Universitatea
Transilvania
din Brașov
FACULTATEA DE INGINERIE ELECTRICĂ
ȘI ȘTIINȚA CALCULATOARELOR

PROIECT DE DIPLOMĂ

Conducător științific:
Conf. Dr. Ing. CIOBANU Cătălin
Colaborare NXP Semiconductors Rmânia:
Dr. Ing. ANTOCHI Iosif

Absolvent:
MITU Mariana-Luciana

Brașov, 2025

Departamentul de Electronică și Calculatoare
Programul de studii: Calculatoare

MITU Mariana-Luciana

Accelerarea Hardware a conversiei spațiilor de culoare YUV-RGB pe procesoarele RISC-V

Conducător științific:
Conf. Dr. Ing. CIOBANU Cătălin
Colaborare NXP Semiconductors Rmânia:
Dr. Ing. ANTOCHI Iosif

Brașov, 2025

Cuprins

FIGURI

TABELE

CODURI SURSĂ

ADC - Analog-to-Digital Converter;
AI - Artificial Intelligence;
APB - Advanced Peripheral Bus;
ATSC - Advanced Television Systems Committee;
AXI - Advanced eXtensible Interface;
AVC - Advanced Video Coding;
CIE - Commission Internationale de l'Éclairage;
CMY - Cyan, Magenta, Yellow;
CMYK - Cyan, Magenta, Yellow, Black;
CSR - Control Status Registers;
DMA - Direct Memory Access;
DTO - Device Tree Overlay;
DSP - Digital Signal Processing;
DVB - Digital Video Broadcasting;
ENIG - Electroless Nickel Immersion Gold;
FIC - Fabric Interface Controller;
FPGA - Field Programmable Gate Array;
FOURCC - Four Character Code;
HD - High Definition;
HEVC - High-Efficiency Video Coding;
HPC - High-Performance Computing;
HSS - Hart Software Services;
HSI - Hue, Saturation, Intensity;
IP - Intellectual Property (core);
LED - Light Emitting Diode;
LE - Logical Elements;
LPDDR - Low-Power Double Data Rate (memory);
LSB - Least Significant Byte;
LSRAM - Local Static Random Access Memory;
LUT - Look Up Table;
MIPI-CSI - Mobile Industry Processor Interface – Camera Serial Interface;
MSB - Most Significant Byte;
NTSC - National Television System Committee;
PAL - Phase Alternating Line;
PCB - Printed Circuit Board;
PCIe - Peripheral Component Interconnect Express;
PLIC - Platform-Level Interrupt Controller;
RAM - Random Access Memory;
RGB - Red, Green, Blue;
RISC - Reduced Instruction Set Computer;
RoHS - Restriction of Hazardous Substances;
SBC - Single-Board Computer;
SGMII - Serial Gigabit Media Independent Interface;
SoC - System-on-Chip;
TCL - Tool Command Language;
UART - Universal Asynchronous Receiver Transmitter;

UHD - Ultra High Definition;
USB - Universal Serial Bus;
YUV - Luminance (Y), Chrominance Blue (U), Chrominance Red (V);

1 INTRODUCERE

1.1 PREZENTAREA TEMEI PROIECTULUI

Proiectul propune o aplicație practică de implementare a conversiei între spațiile de culoare YUV și RGB folosind resursele FPGA (Field Programmable Gate Array) ale plăcii BeagleV®-Fire, precum și avantajele oferite de arhitectura RISC-V. Abordarea folosită permite o paralelizare extinsă și implementarea de tehnici pipeline, esențiale pentru atingerea performanțelor necesare în aplicațiile de procesare a imaginilor. Conversia spațiilor de culoare reprezintă un proces fundamental în cadrul sistemelor de procesare a imaginilor, având o importanță majoră în asigurarea redării fluide și eficiente a fluxurilor video în timp real. Datorită cerințelor tot mai stringente impuse de aplicațiile moderne, cum ar fi transmisiunile de înaltă rezoluție și aplicațiile critice precum monitorizarea video sau mijloacele de transport autonome, eficiența conversiei între diverse spații de culoare a devenit o preocupare majoră.

Modelul RGB (Red, Green, Blue), bazat pe o reprezentare aditivă a culorilor, este optim pentru afișarea pe dispozitive electronice precum monitoare, televizoare și panouri LED, datorită corespondentului său direct cu percepția vizuală umană. Pe de altă parte, spațiul de culoare YUV, derivat din modelul YCbCr, este utilizat frecvent în transmisii și compresie video datorită separării componentelor astfel: luminanță (Y), crominanță (U, V). Această structură permite o compresie mai eficientă a datelor, reducând lățimea de bandă necesară fără a compromite semnificativ calitatea percepută a imaginii.

Conversia între cele două spații de culoare are la bază operații matematice aparent simple, dar care implică un număr semnificativ de înmulțiri cu coeficienți reali, ceea ce poate deveni problematic în implementările hardware din cauza complexității și a timpului de calcul. Pentru a aborda această provocare, o soluție eficientă presupune înlocuirea înmulțirilor cu numere reale cu operații de shiftare și adunare cu numere întregi, tehnici care permit reducerea complexității circuitului hardware. Această abordare se bazează pe aproximarea coeficienților reali prin fracții binare, ceea ce facilitează implementarea directă la nivel de circuit logic.

Un element cheie al proiectului îl reprezintă placa de dezvoltare BeagleV®-Fire, care integrează un procesor RISC-V. Arhitectura RISC-V reprezintă o arhitectură a setului de instrucțiuni (ISA) disponibilă publicului larg (opne-source) spre dezvoltarea de aplicații proprii fără a fi necesară licențierea. Această arhitectură a fost introdusă în anul 2010, devenind populară atât în industrie, cât și în domeniul academic și al cercetării datorită asociației RISC V International, ce se ocupă de dezvoltarea și extinderea acesteia [1]. Conceptul RISC-V a fost inițiat ca o arhitectură complet nouă, ghidată de o serie de obiective majore care urmăreau crearea unei arhitecturi de set de instrucțiuni RISC scalabile, capabile

să acopere un spectru larg de aplicații – de la dispozitive embedded cu consum energetic extrem de redus, până la multiprocesoare de înaltă performanță pentru servere cloud [2].

Proiectul de față este conceput atât pentru a atinge performanțe ridicate în procesul de conversie a spațiilor de culoare, cât și pentru a aduce un plus de cunoștințe mediului academic legat de cum se poate folosi eficient arhitectura RISC-V. Având în vedere accesibilitatea sa ridicată, indiferent de domeniul de aplicabilitate, în contrast cu majoritatea ISA-urilor comerciale existente, care sunt protejate de drepturile de proprietate intelectuală ale companiilor dezvoltatoare, RISC-V reprezintă cea mai favorabilă opțiune pentru acest proiect. În plus, considerând direcția actuală de dezvoltare a industriei semiconductoarelor și faptul că tot mai multe tehnologii emergente adoptă arhitectura RISC-V ca fundament pentru inovație, familiarizarea cu această arhitectură nu mai reprezintă doar un avantaj competitiv, ci o necesitate esențială pentru a răspunde cerințelor pieței tehnologice contemporane.

1.2 Motivația practică pentru alegerea temei

Ochiul uman percepe culorile prin intermediul unor celule fotoreceptoare numite conuri, care se găsesc la nivelul zonei centrale a retinei și conțin în mod normal 3 tipuri de pigmenți: roșu, verde și albastru. Prin stimularea acestor celule, creierul uman percepe culorile, care sunt, de fapt, o combinație în proporții diferite a celor trei nuanțe de bază. Din acest punct de vedere, spațiul de culoare RGB este mai apropiat de modul în care oamenii percep culorile, față de celelalte variante. Însă, având în vedere că în spațiul RGB fiecare pixel este reprezentat sub formă de 3 valori, fiecare având același lățime de bandă, sunt necesare resurse consistente în ceea ce privește stocarea și transferul datelor. Luând în considerare faptul că ochiul uman este mai sensibil la modificările luminozității în detrimentul factorilor de culoare și că aproximativ 60%-70% din informația luminoasă se găsește în componenta verde a unui pixel, se poate reduce semnificativ lățimea de bandă necesară prin eliminarea informației luminoase din componentele roșu și albastru, rezultând formatul YUV. Astfel, formatul YUV permite reducerea semnificativă a necesităților de stocare și lățime de bandă, fără a produce diferențe vizibile în calitatea imaginii.

În contextul actual, unde cerințele de prelucrare video în timp real sunt tot mai riguroase, procesarea imaginilor în spațiul RGB se dovedește a fi mai complexă și mai costisitoare din punct de vedere al resurselor. Fiecare pixel trebuie să fie procesat în întregime pentru toate cele trei componente RGB, ceea ce implică mai multe calcule și o utilizare intensivă a memoriei. În schimb, procesarea datelor YUV este mult mai eficientă, deoarece schimbările de culoare pot fi aplicate doar pe componentele U și V, în timp ce Y (luminanța) rămâne constantă, ceea ce permite reducerea complexității operațiunilor de calcul și a necesităților de stocare.

În această lumină, accelerarea hardware a conversiei YUV-RGB este o necesitate, mai ales în scenariile care implică procesare video în timp real, cum ar fi televiziunea digitală, videoconferințele și aplicațiile de procesare video avansată. Aceste aplicații impun o gestionare eficientă a resurselor, reducerea latenței și maximizarea performanței. Utilizarea unui FPGA pentru realizarea conversiei hardware între YUV și RGB, așa cum este propus în acest proiect, oferă avantaje considerabile în termeni de viteză și eficiență față de soluțiile software tradiționale, care pot introduce latență și consum mare de energie.

De asemenea, un factor esențial ce a dus la alegerea acestui proiect îl reprezintă dorința de adaptare a cunoștințelor dobândite în mediul academic, la avansul tehnologic actual al industriei. Astfel, s-a ales o implementare având drept resursă hardware o placă de dezvoltare având un procesor cu arhitectură RISC-V, care prezintă, totodată, și o structură FPGA, potrivită datorită capacității sale de procesare paralelă.

1.3 Obiectivele proiectului

Scopul principal al acestui proiect constă în dezvoltarea și implementarea unei soluții hardware pentru conversia imaginilor de diferite rezoluții din formatul YUV422 în formatul RGB, utilizând atât limbajul de descriere hardware Verilog pentru modelarea semnalelor pe FPGA, cât și limbajul de programare C pentru furnizarea datelor de intrare de către un procesor cu arhitectură RISC-V. Aceeași conversie va fi realizată, în paralel folosind tehnici pur software, fără implicarea structurii FPGA. Vor fi comparate performanțele acestor două abordări în termeni de viteză de procesare, lățime de bandă și utilizare a memoriei.

Primul pas în procesul de dezvoltare a proiectului este analiza algoritmilor existenți pentru conversia YUV-RGB oferind o bază teoretică solidă pentru implementarea propusă. Se va prezenta, apoi, placa BeagleV®-Fire, evidențiind caracteristicile sale relevante pentru proiect, și se vor detalia resursele FPGA disponibile, esențiale pentru implementarea conversiei. Mai departe, este descrisă arhitectura sistemului propus, incluzând diagrama bloc și explicarea fluxului de date și control, pentru a oferi o înțelegere clară a structurii și funcționării acestuia. Urmează detalierea procesului de dezvoltare a proiectului și analiza rezultatelor obținute în termeni de performanță, pentru a stabili concluziile finale. Acest proces va permite identificarea avantajelor și limitărilor fiecărei abordări, oferind o bază solidă pentru selecția soluției optime în funcție de cerințele specifice ale aplicațiilor de procesare ale imaginilor.

2 STUDIUL LITERATURII DE SPECIALITATE

2.1 ARHITECTURA RISC-V

Arhitectura și proiectarea procesoarelor influențează în mod semnificativ eficiența și performanța dispozitivelor electronice. De-a lungul timpului, au fost dezvoltate diverse seturi de instrucțiuni (Instruction Set Architectures – ISAs), fiecare fiind conceput pentru aplicații specifice. Printre acestea se numără și RISC-V ISA, dezvoltată de Fundația RISC V, care este disponibilă gratuit pentru oricine dorește să o utilizeze, modifice sau implementeze, fără a fi necesare costuri de licențiere [3]. RISC-V este o arhitectură modulară, oferind un grad ridicat de flexibilitate și opțiuni de personalizare. Dezvoltatorii pot adapta setul de instrucțiuni la cerințele specifice ale aplicației lor, fără a fi constrânși de limitări. Singura cerință obligatorie este suportul pentru setul de instrucțiuni de bază pentru operații cu numere întregi (RV32I sau RV64I). Pentru a adăuga capabilități suplimentare de procesare, ISA-ul de bază poate fi extins prin integrarea unor extensii standardizate, fiecare dintre acestea adresându-se unor cazuri de utilizare specifice. RISC-V oferă dezvoltatorilor libertatea de a alege doar extensiile necesare, în funcție de aplicația vizată [4]. În plus, arhitectura permite definirea de extensii personalizate ale setului de instrucțiuni, oferind posibilitatea de a adapta ISA-ul prin introducerea unor instrucțiuni speciale, optimizate pentru cerințele aplicației respective [5]. Printre cele mai uitlizate extensii se numără cele prezentate în Tabelul 1.

În arhitectura RISC-V, registrele au o dimensiune standard de 32 de biți, iar aceste grupări de biți sunt atât de frecvent utilizate încât primesc denumirea de cuvânt (word). O altă dimensiune des întâlnită este cea de 64 de biți, cunoscută sub numele de dublu-cuvânt (doubleword). În implementările uzuale ale arhitecturii RISC-V, procesorul pune la dispoziția programatorului un set fix de 32 de registre, fapt ce impune o gestionare atentă a resurselor în timpul execuției instrucțiunilor [6]. Un număr foarte mare de registre poate duce la creșterea duratei unui ciclu de ceas, întrucât semnalele electronice au nevoie de mai mult timp pentru a parcurge distanțe mai mari în cadrul circuitului. Un alt motiv pentru care nu se utilizează mai mult de 32 de registre este legat de spațiul necesar în formatul instrucțiunilor - un număr mai mare de registre ar presupune un câmp mai larg pentru adresarea acestora, crescând astfel dimensiunea totală a instrucțiunii [6].

Tabel 1: Extensii uzuale ale setului de instrucțiuni RISC V [6], [7]

Cod	Denumire	Descriere
I	Integer Base Instruction Set	Setul de instrucțiuni de bază pentru operații cu numere întregi pe 32 de biți; obligatoriu în orice implementare RISC-V.
M	Integer Multiplication and Division	Adaugă suport pentru operații de multiplicare și împărțire între numere întregi.
A	Atomic Instructions	Permite operații atomice necesare pentru programarea concurentă și sistemele multi-core.
F	Single-Precision Floating-Point	Oferă suport pentru calcule în virgulă mobilă pe 32 de biți, conform IEEE 754.
D	Double-Precision Floating-Point	Extinde extensia F pentru a permite operații pe 64 de biți.
C	Compressed Instructions	Permite utilizarea de instrucțiuni pe 16 biți pentru a reduce dimensiunea codului și a îmbunătăți eficiența memoriei.
B	Bit-Manipulation	Adaugă instrucțiuni pentru manipularea eficientă a biților, utile în criptografie și procesare de semnal.
V	Vector Extension	Suportă operații vectoriale paralele, esențiale pentru aplicații HPC, AI și grafică.
Zicsr	Control and Status Registers	Instrucțiuni pentru accesul la registrele CSR, utilizate în mod privilegiat.
Zifencei	Instruction-Fetch Fence	Asigură coerența între memoria de instrucțiuni și execuția codului, necesară în medii partajate.

Arhitectura RISC-V a fost proiectată încă de la început pentru a executa instrucțiunile folosind tehnica pipeline, reprezentată în Figura 1 [8]. În primul rând, toate instrucțiunile RISC-V au aceeași lungime. Această constrângere facilitează semnificativ etapele de preluare a instrucțiunii (fetch) în prima etapă a pipeline-ului și de decodare în etapa a doua. În al doilea rând, RISC-V utilizează doar câteva formate de instrucțiuni, iar câmpurile pentru registrele sursă și destinație sunt poziționate în aceleași locații în fiecare format, simplificând astfel procesul de decodare. În al treilea rând, operanzii din memorie apar exclusiv în instrucțiunile de tip load și store. Această restricție permite utilizarea etapei de execuție (EX) pentru calculul adresei efective de memorie, urmată de accesul efectiv la memorie în etapa următoare (MEM)[6].

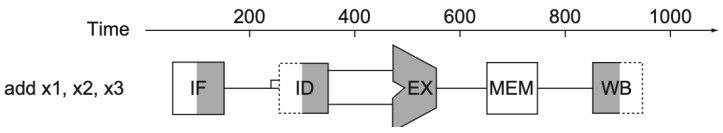


Figura 1: Reprezentare grafică a etapelor de execuție a unei instrucțiuni în arhitectura RISC V [6]

Arhitectura RISC-V reprezintă o abordare modernă, flexibilă și deschisă în proiectarea procesoarelor, fiind concepută pentru a răspunde atât cerințelor simplității hardware, cât și nevoilor de extindere și adaptabilitate. Prin structurarea clară a setului de instrucțiuni, folosirea registrelor de uz general pe 32 de biți și execuția eficientă prin pipeline, RISC-V oferă o bază solidă pentru implementări variate - de la microcontrolere simple până la sisteme complexe de calcul paralel [9].

Faptul că arhitectura este gratuită și modulară facilitează inovația și facilitează dezvoltarea de procesoare, atât în mediul academic, cât și în industrie. Prin aceste caracteristici distinctive, RISC-V devine nu doar o alternativă la arhitecturile tradiționale, ci o fundație viabilă pentru viitorul arhitecturii de calcul.

2.2 Procesarea de imagini colore

Unul dintre cele mai importante aspecte în procesarea imaginilor digitale îl reprezintă utilizarea culorii, atât ca element descriptiv, cât și ca suport esențial în analiză și compresie. Așa cum este evidențiat în lucrarea fundamentală „Digital Image Processing" [10], culoarea servește două scopuri esențiale: în primul rând, facilitează identificarea și extragerea obiectelor dintr-o scenă, iar în al doilea rând, se aliniază capacității vizuale a omului de a distinge mii de nuanțe cromatice. Percepția culorilor la om este rezultatul unui proces complex cu caracter fiziopsihologic, care nu este pe deplin înțeles nici în prezent. Totuși, acest fenomen poate fi explicat teoretic și analizat prin prisma rezultatelor experimentale. Un punct de plecare în acest sens este descoperirea lui Isaac Newton [11] legată de trecerea unei raze de lumină albă printr-o prismă din sticlă și descompunerea sa într-un spectru de culori împărțit în șase regiuni principale: violet, albastru, verde, galben, portocaliu și roșu. Astfel, culorile percepute de ochiul uman sunt determinate de natura radiației reflectate de acel obiect. Lumina vizibilă reprezintă doar o mică porțiune a spectrului electromagnetic, iar un corp care reflectă uniform toate lungimile de undă vizibile va fi perceput ca alb. În schimb, un obiect care reflectă doar o anumită gamă de lungimi de undă va fi perceput într-o anumită culoare, așa cum este evidențiat în Figura 2.

Conform Figurii 2, lumina cromatică acoperă spectrul vizibil de culoare ce se încadrează în intervalul [400nm, 700nm]. Lumina cromatică este caracterizată prin trei concepte fundamentale: radianța, luminanța și strălucirea. Radianța reprezintă cantitatea totală de energie emisă de o sursă de lumină pe unitatea de timp și este exprimată în watt (W). Luminanța, măsurată în lumeni (lm), descrie energia percepută de un observator uman, fiind dependentă de sensibilitatea ochiului la diferite lungimi de undă. Strălucirea, spre deosebire de celelalte două, este un parametru subiectiv, fiind asociat cu percepția individuală a intensității luminii. Aceasta reflectă o noțiune acromatică – adică independentă

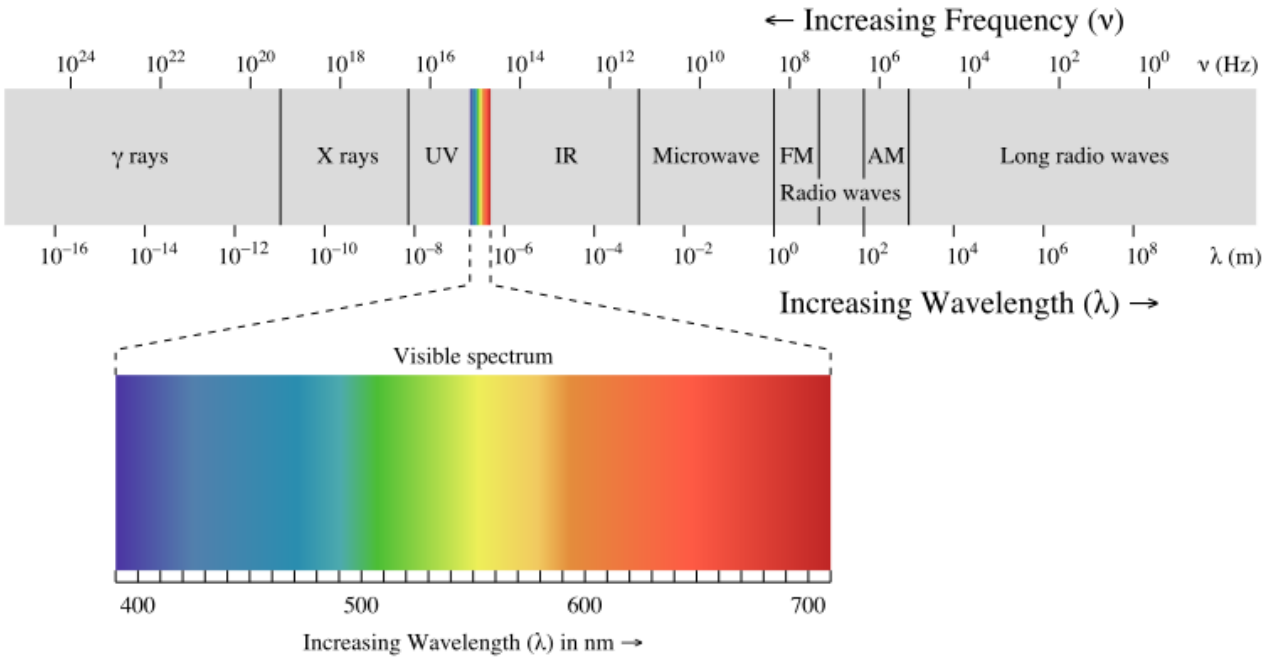


Figura 2: Banda de frecvențe a spectrului electromagnetic [12]

de culoare – și are un rol esențial în formarea senzației vizuale de culoare.

Percepția umană a culorii este rezultatul stimulării celulelor cu con ale retinei, care răspund la lungimi de undă corespunzătoare celor trei culori primare: roșu (R), verde (G) și albastru (B). Această tricomponentă stă la baza modelului RGB, unul dintre cele mai utilizate modele în arhitectura diverselor ecrane folosite în procesarea și redarea imaginilor color.

