Laboratorul 3

Introducere în Arhitectura Zynq: Manipularea perifericelor folosind Sistemul de Procesare

1. Noțiuni generale legate de sistemele bazate pe Zynq si diferența față de alte sisteme de calcul

Sistemele de tipul System-on-Chip (SoC) reprezintă o inovație majora in proiectarea sistemelor de calcul moderne, reunind într-un singur cip toate elementele necesare pentru realizarea unui sistem complet cum ar fi procesorul, memoria, interfețe de comunicare si metode de accelerare hardware precum logica programabila. SoC-urile moderne reduc complexitatea hardware, dimensiunea cipului, costurile de producție ale sistemului si eficienta energetica prin eliminarea nevoii de a integra componente separate pe un cablaj (PCB). Un exemplu de SoC îl reprezintă cel prezent pe plăcile Zybo 7000 sau Zybo-Z20, care combina sinergic puterea de procesare a unui procesor cu flexibilitatea unui FPGA modern.
Un SoC modern include mai multe subsisteme esențiale:

- **Sistemul de procesare (PS)** – in cazul Zynq aceasta componenta este reprezentata de un procesor dual-core Cortex-A9 ARM care oferă performanta de procesare ridicata si suport pentru sisteme de operare complexe cum ar fi sistemul de operare Linux. Mai mult pentru eficientizare procesorul are cache-uri de nivel 1 si 2, o unitate de management a memoriei (MMU) si o unitate de control a snoop-ului(SCU) care ajuta menținerea coerentei cache-ului intre nuclee pentru a asigura o viteza de procesare cat mai mare. Aceasta componenta, SCU, este importanta deoarece se asigura ca toate nucleele din sistem au acces la aceeași versiune a datelor, iar atunci când un nucleu modifica o informație in cache, SCU trimite semnale către celălalt nucleu pentru a actualiza sau a invalida datele din cache-ul lor, un proces cunoscut sub numele de snooping. De asemenea, SCU gestionează si accesul la cache-ul de nivel 2 care este partajat intre nuclee si se asigura ca accesările la cache sunt corecte si coerente.
- **Logica Programabila (PL)** – Sistemul Zynq are integrat un FPGA care permite implementarea unor componente/funcționalități personalizate cum ar drivere de motoare, interfețe de comunicare care nu se găsesc pe aceasta placa sau acceleratoare hardware pentru procesarea paralela a semnalelor/imaginilor sau pentru calcule complexe care ar putea rula încet sau ar încetini un procesor tradițional. De asemenea, logica programabila poate fi utilizata si pentru a implementa procesoare soft cum ar fi MicroBlaze. Aceste procesoare mai pot fi folosite pentru taskuri mai puțin intensive cum ar fi filtrări de baza pe semnale înainte de a fi procesate pe ARM, sisteme de criptare care nu necesita resursele Cortexului A9, sisteme de control precum PID etc.
- **Interfețe de comunicare –** SoC-ul oferă o gama variata de interfețe de comunicare intre placa si alte dispozitive sau periferice. Printre acestea se numără USB, Ethernet, UART, I2C si SPI.

Integrarea componentelor de pe Zynq permite o comunicare rapida intre procesor si FPGA cu o latenta scăzută, prin intermediul interfețelor de tip AXI(advanced eXtensible Interface). De asemenea, întrucât elementele Zynq sunt interconectate pe același cip spre deosebire de sisteme in care FPGA-ul si procesorul sunt conectate fizic prin fire sau interfețe, in dispozitivele Zynq de tip SoC eficienta energetica este mult mai buna, dimensiunea fizica a dispozitivelor e mai mica si costurile de producție sunt mai mici. Prezenta PL-ului in Zynq oferă flexibilitatea de a personaliza hardware-ul pentru a îndeplini cerințe specifice, fără a compromite performanta sistemului. Spre deosebire de soluțiile in care exista doar FPGA, cu un procesor este implementat in logica

programabila, cum ar fi Microblaze, sistemele bazate pe Zynq SoC beneficiază de performanta ridicata a unui procesor ARM dedicat care poate rezolva taskuri complexe cu algoritmi sofisticaţi in timp real si pe care se poate rula un sistem de operare complex precum Linux.

Mai mult in comparaţie cu sistemele care utilizează doar un procesor avansat, Zynq oferă avantajul suplimentar al logicii programabile care poate fi utilizat pentru taskuri de accelerare hardware care ar consuma multe resurse si ar dura mult timp daca ar fi implementate doar in software. In Figura 3.1, preluata din [1], se poate observa schema bloc generala a unui sistem Zynq SoC cu componentele sale principale: PS, PL si interfeţe de comunicare.

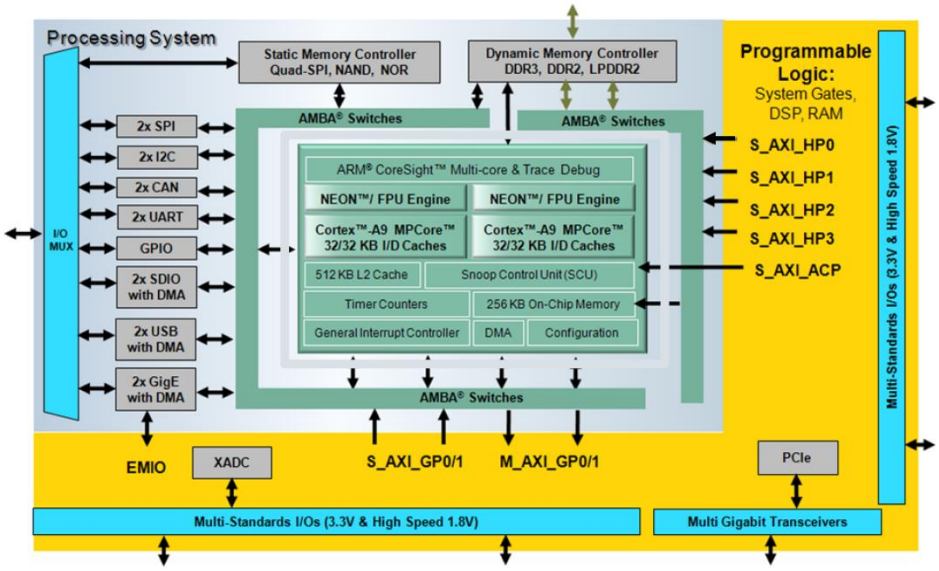

Figura 3.1. Diagrama bloc a unui system Zynq SoC [1]

Aşadar plăcile bazate pe Zynq aduc o valoare considerabila in dezvoltarea de sisteme embedded, oferind o serie de avantaje fata de sistemele clasice bazate doar pe procesor sau doar pe FPGA cum ar fi: performanta superioara, flexibilitate, eficienta energetica si cost redus. Avantajele unui astfel de sistem de calcul cu o arhitectura SoC integrata care oferă o combinaţie de putere de procesare si logica programabila permite crearea de aplicaţii inovative in domenii precum procesarea semnalelor/imaginilor, automatizări industriale si aplicaţii de inteligenta artificiala.

2. Crearea unei Aplicaţii care Manipulează Elemente de Intrare/Ieşire din PS

2.1 Proiectarea Hardware a Sistemului

Pentru realizarea acestei aplicaţii vom avea nevoie de o placa cu cipul Zync si mediile de dezvoltare Vivado 2024.1 si Vitis 2024.1. Creaţi un nou proiect si numiţi-l GPIO_PS_Zynq. Proiectul trebuie sa fie de tipul RTL Project precum se poate observa in Figura 3.2

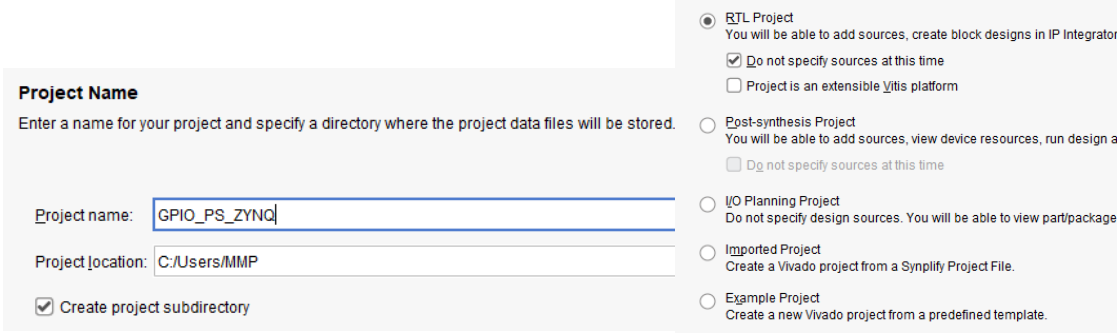

Figura 3.2. Crearea unui nou proiect de tip RTL Project

Apăsaţi butonul **Next** si in următoarea fereastra selectaţi tabul **Boards** si selectaţi placa pentru care faceţi proiectul, in mod similar cum aţi făcut si in laboratorul 1 si cum a fost ilustrat in Figura 1.11. Apăsaţi apoi butoanele **Next** si apoi **Finish**. După ce aţi creat proiectul, in noul proiect gol, daţi click pe butonul **Create Block Design** din Flow navigator si in noua fereastra care apare puteţi lasă numele implicit, design_1, aşa cum se vede in Figura 3.3

