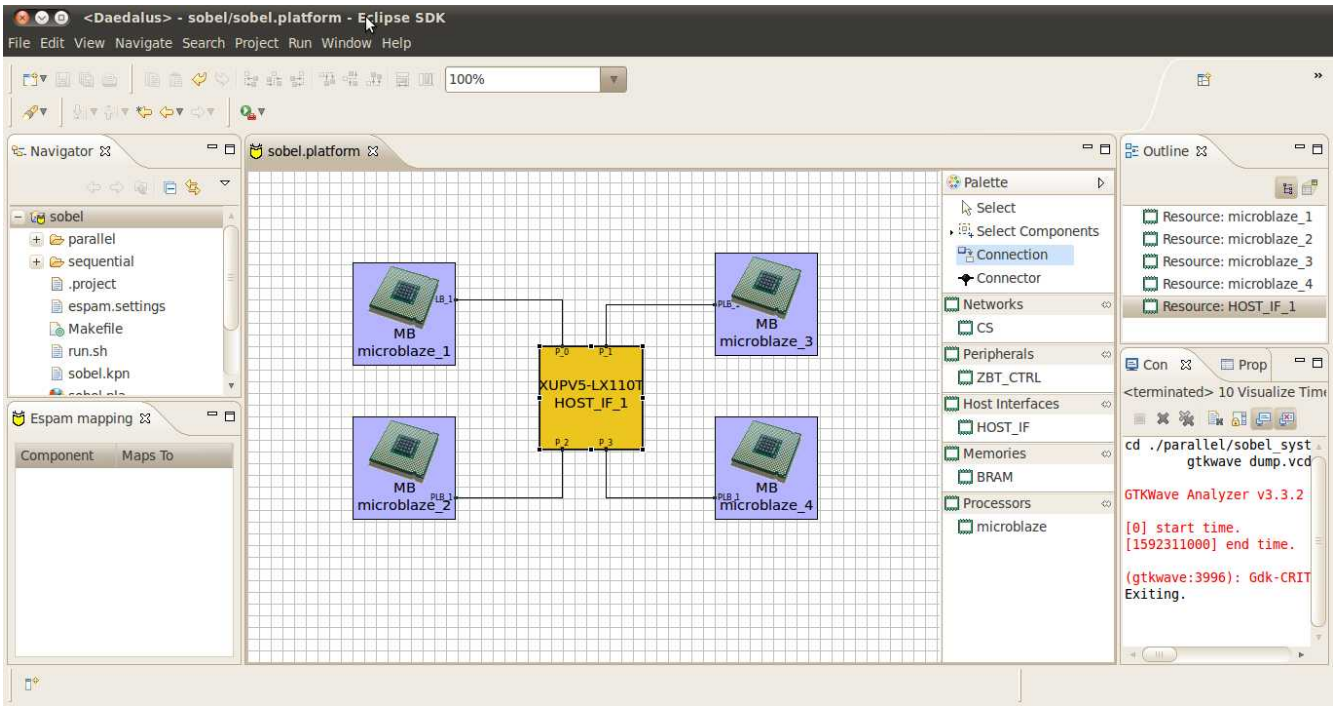
./eclipse -vm /usr/lib/jvm/java-6-openjdk-i386/bin/java

Също така десен бутон върху top-level директорията → Properties → Espam Project Settings → уверете се, че в полетата Platform Lib и Application Lib е указан правилния път!

14. Създава се платформена диаграма, която се използва за конфигуриране на хардуерната част. Досега се използваше празен платформен файл, за да се стартират симулациите. В платформената диаграма потребителя начертава блоковата схема на бъдещата система, която ще бъде заредена в FPGA. За целта се избира проекта sobel от дървото и след това File → New → Other → Espam Designer → Platform Diagram → File Name: името на проекта – в случая sobel.platform → Finish. Отваря се редактор за блокова схема в средата на екрана.

15. В настоящото упражнение ще се синтезира система с четири процесора.
От “Palette” вдясно на екрана се избира microblaze микропроцесор и се разполагат 4 броя в полето на платформената диаграма (**фиг.3**). Аналогично се слага блок HOST_IF, който указва вида на използвания интерфейсен модул на FPGA. Връзката между microblaze_1, microblaze_2, microblaze_3, microblaze_4 и HOST_IF се извършва с помощта на Connection инструмента от Palette. При свързването се появява диалогов прозорец, с помощта на който се указва коя шина

на процесора да бъде свързана с HOST_IF. Избира се PLB, което е съкратено от Processor Local Bus. Натиска се OK.
Натиска се Ctrl + S, с което се запазва sort.platform и автоматично се генерира необходимият ни за синтеза sort.pla файл. Избира се Project → Properties → Espam Project Settings → в полето Platform се указва sobel.platform от top-level директорията чрез browse → OK.