Pentru a diferenția culorile între ele, se recurge, în mod uzual, la trei caracteristici fundamentale: strălucirea (brightness), nuanța (hue) și saturația (saturation) [13]. Strălucirea reflectă componenta acromatică a unei culori, fiind corelată cu intensitatea luminozității percepute – similar cu tonurile de gri într-o imagine monocromă.

Nuanța este asociată cu lungimea de undă dominantă dintr-un amestec de radiații luminoase. Aceasta determină culoarea de bază percepută de observator. Pe de altă parte, saturația exprimă puritatea culorii – adică gradul în care culoarea este diluată de lumină albă. Împreună, nuanța și saturația formează ceea ce se numește cromaticitate (chromaticity), prin urmare orice culoare poate fi descrisă prin cromaticitate și strălucire.

În modelarea matematică a culorilor, se utilizează conceptul de valori tristimulus, notate cu X, Y și Z, ale căror ecuații sunt prezentate în 1, 2, 3, 4 , care reprezintă cantitățile necesare din cele trei componente de bază (roșu, verde, albastru) pentru a reproduce o anumită culoare. Pe baza acestor valori se calculează coeficienții tricromatici, care permit reprezentarea unică a unei culori în spațiul colorimetric standardizat de Comisia Internațională de Iluminare (CIE – Commission Internationale de

l'Éclairage), autoritatea globală în domeniul științei culorii și iluminării.

$$x = \frac{X}{X + Y + Z} \tag{1}$$

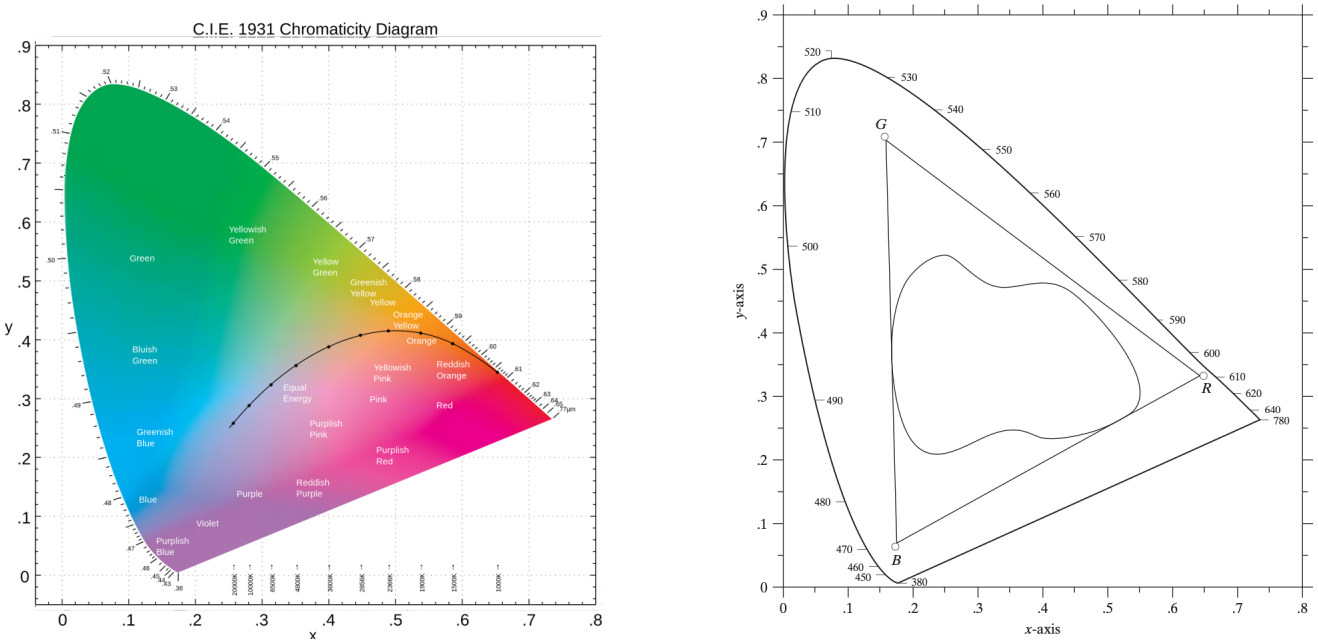
$$y = \frac{Y}{X + Y + Z} \tag{2}$$

$$z = \frac{Z}{X + Y + Z} \tag{3}$$

Se poate observa că :

$$x + y + z = 1 \tag{4}$$

Pe baza ecuațiilor fundamentale și a analizei diagramei de cromaticitate CIE [14] din Figura 3a, care descrie compoziția unei culori prin coordonatele x (corespunzătoare componentei roșii) și y (corespunzătoare componentei verzi), se poate realiza o definire riguroasă și standardizată a unei culori date. Diagrama este deosebit de utilă și în analiza amestecurilor de culori, întrucât o linie trasată între două puncte din diagramă indică toate combinațiile liniare posibile între acele două culori. În mod analog, pentru trei culori distincte, zona delimitată de triunghiul format prin conectarea celor trei puncte corespunde gamutului cromatic (spațiului de culori) generabil prin amestecul acestora. Figura 3b ilustrează această idee, prezentând gama cromatică asociată spațiului RGB.



(a) Diagrama de cromaticitate CIE 1931 [15]

(b) Gama cromatică a modelului RGB [10]

Figura 3: Reprezentarea diagramei cromaticității și gamei cromatice RGB

2.2.1 Spații de culoare

Un spațiu de culoare, denumit și sistem sau model de culoare, reprezintă o specificație a unui sistem de coordonate și a unui subspațiu în cadrul acestuia, în care fiecare culoare este reprezentată de un punct unic [10].

În prezent, modelele de culoare utilizate în procesarea imaginilor pot fi clasificate în funcție de scopul pentru care au fost dezvoltate: modele orientate spre hardware, cum ar fi cele utilizate la monitoare, camere video sau imprimante, și modele orientate spre aplicații, folosite în manipularea culorilor pentru grafică, animație sau vizualizări perceptuale [16]. În categoria celor orientate hardware, cele mai utilizate modele sunt:

- RGB - modelul standard folosit pentru reprezentarea culorilor pe monitoare și dispozitive electronice;
- CMY și CMYK (Cyan, Magenta, Yellow, [Black]) – specifice imprimantelor color, unde amestecul substractiv de pigmenți este esențial;
- HSI (Hue, Saturation, Intensity) – un model apropiat de modul în care oamenii percep și descriu culorile, care separă informația cromatică (nuanță și saturație) de intensitatea luminii, permițând aplicarea directă a tehnicilor de prelucrare a imaginilor în tonuri de gri asupra componentei de intensitate;

Există o varietate considerabilă de modele de culoare, acest fapt reflectând complexitatea domeniului de procesare a imaginilor colore, care intersectează numeroase arii aplicative: de la vizualizare și estetică, până la transmisiuni video și analiză spectrală. Deși unele modele pot părea pur teoretice, ele oferă perspective utile și relevante asupra modului în care culoarea poate fi reprezentată, prelucrată și optimizată în diferite contexte tehnologice.

2.2.2 Spațiul de culoare RGB

Modelul RGB (Red, Green, Blue) este unul dintre cele mai utilizate spații de culoare în procesarea digitală a imaginilor și se bazează pe componentele spectrale primare: roșu, verde și albastru. Acest model este reprezentat într-un sistem de coordonate cartezian tridimensional, sub forma unui cub evidențiat în Figura 4, în care fiecare culoare este definită printr-o combinație de valori asociate celor trei componente fundamentale [10]. Pentru simplitate, se consideră de obicei că valorile RGB sunt normalizate în intervalul [0,1], ceea ce face ca acest spațiu să fie echivalent cu un cub unitar. Fiecare culoare din acest model este reprezentată de un vector cu originea în punctul (0,0,0) și direcția către

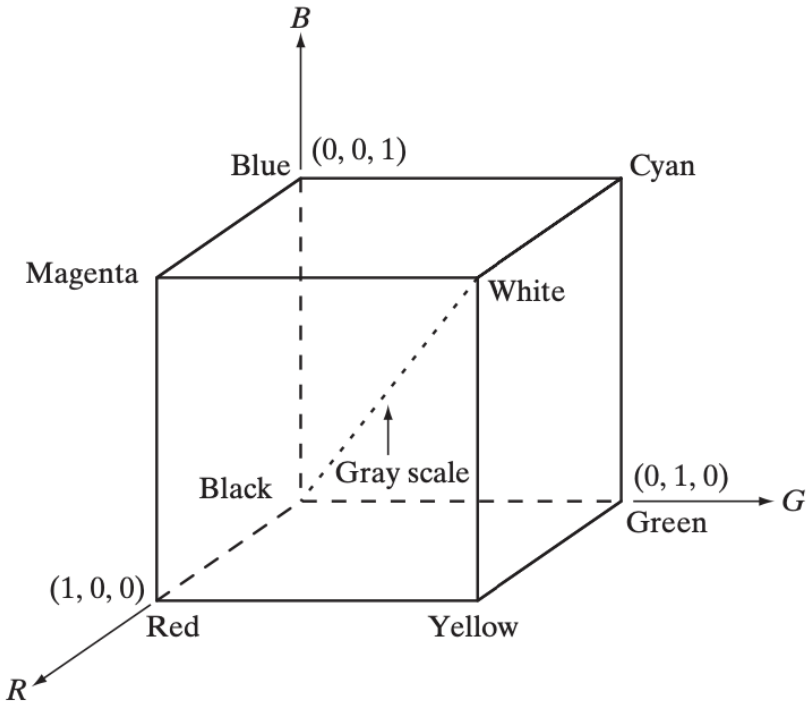


Figura 4: Reprezentarea sub formă de cub unitar a spațiului de culoare RGB [10]

un punct din interiorul sau de pe suprafața cubului. În această reprezentare, negrul se află în origine, unde toate cele trei componente au valoarea minimă (0,0,0), iar albul se situează în colțul opus, unde valorile sunt maxime (1,1,1). Culorile secundare – cyan, magenta și galben – ocupă celelalte colțuri ale cubului, rezultând din combinații de câte două componente primare la valoare maximă. Tonurile de gri sunt reprezentate de punctele de pe diagonala principală a cubului, între negru și alb, unde valorile R, G și B sunt egale. Orice altă culoare se regăsește într-un punct situat în interiorul sau pe suprafața cubului.

Imaginile reprezentate în modelul RGB sunt compuse din trei planuri de culoare, câte unul pentru fiecare componentă primară. Acestea sunt afișate simultan de către un monitor RGB, rezultând astfel imaginea color compusă. Cantitatea de informație asociată fiecărui pixel este determinată de adâncimea de culoare a pixelului (pixel depth).

De exemplu, o imagine RGB în care fiecare canal (R, G și B) este reprezentat pe 8 biți va avea o adâncime totală de 24 de biți per pixel (3 canale × 8 biți). Aceasta este cunoscută în mod obișnuit ca o imagine color completă (full-color image). Numărul total de culori care pot fi reprezentate într-o astfel de imagine este $(2^8)^3 = 16.777.216$ culori distincte.

Deși afișajele moderne de înaltă performanță și plăcile grafice profesionale permit redarea fidelă a culorilor dintr-o imagine RGB pe 24 de biți, o parte considerabilă a sistemelor aflate încă în uz operează cu palete limitate, uneori restrânse la doar 256 de culori. În plus, există aplicații – precum procesarea

imaginilor în culori pseudocolor – în care utilizarea unui număr redus de culori este suficientă și chiar preferabilă, din considerente de eficiență și compatibilitate [17]. În acest context, s-a conturat conceptul de culori sigure RGB (safe RGB colors), adică un subset de culori care pot fi afișate în mod consecvent și fidel pe majoritatea dispozitivelor, indiferent de capabilitățile hardware ale utilizatorului. În aplicațiile web, aceste culori sunt cunoscute sub denumirea de culori sigure pentru browsere (safe Web colors), fiind standardizate pentru a asigura o redare uniformă pe toate platformele.

Deși modelul RGB este intuitiv și larg utilizat în reprezentarea imaginilor color, acesta nu este optim din punctul de vedere al eficienței în procesare și transmisie, în special în aplicațiile care implică lățime de bandă restricționată. Fiecare componentă (roșu, verde și albastru) trebuie transmisă sau procesată cu aceeași rezoluție și adâncime de culoare, ceea ce impune un consum constant și ridicat de resurse [18].

În scenarii precum redarea video sau compresia imaginilor, modificarea proprietăților unui pixel (cum ar fi intensitatea sau nuanța) presupune accesarea și recalcularea tuturor celor trei componente RGB, urmată de rescrierea valorilor în memorie. Acest proces este ineficient în comparație cu o abordare care separă clar informația de luminanță de cea de culoare (cromatică).

Din acest motiv, multe standarde de compresie și transmisie video adoptă spații de culoare care utilizează o componentă de luminanță (luma, Y) și două componente de diferență de culoare. Unul dintre cele mai comune astfel de modele este spațiul de culoare YUV, care permite o reprezentare mai compactă și mai eficientă a imaginilor, păstrând fidelitatea vizuală și reducând cerințele de procesare și stocare.

2.2.3 Spațiul de culoare YUV

Spațiul de culoare YUV a fost dezvoltat inițial pentru a permite compatibilitatea dintre transmisiunile video alb-negru și cele color, în contextul televiziunii analogice. Scopul său principal a fost separarea informației de luminozitate (luminanță) de informația de culoare (cromanță), astfel încât semnalul alb-negru să fie transmis fără degradare, iar componentele de culoare să fie adăugate adițional pentru dispozitivele capabile să le interpreteze [19].

Primele implementări la scară largă ale acestui model de culoare s-au regăsit în standardele PAL (Phase Alternating Line) - Europa și NTSC (National Television System Committee) - SUA, utilizate în sistemele de televiziune din diverse regiuni ale lumii. Odată cu progresul tehnologic și tranziția către formatele video digitale, spațiul YUV a fost adoptat pe scară largă în cadrul algoritmilor de compresie video. Standardele MPEG (Moving Picture Experts Group), care au revoluționat modul de codare și transmisie a conținutului video, au integrat YUV ca model de culoare principal. Capacitatea acestuia

de a menține o calitate vizuală ridicată în paralel cu o reducere semnificativă a dimensiunii fișierelor l-a transformat într-un element esențial în era multimedia digitală [20].

Spațiul de culoare YUV continuă să joace un rol esențial în multiple domenii, în special în procesarea video, serviciile de streaming și transmisiile digitale. Importanța sa este accentuată de transmisia conținutului de înaltă definiție (HD) și ultra-înaltă definiție (UHD), în care gestionarea eficientă a datelor devine crucială pentru performanță.

În domeniul transmisiunilor digitale, YUV este integrat în standarde internaționale precum DVB (Digital Video Broadcasting) și ATSC (Advanced Television Systems Committee), folosite pentru difuzarea semnalelor TV de înaltă definiție prin mijloace terestre, satelit și prin cablu. Platformele de streaming, precum Netflix, Hulu sau YouTube, beneficiază în mod direct de avantajele oferite de YUV. Prin algoritmii de compresie bazate pe acest model de culoare, aceste servicii reușesc să optimizeze lățimea de bandă, asigurând redarea fluidă a conținutului video chiar și în condiții de conexiune slabă la internet.

Pentru a valorifica pe deplin avantajele oferite de spațiul de culoare YUV în aplicațiile digitale, a fost introdus un mecanism esențial: eșantionarea diferențiată a componentelor de culoare, cunoscută sub numele de chroma subsampling [21]. Acest mecanism are la bază reducerea rezoluției componentelor de cromatică pentru a face posibilă optimizarea dimensiunii fișierelor video și a ratei de transfer a datelor, fără degradarea semnificativă a calității percepute a imaginii. Astfel, se pot distinge formatele de eșantionare prezentate în Tabelul 2 și în Figura 5.

Tabel 2: Formate YUV de subeșantionare [21]

Format YUV	**Descriere**
4:4:4	Fără subeșantionare; rezoluție completă pentru luminanță și crominanță
4:2:2	Subeșantionare orizontală (factor 2) pentru crominanță; fără subeșantionare verticală Fiecare linie are 4 eșantioane Y și 2 eșantioane U/V
4:2:0	Subeșantionare orizontală și verticală (factor 2) pentru crominanță
4:1:1	Subeșantionare orizontală (factor 4); fără subeșantionare verticală Fiecare linie are 4 eșantioane Y și 1 eșantion U/V
4:1:0	Subeșantionare orizontală și verticală (factor 4) pentru crominanță

Dincolo de această clasificare conceptuală, modul în care aceste formate sunt efectiv reprezentate în memorie are un impact semnificativ asupra procesării grafice. Stocarea în memorie depinde de următoarele concepte:

- Originea suprafeței (Surface Origin): Pentru toate formatele YUV, originea imaginii în memorie este considerată a fi colțul din stânga sus (coordonata (0,0)). Acest punct marchează începutul

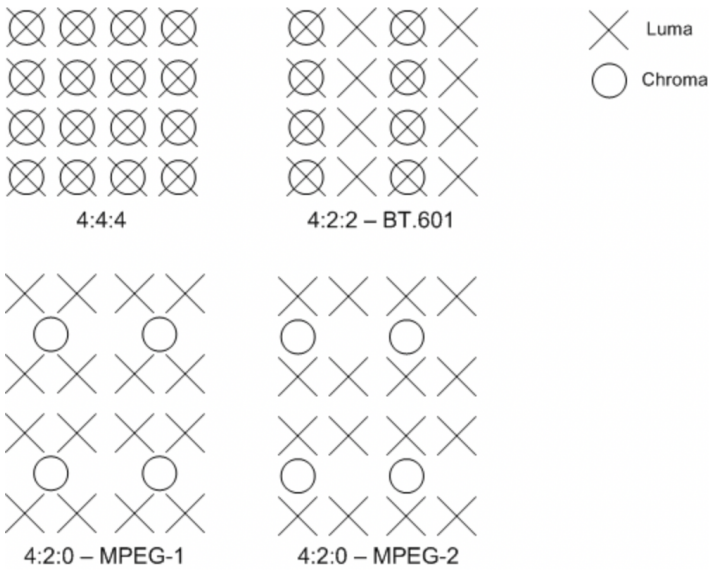


Figura 5: Reprezentarea schematică a formatelor de subeșantionare YUV [21]

fiecărei linii de pixeli pe verticală.

- Stride: Stride-ul, cunoscut și ca pitch, reprezintă lățimea unei linii de pixeli în memorie, exprimată în octeți. Stride-ul nu reflectă neapărat lățimea logică a imaginii în pixeli, deoarece acesta poate include un spațiu suplimentar de aliniere. Având originea suprafeței în colțul din stânga sus, valoarea stride-ului este întotdeauna pozitivă.

- Alinierea suprafeței (Alignment): Din motive de performanță și compatibilitate cu hardware-ul grafic, fiecare linie de pixeli dintr-o imagine trebuie să fie aliniată la o graniță de 32 de biți. În unele cazuri, alinierea poate fi și mai mare, în funcție de cerințele specifice ale plăcii grafice.

- Formate packed și planar: În funcție de cum sunt stocate componentele Y, U și V în memorie, formatele YUV pot fi împărțite în:

  - Formatul packed: componentele Y, U și V sunt stocate intercalat într-un singur tablou de date, organizate în grupuri de pixeli (macropixeli), a căror structură variază în funcție de formatul specific [21].
  - Formatul planar: fiecare componentă (Y, U, V) este stocată într-un plan separat, ceea ce permite procesarea individuală a fiecărei componente, fiind des întâlnit în aplicațiile profesionale de procesare video [21].

Fiecare dintre formatele YUV menționate anterior este asociat cu un cod identificator unic numit FOURCC (Four Character Code). Acesta este un identificator de 32 de biți, format prin concatenarea a patru caractere ASCII, folosit în mod obișnuit pentru a desemna formatele video în cadrul API-

urilor multimedia, sistemelor de codare și decodare, dar și în driverele plăcilor grafice. Utilizarea codurilor FOURCC facilitează identificarea și manipularea automată a datelor video de către software și hardware, oferind o standardizare clară în procesele de randare și procesare video.

În cadrul acestui proiectului s-a optat pentru formatul YUV 4:2:2, un standard frecvent utilizat în aplicații video profesionale datorită echilibrului său între calitatea imaginii și eficiența în stocare și transmisie. Acest format este asociat în cu codurile FOURCC YUY2 și UYVY, ambele fiind formate packed [21].

În formatul YUY2, datele sunt organizate asemenea Figurii 6. Primul octet conține componenta Y pentru primul pixel, al doilea octet conține componenta U comună pentru ambii pixeli, al treilea octet conține componenta Y pentru al doilea pixel, iar al patrulea octet conține componenta V, rezultând un model de tip Y0 U Y1 V. Atunci când memoria este adresată ca un tablou de valori de tip WORD (16 biți) în little-endian, primul WORD conține Y0 în LSB și U în MSB, iar al doilea WORD conține Y1 în LSB și V în MSB.

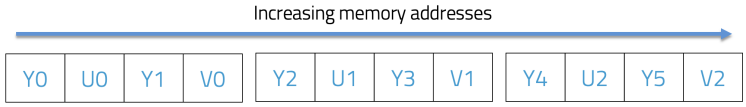


Figura 6: Reprezentarea a schematică modului de stocare pentru formatul YUY2

De cealaltă parte, UYVY păstrează același model de subeșantionare și aceeași logică de împachetare, însă ordinea octeților este inversată, după modelul U Y0 V Y1, asemenea Figurii 7. În reprezentarea pe WORD little-endian, primul WORD conține U în LSB și Y0 în MSB, iar al doilea WORD conține V în LSB și Y1 în MSB.

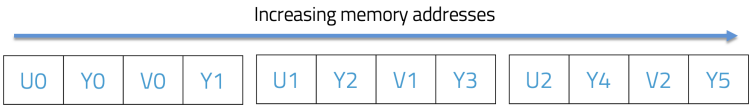


Figura 7: Reprezentarea schematică a modului de stocare pentru formatul UYVY

2.3 Conversia între spațiile de culoare RGB și YUV 4:2:2

Conversia din YUV în RGB este un pas esențial în procesarea video, întrucât majoritatea dispozitivelor de afișare (monitoare, televizoare, ecrane LED etc.) operează în spațiul RGB.

Conversia din spațiul de culoare YUV către RGB se bazează pe relații matematice bine definite, care țin cont de diferențele perceptuale dintre componentele de luminanță (Y) și crominanță (U și V). În cazul formatului YUV 4:2:2, componenta Y este eșantionată pentru fiecare pixel, în timp ce U și V sunt

partajate între doi pixeli adiacenți pe orizontală. Acest lucru presupune o reconstrucție intermediară a valorilor U și V pentru fiecare pixel individual, înainte de aplicarea transformării către RGB.

Standardul ITU-T T.871 [22] definește componentele crominanței (U și V) în intervalul 0-255, cu o valoare centrală de 128 corespunzând absenței culorii (zero crominanță). Pentru a aplica corect ecuațiile de conversie, este necesară translatarea acestor valori într-un interval simetric în jurul lui zero (-128 la 127). Astfel, din fiecare valoare U și V se scade 128. Scăderea lui 128 aliniază valorile U și V cu forma ecuațiilor BT.601 [23], unde termenii (U – 128) și (V – 128) sunt utilizați direct. Componenta de luminanță Y nu necesită offset, deoarece este deja definită în intervalul 0-255 ca o valoare absolută.