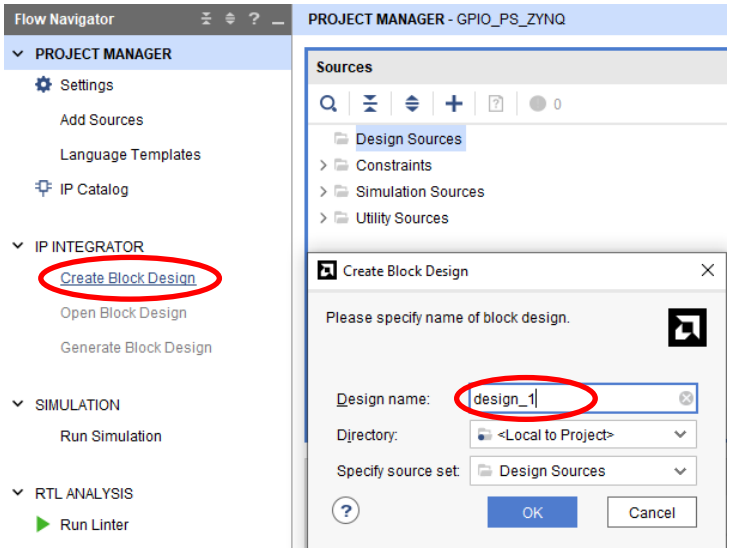

Figura 3.3 Daţi click pe opţiunea **Create Block Design** si in noua fereastra apăsaţi **OK**

Apăsaţi pe butonul „+”, care arata ca in Figura 3.4, in noua fereastra afişata pentru a adăuga componentele (IP-urile) de care avem nevoie. Cuvântul „IP” vin din engleza si se refera la „Intelectual Property”.

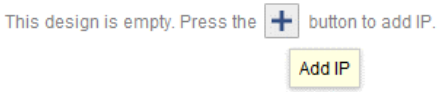

Figura 3.4 Adăugarea modulelor in Block designul gol creat

In noua fereastra care apare după apăsarea butonului „+”, in bara de căutarea (search) scrieţi numele procesorului si anume „Zynq”, in momentul in care va apare aceasta componenta, ca in Figura 3.5, daţi dublu click pe ea pentru a-l adăuga la designul vostru.

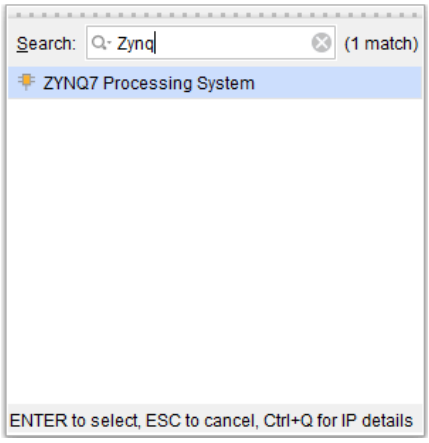

Figura 3.5 Adăugarea unei componente Zynq la Block Design

După adăugarea acestei componente, daţi click pe opţiunea **Run Block Automation**, pentru a va asigura ca Zynq PS este configurat corect si ca toate conexiunile necesare sunt realizate automat, astfel încât sa puteţi continua dezvoltarea designului fără a fi nevoie sa configuraţi manual fiecare componenta si interfaţa. Aceasta opţiune va apărea **fie cu verde in partea de sus** a proiectării hardware sau va trebui sa **daţi click dreapta** sa o selectaţi.
In Figura 3.6 apar ambele variante.

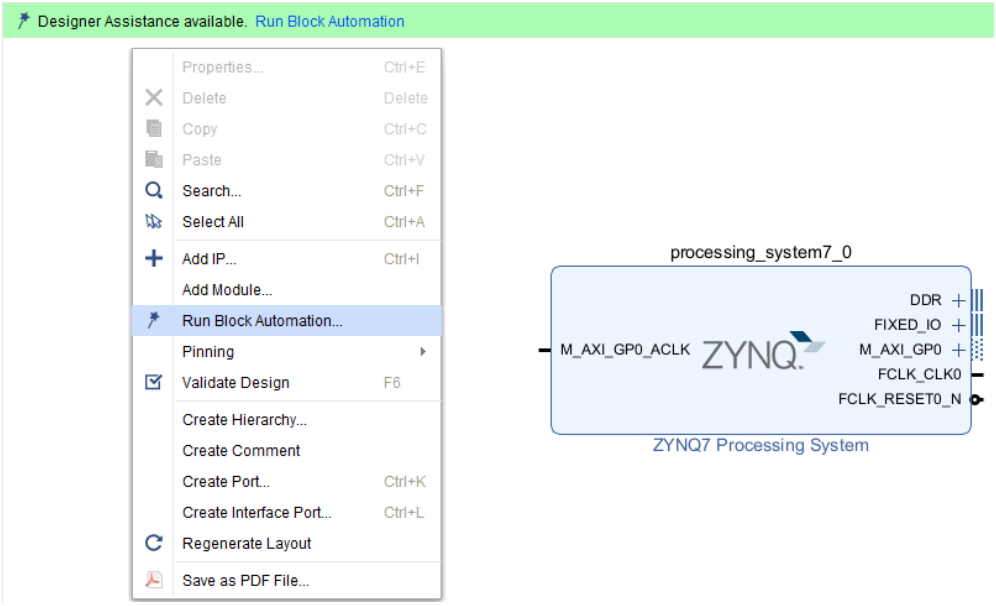

Figura 3.6. Opţiunea Run Block Automation va trebui selectata după adăugarea IP-ului

După apăsarea Run Block Automation va apărea o noua fereastra unde vor trebui setate opţiunile similar ca in Figura 3.7, iar apoi apăsaţi butonul OK. După acest pas veţi observa ca designul nostru s-a schimbat si acum sunt puse interfeţe externe precum DDR si FIXED_IO.

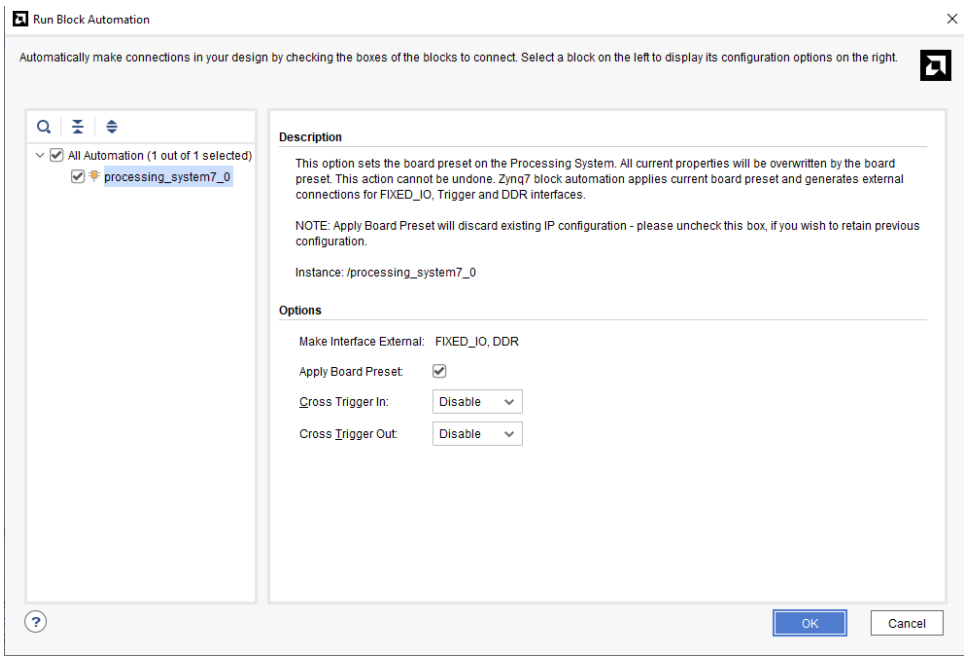

Figura 3.7 Configuraţia care trebuia făcută după apăsarea pe Run Block Automation

In continuare vom adaug un modul GPIO apăsând pe simbolul „+” din bara de opţiuni a diagramei prezentata in Figura 3.8.

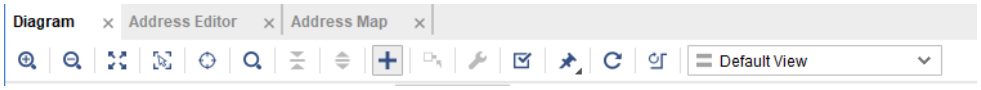

Figura 3.8. Adăugați un nou IP apăsând pe simbolul in forma de “+”

In noua fereastra scrieți in bara de căutare GPIO (prescurtarea de la „general purpose input output”), si selectați componenta AXI GPIO si dați dublu click pe ea pentru a o importa in designul vostru. După ce ați importat componenta trebuie sa o configuram. In Vivado, AXI GPIO are opțiunea **Enable Dual Channel** care permite activarea a doua canale ce pot fi configurate fiecare in parte fie ca intrări sau ieșiri, fiecare configurație fiind independenta de cealaltă. In cazuri simple in care nu este nevoie de multiple seturi de GPIO, Dual Channel va fi dezactivat. In cazul in care se activează canalul dublu vom avea 2 seturi de pini care vor comunica pe aceeași interfața AXI, dar fiecare canal va avea proprii lui registrii de control si de date. În funcție de configurația fiecărui canal, AXI GPIO va expune două seturi de adrese pentru regiștrii de date corespunzători fiecărui canal. In aplicația noastră dorim sa utilizam switchurile, butoanele si ledurile de pe placa. Așadar vom folosi un modul AXI GPIO dual Channel, care va avea un canal pentru intrerupatoare si unul pentru butoane, iar alt modul AXI GPIO cu un singur canal care va fi destinat operațiilor de afișare având ieșirea pe leduri. Dați dublu click pe componenta axi_gpio_0 nou creata pentru a configura modulul si navigați la tabul IP Configuration. Setați modulul ca si in Figura 3.9.