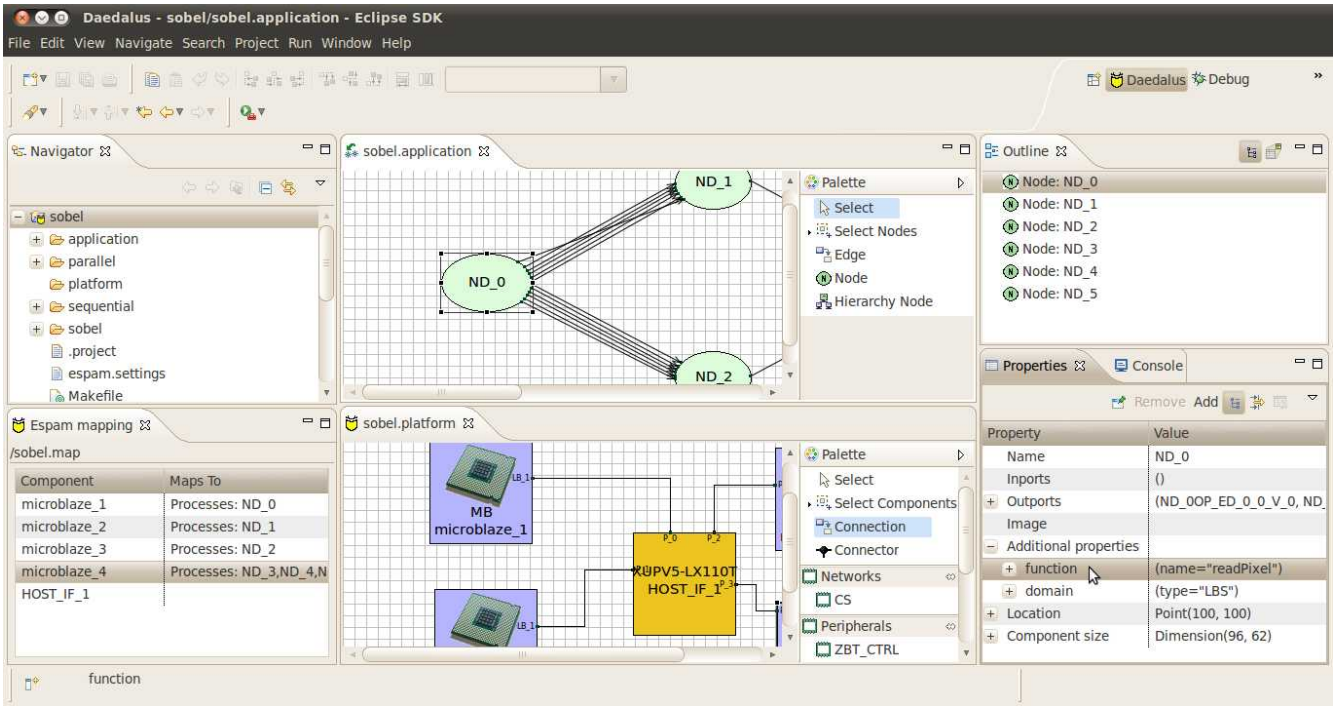
Фиг. 3 – Блокова схема на системата.

16. Следва да се генерира mapping файл. Досега се използваше празен mapping файл, за да се стартират симулациите. Mapping файла указва кой процес от паралелната програма на кой процесор да бъде програмиран. Затова в тази стъпка ние ще укажем на средата, че искаме софтуерния код за процеси 1, 2, 3 и 4 да се изпълни от съответно микропроцесори microblaze_1, microblaze_2, microblaze_3 и microblaze_4, разположени в платформената диаграма.
Вляво на екрана, полето Espam mapping ще бъде попълнено с имената на блоковете от платформената диаграма. В случая имаме полетата – microblaze_1, microblaze_2, microblaze_3, microblaze_4 и HOST_IF. Избира се microblaze_1 и в колоната Maps To се щраква с мишката, за да се появи прозорец за редактиране. Имената на процесите са ND_0, ND_1, ND_2 и ND3. Избира се ND_0 и с бутона - > се премества в дясната колонка, която съдържа процесите, които ще изпълнява microblaze_1. Натиска се бутон OK. Аналогично се присвояват процеси ND_1 → microblaze_2, ND_2 → microblaze_3 и ND_3, ND_4, ND_5 → microblaze_4. HOST_IF не се модифицира. FIFO каналите (в полето Available Channels) за междупроцесорната комуникация не се модифицират, защото те ще бъдат автоматично свързани към съответния процесор. След това с CTRL + S се генерира и записва автоматично mapping файл.
Забележка: имената на функциите, които отговарят на ND_0, ND_1, ND_2 и т.н. могат да бъдат проверени в прозореца sobel.application (точно над sobel.platform). Натиска се даден процес и долу вдясно на екрана се избира таб Properties (**фиг. 4**). Оттам в колонката Properties се търси ред Additional, натиска се + отстрани на него и трябва да се види ред “function”. В колонката Value на

същия този ред трябва да е написано името на функцията, което отговаря на този процес.