Ecuațiile ITU-T T.871 [22] specifică coeficienții prezentați în ecuațiile 5, 6, 7, 8 pentru conversia YUV-RGB:

$$1.402 \times 1024 = 1435.648 \approx 1436 \quad \Rightarrow \quad 16'h059C \tag{5}$$

$$0.344 \times 1024 = 352.256 \approx 352 \quad \Rightarrow \quad 16'h0160 \tag{6}$$

$$0.714 \times 1024 = 731.136 \approx 731 \quad \Rightarrow \quad 16'h02DB \tag{7}$$

$$1.772 \times 1024 = 1814.528 \approx 1814 \quad \Rightarrow \quad 16'h0716 \tag{8}$$

Standardul ITU-T T.871 [22] specifică Ecuațiile 9, 10 și 11 pentru conversia YUV-RGB:

$$R = Y + 1.402 \cdot (V - 128) \tag{9}$$

$$G = Y - 0.344 \cdot (U - 128) - 0.714 \cdot (V - 128) \tag{10}$$

$$B = Y + 1.772 \cdot (U - 128) \tag{11}$$

3 DESCRIEREA PLATFORMEI HARDWARE

3.1 PREZENTAREA PLĂCII BEAGLEV®-FIRE

BeagleV®-Fire, evidențiată în Figura 8, este o placă de dezvoltare de tip single-board computer (SBC), ce integrează un System-on-Chip (SoC) Microchip PolarFire® MPFS025T, care include un total de cinci nuclee RISC-V: patru nuclee RV64GC (U54-MC) pentru aplicații generale și un nucleu RV64IMAC (E51) dedicat funcțiilor de monitorizare și boot, alături de o structură FPGA integrată, compusă din 23.000 de elemente logice (LUT-uri de 4 intrări și bistabile), 68 de blocuri matematice (multiplicatoare 18×18 MACC) și patru canale SerDes cu viteze de până la 12,7 Gbps [24].

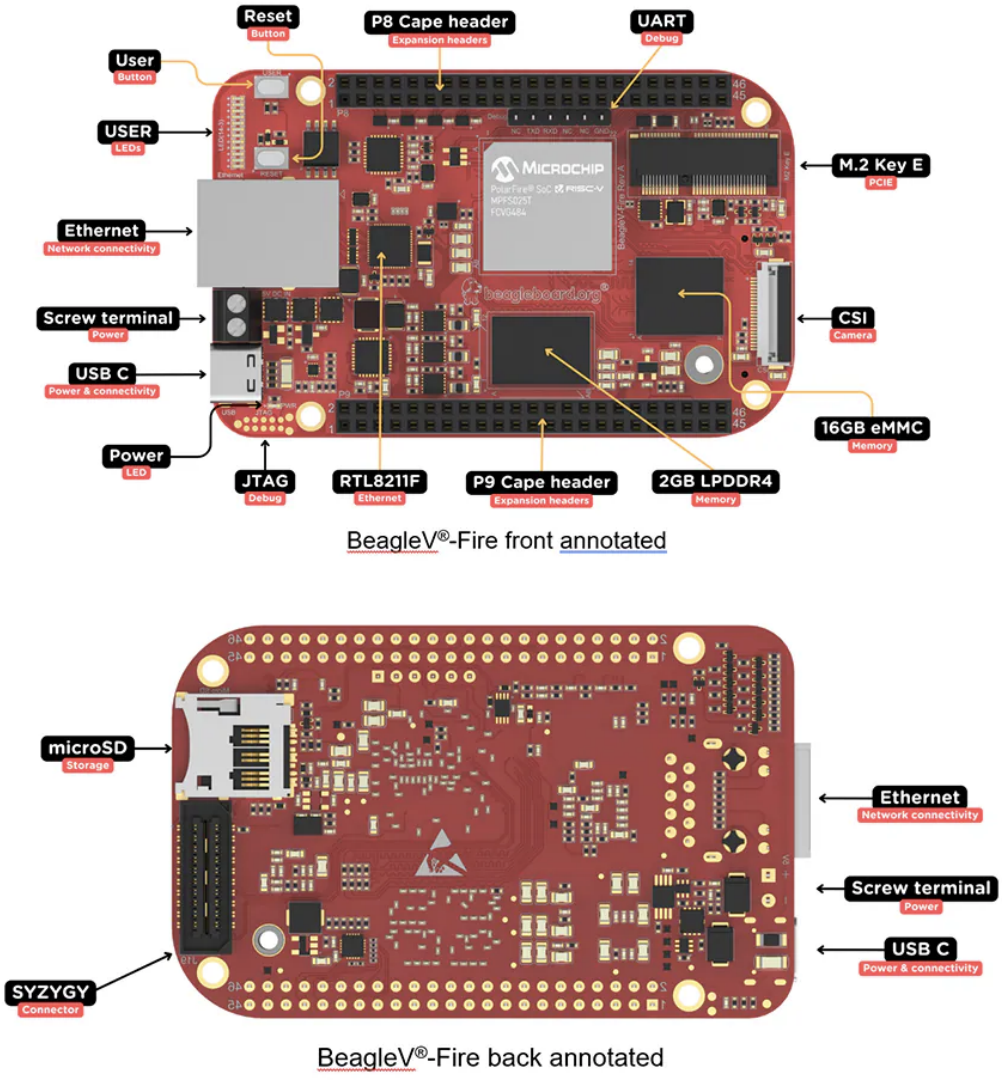


Figura 8: Placa BeagleV®-Fire [24]

3.1.1 Carcateristici fizice

BeagleV®-Fire, având diagrama bloc prezentată în Figura 9, are dimensiuni de 86.38 * 54.61 * 18.8 mm și o grosime a PCB-ului de 1.6mm (12 straturi). Placa este compatibilă RoHS și cântărește aproximativ 45.8g [24].

La proiectarea plăcii s-a utilizat PCB FR-4 Tg150, un material cu stabilitate termică ridicată (Tg = 150°C), ideal pentru medii industriale. Finisajul ENIG (Electroless Nickel Immersion Gold) asigură o suprafață de contact fiabilă pentru conectori, rezistentă la oxidare. BeagleV®-Fire integrează pad-uri termice pentru atașarea heatsink-urilor externe, esențiale în aplicații cu încărcare continuă. SoC-ul PolarFire are senzori interni de temperatură (±1°C precizie) și mecanisme de throttling dinamic pentru evitarea supraîncălzirii.

De asemenea, placa este prevăzută cu un layout intuitiv care facilitează timpul de învățare și monitorizarea erorilor. Aceasta este prevăzută cu:

- LED-uri de stare (alimentare, activitate FPGA, boot), ce oferă feedback imediat;
- Conectorii periferici (USB-C, Ethernet, M.2) sunt poziționați pe margini pentru acces facil;
- Găuri de montaj (4×) permit fixarea mecanică stabilă în carcasă.

Placa este proiectată pentru 0°C până la +75°C [24], fiind potrivită pentru aplicații din domeniile industriale sau automotive. Absența componentelor cu piese mobile (ventilatoare) crește fiabilitatea pe termen lung.

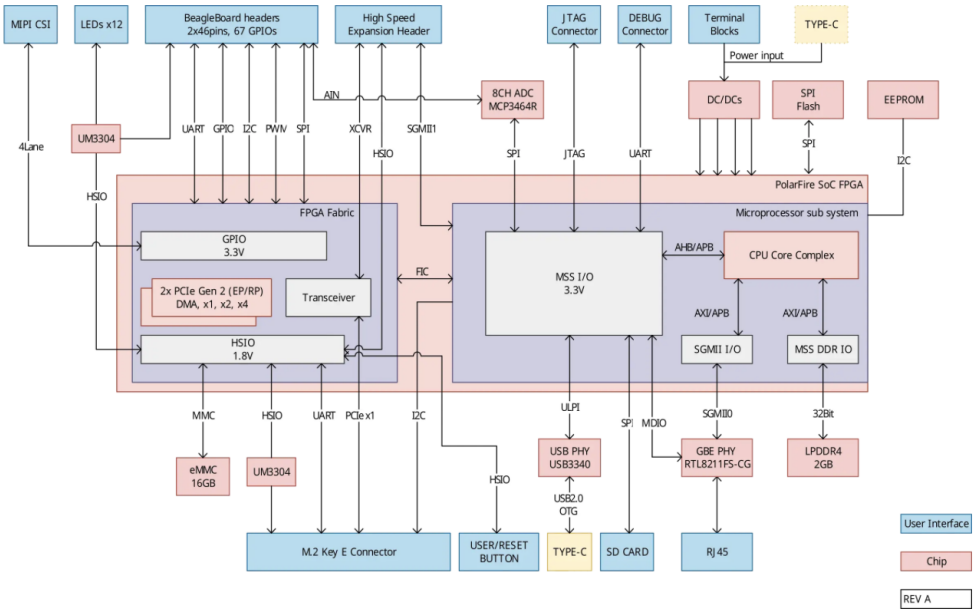


Figura 9: Diagrama bloc a plăcii BeagleV®-Fire [24]

Tabel 3: Conectorii plăcii BeagleV-Fire și funcțiile lor [24]

Conector	Descriere
USB Type-C	◘ Alimentare (5V/3A) și transfer date ◘ Suportă USB 2.0 și Power Delivery ◘ Interfață pentru debug (serial console)
P8/P9 Headers	◘ 92 de pini compatibili cu BeagleBone ◘ Include: 12 ADC, 4 UART, 2 I2C, SPI ◘ Suport pentru capes de expansiune
M.2 Key E	◘ Slot pentru module WiFi/Bluetooth ◘ Interfață PCIe x1 Gen2 (5Gbps) ◘ USB 2.0 pas-through
SYZYGY	◘ Interfață high-speed pentru FPGA ◘ 16 pini LVDS (1.2Gbps/pin) ◘ Alimentare 12V/3A
MIPI CSI-22	◘ Interfață pentru camere embedded ◘ Configurație 2-lane (2.5Gbps/lane) ◘ Compatibil cu senzori IMX219

3.1.2 Carcateristici tehnice

BeagleV®-Fire adoptă o arhitectură modernă bazată pe RISC-V, integrând 4+1 nuclee RV64GC (64-bit) ce suportă seturile de instrucțiuni IMAFDC — întreg, multiplicare, operații atomice, floating-point în dublă precizie și compresie. Frecvența nucleelor este configurabilă între 625 și 667 MHz [25], ceea ce oferă flexibilitate în raportul performanță-consum. Subsistemul de cache se remarcă printr-o ierarhie eficientă, cu 32KB pentru nivelul L1 de instrucțiuni și 32KB pentru nivelul L1 de date per nucleu, alături de un cache L2 partajat de 2MB, contribuind semnificativ la reducerea latenței și creșterea vitezei de execuție [24]. BeagleV®-Fire este echipat cu 2GB LPDDR4 funcționând la 1866MHz, oferind o lățime de bandă totală de 14.9 GB/s, suficientă pentru majoritatea aplicațiilor embedded avansate. În ceea ce privește stocarea non-volatilă, sunt disponibile 16GB eMMC 5.1 (cu viteză de până la 400 MB/s), dar și un slot microSD compatibil UHS-I (bootabil) și o memorie SPI Flash de 128MB utilizată pentru configurare și boot rapid [24].

3.2 Resurse FPGA disponibile

Placa BeagleV®-Fire este echipată cu SoC-ul Microchip PolarFire MPF025T, dispune de un FPGA de tip non-volatile, optimizat pentru aplicații embedded cu consum redus de energie. Acest FPGA oferă aproximativ 23.000 de logic elements (LEs) pentru implementarea circuitelor logice digitale, 1,8 Mbits de memorie RAM distribuită (disponibilă prin intermediul blocurilor de memorie integrată), precum și 390 de blocuri DSP utilizabile pentru operații matematice intensive, cum ar fi multiplicări paralele și procesare digitală a semnalului [25]. Aceste resurse permit dezvoltarea de sisteme complexe, precum acceleratoare hardware, filtre digitale sau controlere personalizate.

O componentă esențială a SoC-ului este procesorul RISC-V U54-MC de 64 de biți, cu 4 nuclee, care funcționează în paralel cu logica reconfigurabilă a FPGA-ului. Comunicarea între procesor și FPGA se realizează printr-o magistrală AXI (Advanced eXtensible Interface) internă, care permite schimbul eficient de date și control bidirecțional. Pentru accesul la perifericele lente sau controlul modulelor interne din FPGA, interfața AXI este convertită printr-un bridge AXI-APB (Advanced Peripheral Bus) [25]. Astfel, procesorul accesează perifericele implementate în logica FPGA prin interfața APB, care este mai simplă, sincronă și optimizată pentru consum redus și latență mică. FPGA-ul poate funcționa ca un coprocesor configurabil, accelerând anumite sarcini grele din punct de vedere computațional, în timp ce procesorul RISC-V gestionează logica de control și aplicația principală.

Tabel 4: Resurse FPGA disponibile ale SoC-ului MPFS025T [25]

Resurse	**MPFS025T**
Elemente Logice (4LUT + DFF)	23.000
Interfețe Logice	7.920
Blocuri LSRAM (20Kb / bloc)	84
Blocuri μSRAM (768 biți / bloc)	204
RAM Total	1.8 Mb
Blocuri Matematice (18x18 MACC)	68
μPROM (Kb)	194

4 CONFIGURAȚIA EXPERIMENTALĂ

Primul pas în dezvoltarea proiectului de conversie între spații de culoare este familiarizare cu placa de dezvoltare BeagleV®-Fire. Placa prezintă un design compact, cu un conector USB C utilizat atât pentru alimentare, cât și pentru comunicație serială. Astfel, s-a folosit un cablu USB C - USB A pentru a realiza conexiunea serială placă - laptop, respectiv alimentarea plăcii, plus terminalul Putty pentru a intermedia comunicarea între placă și laptop. În Device Manager s-a putut identifica noul port COM, folosit pentru a stabili conexiunea serială. Documentația oficială [24] precizează că rata de transmitere ce trebuie folosită este 115200 biți/s.

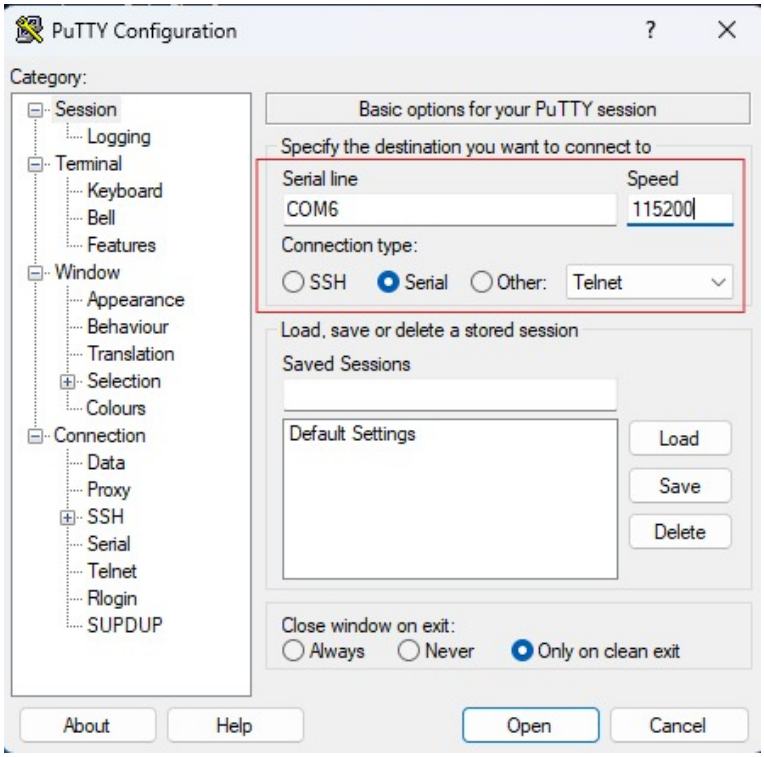


Figura 10: Conectare serială folosind Putty

Placa vine preinstalată cu o distribuție Ubuntu 20.04, ce utilizează inițial kernelul Linux 5.4. Această versiune, deși stabilă, este învechită și nu suportă actualizarea pachetelor. Tocmai de aceea este necesară actualizare către o nouă versiune a imaginii de Linux. Versiunea recomandată este Ubuntu 24.04, care utilizează kernelul Linux 6.6, care aduce suport extins pentru arhitectura RISC-V. Pentru acest pas s-a folosit programul Balena Etcher v1.18.12 [26] (aceasta este versiunea recomandată) pentru a scrie noua imagine pe eMMC. Pentru a realiza acest proces este necesar un modul USB-UART conectat la pinii de debug ai plăcii. Placa va comunica serial cu laptopul prin modulul USB-UART și va fi alimentată prin cablul USB C - USB A. După stabilirea conexiunii seriale, placa va intra în procesul de bootare și este necesară apăsarea oricărei taste pentru a-l opri și a putea accesa promptul HSS (Hart Software Services). Conținutul eMMC-ului plăcii este scris de HSS folosind comanda usbdmsc.

Această comandă expune eMMC-ul ca un dispozitiv de stocare de tip USB mass storage prin conectorul USB de tip C. În continuare se vor executa comenzile din Secțiunea de Cod 1.

```
>> mmc
>> usbdmsc
```

Secvență de Cod 1: Comenzi pentru expunerea eMMC-ului ca mass storage

În Balena Etcher se selectează opțiunea „Flash from file" și se slectează imaginea de Linux dorită, descărcată de pe repository-ul oficial [27].

În etapa următoare, se apasă pe „Select target", unde va apărea lista dispozitivelor de stocare disponibile. Se identifică și se selectează dispozitivul denumit MCC PolarFireSoC_msd, care reprezintă eMMC-ul plăcii BeagleV®-Fire. După confirmarea selecțiilor, se apasă butonul „Flash!" pentru a începe procesul de scriere a imaginii pe eMMC. Balena Etcher va efectua automat verificarea integrității imaginii după scriere, asigurându-se că procesul s-a realizat corect.

4.1 PROGRAMARE FPGA

Gateware-ul joacă un rol esențial în configurarea hardware-ului și definirea funcționalității FPGA-ului. Gateware-ul constituie interfața hardware între procesorul RISC-V și rețeaua de interfețe fizice ale plăcii.

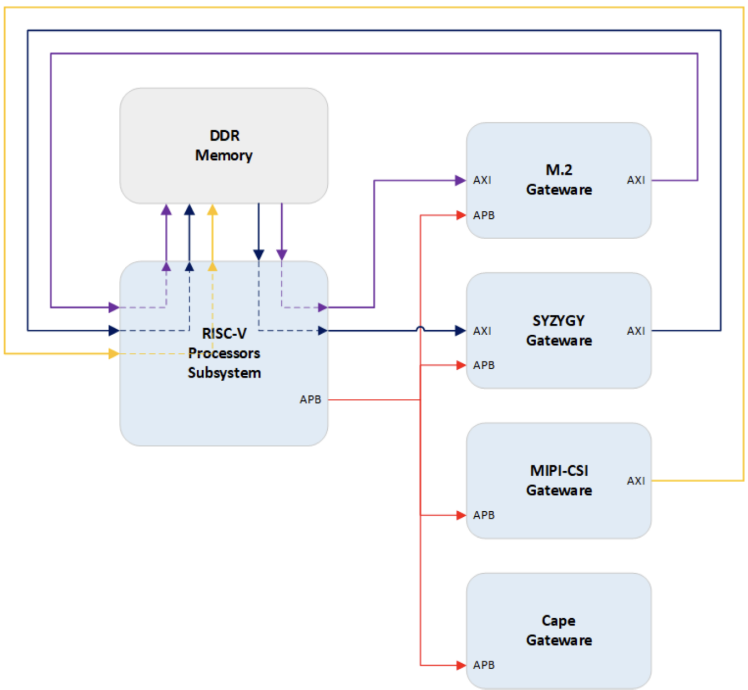


Figura 11: Structura gateware a plăcii BeagleV®-Fire [24]

Structura gateware-ului BeagleV-Fire este organizată în blocuri funcționale,vizibile în Figura 11, fiecare asociat unui conector specific al plăcii. Fiecare bloc dispune de o interfață AMBA APB, permițând

accesul software la registrele de control și status.

Pentru a asigura sincronizarea și funcționarea corectă a întregului sistem, toate aceste blocuri sunt sincronizate printr-un bloc centralizat denumit „Clock and Resets". Deși acest bloc nu este prezent în diagramă, rolul său este crucial în gestionarea semnalelor de ceas și reset pentru fiecare bloc funcțional. În cadrul acestui sistem, sunt disponibile două semnale de ceas: unul de 125 MHz și altul de 160 MHz, care sunt distribuite către blocurile gateware-ului conform necesităților fiecăruia.

Blocurile din structura gateware-ului sunt [24]:

- Cape Gateware: gestionează semnalele de pe conectorii P8 și P9. Aceasta este structura ce va fi modificată și personalizată în cadrul proiectului.
- SYZYGY Gateware: se ocupă de semnalele de pe conectorul high-speed SYZYGY. Acest conector include: până la trei transceivere capabile de comunicații la 12,7 Gbps, o interfață SGMII, 10 I/O-uri de mare viteză și intrări de ceas.
- MIPI-CSI Gateware: gestionează semnalele provenite de la interfața camerei.
- M.2 Gateware: implementează interfața PCIe utilizată pentru modulele Wi-Fi.
- Subsistemul de procesoare RISC-V: gestionează expunerea interfețelor de tip AMBA pentru ca celelalte blocuri de gateware să se poată atașa la acestea.

Componentele unui gateware sunt [24]:

- Design-ul digital pentru FPGA: cuprinde codul HDL/Verilog care definește logica aplicației personalizate, scriputri TCL care definesc configurarea blcurilor IP, scriputri TCL care realizează conexiunea între blocuri, lista de porturi ale MSS și constrângeri de timp, de pini și de plasare.
- Configurarea subsistemului de procesoare: Conține valori ale regiștrilor de configurare care stabilesc comportamentul subsistemului RISC-V.
- Bootloader de nivel zero (HSS): inițializează SoC-ul și gestionează tranziția către etapele ulterioare de boot.
- Device tree overlays: descriu maparea zonei de memorie utilizată de logica FPGA pentru sistemul de operare.

Procesul de dezvoltare a unui gateware, descris în Figura 12, cuprinde următorii pași:

- Scrierea codului HDL/Verilog corespunzător aplicației personalizate.

- Configurarea blocurilor IP folosind scripturi TCL.
- Stabilirea constrângerilor de pini, plasare și timp.
- Compilarea designului într-un fișier bitstream care poate fi programat pe FPGA.
- Generarea fișierelor care descriu noua logică FPGA pentru sistemul de operare (device tree overlays).
- Asigurarea că bootloader-ul HSS este configurat pentru a inițializa corect noul design.