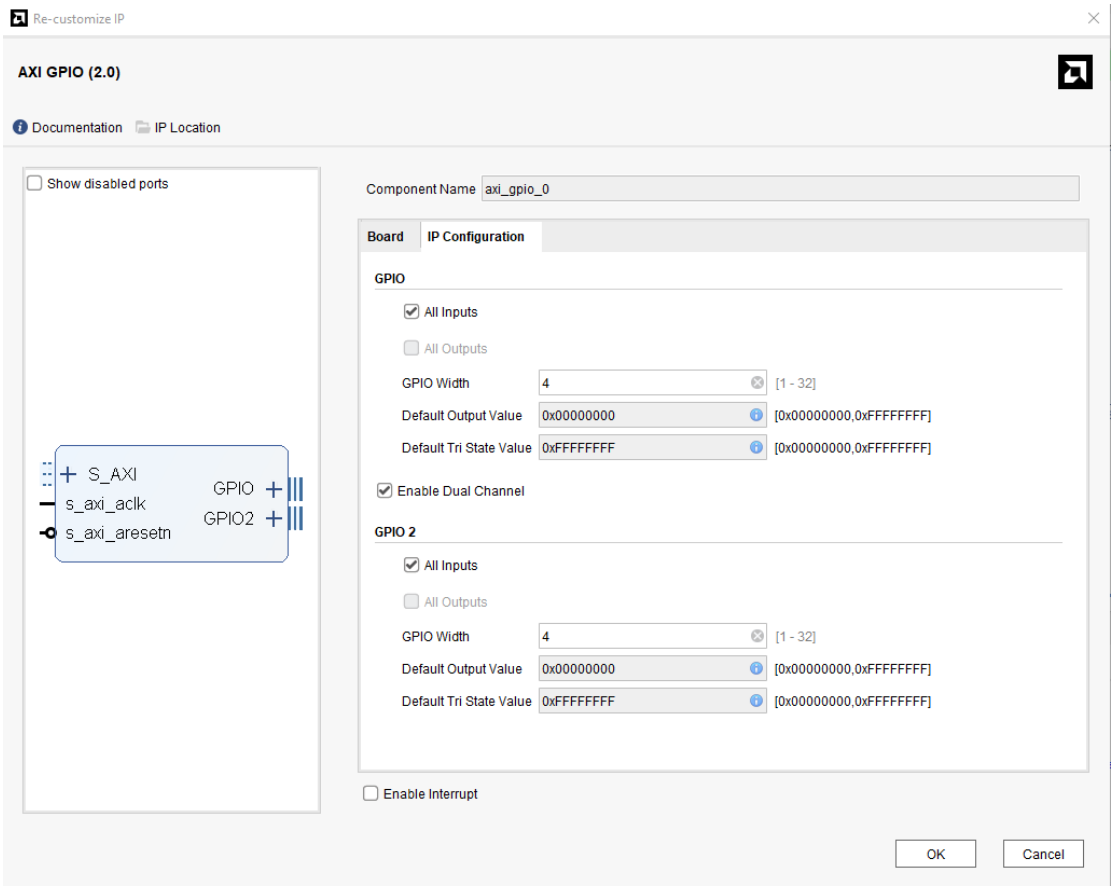

Figura 3.9. Setările AXI GPIO pentru modulul responsabil de întrerupătoare si butoane

In aceasta configurație fiecare canal a fost marcat ca si cana de input, numărul de biți al fiecărui semnal GPIO si GPIO2 poarta numele de GPIO width si este 4 in fiecare caz întrucât pe plăcile Zybo avem 4 butoane si 4 întrerupătoare. De asemenea, după ce ați făcut setările din IP Configuration, navigați in tabul Board si din meniul de tip drop down de la fiecare interfața selectați la ce va fi

utilizata acea interfaţa. In cazul nostru GPIO va fi utilizat pentru a lucra cu butoanele si GPIO 2 cu întrerupătoarele (switch). In figura 3.10 sunt afişate configuraţiile pentru cele doua canale.

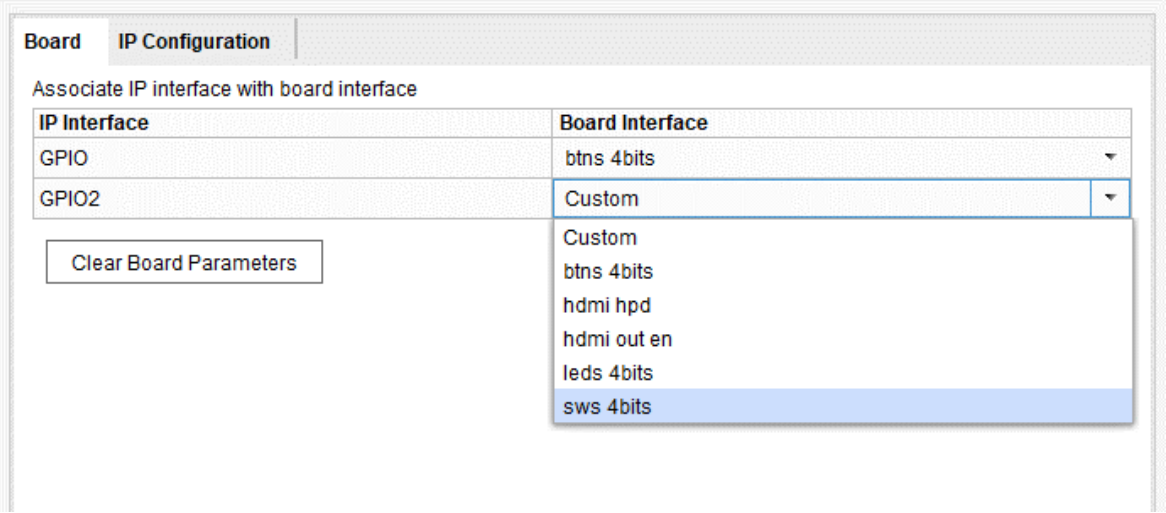

Figura 3.10 Configurarea canalelor GPIO si GPIO2

Adăugaţi un al doilea modul AXI GPIO repetând procedura explicata anterior. Acest al doilea modul va fi folosit pentru afişarea pe leduri. Daţi dublu click pe acest modul si configuraţi-l pentru a fi folosit cu cele 4 leduri de pe placa Zybo. In figura 3.11 aveţi exemplul de configurare pentru al doilea modul AXI GPIO.

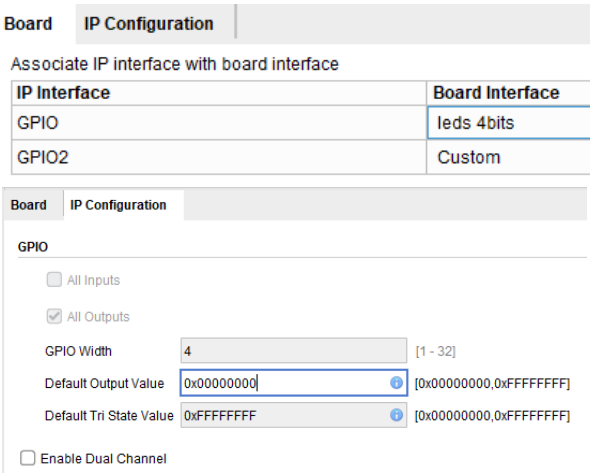

Figura 3.11. Configurarea celui de-al doilea modul AXI GPIO

După realizarea setărilor pentru cele doua module GPIO, apăsaţi din nou pe „**Run Connection Automation**”. Va apărea o fereastra noua in care asiguraţi-va ca aţi bifat toate componentele introduse anterior ca si in Figura 3.12 si apoi apăsaţi butonul **OK**.

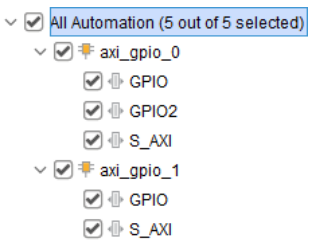

Figura 3.12 Bifați toate elementele când apare fereastra de configurare

Daţi click dreapta pe fereastra cu Block designul si selectaţi opţiunea „**Validate Design**” pentru a va asigura ca nu aveţi erori. Alternativ puteţi apăsa pe butonul F6. In Figura 3.13 este prezentat designul final al diagramei bloc create.