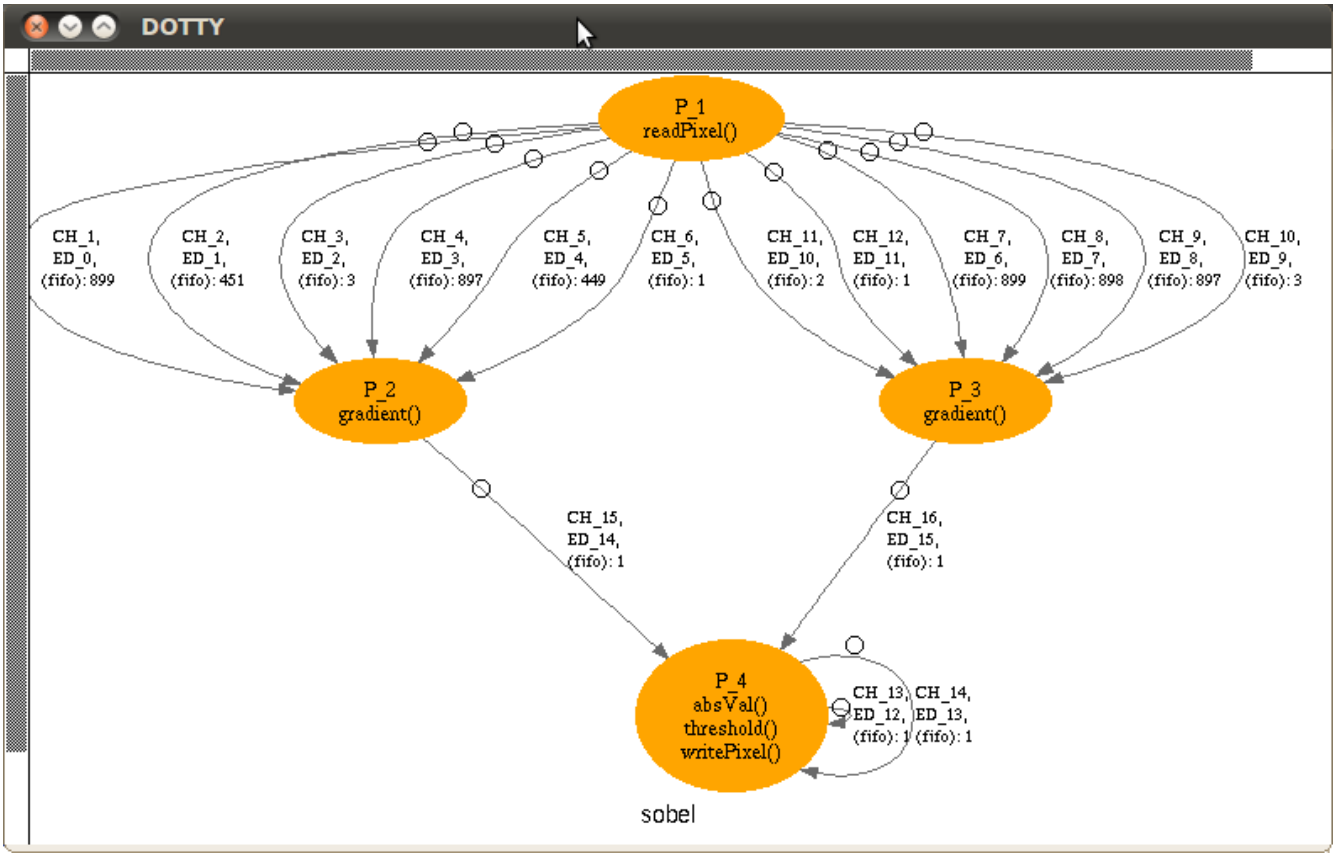
Избира се Project → Properties → Espam Project Settings и се попълва полето Map със sobel.map от top-level директорията. Сега полетата Platform, Application и Map трябва да се попълнени.

Трите най-важни файла – platform, application (в случая KPN файла, показан в прозореца sort.application) и mapping са генерирани и сега може да се премине към синтез.



Фиг. 4 – Връзка между името на процеса от мрежата и името на функцията.