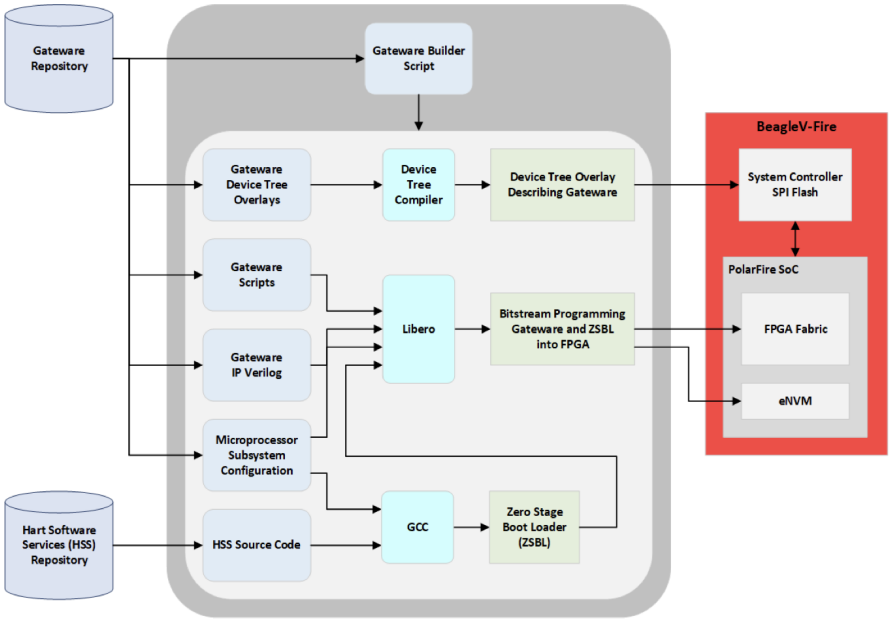


Figura 12: Procesul de generare și programare al gateware-ului [24]

Este esențial ca toate componentele gateware-ului să fie sincronizate pentru a asigura funcționarea corectă a sistemului. De aceea, se recomandă utilizarea unui sistem de construire a gateware-ului care gestionează toate aceste componente într-un mod coordonat.

Acest proces este facilitat de documentația oficială [24] care oferă un template pentru definirea propriului gateware. Pornind de la acesta, s-a dezvoltat un proiect personalizat care a putut fi încărcat pe platforma oficială de versionare, Gitlab la link-ul [28], unde se poate realiza procesul de build într-un container de Docker. Astfel generarea bitstream-ului este mult mai rapidă și nu depinde de resursele locale.

În urma procesului de build se produce un artefact ce conține atât log-urile de sinteză și plasare și rutare, cât și bitstream-ul cu care va fi programat FPGA-ul. Acest bitstream cuprinde bitsetram-ul cu care vor fi programate FPGA și eNVM (mpfs_bitstream.spi) și device tree overlay care descrie conținutul maparea zonelor de memorie utilizate (mpfs_dtbo.spi).

Acest bitstream se poate transfera pe placă folosind următoarea comanda din Secvența de Cod 2.

```
scp -r ./my_custom_fpgs_design beagle@192.168.7.2:/home/beagle/
```

Secvență de Cod 2: Transferul bitstream-ului de pe laptop pe placă

Iar apoi utilizând scriptul oficial de încărcare a gateware-ului, se programează FPGA-ul cu noul bitstream, folosind comanda din Secțiunea de Cod 3.

```
sudo /usr/share/beagleboard/gateware/change_gateware.sh ./
my_custom_fpga_design
```

Secvență de Cod 3: Programarea FPGA-ului cu nou bitstream

4.2 STRUCTURA DE DIRECTOARE A PROIECTULUI

Structura, evidențiată în Secvența de Cod 4, este modulară și reflectă separarea între partea de construcție hardware, configurație software, scripturi de generare automată și sursele efective ale designului.

```
+---bitstream             // contine metode de programare a FPGA-ului
|   +---FlashProExpress   // fisiere pentru utilizarea programatorului FlashPro
|   +---LinuxProgramming  // fisiere pentru programarea FPGA, eNVM și DTO
+---build-options         // fisiere de selectare a anumitor configuratii ale
    componentelor proiectului
+---debian                // contine diverse versiuni ale scripturilor de
    modificare si actualizarea a gateware-ului in functie de kernel
+---gateware_scripts      // contine script-uri de build pentru gateware
+---recipes               // contine scripturi pentru generarea proiectului Libero
|   +---libero-project
+---software              // contine fisierele C pentru comandarea conversiei,
    utilizate de proiectul curent
+---sources
|   +---FPGA-design
|   |   +---script_support
|   |       +---additional_configurations
|   |       +---components
|   |       |   +---BVF_RISCV_SUBSYSTEM
|   |       |   +---CAPE
|   |       |   |   +---DEFAULT
|   |       |   |   +---YUV422_RGB     // CAPE persoanlizat pentru operatia de
    conversie
|   |       |   |   |   +---constraints // contine fisierul de constrangeri
|   |       |   |   |   +---device-tree-overlay  // maparea zonelor de memorie
|   |       |   |   |   +---HDL  // fisierele Verilog cu implementarea conversiei
|   |       |   +---CLOCKS_AND_RESETS  // logica de generare/resetare a ceasurilor
     sistemului
|   |       |   +---M2
|   |       |   +---MIPI_CSI
|   |       |   +---MSS
```

```
|   |       |   +---SYZYGY
|   |       +---HDL // surse HDL implicite pentru arbitrare APB, module IP pentru conectivitate si testare
|   +---HSS
|   +---MSS_Configuration
```

Secvență de Cod 4: Structura de directoare a poriectului

Elementele cheie ale proiectului sunt următoarele:

- my_custom_fpga_design.yaml - servește drept punct central de configurare pentru generarea unui gateware personalizat, permițând integrarea componentelor hardware alese și specificând sursa codului HSS (Hart Software Services). Structura sa modulară reflectă două secțiuni principale: HSS și gateware, evidențiate în Secvența de Code 5 și Secvența de Cod 6.

```
HSS:
  type: git
  link: https://openbeagle.org/beaglev-fire/hart-software-services.git
  branch: main-beaglev-fire
  board: bvf
```

Secvență de Cod 5: Secțiunea HSS din fișierul de configurare

Secvența de Code 5 specifică sursa codului pentru Hart Software Services.

```
gateware:
  type: sources
  build-args: "M2_OPTION:NONE CAPE_OPTION:YUV422_RGB"
```

Secvență de Cod 6: Secțiunea gateware din fișierul de configurare

Secvența de Cod 6 specifică faptul că sursele HDL/Verilog și scripturile de configurare vor fi preluate local, din directorul sources/. De asemenea, sunt specificate argumentele de build care permit selecția opțională a componentelor active în designul hardware. În proiectul de față se exclude interfața M.2 PCIe (utilizată pentru module Wi-Fi), ceea ce eliberează transceiverele FPGA și se activează suportul pentru cape-ul care convertește imaginile YUV422 în RGB, ceea ce implică logică dedicată și suport în device tree.

- Fiecare subdirector din directorul CAPE (componentă) definește o configurație hardware specifică pentru un cape compatibil. Aceste denumiri de directoare funcționează ca identificatori ai opțiunilor și sunt transmise automat scriptului principal de build Libero – BUILD_BVF_GATEWARE.tcl – sub forma argumentelor de build. De asemenea, aceste opțiuni sunt preluate și din fișierele .yaml utilizate de sistemul de build al bitstream-ului, cum este și cazul fișierului my_custom_fpga_design.yaml.

Gateware-ul poate fi personalizat sau extins foarte ușor. Pentru a adăuga o nouă configurație hardware, este suficientă crearea unui nou director în interiorul componentei CAPE, spre exemplu noua componentă a proiectului YUV422_RGB.

- Fiecare opțiune pentru componente conține subdirectoare pentru: constrângeri, DTO și fișiere sursă HDL. În plus, conține scripturi TCL utilizate de mediul de dezvoltare Libero SoC Design Suite pentru a construi și integra logica digitală specifică în proiectul complet. Fiecare opțiune de componentă include un fișier TCL numit convențional ADD_<NUME_COMPONENTĂ>.tcl, de exemplu ADD_CAPE.tcl. Acesta este un script scris manual care definește cum componenta respectivă este integrată în designul global al FPGA-ului și asigură legătura logică între modulele funcționale.

- Fișierul DTO traduce configurația hardware din FPGA într-o structură recunoscută de kernelul Linux, permițând utilizarea efectivă a componentelor adăugate în designul digital.

```
&{/chosen} {
    overlays {
        YUV422_RGB-GATEWARE = "GATEWARE_GIT_VERSION";
    };
};
```

Secvență de Cod 7: Adăugarea unui nou overlay la Device Tree

În Secvența de Cod 7 se adaugă o nouă informație la nodul chosen prin care se specifică că în această sesiune de boot este activ un gateware cu funcționalitate specifică (YUV422_RGB).

```
&{/fabric-bus@40000000} {
        #address-cells = <2>;
        #size-cells = <2>;

        cape_gpios_p8: gpio@41100000 {
                compatible = "microchip,core-gpio", "
    microchip,coregpio-rtl-v3";
                reg = <0x0 0x41100000 0x0 0x1000>;
                clocks = <&fabric_clk3>;
                interrupt-parent = <&plic>;
                gpio-controller;
                #gpio-cells = <2>;
                ngpios = <16>;
                status = "okay";
                interrupts = <129>, <130>, <131>, <132>,
                             <133>, <134>, <135>, <136>,
                             <137>, <138>, <139>, <140>,
                             <141>, <142>, <143>, <144>;

```

```
19                  gpio-line-names = "P8_31", "P8_32", "", "P8_34",
20                                    "", "P8_36", "P8_37", "P8_38",
21                                    "P8_39", "P8_40", "P8_41", "P8_42",
22                                    "P8_43", "P8_44", "P8_45", "P8_46";
23          };
24 };
```

Secvență de Cod 8: Adăugarea unui nou overlay la Device Tree

În Secvența de Cod 8 se definește un nod pentru un nodul GPIO personalizat, implementat în logica FPGA, mapat la adresa 0x41100000, care reprezintă baza, iar dimensiunea este dimensiunea 0x1000 (4KB). GPIO-ul folosește ceasul fabric_clk3 și poate genera întreruperi, iar acestea sunt gestionate prin controlerul de întreruperi PLIC (Platform-Level Interrupt Controller). Fiecare pin are asociat un număr de întrerupere.

- Fișierele HDL ce conțin design-ul digital.

4.3 ModelSim și Libero SoC Design Suite

În cadrul proiectului sunt utilizate două aplicații principale - ModelSim și Libero SoC Design Suite.

ModelSim [29] este un simulator logic utilizat pentru verificarea funcțională a designurilor HDL (scrise în VHDL sau Verilog/SystemVerilog). În contextul proiectelor FPGA este folosit pentru a simula comportamentul modulelor hardware înainte de implementarea fizică. Acesta permite scrierea și rularea de testbench-uri pentru validarea funcțională și oferă o interfață grafică (waveform viewer) pentru urmărirea evoluției semnalelor în timp.

Libero SoC Design Suite [30] este suita oficială de dezvoltare oferită de Microchip pentru proiectarea și implementarea de sisteme pe cip (SoC) care integrează atât logica programabilă (FPGA), cât și componente software. Aceasta oferă un mediu unificat pentru: sinteză logică (synthesis), plasare și rutare (place & route), generarea de imagini de configurare FPGA (bitstream), integrarea de blocuri IP predefinite, proiectarea și integrarea de microcontrolere soft și instrumente de analiză și constrângeri temporale. Libero SoC include și suport pentru configurarea interfețelor AXI/APB și pentru generarea automată de scripturi și fișiere necesare pentru integrarea hardware-software, fiind optimizat pentru familia PolarFire SoC [30].

5 ARHITECTURA PROIECTULUI

În cadrul proiectului de accelerare a conversiei YUV-RGB s-au realizat mai multe implementări, atât în versiunea în care conversia este realizată doar de procesor, cât și în versiunea în care conversia este realizată de FPGA. Astfel, în continuare este prezentată arhitectura a două versiuni de implementare a conversiei pe FPGA:

- una în care procesorul furnizează câte 4 pixeli către FPGA pentru conversie și îi citește imediat după ce sunt convertiți, stocându-i într-un buffer, iar, în final, când întreaga imagine este convertită, copiază conținutul buffer-ului într-un fișier de ieșire.
- cealaltă în care conversia este controlată prin intermediul unui DMA Controller pentru a reduce semnificativ timpul de folosire al procesorului. Această implementare presupune stocarea întregii imagini de intrare într-o o zonă mapată de memorie și transmiterea unui semnal de inițiere a conversiei, respectiv semnalele de configurare, de la procesor către DMA Controller pentru a-l anunța faptul că se poate începe conversia. În momentul în care primește startul, DMA Controller începe să citească valorile din memorie și să le transmită în rafală către modulul de conversie. Pe măsură ce datele sunt primite de FPGA, ele sunt convertite și se scrie rezultatul într-o altă zonă de memorie. În final, se va transmite o întrerupere procesorului pentru a-l anunța că poate citi imaginea convertită din memorie.

5.1 DIAGRAMA BLOC A SISTEMULUI

5.1.1 Implementara conversiei fără DMA

Această variantă de implementare presupune instanțierea modulului yuv422_to_rgb în modulul superior YUV_TO_RGB_CONVERTER având funcția de wrapper în sistem. Modulul yuv422_to_rgb prezintă o interfață tipică a protocolului APB în care se transmit date pe 32 de biți, împreună cu un semnal de întrerupere irq, folosit pentru semnalizarea finalizării conversiei a 4 pixeli. S-a ales conversia a 4 pixeli simultan pentru a evidenția subeșantionarea specifică YUV422 evidențiată în Figura 5. În acest modul sunt prezente cele 4 instanțe ale modulului de conversie yuv_to_rgb care primesc semnalul de începere a conversiei și la finalizarea acesteia transmit înapoi valorile RGB obținute, respectiv o semnalizare de finalizare. Deși semnalul irq poate fi conectat la o întrerupere a MSS pentru a alerta procesorul de fianlizarea conversiei, pentru această implementare s-a ales identificare finalizării prin polling al stării FSM-ului ce controlează conversia. Figura 13 înfățișează reprezentarea grafică a schemei bloc a sistemului. Blocurile MSS, APB_ARBITER și FICO_INITIATOR [31] sunt structuri preluate din catalogul oferit de Microchip.

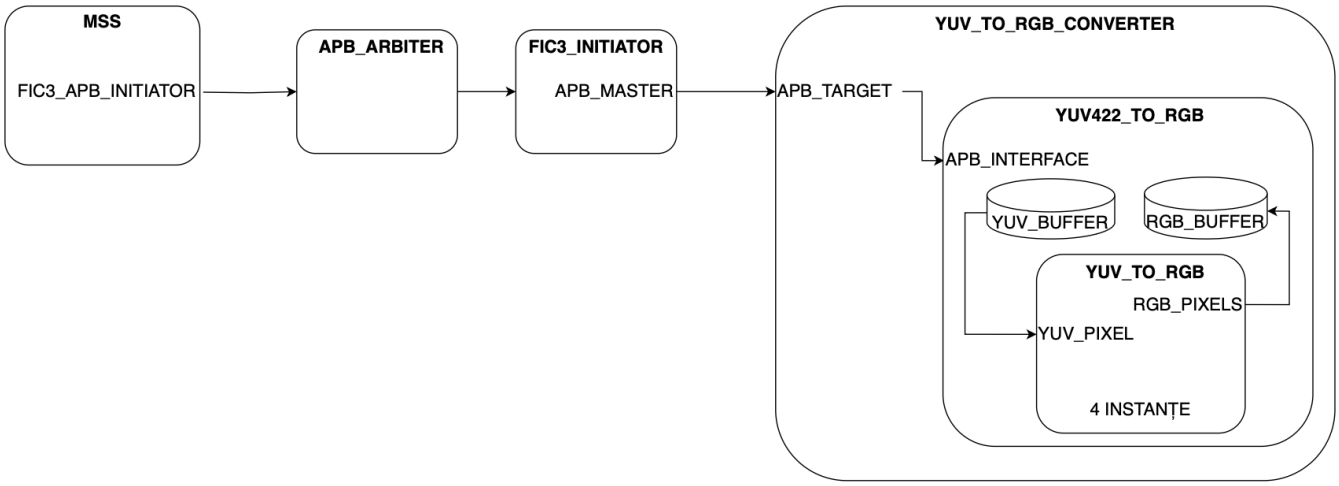


Figura 13: Diagrama bloc a sistemului de conversie - versiune fără DMA

5.1.2 Implementara conversiei cu DMA

Implementarea conversiei spațiilor de culoare folosind un controller DMA are o arhitectură complexă, evidențiată în Figura 14, și presupune folosirea a două controller-e pentru gestionarea magistralelor de comunicare cu structura FPGA (FIC - Fabric Interface Controller):

- FIC0 este controller-ul pentru una dintre cele două magistrale de comunicație de tip AXI4 pe 64 de biți, care realizează legătura directă între MSS și logica programabilă din FPGA [31]. Aceasta poate fi accesată atât de MSS, cât și de FPGA în calitate de master, ceea ce permite un model de comunicație flexibil și bidirecțional, perfect pentru scrierea, respectiv citirea datelor din memoria LSRAM. Un mare plus îl aduce frecventa cu care această magistrală operează, putând atinge o valoare de $\frac{F_{\text{CPUCLK}}}{2}$[32], în cazul de față folosindu-se frecvența de 250 MHz. De asemenea, în cadrul implementării se beneficiază și de avantajele oferit de protocolul AXI4 - latență redusă și suport pentru transferuri în rafală (burst) [33].

- FIC3 este controller-ul pentru magistrala de tip APB pe 32 de biți, destinată comunicației între MSS și FPGA. Acesta operează ca o interfață unidirecțională, în care MSS acționează ca master, iar modulele implementate pe FPGA sunt slave [31]. Interfața APB este utilizată pentru controlul și configurarea conversiei, permițând procesorului să transmită către DMA Controller adresa de bază a zonei de memorie unde este stocată imaginea de intrare (format YUV), adresa de bază a zonei de memorie unde trebuie stocată imaginea de ieșire (format RGB), dimensiunile imaginii YUV - WIDTH și HEIGHT, respectiv un semnal de start pentru a iniția conversia. Spre deosebire de AXI4, APB este o magistrală simplificată, fără suport pentru acces burst sau pipelining, dar este mai eficientă în ceea ce privește consumul de resurse.

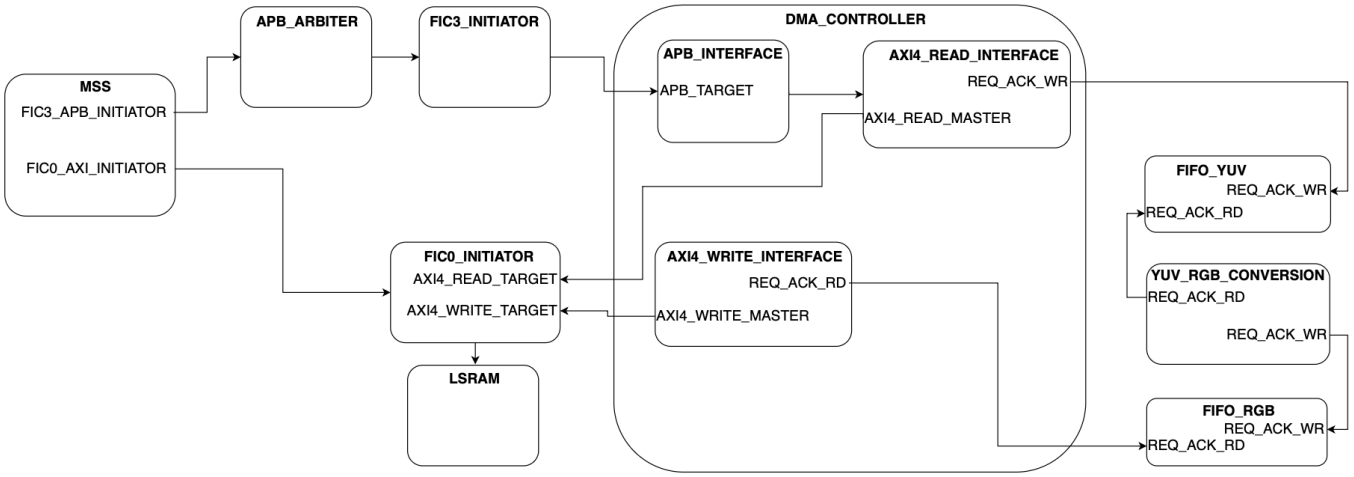


Figura 14: Diagrama bloc a sistemului de conversie - versiune cu DMA

Blocurile MSS, APB_ARBITER, FIC0_INITIATOR, FIC3_INITIATOR [31] sunt structuri preluate din catalogul oferit de Microchip. În schimb, modulele din cadrul structurii DMA_CONTROLLER sunt elemente proprii ale proiectului de față și au fost implementate în mod distinctiv și specific acestuia.

Modulul APB_INTERFACE prezintă o interfață APB prin care procesorul transmite structurii DMA semnalele de control și gestiune a conversiei prin 4 transferuri succesive de scriere. Primele 3 transferuri pot avea o ordine aleatorie (se pot transfera oricare dintre valorile: dresa de bază a zonei de memorie unde este stocată imaginea de intrare (format YUV), adresa de bază a zonei de memorie unde trebuie stocată imaginea de ieșire (format RGB), dimensiunile imaginii YUV - WIDTH și HEIGHT), însă ultimul transfer trebuie neapărat să fie cel de start a conversiei. În cazul în care nu se respectă această cerință, FSM ce gestionează stările transferurilor va intra în starea de eroare, iar pe interfața APB se va seta semnalul pslverr. Ca urmare, procesul de transfer a informațiilor se va relua de la început. Odată primite informațiile, se calculează numărul de octeți necesari imaginii YUV, respectiv RGB, iar împreună cu semnalul de start se transmit către modulul AXI_READ_INTERFACE care va iniția tranzacții în burst de la adresa de bază a imaginii YUV, fiecare rafală având 256 de tranșe a câte 64 de biți (8 octeți $\Rightarrow$4 pixeli) până se va atinge numărul necesar de octeți. Numărul tranșelor este ajustat în cazul ultimei tranzacții în cazul în care numărul octeților ce trebuie citiți este mai mic de 256 * 8 = 2048. S-au ales tranșe de câte 64 de biți pentru a utiliza la maxim magistrala AXI4 ce poate transfera date de 64 de biți. Pe măsură ce datele sunt citite din memorie, acestea se înscriu într-un buffer ce funcționează pe principiul first-in-first-out și fiind gestionat prin doi pointeri, unul de scriere, respectiv unul de citire. În acest buffer pot fi stocate până la 256 de date de 64 de biți. Buffer-ul este folosit ca un mecanism de protecție împotriva desincronizărilor sau întârzierilor ce pot apărea în modulul de conversie. Pe măsură ce datele sunt scrie în FIFO_YUV, aceste sunt preluate de către modulul de conversie printr-un protocol request-acknowledge, apoi sunt convertite și transmise mai

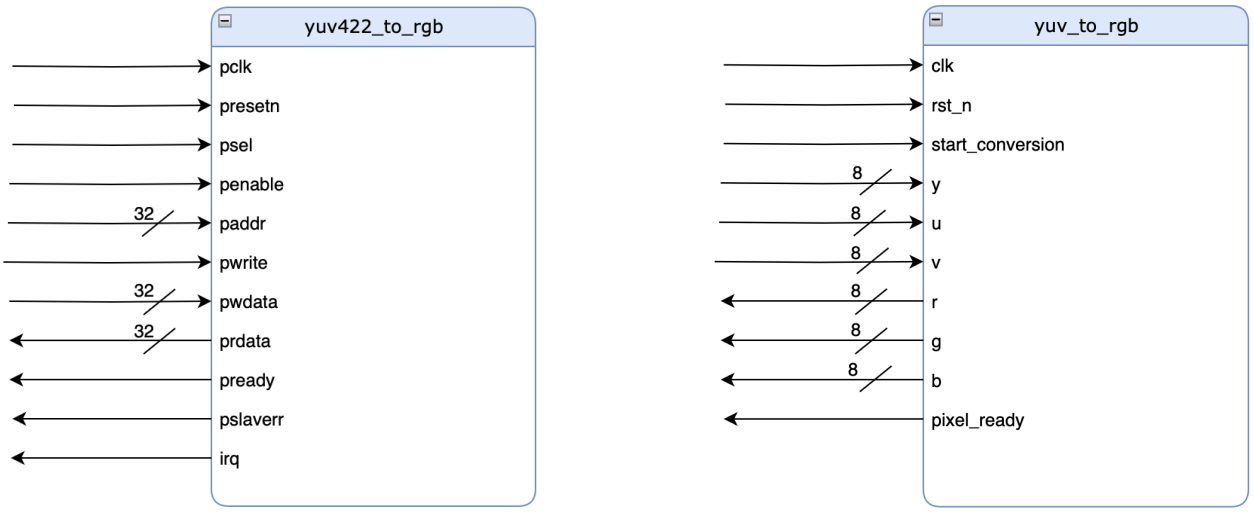
departe în buffer-ul de FPGA_RGB de unde sunt preluate de modulul AXI_WRITE_INTERFACE, care va realiza tranzacții în rafală de scriere în memorie.