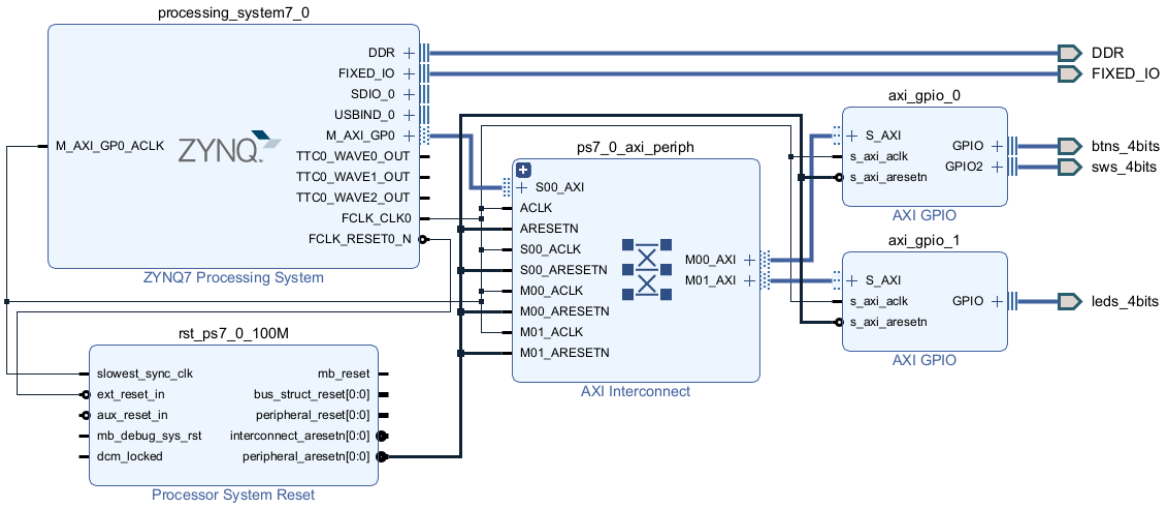

Figura 3.13 Schema bloc finala pentru designul nostru

Este important de menţionat faptul ca atunci când se utilizează switch-urile, butoanele şi LED-urile în cadrul PS (Processing System) într-un sistem Zynq, acestea sunt deja conectate la perifericele GPIO prin interfaţa de AXI, conform designului hardware prezentat. În acest caz, nu mai este necesar să se specifice constrângeri XDC suplimentare pentru aceste periferice, deoarece Vivado ştie deja să le conecteze automat la resursele PS corespunzătoare (cum ar fi GPIO din cadrul blocului AXI). Cu toate acestea daca s-ar dori sa se adauge periferice suplimentare care sa fie conectate la pinii fizici ai FPGA-ului sau sa existe o interacţiune cu alte componente care nu sunt incluse in PS, atunci acestea vor trebui definite intr-un fişier de constrângeri XDC pentru a corela codul VHDL cu pinii respectivelor componente. Odată validat designul, navigaţi la tabul Sources si daţi **click dreapta** pe designul creat si selectaţi opţiunea **„Create HDL Wrapper „**. Acest proces este vizibil in Figura 3.14.

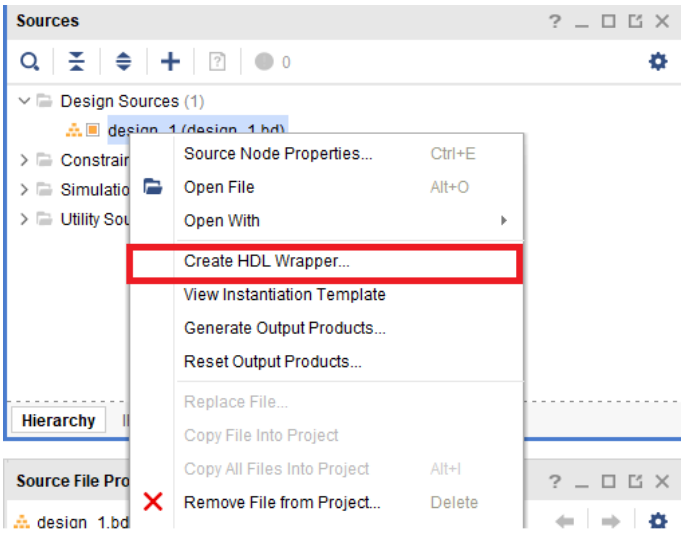

Figura 3.14 Crearea invelitorii HDL pentru proiectarea creat.

Rolul unui HDL wrapper (invelitori) este de a transforma designul la un nivel abstract (de exemplu, blocuri IP, procesor Zynq) într-o descriere sintetizabilă care poate fi utilizată în procesul

de sinteză şi implementare pe FPGA. După apăsarea pe acea opţiune, selectaţi butonul radio in care este scris mesajul „**Let Vivado manage wrapper and auto-update**” si apoi apasati **Ok**.

După ce efectuaţi aceasta operaţie, apăsaţi pe butonul **Generate Bitstream**, apoi in fereastra care apare apăsaţi butonul **Yes** si fereastra următoare apăsaţi **Ok**, pentru a genera fişierul *.bit. După finalizarea generării bitstreamului, va trebui sa exportam hardware-ul creat. Navigaţi la meniul principal si apăsaţi pe **File** si selectaţi opţiunea **Export Hardware** cum se vede si in Figura 3.15

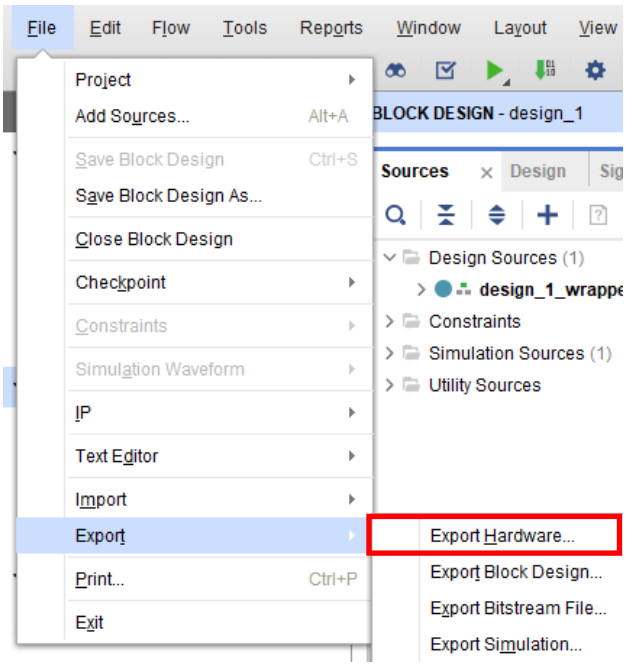

Figura 3.15. Exportarea hardware-ului creat

In fereastra care va apărea apăsaţi butonul **Next** si apoi selectaţi butonul radio unde apare mesajul „**Include bitstream**” si apăsaţi **Next de 2 ori,** si apoi **Finish**. In cazul in care aţi urmat paşii menţionaţi mai sus in consola TCL va apărea mesajul ca hardware-ul a fost salvat cu succes in locaţia indicata de dumneavoastră intr-un fişier cu extensia ***.xsa**. Reţineţi unde este salvat acest fişier întrucât va fi nevoie de el intr-un pas ulterior. De asemenea, este recomandat sa nu închideţi mediul Vivado fiindcă in cazul in care aţi făcut o eroare la generarea fişierului *.xsa sau daca va mai trebui adăugat ceva sa aveţi totul la îndemâna.

2.2 Crearea Platformei Suport in Vitis

Acum deschideţi mediul de dezvoltare Vitis IDE. Acest lucru il puteţi realiza din Vivado din meniul principal selectând opţiunile **Tools->Launch Vitis IDE** sau dând dublu click pe icoana corespunzătoare Vitis 2024.1 IDE-ului.

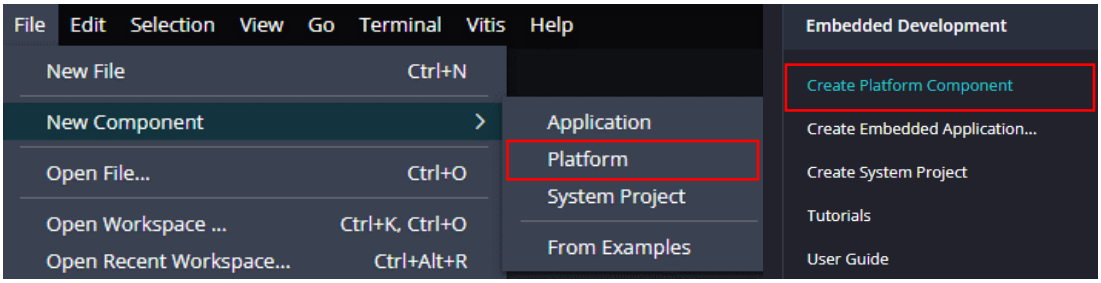

Figura 3.16. Cele 2 modalităţi de a crea un proiect de tip platforma

In Vitis va trebui prima data sa cream un proiect de tip platforma. Pentru a realiza acest lucru navigaţi la meniul **File->New Component->Platform** sau din fereastra de **Welcome** din secţiunea **Embedded Development** selectaţi **Create Platform Component**.
In figura 3.16 sunt ilustrate cele 2 modalităţi de a crea un nou proiect de tip platforma. În Vitis, este necesar să cream o platformă înainte de a realiza un proiect propriu-zis, deoarece platforma serveşte ca fundaţie hardware şi software pe care se va construi aplicaţia dumneavoastră. Aceasta va oferă un mediu predefinit care gestionează comunicarea între partea de procesare (PS - Processing System) şi logica programabilă (PL - Programmable Logic), precum şi accesul la resursele hardware şi software disponibile. In fereastra nou apăruta daţi un nume platformei, si selectaţi spaţiul de lucru iar apoi apăsaţi butonul **Next**. In fereastra nou apăruta, selectaţi (daca nu este deja selectat) butonul radio „Hardware Design" si la bara de căutarea cu mesajul „Hardware Design (XSA) For implementation" navigaţi la locaţia unde aţi salvat fişierul *.xsa din Vivado si selectai-l (daca nu mai ţineţi minte unde este salvat puteţi sa consultaţi mesajul din consola TCL din Vivado). Opţiunile descrise anterior se pot observa in Figura 3.17. După ce aţi selectat platforma *.xsa apăsaţi butonul **Next**.