17. Синтезира се MPSoC системата. За целта се избира Run → External Tools → Synthesize MPSoC. Ако в прозореца Console не се изведе съобщение за грешка, значи че синтезът е преминал успешно и директорията /home/student-tu/workspace/sobel/sobel не трябва да е празна. Процесната мрежа на системата може да бъде визуализирана с Run → External tools → Generate SystemC Timed и след това Run → External tools → 10_Visualize KPN (**фиг. 5**). Да се направи сравнение с процесната мрежа от **фиг. 1**.



Фиг. 5 – Процесна мрежа на системата.

18. Следващата стъпка е да се създаде Xilinx проект с вече готовите файлове и да се генерира bitstream файл, който да се зареди във FPGA. Стартира се терминал, като се натисне комбинацията CTRL + ALT + T. Подготвя се средата на операционната система за работа с Xilinx – за целта се изпълнява командата в терминала:

```
student-tu@ubuntu:~$ source .xilinx
```

която трябва да изпише следните редове:

```
./home/student/ProgramFiles/Xilinx/13.4/ISE_DS/EDK/.settings64.sh
/home/student/ProgramFiles/Xilinx/13.4/ISE_DS/EDK
./home/student/ProgramFiles/Xilinx/13.4/ISE_DS/ISE/.settings64.sh
/home/student/ProgramFiles/Xilinx/13.4/ISE_DS/ISE
./home/student/ProgramFiles/Xilinx/13.4/ISE_DS/PlanAhead/.settings64.sh
/home/student/ProgramFiles/Xilinx/13.4/ISE_DS/PlanAhead
./home/student/ProgramFiles/Xilinx/13.4/ISE_DS/SysGen/.settings64.sh
/home/student/ProgramFiles/Xilinx/13.4/ISE_DS/SysGen
./home/student/ProgramFiles/Xilinx/13.4/ISE_DS/common/.settings64.sh
/home/student/ProgramFiles/Xilinx/13.4/ISE_DS/common
```

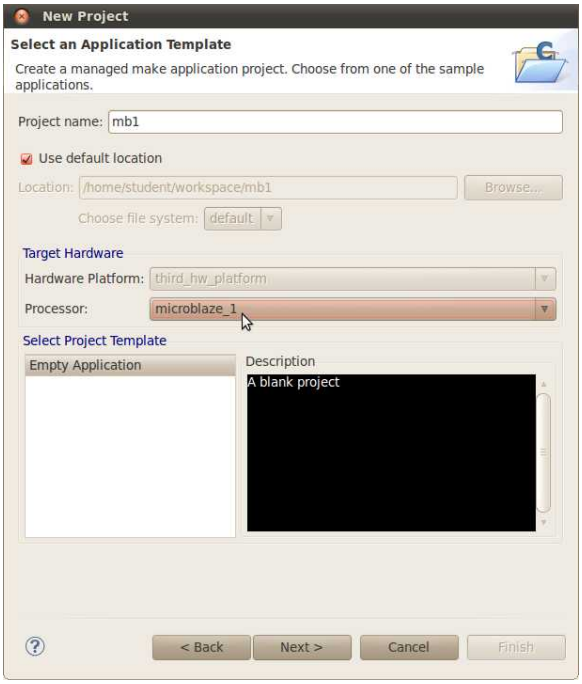
След това се изпълнява командата xps. Тя стартира средата Xilinx Platform Studio. В нея се избира File → Open Project и се указва пътя до /home/student-tu/workspace/sobel/sobel

/system.xmp. Появява се прозорец, който ни информира, че проектът, който отваряме е за по-стара версия на Xilinx и ни пита дали искаме да го конвертираме за новата версия. Отговаря се с Yes и започва процеса на преобразуване. Следват се указанията на диалоговите прозорци, като единственото нещо, на което трябва да се обърне внимание е на прозореца “New, Compatible Revisions of Cores”. Тук се маха отметката на mpmc модула, който представлява контролер на памет и е чувствителен към промени. За да се създаде успешно Xilinx проект е необходимо да се запази старата версия на библиотеката на този контролер.
Ако конвертирането е преминало успешно ще се отвори новият проект в Xilinx Platform Studio.

19. Компилира се фърмуера за микропроцесорите. Това става с помощта на XSDK. Преди да се премине нататък трябва първо да се затвори ESPAM проекта и версията на Eclipse, модифицирана за работа с ESPAM (т.е. средата, в която синтезирахме MPSoC). След това от XPS се избира Project → Export Hardware Design To SDK и в новопоявилият се прозорец се маха отметката на Include Bitstream and BMM file (**фиг. 6**).
Натиска се бутонът Export & Launch SDK. Ако не са възникнали грешки, ще се отвори XSDK и ще се създаде един проект sort_hw_platform, съдържащ хардуерното описание на системата (описано в .MHS файла).