5.2 INTERFEȚE

Pentru înțelegerea mai detaliată a implementării, în continuare sunt prezentate interfețele modulelor de comunicare cu procesorul, folosind protocolul APB, respectiv de conversie.

5.2.1 Interfețele conversiei fără DMA Controller

Interfața modulului yuv422_to_rgb este vizualizată grafic în Figura 15a și detaliată în cadrul Tabelului 5. Interfața modulului yuv_to_rgb este vizualizată grafic în Figura 15b și detaliată în cadrul Tabelului 6.



(a) Interfața modulului yuv422_to_rgb

(b) Interfața modulului yuv_to_rgb

Figura 15: Interfețele modulelor sistemului de conversie fără DMA

Tabel 5: Semnalele interfeței APB pentru modulul `yuv422_to_rgb`

Denumire semnal	Direcție	Width	Descriere
pclk	INPUT	1	Ceas APB, frecvență de 125MHz
presetn	INPUT	1	Reset asincron APB, activ pe 0 logic
psel	INPUT	1	Semnal de selecție a slave-ului
penable	INPUT	1	Semnal de control pentru validarea transferului
paddr	INPUT	32	Adresa de memorie la care se realizează operația
pwrite	INPUT	1	Semnal ce indică operația care se realizează: 1 - scriere; 0 - citire
pwdata	INPUT	32	Data transmisă de procesor, cuprinde câte un byte pentru fiecare din valorile Y0, U0, Y1, V1
prdata	OUTPUT	32	Data transmisă de FPGA după conversie, cuprinde câte un byte pentru fiecare din valorile R, G, B + un byte cu valoarea 0
pready	OUTPUT	1	Semnal de indicare a acceptului tranzacției. În implementarea actuală este pus în 1 tot timpul.
pslaverr	OUTPUT	1	Semnal sub formă de puls care este setat în cazul apariției unei erori. În implementarea actuală este pus în 0 tot timpul.
irq	OUTPUT	1	Puls de întrerupere transmis către procesor la finalizarea conversiei a 4 pixeli.

Tabel 6: Semnalele modulului de conversie `yuv_to_rgb`

Denumire semnal	Direcție	Width	Descriere
clk	INPUT	1	Ceas APB, frecvență de 125MHz
rst_n	INPUT	1	Reset asincron APB, activ pe 0 logic
start_conversion	INPUT	1	Puls pentru a indica faptul că datele transmise drept input sunt stabile și se poate realiza conversia
y	INPUT	8	Componenta de luminanță a unui pixel
u	INPUT	8	Componenta de crominanță a unui pixel
v	INPUT	8	Componenta de crominanță a unui pixel
r	OUTPUT	8	Componenta roșie a unui pixel
g	OUTPUT	8	Componenta verde a unui pixel
b	OUTPUT	8	Componenta albastră a unui pixel
pixel_ready	OUTPUT	1	Puls pentru a indica faptul că valorile convertite pentru un pixel sunt stabile și gata de citire

5.2.2 Interfețele conversiei cu DMA Controller

Interfețele modulelor prezentate în Figura 14 sunt evidențiate în Tabelele 7, 8, 9, 10, 11, ce prezintă atât direcția acestora, dimensiunea în biți, cât și o descriere cu rolul lor în modulul respectiv.

Tabel 7: Semnalele modulului `APB_INTERFACE`

Denumire semnal	Direcție	Width	Descriere
pclk	INPUT	1	Semnal de ceas pentru magistrala APB
presetn	INPUT	1	Reset asincron, activ pe 0 logic
paddr	INPUT	32	Adresa utilizată pentru citiri/scrieri pe magistrala APB
pwdata	INPUT	32	Datele de intrare (scriere) pe magistrala APB
prdata	OUTPUT	32	Datele de ieșire (citire) de pe magistrala APB
pwrite	INPUT	1	Semnal care indică dacă transferul este de scriere (1) sau citire (0)
psel	INPUT	1	Semnal de selecție a slave-ului APB
penable	INPUT	1	Semnal care indică momentul în care transferul APB este activ
pslverr	OUTPUT	1	Semnal care indică o eroare de comunicație APB
pready	OUTPUT	1	Semnal de handshake – indică dacă slave-ul este pregătit pentru transfer
dma_base_addr_yuv	OUTPUT	32	Adresa de start din memorie pentru citirea imaginii YUV
dma_base_addr_rgb	OUTPUT	32	Adresa de start din memorie pentru scrierea imaginii convertite RGB
dma_transfer_len_yuv	OUTPUT	32	Numărul total de octeți ce trebuie transferați pentru imaginea YUV
dma_transfer_len_rgb	OUTPUT	32	Numărul total de octeți ce trebuie transferați pentru imaginea RGB
dma_start	OUTPUT	1	Semnal de control care inițiază transferul DMA
dma_busy	INPUT	1	Semnal care indică faptul că DMA-ul este ocupat și transferul este în desfășurare

Tabel 8: Semnalele modulului `AXI4_WRITE_INTERFACE`

Denumire semnal	Direcție	Width	Descriere
Control de la `APB_INTERFACE`			
base_addr_rgb	INPUT	32	Adresa de bază în memorie unde se vor scrie datele RGB
transfer_len_rgb	INPUT	32	Lungimea totală a transferului exprimată în octeți (width * height * 3)
start_write_rgb	INPUT	1	Semnal care inițiază operația de scriere AXI
full_image_written_done	OUTPUT	1	Semnal care indică încheierea completă a scrierii datelor RGB în memorie
Interfață AXI4 – Canal de adresă de scriere			
M_AXI_ACLK	INPUT	1	Ceasul sistemului AXI4
M_AXI_ARESETn	INPUT	1	Reset activ pe 0 logic pentru magistrala AXI
M_AXI_AWADDR	OUTPUT	32	Adresa de scriere generată de master AXI
M_AXI_AWLEN	OUTPUT	8	Numărul de transferuri într-un burst
M_AXI_AWSIZE	OUTPUT	3	Dimensiunea fiecărui transfer (în biți)
M_AXI_AWBURST	OUTPUT	2	Tipul burstului (INCR - incremental)
M_AXI_AWVALID	OUTPUT	1	Semnal care indică validitatea adresei de scriere
M_AXI_AWREADY	INPUT	1	Slave-ul este pregătit să accepte adresa de scriere
Interfață AXI4 – Canal de date de scriere			
M_AXI_WDATA	OUTPUT	64	Datele ce vor fi scrise către memoria principală
M_AXI_WVALID	OUTPUT	1	Semnal care indică validitatea datelor
M_AXI_WREADY	INPUT	1	Slave-ul este pregătit să accepte datele de scriere
M_AXI_WLAST	OUTPUT	1	Semnal care marchează finalul burstului de date
Interfață AXI4 – Canal de răspuns la scriere			
M_AXI_BRESP	INPUT	2	Răspunsul returnat de slave (OKAY, ERROR etc.)
M_AXI_BVALID	INPUT	1	Semnal care indică faptul că răspunsul este valid
M_AXI_BREADY	OUTPUT	1	Semnal care indică faptul că masterul acceptă răspunsul
Interfață cu FIFO			
rgb_pixels	INPUT	64	Datele convertite RGB preluate din FIFO
read_req	OUTPUT	1	Cerere de citire a unui nou cuvânt din FIFO
read_ack	INPUT	1	Confirmarea primirii cererii de citire
fifo_full	INPUT	1	Indică faptul că FIFO este plin
fifo_empty	INPUT	1	Indică faptul că FIFO este gol

Tabel 9: Semnalele modulului `AXI4_READ_INTERFACE`

Denumire semnal	Direcție	Width	Descriere
Control de la `APB_INTERFACE`			
base_addr_yuv	INPUT	32	Adresa de bază în memorie pentru imaginea YUV ce urmează a fi citită
transfer_len_yuv	INPUT	64	Lungimea totală a transferului, exprimată în octeți (width * height * 2)
start_read	INPUT	1	Semnal care inițiază transferul AXI de citire
full_image_read_done	OUTPUT	1	Semnal care indică finalizarea transferului complet
start_conversion	OUTPUT	1	Semnal care inițiază conversia YUV–RGB după citire
Interfață AXI4 – Canal de adresă de citire			
M_AXI_ACLK	INPUT	1	Ceasul sistemului AXI4
M_AXI_ARESETn	INPUT	1	Reset activ pe 0 logic pentru magistrala AXI
M_AXI_ARADDR	OUTPUT	32	Adresa de început a tranzacției de citire
M_AXI_ARLEN	OUTPUT	8	Numărul de transferuri într-un burst
M_AXI_ARSIZE	OUTPUT	3	Dimensiunea fiecărui transfer
M_AXI_ARBURST	OUTPUT	2	Tipul burstului (INCR)
M_AXI_ARVALID	OUTPUT	1	Semnal care validează cererea de citire
M_AXI_ARREADY	INPUT	1	Semnal de la slave care confirmă acceptarea adresei
Interfață AXI4 – Canal de date de citire			
M_AXI_RDATA	INPUT	64	Datele citite de la slave
M_AXI_RVALID	INPUT	1	Semnal care indică validitatea datelor
M_AXI_RREADY	OUTPUT	1	Semnal care indică disponibilitatea masterului de a prelua date
M_AXI_RLAST	INPUT	1	Semnal care indică finalul burstului curent
M_AXI_RRESP	INPUT	2	Răspunsul returnat de slave (OKAY, ERROR etc.)
Interfață cu FIFO			
yuyv_4pixels	OUTPUT	64	Grup de 4 pixeli YUYV citiți din memorie
write_req	OUTPUT	1	Semnal de cerere pentru scriere în FIFO
write_ack	INPUT	1	Confirmarea că FIFO acceptă datele
fifo_full	INPUT	1	FIFO este plin – nu se mai pot scrie date
fifo_empty	INPUT	1	FIFO este gol – nu sunt date disponibile

Tabel 10: Semnalele modulului `FIFO`

Denumire semnal	Direcție	Width	Descriere
clk	INPUT	1	Semnal de ceas al modulului FIFO
resetn	INPUT	1	Reset asincron activ pe 0 logic
data_in	INPUT	64	Datele care trebuie scrise în FIFO
read_req	INPUT	1	Cerere de citire a datelor din FIFO
write_req	INPUT	1	Cerere de scriere a datelor în FIFO
read_ack	OUTPUT	1	Confirmare că o operație de citire a avut loc
write_ack	OUTPUT	1	Confirmare că o operație de scriere a avut loc
data_out	OUTPUT	64	Datele citite din FIFO
empty	OUTPUT	1	Semnal care indică faptul că FIFO-ul este gol
full	OUTPUT	1	Semnal care indică faptul că FIFO-ul este plin

Tabel 11: Semnalele modulului `YUV_TO_RGB_CONVERSION`

Denumire semnal	Direcție	Width	Descriere
clk	INPUT	1	Semnal de ceas utilizat pentru sincronizarea tuturor operațiilor modulului.
resetn	INPUT	1	Reset asincron, activ pe 0 logic.
start_conversion	INPUT	1	Puls de activare care inițiază conversia YUV–RGB la fiecare scriere în FIFO-ul de intrare.
Interfață cu FIFO RGB			
rgb_pixels	OUTPUT	64	4 pixeli convertiți în format RGB888 (24 biți/pixel) concatenați intercalat.
fifo_full_rgb	INPUT	1	Semnal ce indică faptul că FIFO-ul RGB este plin.
fifo_empty_rgb	INPUT	1	Semnal ce indică faptul că FIFO-ul RGB este gol.
write_req_rgb	OUTPUT	1	Cerere de scriere a datelor convertite în FIFO-ul RGB.
write_ack_rgb	INPUT	1	Confirmare că datele au fost scrise în FIFO-ul RGB.
Interfață cu FIFO YUV			
yuyv_4pixels	INPUT	64	4 pixeli YUYV (YUV 4:2:2) citiți din FIFO-ul de intrare.
fifo_full_yuv	INPUT	1	Semnal ce indică faptul că FIFO-ul YUV este plin.
fifo_empty_yuv	INPUT	1	Semnal ce indică faptul că FIFO-ul YUV este gol.
read_req_yuv	OUTPUT	1	Cerere de citire a unui cuvânt (64 biți) din FIFO-ul YUV.
read_ack_yuv	INPUT	1	Confirmare a citirii datelor din FIFO-ul YUV.

6 IMPLEMENTARE

Conversia datelor video din format YUV422 în RGB este un proces esențial în sistemele de procesare a imaginilor, având aplicații variate, de la afișarea video pe dispozitive standard până la prelucrarea fluxurilor în sisteme incorporate. Performanța acestui proces devine critică atunci când vorbim de imagini în timp real, unde latența și eficiența energetică pot face diferența între o soluție viabilă și una nepractică. Obiectivul acestui capitol este de a descrie și, ulterior, analiza și compara execuția conversiei YUV422-RGB pe două platforme complementare: un nucleu de procesare software cu o arhitectură RISC-V și o implementare hardware pe structura FPGA a sistemului integrat pe chip Microchip PolarFire MPFS025T, cu scopul final de a evalua potențialul ambelor abordări pentru procesarea video în timp real.

Dezvoltarea și testarea modulului de conversie au fost realizate utilizând imagini statice de dimensiuni variate, pornind de la fișiere YUV422 cu rezoluția mică de 176x144 pixeli. Această rezoluție permite o evaluare controlată a performanței, simplificând experimentele și facilitând măsurarea precisă a timpilor de execuție. Experimentele se desfășoară pe placa BeagleV®-Fire, care integrează cinci nuclee RISC-V și o zonă FPGA. Implementarea software rulează pe nucleele RISC-V, fiind dezvoltată în limbajul C și optimizată prin paralelizare cu directiva #pragma omp parallel for. Aceasta permite exploatarea celor patru nuclee U54 disponibile pentru procesarea simultană a grupurilor de pixeli, îmbunătățind eficiența față de o abordare strict secvențială. Implementarea hardware, în schimb, utilizează structura FPGA al aceleiași plăci, fiind proiectată în Verilog pentru a exploata paralelismul intrinsec al arhitecturii FPGA.

În această etapă inițială, sunt comparați timpii de execuție obținuți pentru conversia imaginilor de 176x144 pixeli pe cele două platforme, stabilind o bază de referință pentru analiza diferențelor dintre procesarea software paralelă și cea hardware. Rezultatele obținute cu aceste imagini de dimensiuni reduse vor fi extinse ulterior către fișiere YUV de rezoluții mai mari. Prin această abordare incrementală, se urmărește evidențierea avantajelor specifice fiecărei metode – capacitatea de paralelizare software a nucleelor RV64GC și eficiența energetică asociată, respectiv viteza superioară și paralelismul masiv al FPGA-ului – și fundamentarea optimizărilor viitoare pentru aplicații embeded.

6.1 DEZVOLTAREA MODULULUI PENTRU CONVERSIE - VARIANTA FĂRĂ DMA

Implementarea hardware a conversiei YUV422-RGB se bazează pe două module principale: yuv_to_rgb, care realizează conversia unui pixel individual conform standardului ITU-T T.871 (ITU-R BT.601) [22], și yuv422_to_rgb, care coordonează conversia simultană a patru pixeli utilizând o interfață APB (Advanced Peripheral Bus) pentru comunicarea cu nucleele RISC-V RV64GC (U54). Standardul ITU-T T.871 [22] a fost ales pentru definirea coeficienților de conversie, asigurând compatibilitatea cu specificațiile video larg utilizate și o precizie ridicată a culorilor rezultate.

6.1.1 Modulul yuv_to_rgb

Modulul yuv_to_rgb este responsabil pentru conversia unui singur pixel din spațiul YUV422 în spațiul RGB, utilizând ecuațiile standardizate din ITU-T T.871. Intrările sunt reprezentate pe 8 biți fiecare (y, u, v), iar ieșirile (r, g, b) sunt, de asemenea, limitate la 8 biți, corespunzând intervalului 0-255. Implementarea folosește aritmetică cu virgulă fixă pentru a optimiza performanța pe FPGA și a minimiza utilizarea resurselor.

Standardul ITU-T T.871 [22] definește componentele crominanței (U și V) în intervalul 0-255, cu o valoare centrală de 128 corespunzând absenței culorii (zero crominanță). Pentru a aplica corect ecuațiile de conversie, este necesară translația acestor valori într-un interval simetric în jurul lui zero (-128 la 127). Astfel, din fiecare valoare u și v se scade 128, operație realizată prin definirea semnalelor u_s și v_s pe 16 biți. Scăderea lui 128 aliniază valorile U și V cu forma ecuațiilor BT.601, unde termenii (U – 128) și (V – 128) sunt utilizați direct. Componenta luminozității y nu necesită offset, deoarece este deja definită în intervalul 0-255 ca o valoare absolută, și este extinsă la 16 biți ca y_s pentru consistență în calcule. Alegerea a 16 biți pentru aceste semnale permite stocarea valorilor negative rezultate din offset și oferă spațiu pentru calculele ulterioare fără pierderi premature de precizie. Operațiile de aliniere a valorilor inițiale la standardul precizat sunt realizate în Secvența de Cod 9.

```
1 wire signed [15:0] y_s = y;
2 wire signed [15:0] u_s = u - 128;
3 wire signed [15:0] v_s = v - 128;
```

Secvență de Cod 9: Semnale intermediare cu valorile translate într-un interval simetric

Coeficienții de conversie sunt aproximați în format cu virgulă fixă Q10 (10 biți fracționari) pentru a elimina necesitatea aritmeticii cu virgulă flotantă, care ar consuma resurse excesive pe FPGA. În format Q10, un număr real este multiplicat cu 2^{10} = 1024 și rotunjit la cel mai apropiat întreg. Alegerea a 10 biți fracționari (Q10) reprezintă un compromis între precizie și complexitatea hardware: oferă o aproximare suficient de exactă a coeficienților reali, reducând în același timp dimensiunea

multiplicatoarelor necesare. Coeficienții Q10 sunt calculați manual în Ecuațiile 5, 6, 7, 8 pentru a reduce complexitatea modulului.

Acești coeficienți sunt definiți drept parametrii pe 16 biți în Secvența de cod 10.

```
localparam signed [15:0] C_1_402 = 16'h059C;
localparam signed [15:0] C_0_344 = 16'h0160;
localparam signed [15:0] C_0_714 = 16'h02DB;
localparam signed [15:0] C_1_772 = 16'h0716;
```

Secvență de Cod 10: Definirea coeficienților pentru conversie

Pentru a preveni depășirea și trunchierea prematură în timpul multiplicărilor și adunărilor, rezultatele intermediare sunt calculate pe 32 de biți, după cum este prezentat în Secvența de Cod 11.

```
assign r_temp = y_s + ((v_s * C_1_402) >>> 10);
assign g_temp = y_s - ((u_s * C_0_344) >>> 10) - ((v_s * C_0_714) >>> 10);
assign b_temp = y_s + ((u_s * C_1_772) >>> 10);
```

Secvență de Cod 11: Calcularea rezultatelor intermediare prin aplicarea formulelor (1), (2), (3)

Aceste valori sunt apoi trunchiate deoarece ieșirile RGB trebuie să fie pe 8 biți (0-255). Trunchierea și restricționarea sunt realizate prin blocuri secvențiale pentru a asigura sincronizarea ieșirilor cu ceasul pclk și a transmite rezultatele când semnalele de intrare sunt stabile. Secvența de Cod 12 evidențiază cum s-a realizat trunchierea pentru componenta roșu a unui pixel.

```
always @(posedge clk or negedge rst_n)
  if (~rst_n)                 r <= 8'h00; else
  if (start_conversion) begin
    if (r_temp < 'd0)         r <= 8'h00; else
    if (r_temp > 'd255)       r <= 8'hFF; else
                              r <= r_temp[7:0];
  end
```

Secvență de Cod 12: Calcularea valorilor finale RGB prin trunchierea valorilor intermediare

De asemenea, modulul superior, de conexiune cu procesorul prin interfața APB, este anunțat de finalizarea conversiei prin semnalul pixel_ready. Acest semnal sub formă de puls este setat când semnalul start_conversion este 1, în același timp cu punerea valorilor convertite pe ieșiri, și resetat după un tact de ceas pclk. Secvența de Cod 13 aduce în prim plan setarea semnalului de finalizare a conversiei pentru un pixel după un tact de ceas de la începerea ei, deoarece conversia este realizată combinațional, transmiterea rezultatelor fiind secvențială.

```
always @(posedge clk or negedge rst_n)
  if (~rst_n)             pixel_ready <= 1'b0; else
  if (pixel_ready)        pixel_ready <= 1'b0; else
  if (start_conversion)   pixel_ready <= 1'b1;
```

Secvență de Cod 13: Modelarea semnalului ce indică finalizare conversiei vlorilor YUV ale unui pixel

6.1.2 Modulul yuv422_to_rgb

Modulul yuv422_to_rgb integrează patru instanțe ale modulului yuv_to_rgb pentru a procesa simultan patru pixeli YUV422 (8 octeți: Y0, U0, Y1, V0, Y2, U1, Y3, V1), reflectând natura sub-eșantionată 4:2:2 a formatului. Acesta utilizează o interfață APB pentru a primi datele de intrare de la nucleele RISC-V și pentru a returna rezultatele RGB, fiind sincronizat cu ceasul pclk și resetul presetn, activ pe palierul de 0. Datele de intrare sunt stocate temporar într-un buffer de 8 regiștrii pe 8 biți, iar procesarea este orchestrată de o mașină de stări finite (FSM) cu trei stări: IDLE, LOAD și CONVERT.