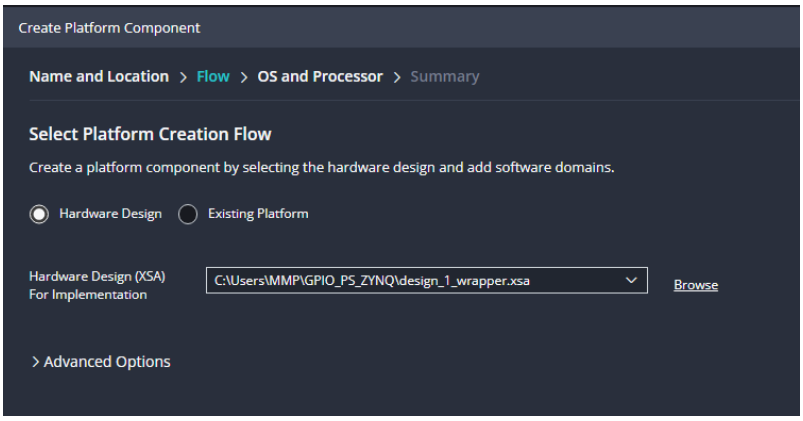

Figura 3.17. Selectarea platformei *xsa

In fereastra care a apărut, după generarea Device Tree-ului din fişierul *.xsa la opţiunea Operating System selectaţi „standalone" întrucât dorim sa facem o aplicaţie de tipul bare metal. Fişierul **.xsa** (Xilinx Support Archive) este fişierul generat de Vivado în cadrul procesului de proiectare hardware pentru platformele FPGA şi SoC. Acest fişier conţine o descriere completă a configuraţiei hardware, care este utilizată pentru a integra designul hardware cu aplicaţiile software dezvoltate in Vitis. Include toate modulele şi perifericele definite în proiectul hardware, configurarea conexiunilor dintre ele şi modul în care acestea sunt mapate în sistemul final spre exemplu blocuri de procesoare MicroBlaze sau ARM Cortex A9 în cazul Zynq, Periferice AXI (cum ar fi GPIO, UART, I2C, SPI), Componente PL etc.

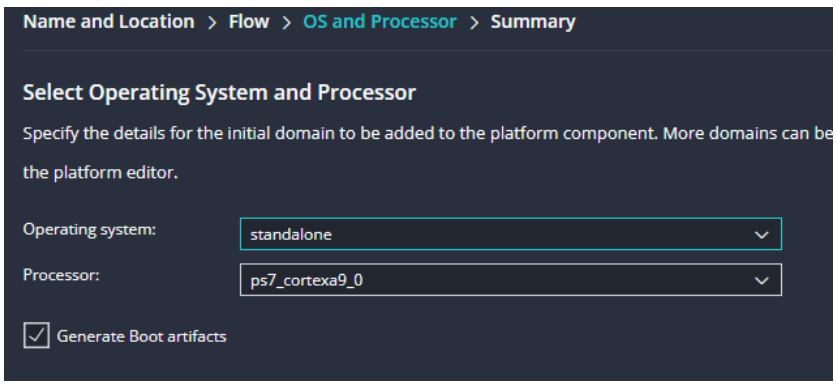

Figura 3.18 Selectarea opţiunilor pentru sistemul de operare si procesorul platformei

La procesor selectaţi unul din procesoarele cortex a9 disponibile, de regula se selectează primul. In Figura 3.18 sunt ilustrate aceste selecţii. Apăsaţi apoi pe butonul **Next** si apoi **Finish** pentru a crea platforma. Aşteptaţi apoi ca platforma sa fie creata. Este important de reţinut ca de regula crearea unei platforme in Vitis este un proces care consuma timp. In practica platforma se creează o singura data si apoi se lucrează ea si doar daca exista modificări semnificative se recreează o alta platforma. Odată creata platforma va apărea in partea stânga la secţiunea Vitis Components. Pentru a vedea ce conţine platforma puteţi naviga prin fişierele platformei create. Pentru a folosi platforma va trebui sa ii dam build, aşadar daţi click pe platforma creata si **din meniul Flow** selectaţi opţiunea **Build** pentru a face build la platforma. Daca buildul a fost realizat cu succes va apărea un mesaj in consola care sa ne spună acest lucru altfel vor apărea mesaje de eroare. In Figura 3.19 se poate observa ca Build-ul platformei noastre a fost realizat cu succes.

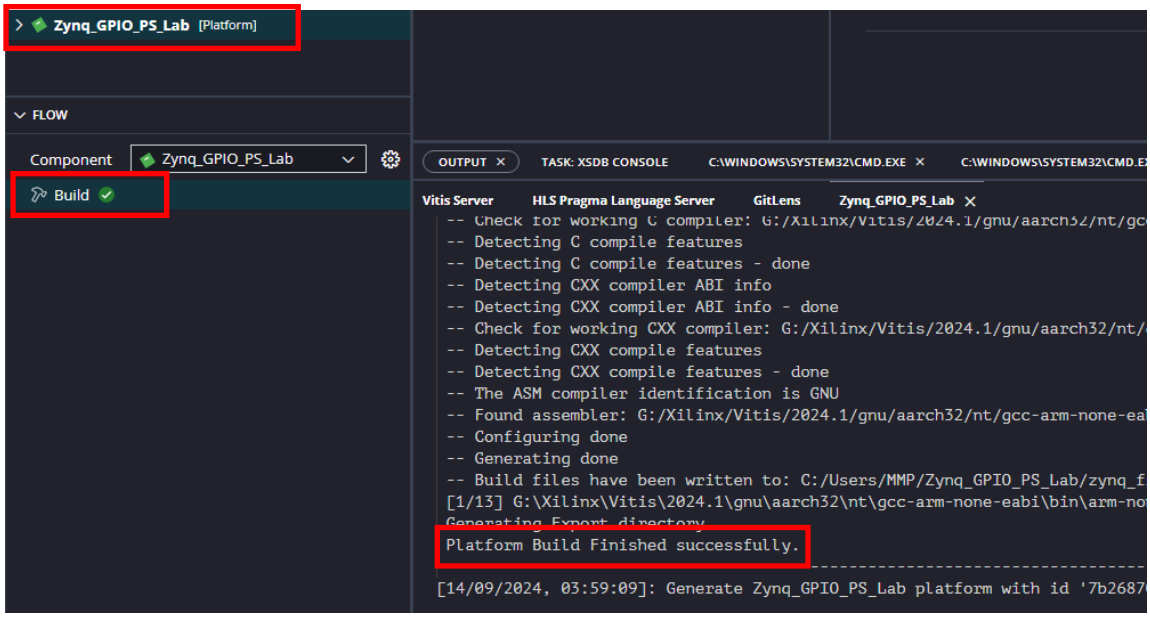

Figura 3.19. Operaţia de Build a platformei nou create este efectuata cu succes

2.3 Crearea Aplicaţiei Embedded

Acum ca avem creata platforma putem crea aplicaţia pe baza platformei. Aplicaţia pe care o vom crea va fi bazata pe un exemplu de aplicaţie de tip „Hello World" pe care îl vom modifica. Aşadar in Vitis IDE, navigaţi la meniul **File->New Component-> From Examples** sau din fereastra de Welcome, din secţiunea Embedded Development, dati click pe **Create Embedded Application -> Create Embedded Application From Example.** Cele 2 variante sunt ilustrate in Figura 3.20.

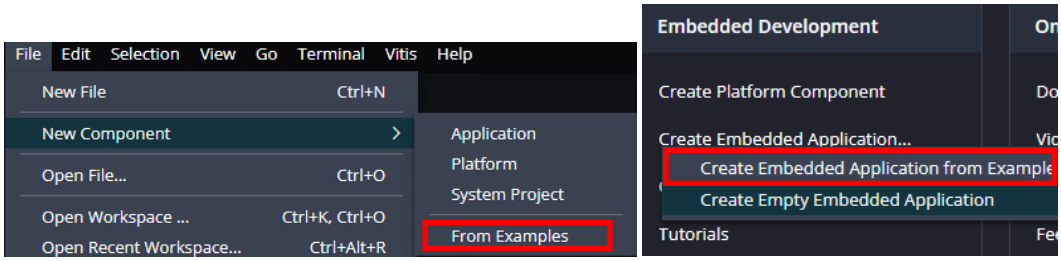

Figura 3.20. Doua variante de crearea aplicatiei embedded

Din panelul cu exemple care vor apărea, identificaţi aplicaţia **Hello World** si daţi click pe aceasta. In noua fereastra daţi click pe butonul **Create Application Component from Template.** Avantajul de a crea aplicaţia in acest fel este acela ca Vitis ne va include automat anumite librarii standard fără a fi nevoie sa le mai căutam sa le introducem noi.