Фиг. 6 – Експорт на проект за фърмуер в Xilinx SDK.

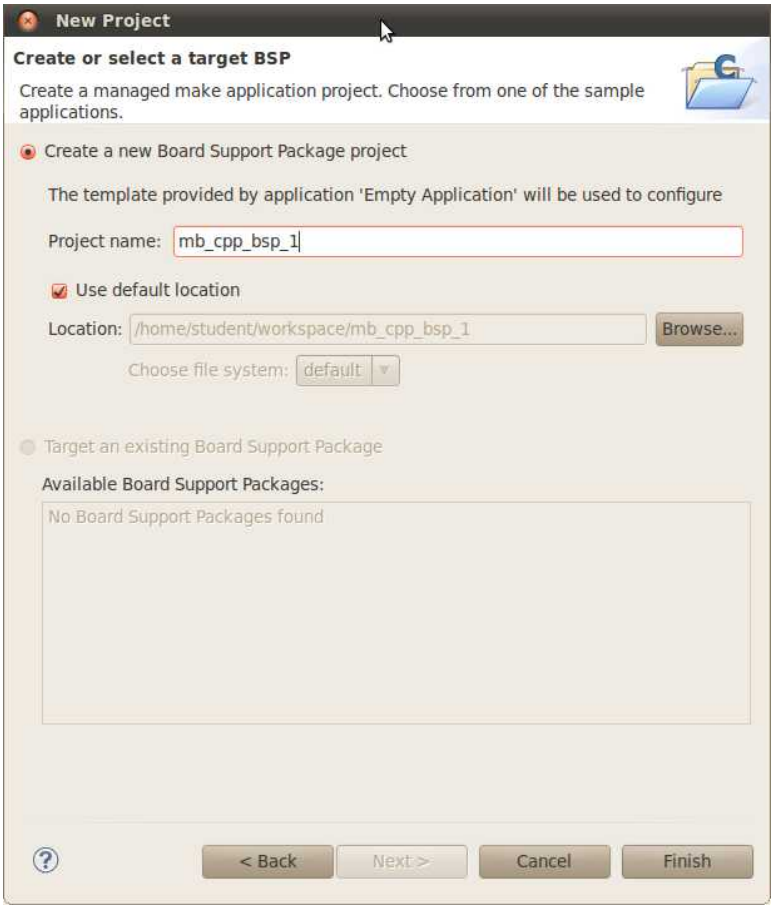
19.1 Създават се нови C++ проекти. За целта се избира File → New → Other → Xilinx C++ Project → Next. В следващия прозорец се указва името на проекта и процесорът, за който ще се пише фърмуера. Нека името да е mb1 за първия (microblaze_1) микропроцесор. От падащото меню се избира микропроцесорът microblaze_1 и се натиска бутон Next (**фиг. 7**).



Фиг. 7– Указване на процесора, за който ще се пише фърмуер.

В следващия прозорец се указва името на нов проект, който ще се създаде автоматично и който съдържа информация за използваната демо платка, на която се намира FPGA (затова типа на проекта се казва “Board Support Package”). Въвежда се например името mb_bsp_1 и се натиска Finish (**фиг. 8**). При създаването на проектите за другите процесори аналогично за всеки се прави по един BSP проект.

19.2 Импортват се сорс файловете, генерирани от ESPAM. За целта от дървото на проекта mb1 се избира директорията src и файла main.cc се изтрива. Дава се десен бутон върху src директорията → Import → File System → Next → From Directory: Browse → указва се пътя /home/student-tu/workspace/sobel/sobel/code/P_1 → OK и се поставя отметка на P_1.cpp → Finish. Аналогично с десен бутон върху src директорията → Import → File System → Next → From Directory: Browse → указва се пътя /home/student-tu/workspace/sobel/sobel/code/func_code → OK → поставя се отметка на sobel_func.cpp и sobel_func.h → Finish.
След това се добавя директорията на файловете aux_func.h и MemoryMap.h, като се избере Project → Properties → C/C++ General → Paths and Symbols → Includes → Add → File System → /home/student-tu/workspace/sobel/sobel/code → OK → OK.
Избира се проекта mb1 от дървото в XSDK. След това Xilinx Tools → Generate Linker Script → Stack Size се променя на 3072 байта (3 kB) → Generate.
Избира се ниво на оптимизация O2 с Project → Properties → C/C++ Build → Settings → Tool Settings → Microblaze g++ compiler → Optimization → Optimization Level: избира се от падащото меню “Optimize more (-O2)” → OK.



Фиг. 8 – Създаване на Board Support Package проект.