Interfața APB este implementată pentru a permite nucleelor RISC-V să scrie datele YUV422 și să citească rezultatele RGB. Semnalul pready este modelat pentru a respecta protocolul APB, care prevede două faze: setup (când psel este activ și penable este inactiv) și access (când penable devine activ). În faza de setup, modulul acceptă datele de intrare, iar în faza de access confirmă finalizarea operațiunii. Semnalul prdata este configurat să returneze valorile RGB pentru fiecare pixel sau informații de stare, în funcție de valoarea lui paddr[7:0]. Având în vedere că bus-ul de APB permite transmiterea a 32 de biți de date, iar valorile RGB sunt formate din 8 biți pentru fiecare componentă, primul byte (LSB) va fi completat cu 0, având reprezentarea grafică în Figura 16.

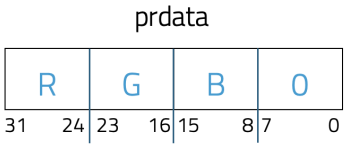


Figura 16: Strunctura semnalului PRDATA

Această împărțire pe subcâmpuri a semnalului de date de ieșire este modelat în Verilog folosind operația de concatenare, după cum se poate observa în Secvența de Cod 14.

```
1
2 always @(posedge pclk or negedge presetn)
3   if (~presetn) prdata <= 32'd0; else
4   if (psel & ~penable & ~pwrite)
5     case(paddr[7:0])
6       // Starea, dupa incarcare si conversie, FSM intra in IDLE (la primirea
    întreruperii)
7       8'h20: prdata <= (state == IDLE) ? {32'hFFFFFFFF} : {32'd0};
8       // Primul pixel
9       8'h30: prdata <= {rgb_out[0][0], rgb_out[0][1], rgb_out[0][2], 8'd0};
10      // Al doilea pixel
11      8'h34: prdata <= {rgb_out[1][0], rgb_out[1][1], rgb_out[1][2], 8'd0};
12      // Al treilea pixel
13      8'h38: prdata <= {rgb_out[2][0], rgb_out[2][1], rgb_out[2][2], 8'd0};
14      // Al patrulea pixel
15      8'h3c: prdata <= {rgb_out[3][0], rgb_out[3][1], rgb_out[3][2], 8'd0};
```

```
16      endcase
```

Secvență de Cod 14: Modelarea semnalului prdata conform protocolului APB

Datele de intrare sunt scrise secvențial prin pwdata în memoria yuv_mem, utilizând paddr[7:0] ca selector pentru pozițiile din memorie yuv_mem în care vor fi stocate datele. În cei 32 de biți transmiși de către procesor se regăsesc valorile YUV pentru 2 pixeli. Procesorul va realiza două astfel de scrieri, transmițând valorile YUV prezentate în Figura 17, iar aceste valori vor fi decodificate de către FPGA conform Secvenței de Cod 15.

Figura 17: Strunctura semnalului PWDATA în funcție de adresa de scriere PADDR

```
1  // Stocare in memorie a datelor scrise - 4 valori / adresa
2  always @(posedge pclk or negedge presetn)
3    if (~presetn) begin yuv_mem[0] <= 8'd0;
4                        yuv_mem[1] <= 8'd0;
5                        yuv_mem[2] <= 8'd0;
6                        yuv_mem[3] <= 8'd0;
7                        yuv_mem[4] <= 8'd0;
8                        yuv_mem[5] <= 8'd0;
9                        yuv_mem[6] <= 8'd0;
10                       yuv_mem[7] <= 8'd0; end else
11   if (psel & pwrite & ~penable)
12     case (paddr[7:0])
13         8'h10: begin yuv_mem[0] <= pwdata[7:0];          // Y0
14                      yuv_mem[1] <= pwdata[15:8];         // U0
15                      yuv_mem[2] <= pwdata[23:16];        // Y1
16                      yuv_mem[3] <= pwdata[31:24]; end    // V0
17         8'h14: begin yuv_mem[4] <= pwdata[7:0];          // Y2
18                      yuv_mem[5] <= pwdata[15:8];         // U1
19                      yuv_mem[6] <= pwdata[23:16];        // Y3
20                      yuv_mem[7] <= pwdata[31:24]; end    // V1
21           endcase
```

Secvență de Cod 15: Încărcarea în memorie a valorilor YUV transmise prin pwdata pe interfața APB

Procesarea pixelilor este controlată de o mașină de stări finite, ce comandă trecerea sistemului de conversie prin 3 stări:

- IDLE: Starea inițială și de așteptare, în care modulul monitorizează scrierea datelor prin APB. Sistemul se regăsește în această stare atât în momentul în care nu se realizează nicio operație

asupra datelor, cât și în momentul în care se așteaptă ca datele convertite să fie citite de către procesor. Tranziția către starea LOAD se realizează imediat ce are loc o scriere pe APB. Acest lucru determină începerea încărcării datelor în memoria yuv_mem și ulterior în registrele ce vor fi transmise către modulul de conversie.

- LOAD: În această stare, datele stocate în memorie sunt transferate către regiștrii intermediari (y0, u0, y1, v0, y2, u1, y3, v1), pregătindu-le pentru conversie. Trecerea către starea CONVERT este realizată după cea de-a doua scriere prin APB, adică în momentul în care s-au primit valorile YUV pentru 4 pixeli.

- CONVERT: Datele din regiștrii intermediari sunt procesate de instanțele yuv_to_rgb, iar rezultatele sunt scrise într-un buffer rgb_out de 4 locații a câte 3 adrese de 8 biți. După finalizare, starea revine la IDLE, pentru ca datele să poată fi citite de procesor și să se poată scrie apoi valorile YUV pentru următorii pixeli.

Pentru a ne asigura că au loc două scrieri APB înainte de a trece la starea de conversie, se folosește semnalul cnt_writes care se incrementează la fiecare tranzacție de scriere. Pentru a indica faptul că valorile YUV au fost primite și se poate începe conversia, se folosește semnalul start_conversion, care este transmis simultan către cele 4 instanțe de conversie. La finalizarea conversiei, fiecare instanță transmite un semnal de finalizare conectat la semnalul result_ready de 4 biți codați one-hot asemenea Figurii 18.

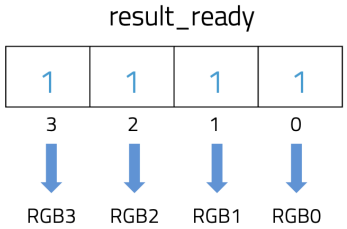


Figura 18: Codare one-hot pentru result_ready

În momentul în care toate instanțele transmit semnalul de finalizare a conversiei, rezultatele se încarcă în memoria rgb_mem și se setează un semnal de întrerupere prin care procesorul este anunțat că valorile RGB sunt gata de citire. Această întrerupere poate fi conectată la una dintre întreruperile sistemului, însă pentru implementarea de față se folosește metoda de polling, prin care procesorul citește constant registrul de stare, iar când aceasta trece în IDLE, poate începe citirea valorilor RGB ale celor 4 pixeli convertiți.

Arhitectura bazată pe FSM separă clar etapele de stocare, transfer și conversie, reducând riscul de erori de sincronizare. Paralelismul este maximizat prin utilizarea a patru instanțe yuv_to_rgb,

procesând cei patru pixeli într-un singur ciclu de conversie, ceea ce minimizează latența comparativ cu o abordare secvențială. Interfața APB asigură o integrare eficientă cu procesorul, permițând flexibilitate în transferul datelor și accesul la rezultate.

6.1.3 Simulări

Pentru testarea sistemului de conversie s-a creat, în faza inițială, un testbench ce a permis simularea semnalelor folosind forme de undă în ModelSim. Astfel s-a putut verifica funcționalitatea corectă a protocolului APB, respectiv corectitudinea valorilor convertite folosind formulele 9, 10,11.

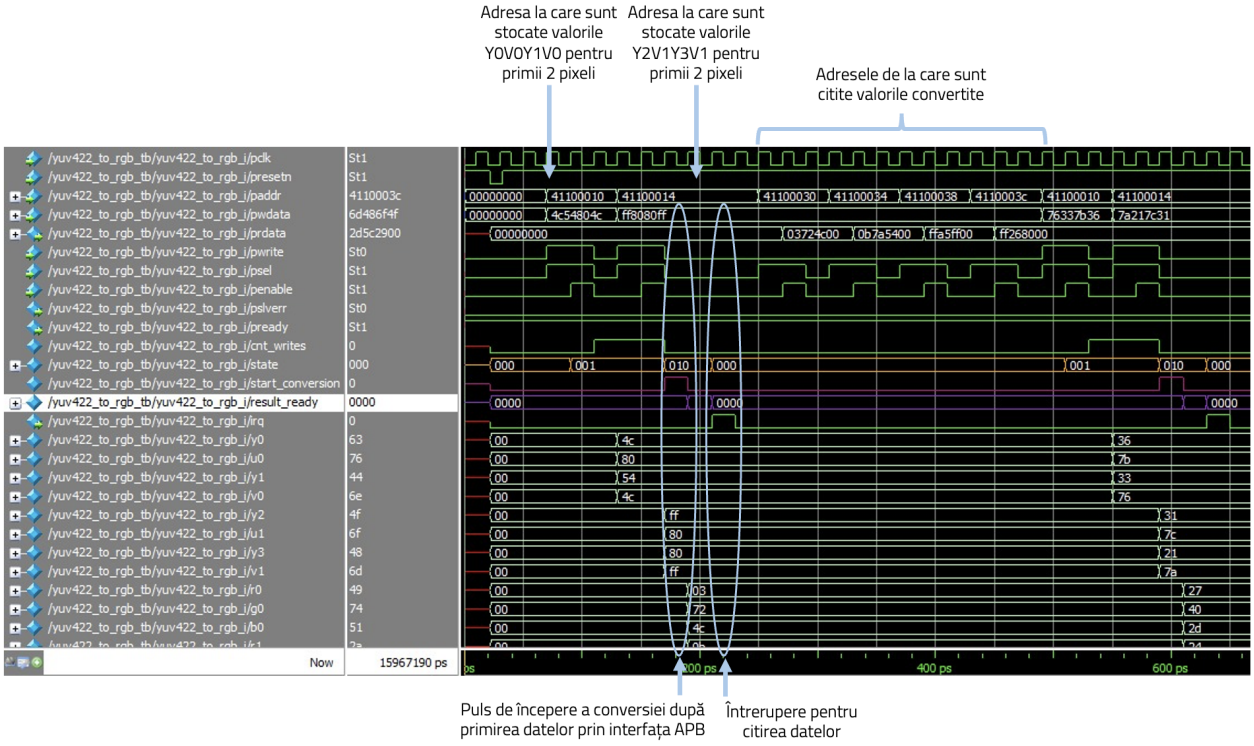


Figura 19: Forme de undă din cadrul simulării procesului de conversie

În cadrul Figurii 19 se poate observa cum au loc două transferuri folosind protocolul APB, scriindu-se cei 4 pixeli YUV. După al doilea transfer, când datele sunt stabile, acestea sunt transmise către instanțele de conversie, împreună cu semnalul de start. Conversia este rapidă, datorită folosirii multiplicatoarelor implementate pe FPGA, finalizându-se cu un semnal de ready, pentru ca datele RGB să poată fi preluate și stocate într-un buffer după care, citite prin interfața APB de către procesor. Pentru această simulare s-a folosit o imagine de rezoluție 176x144.

6.1.4 Optimizări

Din Figura 19 se poate observa faptul că se pierde foarte mult timp pe realizarea transferurilor APB de scriere/citire între procesor și FPGA (aproximativ 20 de tacte de ceas), mai ales că se așteaptă două scrieri pentru a se realiza cele 4 conversii simultan. Această primă implementare nu oferă o soluție optimizată, deoarece nu se utilizează la capacitate maximă proprietatea de paralelizare oferită de FPGA și, mai ales din cauza proprietății half-duplex a magistralei APB. De asemenea, adăugarea biților de umplutură pentru a se putea transfera un pixel de 24 de biți pe magistrala APB de 32 de biți, doar încetinește procesul prin necesitatea de a realiza un transfer/pixel.

Pentru a accelera această implementare se propun următoarele modificări:

- Transmiterea continuă a câte 32 de pixeli YUV (16 scrieri APB) și realizarea simultană a conversiei acestora, ceea ce înseamnă că la fiecare tranzacție APB de scriere se realizează 2 conversii. Modulul primește la adresa 0x41100014 numărul de operații de scriere ce urmează a fi realizate, iar pe baza acestuia calculează câte citiri se vor realiza, după cum se observă în Secvența de Cod 16.

```
// Numar de scrieri prin APB
always @(posedge pclk or negedge presetn)
  if (~presetn)
    number_of_write_transactions <= 'd0; else
  if (psel & (~penable) & pwrite & (paddr[7:0] == 8'h14))
    number_of_write_transactions <= pwdata;

// Numar de citiri prin APB
always @(posedge pclk or negedge presetn)
  if (~presetn)
    number_of_read_transactions <= 'd0; else
  if (psel & penable & pwrite & (paddr[7:0] == 8'h14))
    number_of_read_transactions <= (number_of_write_transactions * 6) >> 2;
    // pt 16 scrieri => (16*6) / 4 = 96/4 = 24 citiri
```

Secvență de Cod 16: Primirea numărului de tranzacții de scriere ce vor fi realizate

După primirea valorii corespunzătoare numărului de tranzacții de scriere, se încep scrierile efective, se preia valoarea primită pe pwdata, și se transmite împreună cu un puls de start către modulul convertor. Acest lucru este evidențiat de Secvența de Cod 17.

```
// Preluarea valorii YUV pentru 2 pixeli si transmiterea catre convertor
always @(posedge pclk or negedge presetn)
  if (~presetn)
    yuyv_value <= 'd0; else
  if (psel & (~penable) & pwrite & (paddr[7:0] == 8'h10))
    yuyv_value <= pwdata;
```

```
7
8 // Puls de inceperea conversiei cand s-au finalizat doua scrieri
9 always @(posedge pclk or negedge presetn)
10   if (~presetn)
11     start_conversion <= 1'b0; else
12   if (start_conversion)
13     start_conversion <= 1'b0; else
14   if (psel & (~penable) & pwrite & (paddr[7:0] == 8'h10))
15     start_conversion <= 1'b1;
```

Secvență de Cod 17: Încărcarea în registrul local a valorii YUV și transmiterea către convertor cu pulsul de start

- Stocarea valorilor RGB obținute pentru cei 32 de pixeli (96 de octeți) într-un buffer intern până la finalizarea celor 16 scrieri. Acest lucru permite transferarea valorilor RGB intercalate, fără a mai fi necesară introducerea biților de umplutură la fiecare pixel. În acest mod se reduce numărul de transferuri de la 2 scrieri YUV/ 4 citiri RGB la 2 scrieri YUV/ 3 citiri RGB. Pe baza a doi pointeri, unul de scriere în buffer, celălalt de citire, se asigură un mecanism ce permite scrieri de 6 octeți și citiri de câte 4, după cum se observă în Secvența de Cod 18.

```
1 assign data_converted  = |result_ready;
2 assign read_req        = psel & ~penable & ~pwrite;
3 assign wr_pointer_next = (data_converted & (~fifo_full)) ? ((wr_pointer +
    BYTES_WRITTEN == DEPTH) ? 'd0 : wr_pointer + BYTES_WRITTEN) : wr_pointer;
4 assign rd_pointer_next = (read_req & (paddr[7:0] == 8'h30) & (~fifo_empty)) ?
     ((rd_pointer + BYTES_READ == DEPTH) ? 'd0 : rd_pointer + BYTES_READ) :
    rd_pointer;
5
6 always @(posedge pclk)
7   if (data_converted)   begin fifo_rgb[wr_pointer] <=  r0;
8                               fifo_rgb[wr_pointer + 1] <=  g0;
9                               fifo_rgb[wr_pointer + 2] <=  b0;
10                              fifo_rgb[wr_pointer + 3] <=  r1;
11                              fifo_rgb[wr_pointer + 4] <=  g1;
12                              fifo_rgb[wr_pointer + 5] <=  b1; end
```

Secvență de Cod 18: Pointeri de scriere și citire și introducerea datelor convertite în buffer

- Având în vedere că valorile YUV sunt convertite imediat ce sunt scrise, la finalizarea scrierilor se poate face direct citirea din buffer a valorilor RGB, garantându-se astfel că întreruperea este mai rapidă și vine imediat după ultima citire. În Secvența de Cod 19 se observă cum s-a implementat întreruperea și cum se detectează aceasta.

```
1 // Puls se intrerupere la finalizarea conversiei - indicator ca datele pot fi
     citite
2 always @(posedge pclk or negedge presetn)
3   if (~presetn)
```

```
    irq <= 1'b0; else
  if (read_req & (paddr[7:0] == 8'h20))
    irq <= 1'b0; else
  if (cnt_writes == (number_of_write_transactions - 'd1))
    irq <= 1'b1;

    always @(posedge pclk or negedge presetn)
  if (~presetn) prdata <= 32'd0; else
  if (read_req)
    case(paddr[7:0])
      8'h20: prdata <= (irq) ? {32'hFFFFFFFF} : {32'd0};
      8'h30: prdata <= {fifo_rgb[rd_pointer + 3], fifo_rgb[rd_pointer + 2],
   fifo_rgb[rd_pointer + 1], fifo_rgb[rd_pointer]};
    endcase
```

Secvență de Cod 19: Puls de întrerupere și transmiterea datelor convertite

În Figura 20 este evidențiată simularea realizată pentru aceste optimizări.

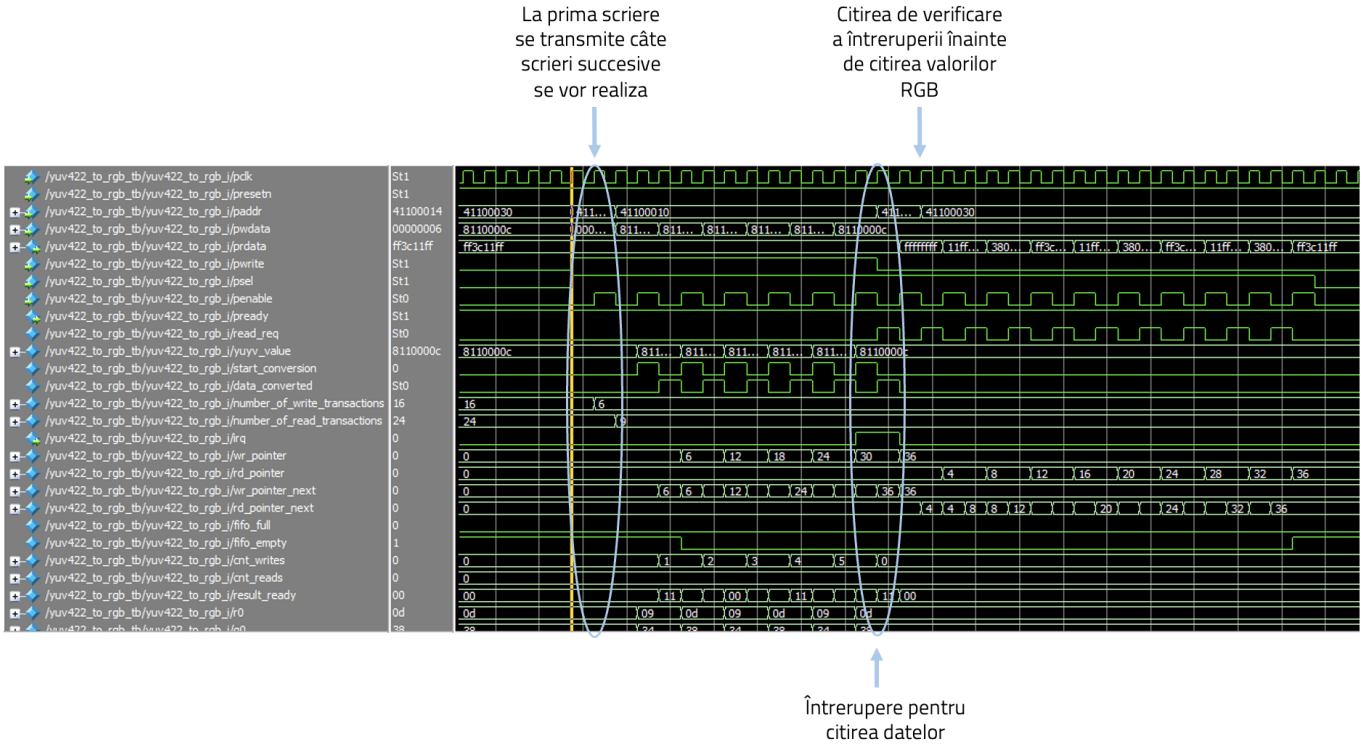


Figura 20: Forme de undă din cadrul simulării procesului de conversie optimizat

6.2 DEZVOLTAREA MODULULUI PENTRU CONVERSIE - VARIANTA CU DMA

Pentru a utiliza structura FPGA oferită de Microchip PolarFire SoC la capacitatea sa maximă, se poate trece cu implementarea conversiei la un nivel superior. Acest lucru implică reducerea timpului de interacțiune cu procesorul prin folosirea unei structuri DMA Controller, ce permite FPGA-ului să comunice direct cu memoria. Principiul acestei implementări este de a a avea doar 4 transferuri inițiale procesor-DMA Controller în care se transmit adresa de bază a zonei de memorie unde este stocată imaginea de intrare, adresa de bază a zonei de memorie unde este stocată imaginea de ieșire, dimensiunile imaginii de intrare și un semnal de start. După aceste 4 scrieri către DMA Controller, procesorul este liber să execute orice altă activitate până când imaginea de ieșire este obținută și stocată în memoria LSRAM. DMA Controller cuprinde două interfețe AXI4, una de citire, care va interacționa cu FIC0 pentru a citi datele de intrare direct din memorie, respectiv una de scriere, care, de asemenea, va comunica cu FIC0 pentru a scrie datele convertite în memorie.

Această implementare aduce beneficii prin independența sa față de procesor și folosirea unui ceas mai rapid pentru conversie - cel al interfeței AXI4 (250 MHz), comparativ cu cel al interfeței APB (125 MHz), ceea ce va duce și la performanțe de timp net superioare față de implementarea fără DMA. În plus se beneficiază de proprietatea full-duplex oferită de magistrala AXI. În continuare este descrisă pas cu pas implementarea modulelor constituente.

6.2.1 Modulul APB_INTERFACE

Acest modul are rolul de APB slave, având interfața prezentată în Tabelul 7. Modulul are în centru 4 scrieri pe magistrala APB, provenite de la procesor. Scrierea adreselor de start și a parametrilor de rezoluție se poate realiza în orice ordine, în schimb, semnalul de start trebuie să fie neapărat ultimul. Pentru a ne asigura de acest lucru se folosește un registru de 4 biți codat one-hot, al cărui biți se setează la fiecare scriere conform Figurii 21.