Acest pas de crearea a aplicației este ilustrat in Figura 3.21.

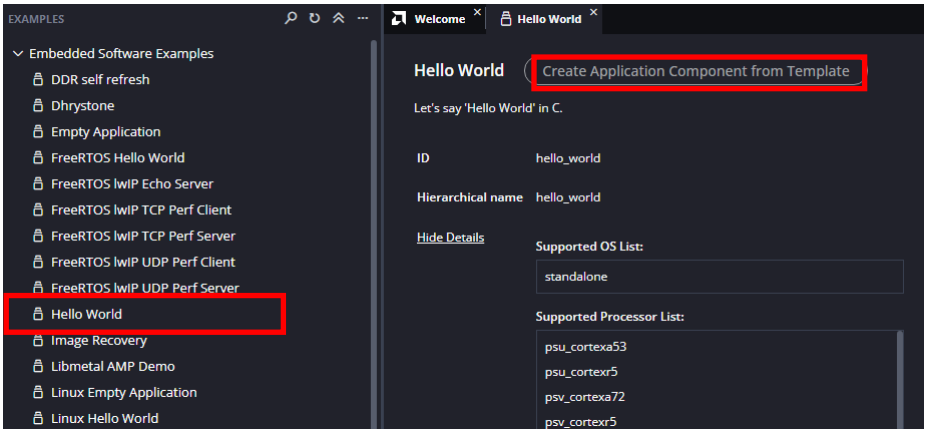

Figura 3.21 Crearea unei aplicații dintr-un template “Hello World”

In fereastra noua care va apărea setați numele nou al aplicației (in loc de hello world) si apăsați butonul **Next**. In Figura 3.22 este prezentat acest pas.

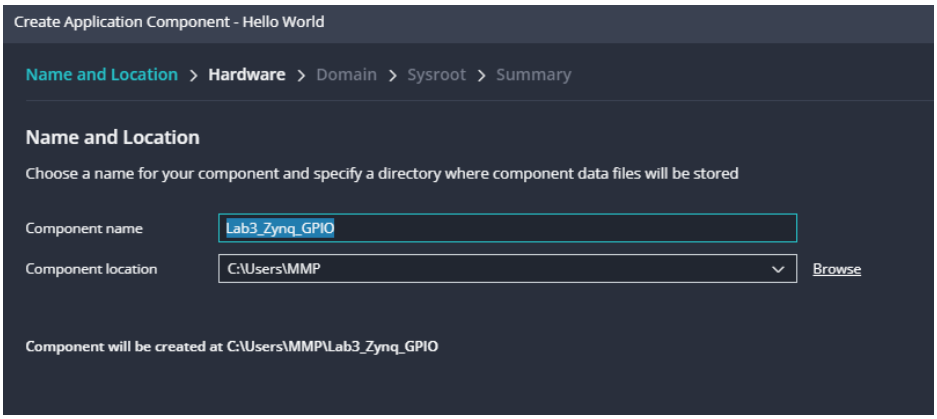

Figura 3.22 Setarea numelui aplicației si a spațiului de lucru unde se afla si platforma create anterior

In următorul pas, la secțiunea **Hardware**, selectați, din lista de platforme existente, platforma pentru care se face aplicația actuala, adică platforma creata de noi in **secțiunea 2.2**. Acest pas este ilustrat in Figura 3.23. Atenție! Daca nu ați realizați operația de build la platforma dumneavoastră, acesta nu va apărea ca opțiune.

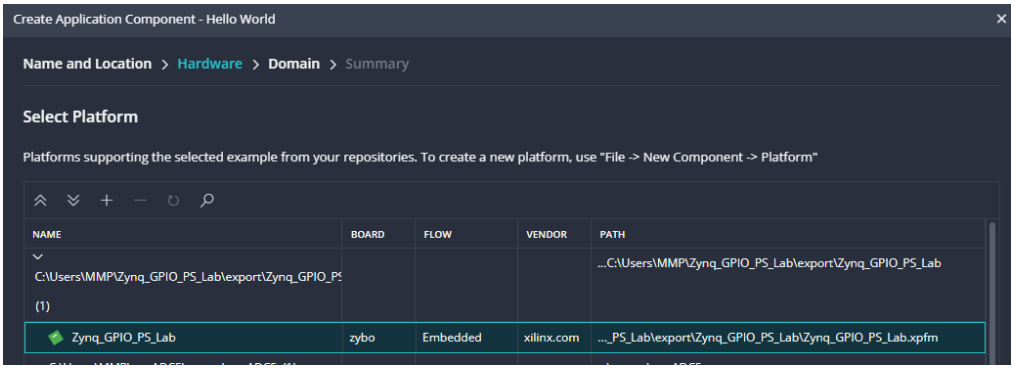

Figura 3.23 Selectarea platformei create anterior pe care vom construi aplicația embeded

După selectarea platformei apăsați butonul **Next**. Următoarea secțiune in configurarea aplicației este data de „Domain”. In Vitis, „Domain” se refera la mediul de execuție in care va rula aplicația dumneavoastră embedded. Acesta poate fi asociat cu tipul de procesor sau cu resursele hardware

alocate pentru executarea codului. Intr-o platforma embedded precum Zynq, care combina un sistem de procesare (PS) si logica programabila (PL) pot exista mai multe domenii potențiale:

1. Bare Metal – Aplicația dezvoltata va rula direct pe hardware, fără a avea un sistem de operare. Aceasta opțiune este frecvent utilizata in cazul in care avem nevoie de aplicații de timp real unde accesul direct la resursele hardware este esențial.
2. FreeRTOS – In acest scenariu este pus la dispoziție un sistem de operare simplu de timp real(RTOS), pe care va rula aplicația, care permite multitasking sau alte funcționalități utile in aplicațiile embedded cum ar fi gestionarea unor procese multiple.
3. Linux – Pe platforma Zynq sau alte sisteme embedded avansate se poate include un sistem de operare linux avansat care are si interfața grafica (ex: Ubuntu) pe care sa ruleze aplicațiile.

In cazul nostru, dorim sa facem o aplicație **Bare Metal** in care aplicația sa ruleze direct pe hardware. Așadar, la secțiunea Domain, asigurați-va ca aveți configurația prezentata in Figura 3.24.

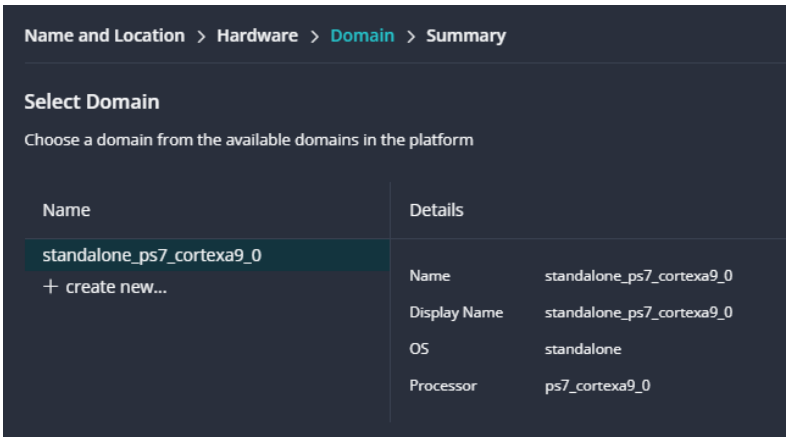

Figura 3.24. Setarea domeniului pentru aplicația noastră

Apăsați apoi butonul **Next** si **Finish** pentru a va genera aplicația. După câteva momente aplicația va fi generata. In directorul **src** vom avea elementele prezente in Figura 3.25, iar in fișierul helloworld.c va fi codul standard pentru aplicația noastră Hello World. Acest cod, când rulează pe plăcuța, va printa la consola, prin intermediul protocolului UART mesajele „Hello World" si „Succesfully ran Hello World application".