19.3 Проектът се компилира с Project → Build Project. На този етап ще се изведе грешка в терминала:

```
../src/sobel_func.cpp:46: error: expected initializer before ‘*’ token
../src/sobel_func.cpp:48: error: ‘fh’ was not declared in this scope
../src/sobel_func.cpp:49: error: ‘mwOpen’ was not declared in this scope
../src/sobel_func.cpp:52: error: ‘fh’ was not declared in this scope
../src/sobel_func.cpp:52: error: ‘mPutc’ was not declared in this scope
make: *** [src/sobel_func.o] Error 1
```

която произлиза от факта, че не сме включили файловете imageIO.c и imageIO.h. Тяхното включване в проекта е безсмислено, защото в микропроцесора няма да бъде заредена операционна + файлова система и функциите fopen( ), fclose( ) и printf( ) губят своя смисъл за файлов вход/изход и принтиране на съобщения в терминала.
Затова всички извиквания на printf( ) трябва да бъдат коментирани, а четенето и запис на данни при обработката на изображение да стават от/във външната RAM памет, започваща от адрес 0x9000 0000. Корекциите трябва да се направят във sobel_func.c на съобтветния процесор. Например четенето на изображението става в P_1, следователно ще се редактира sobel_func.c на проекта mb1 и т.н.
При повторно компилиране грешки няма да има и генерираният изпълним файл е с разширение

ELF (може да се види в дървото на проекта → Binaries → mb1.elf). Файлът mb1.elf е фърмуера, който ще се изпълнява от микропроцесора.

19.4 Аналогично се повтарят стъпки 19.1 ÷ 19.3 за microblaze_2, microblaze_3, microblaze_4 и host_if процесорите. Единствената особеност е за host_if, за който трябва да се добави само един copc файл към проекта и той се намира в директорията /home/student-tu/workspace/sobel/sobel/code/HOST_IF/main_PLB.cpp. Също така за Xilinx Tools → Generate Linker Script → Stack Size трябва да е 1024 байта (1 kB).

20. Генерира се bitstream файл, който ще се зареди в FPGA. За целта първо се указва пътя до фърмуера за микропроцесора.
В средата XPS вляво има прозорец с два tab-а: Project и IP Catalog. Избира се tab-а Project → Project Files → Elf Files → microblaze_1 → Imp Executable: → десен бутон → Browse → указва се пътя /home/student-tu/workspace/mb1/Debug → избира се mb1.elf → Open. Аналогично се указват пътищата до ELF файловете на другите три процесора. След това се указва пътя на интерфейсния процесор host_if → Imp Executable: → десен бутон → Browse → указва се пътя /home/student-tu/workspace/host_if/Debug → избира се host_if.elf → Open.
След това се избира Hardware → Generate Netlist. В долната част на екрана в XPS се избира tab-а Console и се следи изпълнението на процеса. Той отнема много време, защото сега трябва да се синтезира принципната схема на цялата система от предоставените IP библиотеки. Колкото повече процесори имаме в системата, толкова по-бавен е синтезът. Ако синтезът е минал успешно в Console ще се изпише:

```
XST completed
Done!
```

Избира се Hardware → Generate Bitstream. Ако файлът е генериран успешно в Console ще се изпише:

```
Saving bit stream in "system.bit".
Bitstream generation is complete.
Done!
```

21. Зарежда се bitstream файла в FPGA. За целта се стартира терминал, като се натиснат CTRL + ALT + T и в него се изпълнява командата impact. Тя стартира програмата, с помощта на която bitstream-ът ще бъде зареден. Появява се прозорец за създаване на нов проект или отваряне на стар. Натиска се бутон Cancel.
Вляво на екрана се появява дърво iMPACT Flows. В него се избира Boundary Scan и се натиска два пъти с мишката. Вдясно се отваря прозорец, в който пише Right Click to Add Device or Initialize JTAG Chain.
Свързва се USB-JTAG дебъгерът и се включва Virtex-5 XUPV5-LX110T демо платката. От виртуалната машина (менюто горе, по средата на екрана) се избира Virtual Machine → Removable Devices → xilinx → Connect (Disconnect from host).
След това с десен бутон в полето на прозореца се избира Initialize Chain. Показва се прозорец за избор на устройство. Понеже на JTAG веригата са свързани повече от едно устройства, трябва да се избере само FPGA и да се натисне OK (**фиг. 9**).

Фиг. 9 – Избор на устройство от JTAG веригата.