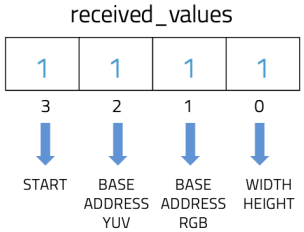


Figura 21: Codare one-hot pentru received_values

În cazul în care ordinea scrierilor nu este respectată, se setează pslverr pentru a alerta procesorul și a indica faptul că trebuie să reia scrierile de la început. De asemenea, registrul received_values se resetează și el, aspect modelat în Secvența de Cod 20.

```
assign transaction_apb     = psel & penable & pwrite & pready;
assign initiating_transfer = psel & (~penable) & pwrite;
assign incorrect_order     = initiating_transfer & (~&received_values[2:0]) & (
    paddr[7:0] == 8'h00);

always @(posedge pclk or negedge presetn)
  if (~presetn)                      received_values <= 4'b0000; else
  if (dma_start | incorrect_order)   received_values <= 4'b0000; else
  if (initiating_transfer)
    case (paddr[7:0])
      8'h00: received_values <= received_values | 4'b1000;
      8'h04: received_values <= received_values | 4'b0100;
      8'h08: received_values <= received_values | 4'b0010;
      8'h0c: received_values <= received_values | 4'b0001;
    endcase
```

Secvență de Cod 20: Primirea valorilor de configurare pentru DMA

La finalizarea celor 4 scrieri se setează semnalul dma_start, pentru a indica interfeței AXI4 de citire că poate începe citirea valorilor din memorie de la adresa de bază. În plus, este setat și semnalul dma_busy pentru a împiedica procesorul să interacționeze cu DMA Controller prin interfața APB (modulul nu mai poate seta pready pe parcursul conversiei).

6.2.1.1 Simulare

În cadrul simulării modulului APB_INTERFACE din Figura 22 se testează atât cazul primirii în ordinea corectă a valorilor de configurare și transmiterea acestora către DMA, cât și cazul primirii incorecte, în care se ignoră valorile primite și se setează semnalul de eroare.

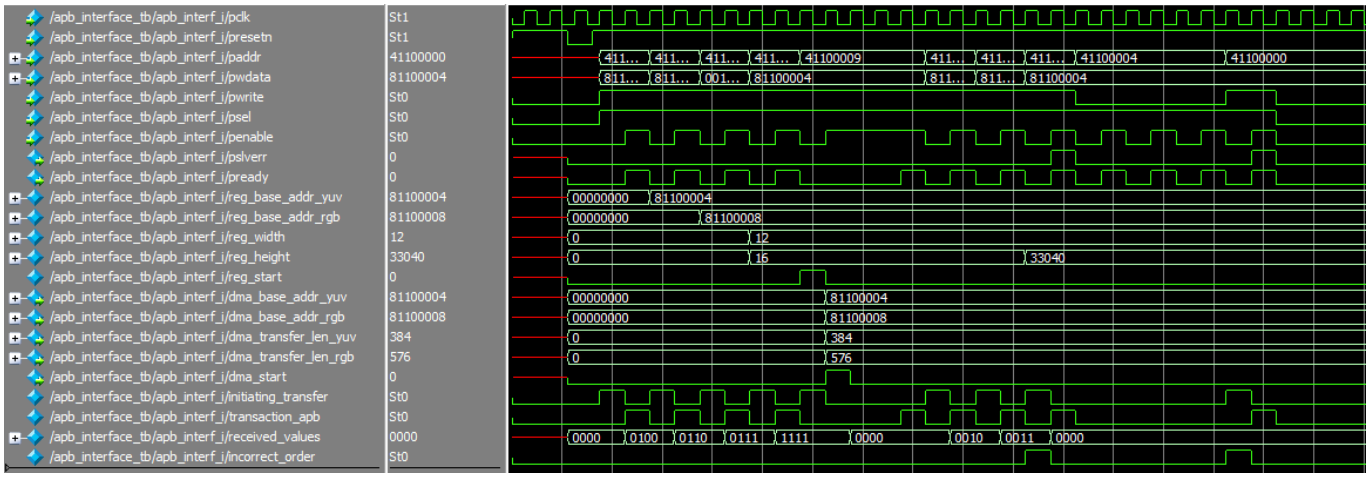


Figura 22: Forme de undă din cadrul simulării modulului APB_INTERFACE

6.2.2 Modulul AXI4_READ_INTERFACE

Acest modul are rolul de AXI4 master, având interfața prezentată în Tabelul 9. Acesta prezintă o interfață de citire specifică protocolului AXI4 [33], plus semnalele de control provenite de la modulul APB_INTERFACE (base_addr_yuv, transfer_len_yuv și start_read). Semnalul transfer_len_yuv conține numărul de octeți YUV necesari a fi citiți din memorie de la adresa de bază. Spre exemplu, pentru o imagine de rezoluție WIDTH x HEIGHT se vor citi:

$$WIDTH * HEIGHT * 2 \tag{12}$$

În momentul în care se primește semnalul de start se inițiază o tranzacție de citire către FIC0 prin setarea valorilor de pe canalul de adresă. Adresa primei tranzacții va fi adresa de bază, urmând ca la fiecare tranșă aceasta să fie incrementată cu 8. Acest lucru este realizat prin setarea semnalului ARBURST cu valoarea 2'b01 (INC - valoarea adresei se incrementează la fiecare tranșă). De asemenea, având în vedere faptul că magistrala AXI operează cu date pe 64 de biți, se vor citi câte 8 octeți (4 pixeli) într-o tranșă, acest lucru fiind setat prin semnalul ARSIZE la 3'b110. În plus, având în vedere că pentru imaginile folosite numărul de octeți depășește ordinul milioanelor, se vor realiza transferuri în rafală de câte 256 tranzacții atâta timp cât se poate, prin setarea semnalului ARLEN la valoarea 255. Pentru a tine evidența câți octeți mai sunt necesari a fi citiți se folosește un numărător cnt_reads_needed, care descrește la fiecare tranșă cu 8. La fiecare nou transfer de citire se verifică valoarea numărătorului și se ajustează numărul de tranșe. De asemenea, având în vedere că la fiecare transfer trebuie precizată adresa de început, se folosește un alt numărător cnt_current_address, care la fiecare tranșă este incrementat cu 8. Implementarea acestor semanle se poate observa în Secvența de Cod 21.

```
assign full_image_read_done = (M_AXI_RREADY & M_AXI_RVALID & M_AXI_RLAST) &
(cnt_reads_needed == 'd0);

always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
  if (~M_AXI_ARESETn)
    cnt_reads_needed <= 'd0; else
  if (start_read)
    cnt_reads_needed <= transfer_len_yuv; else
  if (M_AXI_RREADY & M_AXI_RVALID & (cnt_reads_needed != 'd0))
    cnt_reads_needed <= cnt_reads_needed - 'd8;

always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
  if (~M_AXI_ARESETn)
    cnt_current_address <= 'd0; else
  if (start_read)
    cnt_current_address <= base_addr_yuv; else
```

```
17   if (M_AXI_RREADY & M_AXI_RVALID)
18     cnt_current_address <= cnt_current_address + 'd8;
19
20 always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
21   if (~M_AXI_ARESETn)
22     start_axi_transaction <= 1'b0; else
23   if (start_axi_transaction)
24     start_axi_transaction <= 1'b0; else
25   if (start_read | ((M_AXI_RREADY & M_AXI_RVALID & M_AXI_RLAST) &
26     (|cnt_reads_needed)))
27     start_axi_transaction <= 1'b1;
```

Secvență de Cod 21: Calcularea numărului de citiri necesare și a adresei curente

Pe canalul de citire a datelor, modulul de față va primi date în transe de câte 8 octeți și le va accepta numai dacă există loc în FIFO_YUV, un buffer la care accesul de scriere/citire se realizează prin intermediul protocolului request-acknoledge, implementat în Secvența de cod 22. Modulul curent transmite, simultan cu primirea datelor pe AXI, valori către acest buffer pentru ca acestea să poată fi preluate de modulul de conversie, fără a se pierde valori în cazul în care apar desincronizări sau întârzieri. Imediat ce se citește prima tranșă se setează și semnalul de start pentru conversie și se începe transferul datelor către fifo. În cazul umplerii acestui buffer, citirile pe AXI vor intra în pauză (nu se steză RREADY) până la golirea unei adrese.

```
1 // FIFO signals
2 always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
3   if (~M_AXI_ARESETn)               yuyv_4pixels <= 'd0; else
4   if (M_AXI_RREADY & M_AXI_RVALID)  yuyv_4pixels <= M_AXI_RDATA;
5
6 always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
7   if (~M_AXI_ARESETn)                           write_req <= 'd0; else
8   if (M_AXI_RREADY & M_AXI_RVALID)              write_req <= 'd1; else
9   if (write_ack)                                write_req <= 'd0;
10
11
12 always @(posedge M_AXI_ACLK or negedge M_AXI_ARESETn)
13   if (~M_AXI_ACLK)            start_conversion <= 'd0; else
14   if (write_req & write_ack)  start_conversion <= 'd1; else
15   if (start_conversion)       start_conversion <= 'd0;
```

Secvență de Cod 22: Handshake request-acknowledge pentru scrierea datelor în FIFO$_YUV$

6.2.2.1 Simulare

În cadrul simulării din Figura 23 s-a testat funcționare corectă și sincronizarea citirilor din memorie pe magistrala AXI cu transferul datelor citite în FIFO_YUV. De asemenea, s-au luat în considerare și cazurile în care acest buffer ajunge să se umple și nu se va mai seta semnalul RREADY.

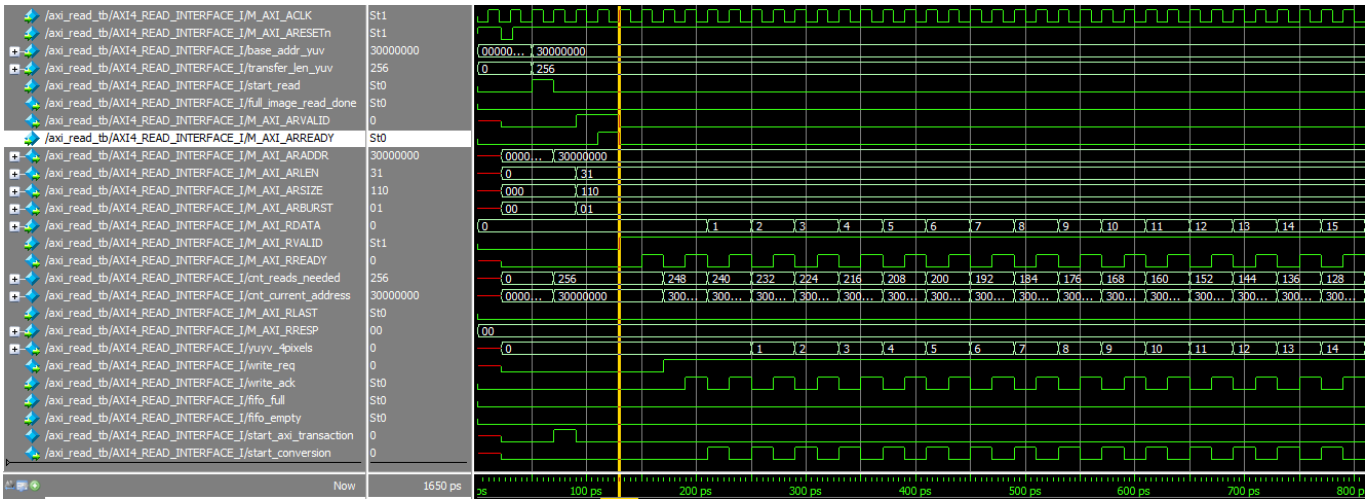


Figura 23: Forme de undă din cadrul simulării modulului AXI_READ_INTERFACE

6.2.3 Modulul AXI4_WRITE_INTERFACE

Acest modul are rolul de AXI4 master, având interfața prezentată în Tabelul 8. Modulul gestionează o interfață de scriere conform specificațiilor protocolului AXI4 [33], precum și semnalele de control provenite de la modulul APB_INTERFACE -base_addr_rgb și transfer_len_rgb. Semnalul transfer_len_rgb conține numărul de octeți RGB necesari a fi scriși în memorie de la adresa de bază. Spre exemplu, pentru o imagine de rezoluție WIDTH x HEIGHT se vor citi:

$$WIDTH * HEIGHT * 3 \tag{13}$$

La primirea semnalului start_write_rgb, modulul inițiază o tranzacție de scriere către interfața AXI (prin FIC0), setând valorile corespunzătoare pe canalul de adresă. Adresa de început este base_addr_rgb, urmând ca pentru fiecare tranșă aceasta să fie incrementată cu 8 octeți, valoare corespunzătoare lățimii magistralei AXI (64 biți). Incrementarea adresei este realizată prin configurarea semnalului AWBURST cu valoarea 2'b01 (incremental burst), iar AWSIZE este setat la 3'b011, indicând transferuri pe 8 octeți.

Pentru a eficientiza transferul, se utilizează tranșe în rafală de câte 256 tranzacții (atât cât permite protocolul AXI), prin setarea semnalului AWLEN la valoarea maximă 8'hFF. Pe măsură ce fiecare tranșă este inițiată, un contor intern cnt_writes_needed scade cu 8, iar un alt contor, cnt_current_address,

este incrementat corespunzător cu fiecare tranșă, pentru a menține adresa de început a următorului transfer.

În ceea ce privește canalul de date, valorile RGB convertite sunt preluate din bufferul FIFO_RGB sub forma a 64 de biți. Modulul va transmite date pe canalul de date doar în momentul în care FIFO_RGB nu este gol și există confirmarea (read_ack) pentru fiecare cerere de citire (read_req) către buffer. Protocolul de tip request–acknowledge asigură sincronizarea corectă între acest modul și bufferul de date.

Semnalul WLAST este setat corespunzător la ultima tranșă din fiecare burst, iar răspunsurile de scriere (BRESP) sunt monitorizate pentru a detecta eventualele erori. În cazul în care bufferul este gol, iar ACK-ul nu este primit, transferul de date este temporar suspendat prin negarea semnalului WVALID până la disponibilitatea următorului pachet de date.

Acest mecanism asigură o scriere robustă și eficientă a datelor RGB convertite în memorie, sincronizată cu disponibilitatea datelor din FIFO_RGB și cu protocolul de transfer AXI.

6.2.4 Modulul FIFO

Acest modul are rolul de buffer intermediar ce păstrează datele în cazul diferențelor de sincronizare între scrierea și citirea lor, având interfața prezentată în Tabelul 10. În cadrul sistemului sunt folosite două astfel de buffere, unul pentru imaginea de intrare, celălalt pentru cea de ieșire. Ambele buffere au dimensiuni standard de 256 de date a câte 64 de biți pentru a se potrivii configurărilor realizate în cadrul tranzacțiilor pe AXI și a se putea realiza un transfer compet de scriere/ citire înainte de a se umple buffer-ul.

Scrierea, respectiv citirea din buffer se realizează pe baza protocolului request-acknowledge, iar pozițiile de unde se citesc sau scriu datele sunt indicate de 2 pointeri, care se incremenetează independent unul de celălalt până în momentul în care se intersectează.

6.2.4.1 Simulare

În cadrul simulării din Figura 24 s-a testat sincronizarea dintre cele două buffere de date - de intrare și ieșire și modulul de conversie YUV_TO_RGB_CONVERSION. Se poate observa cum din momentul în care are loc prima scitire a datelor din memorie și acestea sunt stocate în FIFO_YUV, are loc un transfer continuu sincronizat între cele 3 module. De asemenea, în Figura 25 sunt testate cazulrile în care se realizează scriri fără citiri corespondente și se umplu cele două buffere, ajungându-se la stadiul în care până la primirea unei cereri de citire sistemul rămâne blocat.

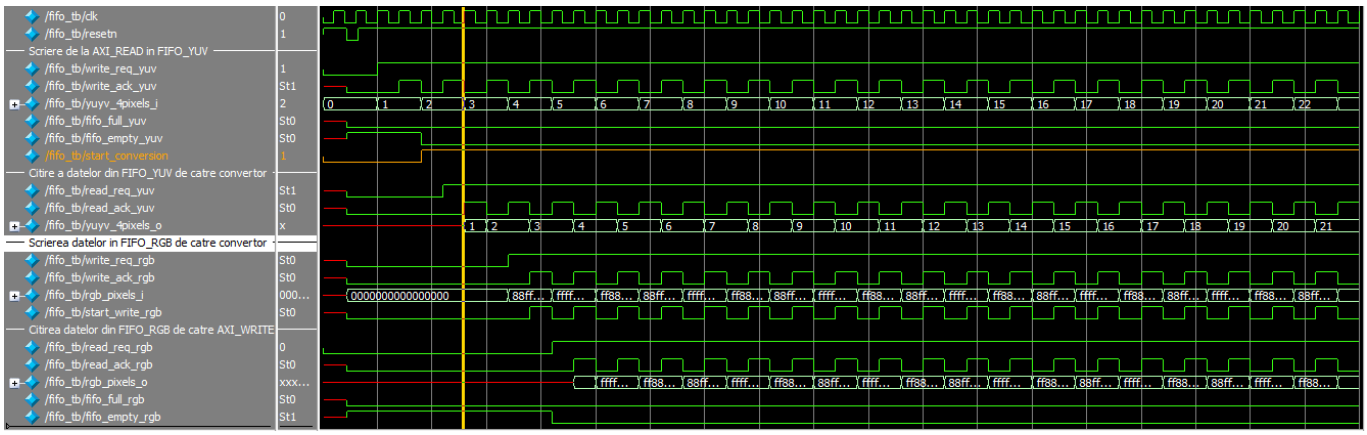


Figura 24: Forme de undă din cadrul simulării modulului FIFO

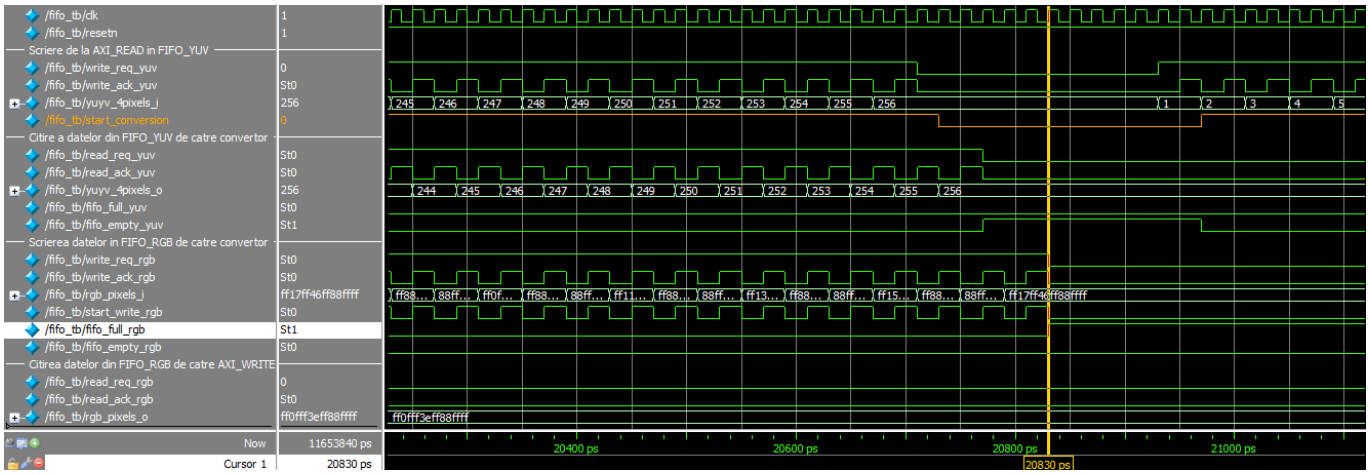


Figura 25: Forme de undă din cadrul simulării modulului FIFO - umplerea buffer-ului FIFO_RGB

6.2.5 Modulul YUV_TO_RGB_CONVERSION

Acest modul are rolul principal în sistem - de a realiza conversia, având interfața prezentată în Tabelul 11. În cadrul acestui modul se înaintează cereri de citire a câte 8 pixeli YUV din buffer-ul corespunzător. Pe măsură ce se realizează aceste cereri de citire, datele YUV sunt transmise spre conversie spre 4 instante ale modulului de conversie, iar la finalizarea acesteia, valorile RGB (96 de biți) sunt stocate într-un buffer intern, conceput special pentru intrcalare valorilor RGB a mai mulți pixeli și a transmite câte 64 de biți către FIFO_RGB. Acest lucru asigură simplitatea și posibilitatea de refolosire a modulului FIFO și, totodată, permite realizarea transferurilor standard pe magistrala AXI la scriere în memorie, fără a fi necesară adăugarea biților de umplutură. Secvența la Cod 23 evidențiază semnalele care stau la baza funcționării corecte a acestei implementări - distanța (se ține cont de numărul de elemente din buffer la momentul curent), full (pentru a stopa cererile de citire din FIFO_YUV pana când nu se citesc date, empty (pentru a se opri cererile de scriere până nu se convertesc noi date).

```
// Semnale BUFFER INTERN
assign distance      = (pointer_rgb_write_pos >= pointer_rgb_read_pos)?
pointer_rgb_write_pos - pointer_rgb_read_pos :
RGB_BUFFER_DEPTH - (pointer_rgb_read_pos - pointer_rgb_write_pos);

assign distance_next = distance + (pixels4_converted ? RGB_BYTES_WRITTEN : 0) - (
    rgb_data_sent ? RGB_BYTES_READ : 0);

assign buffer_full   = (distance_next + RGB_BYTES_WRITTEN) > RGB_BUFFER_DEPTH;

assign buffer_empty  = distance_next < RGB_BYTES_READ;

assign next_read     = pointer_rgb_read_pos + RGB_BYTES_READ;
```

Secvență de Cod 23: Semnale de control pentru buffer-ul intern pentru scrieri de 12 octeți și citiri de 8 octeți

6.2.5.1 Simulare

În cadrul simulării din Figura 26 a fost testat fluxul continuu de scrieri și citiri din memorie cu trecere prin modulul de conversie, asemenea cazului real. S-a verificat corectitudinea rezultatelor stocat în buffer-ul intern și a datelor ce au fost scoase din aceasta și, de asemenea, s-a testat implementarea protocolului request-acknowledge cu cele două module FIFO.