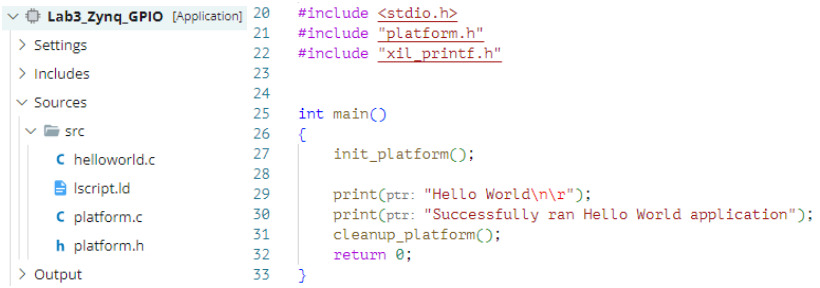

Figura 3.25. Elementele Noii aplicații create

Acum ca avem aplicația creata putem sa o modificam. Putem sa redenumim fișierul principal helloworld.c in Exemplu_GPIO.c pentru a avea un nume mai reprezentativ. Acest lucru se realizează dând click dreapta pe fișierul helloworld.c si selectând opțiunea **Rename**. Introduceți apoi noul nume. Pe lângă fișierul principal pe care l-am redenumit Exemplu_GPIO.c mai întâlnim in directorul src si fișierele lsscript.ld, platform.c, și platform.h. Aceste fișiere sunt parte din structura unei aplicații în Vitis și joacă roluri importante în configurarea și funcționarea proiectului. Fișierul **lsscript.ld** are ca scop principal definirea unor secțiuni din memorie, cum ar

fi locaţia unde se va încărca si se va executa codul, unde vor fi plasate datele statice, cum vor fi iniţializate variabilele globale etc. Acest script de linkare mai este important deoarece controlează unde si cum se încărca secţiunile comune ale programului(.text,.data, .bss, stive, heap etc) in memoria flash sau RAM pentru a utiliza cat mai eficient resursele hardware. Fişierele platform.c si platform.h au rolul principal de a iniţializa resursele hardware esenţiale înainte ca aplicaţia sa înceapă sa ruleze (spre exemplu iniţializarea UART-ului, configurarea clockuri-lor setarea si iniţializarea perifericelor utilizate de aplicaţie etc.). Codul din aceste fişiere este adaptat platformei (procesorului şi perifericelor), asigurându-se că hardware-ul este pregătit pentru execuţia programului principal.

In continuare vom implementa programul nostru in c. Funcţionalitatea programului va fi următoarea: de fiecare data când se va apăsa pe unul din cele 4 butoane se va aprinde un led diferit (butonul 1 aprinde ledul 1, butonul 2 aprinde ledul 2 s.a.), iar de fiecare data când vom acţiona un switch ne va printa la consola switchul care are valorarea „1” logic, iar daca sunt mai multe switchuri care sunt pe „1” logic se va transmite un mesaj care sa informeze in legătura cu acest lucru. Codul corespunzător funcţionalităţii de mai sus este prezentat in caseta de mai jos.

```
#include <stdio.h> //Include biblioteca standard pentru input/output
//Include definiții și funcții pentru inițializarea și curățarea
//platformei
#include "platform.h"
//Include funcții pentru gestionarea GPIO-urilor
#include <xgpio.h>
//Include adrese de bază și alte setări pentru periferice
#include "xparameters.h"
//Include funcția usleep pentru a întârzia execuția
#include "sleep.h"

int main()
{
//Declară două structuri XGpio pentru pinii de intrare și ieșire
 XGpio input, output;
 int button_data = 0;//Variabilă pentru stocarea stării butoanelor
 int switch_data = 0; //Variabilă pentru stocarea stării switch-urilor

// Inițializează pinii de intrare (GPIO0) cu adresa de bază
//specificată în xparameters.h
   XGpio_Initialize(&input, XPAR_AXI_GPIO_0_BASEADDR);
// Inițializează pinii de ieșire (GPIO1) cu adresa de bază specificată
//în xparameters.h
   XGpio_Initialize(&output, XPAR_AXI_GPIO_1_BASEADDR);

// Setează canalul 1 al primului GPIO ca intrare (tristate buffer
//pentru citire)
   XGpio_SetDataDirection(&input, 1, 0xF);
// Setează canalul 2 al primului GPIO ca intrare (tristate buffer
//pentru citire)
   XGpio_SetDataDirection(&input, 2, 0xF);
```

```
// Setează unicul canal al GPIO-ului de ieșire (tristate buffer pentru
//scriere)
XGpio_SetDataDirection(&output, 1, 0x0);
init_platform();//Inițializează platforma hardware
//Afișează un mesaj la începutul execuției
print("Incepem procesarea \r\n");
// Buclează infinit pentru a citi și scrie date in continuu
while(1){
// Citește datele de la switch-uri de pe canalul 2 al GPIO-ului de
//intrare
      switch_data = XGpio_DiscreteRead(&input, 2);
// Citește datele de la butoane de pe canalul 1 al GPIO-ului de
//intrare
      button_data = XGpio_DiscreteRead(&input, 1);
//Scrie datele de la butoane pe canalul 1 al GPIO-ului de ieșire
//(LED-uri)
      XGpio_DiscreteWrite(&output, 1, button_data);
//Mai jos sunt condiții pentru a afișa mesajul corespunzător în
//funcție de switch-ul acționat
// Dacă niciun switch nu este apăsat, nu face nimic
      if(switch_data == 0b00000){}
 //Dacă switch-ul 0 este apăsat afizeaza mesajul corespunzator
      else if(switch_data == 0b00001)
          print("switchul 0 a fost actionat \r\n");
 //Dacă switch-ul 1 este apăsat afiseaza mesajul
      else if(switch_data == 0b00010)
          print("switchul 1 a fost actionat \r\n");
 //Dacă switch-ul 2 este apăsat afiseaza mesajul si asa mai departe
      else if(switch_data == 0b00100)
          print("switchul 2 a fost actionat \r\n");
      else if(switch_data == 0b01000)
          print("switchul 3 a fost actionat \r\n");
 //Daca sunt mai multe switchuri apasate mentioneaza acest lucru
      else
          print("mai multe switchuri au fost actionate \r\n");
 //Întârzie execuția buclei pentru 200 de milisecunde
      usleep(200000); //delay
   }
//Curăță resursele platformei hardware înainte de ieșirea din program
    cleanup_platform();
//Returnează 0 pentru a indica execuția finalizata cu succes
    return 0;
}
```

După ce aţi implementat codul de mai sus, va trebui sa ii facem build, apăsând pe butonul **Build** din secţiunea **Flow**. Daca operaţia a fost finalizata cu succes, va apărea o bifa verde lângă butonul de Build ca si in Figura 3.26.

De asemenea, informaţia ca acest cod a fost compilat si procesul de build a fost realizat fără erori poate fi văzut si in consola asociata aplicaţiei (si aceasta informaţie poate fi văzută in Figura 3.26).

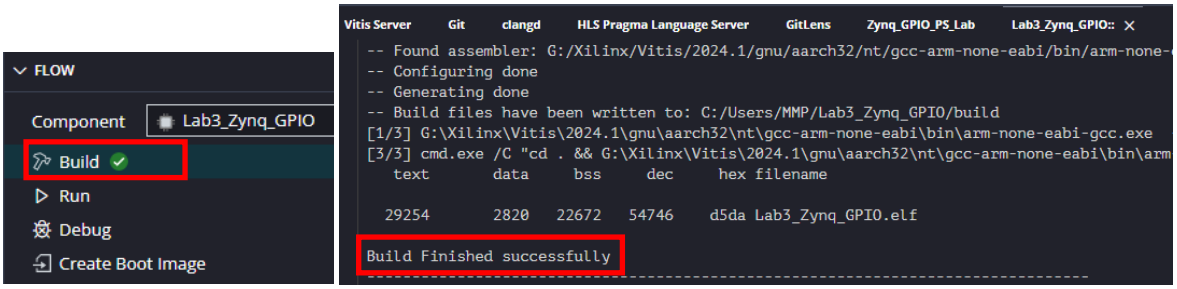

Figura 3.26. Procesul de build al aplicatiei de mai sus a fost executat cu succes

Pentru a programa cipul exista doua variante: fie apăsând **Run** sau din meniul principal de la opţiunea **Vitis->Program Device**. Diferenţa cheie consta in faptul ca atunci când executam **Run** doar codul aplicaţiei software este descărcat şi rulează temporar în RAM-ul procesorului; resetarea dispozitivului îl va şterge. Când executam Program Device se configurează hardware-ul FPGA şi (opţional) se programează permanent bootloader-ul sau logica hardware, astfel încât să fie păstrat după resetare. De regula, pentru debugging sau rulări rapide se foloseşte opţiunea Run.

Conectaţi plăcută la calculator, si identificaţi portul COM pe care este conectat aceasta (cum a fost prezentat in laboratorul 1). După conectare apăsaţi butonul **Run** (vizibil si in figura 3.26). Deschideţi aplicaţia de tip terminal preferata, pentru a putea vedea outputul programului. In cazul nostru se va deschide aplicaţia **HTerm**, se va selecta portul corespunzător plăcutei si se va configura plăcuţa cu setările care sunt vizibile si in Figura 3.27.

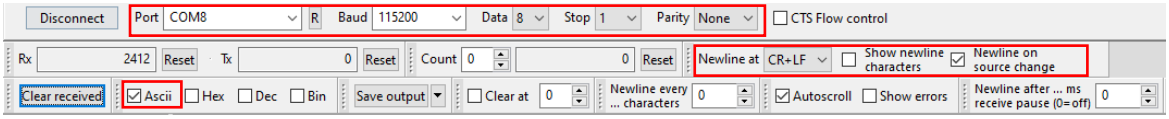

Figura 3.27. Setările pentru comunicarea seriala. Atenţie la elementele încadrate in dreptunghiuri roşii.

In Figura 3.28 se observa starea întrerupătoarelor si rezultatul pe leduri al apăsării unui buton, si imaginea plăcutei împreuna cu outputul in terminal.