Сега в празния прозорец трябва да са изобразени схематично всички устройства от JTAG веригата и само FPGA, в случая xc5vlx110t, да е зелен. С десен бутон върху FPGA се избира Assign New Configuration File и се указва пътя до bitstream файлът, който генерирахме /home/student-tu/workspace/sort/sort/implementation/download.bit. Появява се прозорец “Attach SPI or BPI PROM”, на който се отговаря с No (bitstream-ът ще бъде зареден директно в матрицата през JTAG интерфейса).
След това отново с десен бутон се избира Program → Device 5 (FPGA xc5vlx110t) → OK и се появява прозорец, показващ процеса на зареждане (**фиг. 10**).

Фиг. 10 – Зареждане на bitstream във FPGA.

Ако зареждането мине успешно в прозореца вдясно на iMPACT трябва да се покаже надпис със сини букви Program Succeeded.

22. Стартира се програмата serial_XUPV5_sobel, която извършва всичко за нас автоматично. Тя записва car_gray.B файла във външната RAM памет на адрес 0x9000 0000 през RS232 порта на компютъра. След това се стартират четирите процесора на FPGA и се изчаква тяхното изпълнение да завърши. Когато това стане, програмата ще продължи с четене на обработеното изображение от адрес 0x9010 0000, отново през RS232. Накрая полученият .raw файл ще бъде записан като car_sobel.raw и конвертиран в JPEG формат, получавайки файл car_sobel.jpg. Последният се визуализира с програма от Линукс и полученото изображение трябва да е същото, като изображението от стартиране на sequential програмата.
23. Да се измери времето на изпълнение на програмата, като се използва програмата serial_XUPV5_sort от предходното упражнение. За целта да се избере команда 3) за изпълнение

на програмата и след това 5) - Get total measured time. Един примерен резултат е:

```
Execution time = 26390840
```

или около 26.4 млн системни такта.

26. Да се измерят времената на изпълнение на всяко едно извикване на функциите readPixel, gradient, absVal, threshold, writePixel. Измерванията за writeFSL и readFSL да се вземат от предходното упражнение (използват се същите процесори). Преди извикването на всяка функция да се нулира независим сумиращ брояч, който да увеличава стойността си на всеки процесорен такт. Понеже всяка функция се намира във for(;;) цикъл, нека резултата от брояча да се натрупва и една променлива да отмерва броя на натрупванията. Така в края на процеса може да се изчисли средноаритметичната стойност и да се запише във външната RAM памет.
Резултатите да се прочетат с команда 4) и да се приложат в SystemC timed симулацията, като се редактира файла /home/student-tu/workspace/sobel/parallel/ /sobel_systemc/workload.h. Примерни резултати са:

```
readPixel = 51
gradient = 22
absValue = 145
threshold = 14
writePix = 26
```

27. Да се извърши SystemC timed симулация, като се изпълнят **само** двете команди Run → External Tools → Simulate SystemC Timed и Run → External Tools → Visualize Timed Simulation. Да се сравнят резултатите за Total Execution Time от симулацията и от измерванията.

SystemC Timed симулацията, използвайки горния пример, даде резултат за TOTAL EXECUTION TIME: 26596886. Ако сравним измерените резултати и симулираните ще видим, че:

$t_{sim} > t_{meas}$

като $\Delta t = 206046$.

```
/*********************************************************************
************************ sobel.c *********************************
*********************************************************************/

#include "sobel_func.h"

#define N 450
#define M 275

int main(void)
{
   int i, j;

   static int image[1000][1000];
   static int Jx[1000][1000];
   static int Jy[1000][1000];
   static int av[1000][1000];


   for (j=0; j < M; j++) {
      for (i=0; i < N; i++) {
         readPixel(&image[j][i]);
      }
   }

   for (j=1; j < M-1; j++) {
      for (i=1; i < N-1; i++) {
           gradient( &image[j-1][i-1], &image[j][i-1], &image[j+1][i-1], &image[j-
1][i+1], &image[j][i+1], &image[j+1][i+1], &Jx[j][i] );
      }
   }

   for (j=1; j < M-1; j++) {
      for (i=1; i < N-1; i++) {
                  gradient(  &image[j-1][i-1],  &image[j-1][i],  &image[j-1][i+1],
&image[j+1][i-1], &image[j+1][i], &image[j+1][i+1], &Jy[j][i] );
      }
   }

   for (j=1; j < M-1; j++) {
      for (i=1; i < N-1; i++) {
           absVal( &Jx[j][i], &Jy[j][i], &av[j][i] );
           threshold( &av[j][i], &av[j][i] );
      }
   }

   for (j=1; j < M-1; j++) {
      for (i=1; i < N-1; i++) {
          writePixel( &av[j][i] );
      }
   }

   return (0);
}
```

```
/*****************************************************************************
*****************************sobel_func.c*************************************
*****************************************************************************/

#include <stdlib.h>
#include "imageIO.h"
//#include "MemoryMap.h"
#include "sobel_func.h"

/*
//volatile int *source_ptr = (volatile int *)0x90000000;
//volatile int *dest_ptr = (volatile int *)0x90100000;
/*/
volatile unsigned char *source_ptr = (volatile unsigned char *)0x90000000;
volatile unsigned char *dest_ptr = (volatile unsigned char *)0x90100000;
//*/

void gradient( const int *a1, const int *a2, const int *a3, const int *a4, const
int *a5, const int *a6, int *output ){

    *output = ( ((*a4)+((*a5)<<1)+(*a6)) - ((*a1)+((*a2)<<1)+(*a3)) );

}


void absVal( const int *x, const int *y, int *output ) {

    *output = ( (abs(*x)+abs(*y))/4 );

}

void threshold( const int *x, int *output ) {

   if( *x > 32 )
      *output = 0;
   else
      *output = 255;
}


void readPixel( int *output ) {
//*
        int pp;
      static FILE *fh = NULL;

      if (fh == NULL) {
           fh = mrOpen("car_gray.B");
      }

      pp = mGetc(fh);
      *output =  pp;
/*/
        static int addr = 0;
        *output = source_ptr[addr++];
```

```
//*/
}

void  writePixel( const int *pp ) {
//*
      static FILE *fh = NULL;

      if (fh == NULL) {
           fh = mwOpen("car_gray_sobel.raw");
      }

      mPutc(fh, *pp);
/*/
        static int addr = 0;
        dest_ptr[addr++] = *pp;
//*/
}

/
*****************************************************************************
**********************************sobel_func.h*********************************
*****************************************************************************
*/
#ifndef SOBEL_FUNC_H
#define SOBEL_FUNC_H

void gradient( const int *a1, const int *a2, const int *a3, const int *a4, const
int *a5, const int *a6, int *output );

void absVal( const int *x, const int *y, int *output );

void threshold( const int *x, int *output );

void readPixel( int *output );

void  writePixel( const int *pp );

#endif

/*************************************************************************
*********************** imageIO.c ************************************
*************************************************************************/

#include <stdlib.h>
#include "imageIO.h"

//--------------------------------------------
// mrOpen() opens a file for read
//--------------------------------------------
FILE *mrOpen(char *filename) {

      FILE *fh;
      if ((fh = fopen(filename,"rb"))==NULL) {
          printf("Cannot read input file %s\n", filename);
          exit(1);
```

```
      }
      return(fh);

}

//-------------------------------------------------
// mwOpen() creates a file for write
//-------------------------------------------------
FILE *mwOpen(char *filename) {

      FILE *fh;
      if ((fh = fopen(filename,"wb"))==NULL) {
          printf("Cannot create file %s\n", filename);
          exit(1);
      }
      return(fh);

}


//-----------------------------------
// mClose() closes a file
//-----------------------------------
void mClose(FILE *fh) {

    fclose(fh);

}

//-----------------------------------------------------------
//mGetc() gets a character from a file
//-----------------------------------------------------------
int mGetc(FILE *fh) {

    return( getc(fh) );

}

//-------------------------------------------------
//mPutc puts a character to a file
//-------------------------------------------------
void mPutc(FILE *fh, int a) {

    putc(a, fh);

}
```

```
 /*****************************************************************************
******************** imageIO.h ***********************************************
*****************************************************************************/
#ifndef IMAGEIO_H
#define IMAGEIO_H

#include <stdio.h>

//---------------------------------------------
// mrOpen() opens a file for read
//---------------------------------------------
FILE *mrOpen(char *filename);

//-----------------------------------------------
// mwOpen() creates a file for write
//-----------------------------------------------
FILE *mwOpen(char *filename);

//-----------------------------------
// mClose() closes a file
//-----------------------------------
void mClose(FILE *fh);

//-------------------------------------------------------
//mGetc() gets a character from a file
//-------------------------------------------------------
int mGetc(FILE *fh);

//-----------------------------------------------
//mPutc puts a character to a file
//-----------------------------------------------
void mPutc(FILE *fh, int a);

#endif
```

* * *

[1] Stefanov T., Deprettere E., Nikolov H., Marinov M., Popov A., *„Embedded Systems: components, modelling, design and case study“*. TU-Sofia, 2012, ISBN 978-954-438-975-8.

[2] Bogdanov L., H. Nikolov, T. Stefanov. *„Embedded Multi-processor Systems-on-Chip: Laboratory Experiments and Users Manual”*, TU-Sofia, 2013, ISBN 978-619-167-034-5.

инж. д-р Христо Николов (LIACS), гл. ас. д-р инж. Любомир Богданов, 2021 г.