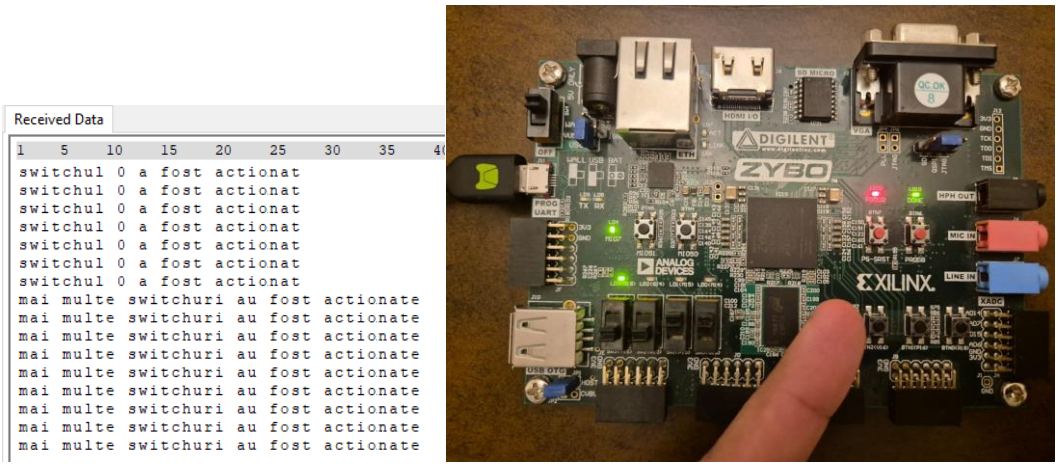

Figura 3.28. Mai multe întrerupătoare au valoarea “1” logic si butonul 4 e apăsat ce face ca ledul 4 sa fie activ.

Pentru a opri rularea aplicaţiei din meniul din stânga selectaţi opţiunea debut care are o imagine cu un triunghi si o insecta, iar apoi in fereastra nou apăruta apăsaţi pe pătratul roşu.

In Figura 3.29 este prezentata imaginea corespunzătoare opririi rulării programului.

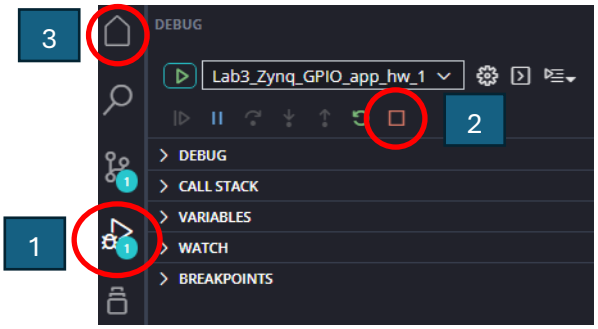

Figura 3.29. Dați click pe icoana de debug indicata de pătratul numărul 1, apăsați butonul de stop indicat de pătratul numărul 2. După ce ați oprit aplicația pentru a reveni la spațiul de apăsați poligonul indicat de pătratul cu numărul 3

2.4 Alte considerații legate de lucrul cu GPIO pe Zybo

Precum s-a putut observa si din aplicația prezentata in secțiunea anterioara, lucrul cu elemente de input si output au anumiți pași care trebuie respectați. In aceasta secțiune vom analiza pașii necesari lucrului cu intrările si ieșirile de pe Zybo.

In primul rând pentru a lucra cu perifericele plăcutei si cu funcțiile asociate acestor periferice trebuie inclusa librăria **xgpio.h** . Aceasta librărie este esențiala pentru a interacționa cu pinii de pe platforma, oferind o interfața simpla pentru configurarea pinilor si abstractizând detaliile de nivel hardware.

În proiecte embedded, trebuie să se știe adresele de memorie unde sunt mapate perifericele, pentru a putea interacționa cu ele. Librăria **xparameters.h** definește atât adresele pentru perifericele folosite precum GPIO, UART, etc. cat si o lista de constante care definesc configurarea hardware specifica sistemului nostru, generate automat din Vivado in timpul procesului de proiectare hardware.

Declararea obiectelor de tip GPIO se face utilizând tipul **XGpio.** După declararea obiectelor de tip GPIO, acestea trebuie inițializate folosind funcția **XGpio_Initialize**. Funcția primește ca parametru o referința la variabila de tip XGPIO si o inițializează la una din adresele de baza create in timpul procesului de proiectare si definita in fișierul xparameters.h.

După acest pas urmează configurarea registrului de direcție pentru variabila noastră (adică la ce fel de operații va fi folosita acea variabila, scriere sau citire). Pentru a seta o variabila ca intrare direcția va trebui setata pe „1” pentru a seta o variabila ca ieșire direcția va trebui setata pe „0”. **Setarea biților de direcție este invers ca si la AVR-uri.** Funcția predefinita care setează direcția pinilor pe Zynq este **XGpio_SetDataDirection.** Aceasta funcție primește ca intrare o referința la variabila XGPIO la care dorim sa facem setarea biților de directive, numărul canalului la care se face acea setare (țineți minte ca putem avea 2 canale fiecare putând fi configurat independent), si apoi setarea valorilor de „0” sau „1” pentru fiecare pin din acel canal in funcție de cum vrem sa ii setam. Spre exemplu noi am folosit valoarea 0xF (sau 0b1111 in binar) deoarece avem 4 întrerupătoare pe canalul 1, pinii de la fiecare întrerupător fiind o pini de intrare.

După efectuarea tuturor setărilor platforma se inițializează folosind funcția **init_platform.** Aceasta funcție asigură că toate resursele de bază ale platformei (comunicarea serială, cache, periferice etc.) sunt inițializate corect înainte de a începe execuția codului principal al aplicației. Fără această inițializare, aplicația poate să nu aibă acces la periferice corect configurate.

Pentru a citi datele de la un canal se folosește funcția **XGPIO_DiscreteRead** care primește ca intrări o variabila de tip XGPIO si numărul canalului de pe care sa citească. Funcția întoarce o valoare care reprezintă starea de pe toți pinii din acel canal. Pentru a lua o valoare individuala a

unui pin va trebui fie sa aplicam o masca binara sau sa comparam cu valori binare incremental (așa cum am făcut in exemplul de cod pentru întrerupătoare). In bucata de cod de mai jos verificam daca butonul 1 a fost apăsat. **XGPIO_DiscreteRead** întoarce o valoare pe 32 de biți, care are pe pozițiile 0,1,2 sau 3 valoarea ‚1' in cazul in care un buton este apăsat. Așadar se observa ca din 32 de biți doar 4 sunt folosiți in acest caz, restul biților având valoarea 0. In cazul in care dorim sa vedem ca butonul 1 este apăsat, va trebui sa avem o masca de forma 0x00000002, care face ca pozițiile 0,2 si 3 sa aibă valoarea ‚0'. In cazul in care butonul e apăsat operația logica „si" button_data & mask va avea un rezultat diferit de 0.

```
int mask = 0x00000002;
int button_data = XGpio_DiscreteRead(&input, 1);

if((button_data & mask) == mask) //facem ceva
```

Pentru a scrie valori pe un canal de ieșire, se folosește funcția **XGpio_DiscreteWrite**. Aceasta funcție primește ca parametrii, numele variabilei de tip XGpio pe care dorim sa facem scrierea, canalul pe care se efectuează scrierea si o valoare binara, in cazul nostru de la 0000 la 1111, pentru fiecare pin din acel canal, ‚0' însemnând oprit si ‚1' însemnând activ sau pornit.

Mai multe informații legate de GPIO-uri puteți găsi in referința [2].

Proiectarea hardware din Vivado, platforma hardware din Vitis si aplicația embedded din Vitis se găsesc la linkul de git:
https://github.com/mirceamp/StructuraSistemelorDeCalcul/tree/main/Lab3

Exerciții

1. Implementați tutorialul prezentat in aceasta lucrare de laborator si încercați sa înțelegeți conceptele prezentate.
2. Modificați aplicația embedded din Vitis astfel încât daca se apăsa pe butonul 1 sa se aprindă ledurile 0 si 2 si restul ledurilor sa se stingă iar daca se apăsa pe butonul 2 se aprind ledurile 1 si 3.
3. Modificați aplicația embedded astfel încât daca întrerupătorul 1 are valoarea ‚1' logic sa se aprindă pe rând ledurile având o pauza de o secunda intre aprinderea fiecărui led. (Se aprinde ledul 0 apoi după o secunda se închide si se aprinde ledul 1 care va sta o secunda aprins si apoi se va închide si el urmând sa se aprindă ledul 2 s.a.)
4. Creați un meniu cu 6 opțiuni si afișați-l la consola. Folosiți butoanele pentru a naviga printre opțiunile din meniu punând in fata opțiunii curente semnul „>". Afișați numărul opțiunii curente(un număr de la 1 la 6) si pe leduri in binar.

Tema

Exercițiile din lucrarea de laborator care nu le veți încheia in timpul laboratorului sa vor finaliza acasă.

Bibliografie

[1] N. Aranzabal et al., 'Design of Digital Advanced Systems Based on Programmable System on Chip', Field - Programmable Gate Array. InTech, May 31, 2017. doi: 10.5772/66579.

[2] Crockett, L. H., Elliot, R. A., Enderwitz, M. A., & Stewart, R. W. (n.d.). *The Zynq book: Embedded processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 all programmable SoC* (1st ed.). Department of Electronic and Electrical Engineering, University of Strathclyde