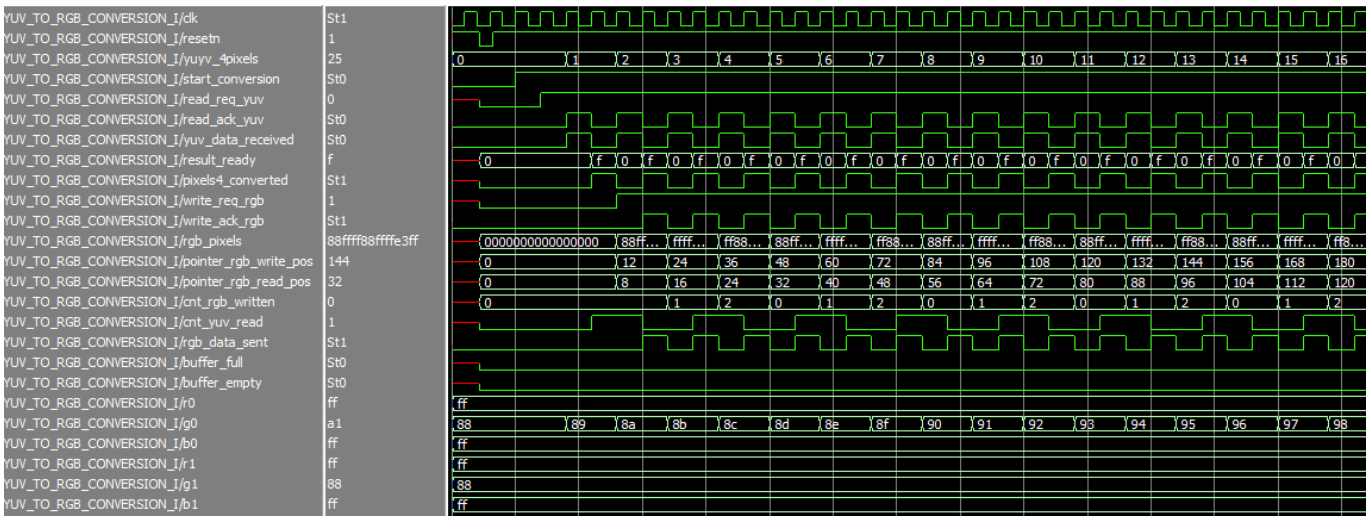


Figura 26: Forme de undă din cadrul simulării modulului YUV_TO_RGB_CONVERSION

6.3 Implementarea comunicării procesor-FPGA în C și verificarea rezultatelor

Pentru a putea realiza comunicarea efectivă dintre procesor și FPGA (în cazul implementării fără DMA) este necesară preluarea imaginii de intrare, în forma .yuv, stocarea acesteia într-un buffer, maparea zonei de memorie către care se transmit pixelii, respectiv transmiterea acestora și așteptarea întreruperii pentru a putea citii rezultatul. Această parte a fost realizată în limbajul de programare C, având structura evidențiată în Secvența de Cod 24, și va fi explicată pas cu pas în această secțiune.

```
main.c          // citirea imaginii de intrare din fisier,
                // apelarea functiei de conversie specificata prin parametrul METHOD
                // scrierea imaginii de iesire într-un nou fisier
conversion.c    // functiile de conversie (varianta procesor),
                // maparea zonei de memorie dintre procesor si FPGA,
                // initierea tranzactiilor de scriere si citire
conversion.h    // declararea interfetelor functiilor si tipurilor de date
Makefile        // automatizeaza compilarea si rularea programului
```

Secvență de Cod 24: Structura de fișiere a programului realizat de către procesor

Programul principal al aplicației este implementat în fișierul main.c Acesta are rolul de a gestiona fluxul de execuție începând de la citirea argumentelor input_file, output_file, WIDTH, HEIGHT și METHOD din linia de comandă. Argumentele sunt parsate și stocate în variabile adecvate, după cum este specificat în Secvența de Cod 25.

```
const char *input_filename = argv[1];
const char *output_filename = argv[2];
uint32_t WIDTH = atoi(argv[3]);
uint32_t HEIGHT = atoi(argv[4]);
```

Secvență de Cod 25: Parsarea și stocarea argumentelor

Argumentul METHOD este folosit pentru a identifica de cine se dorește realizarea conversiei - de către procesor (varianta cpu), de către FPGA (varianta fpga) sau prin ambele metode și compararea rezultatelor (varianta both). Astfel, se compară al cincilea argument cu variantele menționate pentru a determina ce funcții vor fi apelate în continuare, după cum poate fi observat în Secvența de Cod 26.

```
  if (strcmp(argv[5], "cpu") == 0)
    method = CONVERSION_CPU;
  else if (strcmp(argv[5], "fpga") == 0)
    method = CONVERSION_FPGA;
  else if (strcmp(argv[5], "both") == 0)
    method = BOTH;
  else
  {
    fprintf(stderr, "Metoda invalida: 'cpu' - 'fpga' - 'both'\n");
    return 1;
```

```
11  }
```

Secvență de Cod 26: Parsarea și stocarea argumentului METHOD

În continuare se realizează alocarea memoriei pentru buffer-ele ce vor stoca valorile YUV, respectiv RGB ale imaginii de intrare și celei de ieșire. Buffer-ul de stocare al imaginii de intrare va avea o dimensiune de HEIGHT * WIDTH * 2 deoarece formatul YUV422 folosește 2 octeți/pixel, iar buffer-ul de stocare al imaginii de ieșire va avea o dimensiune de HEIGHT * WIDTH * 3 deoarece formatul RGB folosește 3 octeți/pixel. Este folosită funcția fread() pentru a citi în întregime imaginea din fișierul de intrare și a o stoca în buffer, iar apoi, în funcție de metoda selectată se apelează funcțiile corespunzătoare, după cum este prezentat în Secvența de Cod 27.

```
1 switch (method)
2   {
3     case CONVERSION_CPU: { convert_cpu(yuv_buffer, rgb_buffer, WIDTH, HEIGHT);
                             break;}
4     case CONVERSION_FPGA: { convert_fpga(yuv_buffer, rgb_buffer, WIDTH, HEIGHT);
                              break;}
5     case BOTH : { convert_fpga(yuv_buffer, rgb_buffer, WIDTH, HEIGHT);
6                   uint8_t *rgb_buffer_cpu = malloc(rgb_size);
7                   if (!rgb_buffer_cpu)
8                   {
9                     perror("Nu s-a putut aloca memorie!");
10                    return 1;
11                  }
12                  convert_cpu(yuv_buffer, rgb_buffer_cpu, WIDTH, HEIGHT);
13                  compare_rgb(rgb_buffer, rgb_buffer_cpu, WIDTH, HEIGHT);
14                  break;
15                }
16  }
```

Secvență de Cod 27: Selectarea metodei de realizare a conversiei pe baza variabilei metod

După conversie, imagine RGB rezultată este scrisă în fișierul de ieșire și are loc eliberarea memoriei și închiderea fișierelor.

Fișierul conversion.h este header-ul proiectului și are rolul de a declara tipurile de date, funcțiile și structurile necesare pentru conversie. Fișierele main.c și conversion.c includ acest header pentru a avea acces la definițiile comune. În acest fișier sunt incluse bibliotecile standard: stdint.h - pentru tipurile de date de dimensiune fixă uint8_t, uint16_t și uint32_t folossite la formulele de conversie și la stocarea pixelilor, stdio.h - pentru funcții de I/O, stdlib.h - pentru alocarea dinamică de memorie, string.h - pentru lucrul cu șiruri de caractere, time.h, opm.h - pentru paralelizare. De asemenea, este declarat și tipul de enumerație ConversionMethod având structura specificată în Secțiunea de Cod 28 și structura RGB (pentru a stoca componentele de culoare ale fiecărui pixel), având descrierea

specificată de Secțiunea de Cod 28.

```
typedef enum
{
  CONVERSION_CPU,
  CONVERSION_FPGA,
  BOTH
} ConversionMethod;

typedef struct
{
  uint8_t r;
  uint8_t g;
  uint8_t b;
} RGB;
```

Secvență de Cod 28: Definirea enumerației ConversionMethod și a structurii RGB

În plus, în acest fișier sunt definite și prototipurile funcțiilor utilizate.

- long time_diff_us (struct timespec start, struct timespec end); - calculează diferența în microsecunde între două momente temporale, utile pentru trofilarea performanței;
- long sum_conversion_times (const long *durations, size_t count); - calculează suma duratelor de execuție pentru conversia grupurilor de câte 4 pixeli;
- uint8_t clip(uint16_t value); - limitează valorile calculate ale componentelor RGB între 0 și 255 pentru a evita depășirile;
- void yuv422_to_rgb (uint8_t y0, uint8_t u, uint8_t y1, uint8_t v, RGB* pixel0, RGB* pixel1); - implementarea formulelor de calcul pentru conversia YUV-RGB, utilă pentru implementarea software;
- void convert_cpu (uint8_t *yuv_buffer, uint8_t *rgb_buffer, uint32_t width, uint32_t height); - implemntarea conversiei YUV-RGB realizată integral de către procesor;
- int convert_fpga (uint8_t *yuv, uint8_t *rgb_buffer, uint32_t width, uint32_t height); - transmiterea datelor YUV către FPGA prin intermediul zonelor mapate de memorie, declanșând conversia hardware și recepționând rezultatul RGB înapoi;
- void compare_rgb (uint8_t *rgb_buffer_fpga, uint8_t *rgb_buffer_cpu, uint32_t WIDTH, uint32_t HEIGHT); - compară rezultatele celor două tipuri de implementări, verificând corectitudinea acestora.

În fișierul conversion.c, funcțiile cheie sunt:

- yuv422_to_rgb - în care se convertește un grup de 4 octeți în format YUV422 în 2 pixeli RGB. Această funcție este folosită în cadrul conversiei realizate integral de către procesor, iar pentru a fi cât mai apropiată de varianta hardware, s-a ales folosirea coeficienților calculați în ecuațiile 5, 6, 7, 8, pentru a evita înmulțirile cu numere reale, și shiftare, pentru a înlocui împărțirea. Secvența de Cod 29 evidențiază implementarea funcției.

```
void yuv422_to_rgb(uint8_t y0, uint8_t u, uint8_t y1, uint8_t v, RGB* pixel0,
    RGB* pixel1)
{
  uint32_t r_temp, g_temp, b_temp;
  uint16_t r0, g0, b0;
  uint16_t r1, g1, b1;

  r_temp = (1436 * (v - 128)) >> 10;
  g_temp = ((352 * (u - 128)) >> 10) + ((731 * (v - 128)) >> 10);
  b_temp = (1814 * (u - 128)) >> 10;

  r0 = y0 + r_temp;
  g0 = y0 - g_temp;
  b0 = y0 + b_temp;
  r1 = y1 + r_temp;
  g1 = y1 - g_temp;
  b1 = y1 + b_temp;

  pixel0->r = clip(r0);
  pixel0->g = clip(g0);
  pixel0->b = clip(b0);
  pixel1->r = clip(r1);
  pixel1->g = clip(g1);
  pixel1->b = clip(b1);
}
```

Secvență de Cod 29: Implementarea ecuațiilor de conversie YUV-RGB în software

- convert_cpu - în această funcție imaginea este procesată pe segmente de câte 8 octeți (4 pixeli YUV) folosiți în apelarea funcției yuv422_to_rgb, care îi convertește în 4 pixeli RGB, ale căror valori sunt apoi stocate în buffer-ul imaginii de ieșire. S-a folosit structura #pragma omp parallel for pentru a împărți bucla for între mai multe fire de execuție, accelerând astfel conversia și folosind eficient procesorul multi-core. Pentru a măsura timpul de utilizare al procesorului s-a folosit funcția clock_gettime() înainte și după apelarea funcției de conversie.

- convert_fpga - procesorul și FPGA-ul comunică prin intermediul zonelor de memorie mapate, astfel, primul pas necesar este maparea zonei de memorie începând de la adresa de memorie #define BASE_ADDR 0x41100000 pentru a avea acces direct la regiștrii de control ai FPGA-ului.

Pentru a realiza acest lucru se folosește Secvența de Cod 30.

```
1    int mem_fd = open("/dev/mem", O_RDWR | O_SYNC); // O_SYNC - scrieri
    directe, fara buffer/ cache
2    void *map_base = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
     mem_fd, APB_BASE_ADDR); // MAP_SHARED - modificarile afecteaza si memoria
     fizica
```

Secvență de Cod 30: Accesul direct la regiuni de memorie fizică rezervate pentru periferice

/dev/mem este un fișier special în sistemele Linux care oferă acces la întreaga memorie fizică a sistemului. Acesta se deschide cu permisiunile O_RDWR - permite scriere și citirea în regiunea de memorie și O_SYNC - forțează scrierile să fie realizate direct în memorie, fără cache sau buffer, ceea ce este esențial pentru sincronizarea cu hardware-ul.

Următoarea lini mapează regiunea de memorie fizică în spațiul de adrese virtuale al procesului, permițând acces direct la registrii FPGA. Parametrul NULL permite sistemului să aleagă automat adresa de mapare. Parametrul MAP_SIZE indică dimensiunea regiunii de memorie ce va fi mapată (în acest caz 4096 bytes, adică o pagină de memorie). Parametrii PROT_READ | PROT_WRITE specifică drepturile de acces asupra regiunii mapate (citire și scriere). Parametrul MAP_SHARED indică faptul că modificările realizate de proces asupra regiunii mapate vor fi reflectate în memoria fizică și pot fi vizibile și altor procese care mapează aceeași zonă. Urmează descriptorul obținut anterior, respectiv adresa fizică de bază.

Pentru a putea realiza scrierea în memorie se creează un pointer către o adresă fizică specificată din memoria mapată, specificând compilatorului să nu optimizeze accesul la acea adresă de memorie (prin atributul volatile), pentru a evita modificarea valorii din registru în afara programului. Spre exemplu, pointer-ul creat pentru a putea scrie la adresa fizică 0x41100010 este evidențiat în Secvența de Cod 31.

```
1     volatile uint32_t *y0u0y1v0 = (volatile uint32_t *)(map_base + 0
    x00000010);
```

Secvență de Cod 31: Pointer către o adresă fizică specificată din memoria mapată - 0x41100010

Scrierea, respectiv citirea efectivă la adresa de memorie sunt realizate în Secvența de Cod 32.

```
1     *y0u0y1v0 = ((uint32_t)yuv_buffer[i+3] << 24) |
2                 ((uint32_t)yuv_buffer[i+2] << 16) |
3                 ((uint32_t)yuv_buffer[i+1] << 8)  |
4                 ((uint32_t)yuv_buffer[i+0]);
5    volatile uint32_t rgb0_reg;
6    rgb0_reg = *rgb0;
```

Secvență de Cod 32: Scrierea și citirea de la o adresă fizică din memoria mapată

Pentru a verifica dacă conversia a fost finalizată se pot folosi două metode: polling - se citește periodic un registru status de la o anumită adresă de memorie, iar când statusul indică conversie completă se pot citii valorile RGB) sau întrerupere (la finalizarea conversiei, procesorul primește o întrerupere și poate citi datele din memorie).

- compare_rgb - în această funcție se compară octet cu octet valorile RGB de ieșire rezultate în urma apelării celor două funcții de conversie. Se vor afișa diferențele detectate, respectiv numărul final de diferențe după compararea întregii imagini.

Pentru a automatiza procesul de compilare și rulare, se folosește fișierul Makefile, asigurând de asemenea și organizarea fișierelor de ieșire prin adăugarea în denumirea acestora a căii unde vor fi stocate și a unui sufix temporal și tipul de conversie prin care a fost obținut. Astfel, nu va mai fi necesară introducerea numelui fișierului de ieșire la rularea programului în linia de comandă, deoarece acesta va fi automat creat pornind de la denumirea celui de intrare, după cum se poate observa în Secvența de Cod 33.

```
OUTPUT := $(OUT_DIR)/$(BASENAME)$(SUFFIX)_$(TIMESTAMP).rgb
```

Secvență de Cod 33: Formatarea numelui fișierului de ieșier în Makefile

Partea de implementare a constituit puntea esențială între conceptele teoretice și realizarea practică a soluției propuse. Prin integrarea atentă a procesării software și hardware, s-au urmărit nu doar funcționalitatea corectă, ci și obținerea unei baze solide pentru analiza performanțelor.

6.3.1 Optimizări

Având în vedere arhitectura pe 64 de biți a nucleelor RISC-V prezente pe placa de dezvoltare, aceste sunt specializate în realizarea operațiilor cu operanzi de 64 de biți. Însă, având în vedere faptul că magistrala APB permite transferul datelor de 32 de biți, se poate realiza un compromis și se poate trece de la implementarea buffere-lor de stocare a imaginilor de la uint8_t* la uint32_t* asemenea Secvenței de Cod 34.

```
uint32_t *yuv_buffer = aligned_alloc(8, yuv_size);
uint32_t *rgb_buffer = aligned_alloc(8, rgb_size);
```

Secvență de Cod 34: Trecerea de la stocarea în buffer uint8_t* la uint32_t*

De asemenea, se poate reduce numărul de adrese mapate la doar 4 deoarece se pot face transferuri succesive la aceeași adresă ca în Secvența de Cod 35.

```
// se poate realiza scrierea la aceeasi adresa si imediat ce ajung la fpga,
//datele sunt stocate intr-un registru si convertite si puse in buffer de iesre
  volatile uint32_t *y0u0y1v0 = (volatile uint32_t *)(map_base + 0x00000010);
```

```
  volatile uint32_t *transfer_len = (volatile uint32_t *)(map_base + 0x00000014);
  volatile uint32_t *rgb0 = (volatile uint32_t *)(map_base + 0x00000030);
  volatile uint32_t *status = (volatile uint32_t *) (map_base + 0x00000020);
```

Secvență de Cod 35: Reducerea numărului de adrese mapate

În plus, pentru ca acțiunilor procesorului să se poată alinia optimizărilor realizate pe FPGA, este necesară realizarea a câte 16 scrieri succesive, urmate de cele 24 de citiri aferente, vizibile în Secvența de Cod 36. Se ia în considerare și cazul în care dimensiunea imaginii nu este multiplu de 16, iar în acest caz la ultima tranșă se poate reduce numărul de scrieri, respectiv citiri de 32 de biți.

```
for(uint64_t i = 0; i < total_uint32_yuv; i += 16)
  {
    size_t remaining_yuv = total_uint32_yuv - i;
    uint32_t to_send = (remaining_yuv < 16) ? remaining_yuv : 16;

    *transfer_len = to_send;
    // se fac 16 (sau cate mai raman) scrieri de 32 de biti catre FPGA
    for(size_t j = 0; j < to_send; j++)
      *y0u0y1v0 = yuv_buffer[i + j];

    int64_t tries = 100;
    size_t rgb_index_offset = (i / 16) * 24; // avansare cu 24 pentru un pachet
   RGB
    size_t remaining_rgb = total_uint32_rgb - rgb_index_offset;
    size_t to_read = (remaining_rgb < 24) ? remaining_rgb : 24;
    while (tries--)
    {
      if (*status == 0xFFFFFFFF)
      {
        // 24 citiri (sau cate mai raman) de 32 de biti RGB de la FPGA
        for (size_t j = 0; j < to_read; j++)
        {
          rgb_buffer[rgb_index_offset + j] = *rgb0;
        }
        break;
      }
      //usleep(100);
      waiting_loop++;
    }
```

Secvență de Cod 36: Realizarea transferurilor succesive de scriere (16) și citire (24)

7 OBSTACOLE ÎNTÂMPINATE ÎN DEZVOLTAREA PROIECTULUI

În procesul de dezvoltare a acestui proiect, au apărut o serie de provocări tehnice și situații neprevăzute care au necesitat adaptare, documentare suplimentară și uneori reproiectarea anumitor etape. În această secțiune sunt prezentate cele mai relevante obstacole întâlnite și soluțiile identificate pentru a le depăși.

O problemă semnificativă a apărut după rescrierea imaginii de Linux - Ubuntu 24.04 pe placă. Primul semn de întrebare a apărut în momentul în care s-a încercat schimbarea gateware-ului FPGA, însă modificările nu aveau niciun efect. În plus, sistemul a returnat eroarea din Secvența de Cod 37.

```
    Triggering FPGA Gateware Update (/sys/kernel/debug/fpga/microchip_exec_update)
/usr/share/microchip/gateware/update-gateware.sh: line 41:
/sys/kernel/debug/fpga/microchip_exec_update: No such file or directory
```

Secvență de Cod 37: Eroare întâmpinată la rescrierea imaginii de Linux

Această eroare indică faptul că gateware-ul nu era efectiv actualizat, întrucât calea /sys/kernel/debug/fpga/ nu mai exista în sistem. Problema a apărut în urma trecerii la versiunea Ubuntu 24.04, care aduce modificări semnificative asupra structurii directoarelor și scripturilor implicate în procesul de configurare și actualizare a gateware-ului FPGA-ului.

Această problemă poate fi foarte ușor rezolvată prin utilizare unui alt script de actualizare a gateware-ului, adaptat la noua versiune a kernelului, ce se poate găsi în repository-ul oficial Microchip [34].

Un punct critic a fost atins în momentul în care, după o încercare de încărcare a unui nou gateware, sistemul a intrat într-o stare instabilă, iar conexiunea serială prin USB-C a devenit inactivă, ceea ce a făcut imposibilă orice formă de comunicare standard cu placa.

Pentru depanare a fost necesară utilizarea unui convertor USB–UART, prin care s-a putut observa că procesul de boot era întrerupt brusc, iar procesorul intra în stare de stall, afișându-se Secvența de Cod 38.

```
rcu_sched detected stalls on CPUs/tasks
```

Secvență de Cod 38: Eroare întâmpinată la accesarea resurselor indisponibile

Problema este cauzată de activarea unui overlay pentru PCIe (prin comanda run design_overlays în U-Boot) pe o configurație hardware care nu are suport PCIe activ în gateware. Kernelul încearcă să inițializeze interfața PCIe, dar, în lipsa hardware-ului necesar, procesul intră în timeout, generând blocaje și eșuarea procesului de boot. Pentru a se putea ieșii din această stare a fost necesară oprirea scriptului care activează componenta respectivă (în cazul de față PCIe) în U-BOOT și reluare procesului

de boot-are fără componenta respectivă. Soluția acestei probleme a fost găsită pe forum-ul oficial BeagleBoard [35].

Un alt aspect important ce trebuie luat în considerare este generarea bitsream-ului cu care trebuie programată placa. Acesta poate fi generat prin încărcarea proiectului pe GitLab (platformă de version-are) care permite și procesul de build al proiectului într-un container de Docker. Acest lucru este util datorită independenței de resursele locale. Însă, pot apărea situații în care server-ul devine inactiv și nu există acces la platformă. S-a întâmpinat o astfel de situație, care a îngreunat procesul de build, ducând la necesitatea realizării acestuia local. Inițial, pentru build-uirea HSS (Hart Software Services) se clonau și foloseau resursele de pe GitLab potrivite structurii template-ului de proiect oferite de documentația oficială, dar la indisponibilitatea server-ului a fost necesară utilizarea resurselor HSS oferite de Microchip [36], care presupuneau o nouă structură a proiectului.

8 EVALUAREA PERFORMANȚELOR

În acest capitol sunt analizate și comparate rezultatele obținute în urma implementării conversiei din format YUV 4:2:2 în format RGB, atât pe procesor, cât și pe FPGA (cele 3 variante - inițială, optimizată și cu DMA Controller). Obiectivul principal a fost implementarea procesului de conversie utilizând hardware dedicat, în timp ce procesorul este eliberat pentru alte activități.

Pe lângă compararea rezultatelor funcționale, este important să se înțeleagă modul în care resursele FPGA au fost utilizate pentru a implementa această funcționalitate. Analiza resurselor oferă informații despre eficiența designului hardware și poate indica posibilități de optimizare sau extindere viitoare.

8.1 RESURSE FPGA UTILIZATE

8.1.1 Implementarea inițială

În Tabelul 12 este prezentat un extras relevant din raportul de sinteză generat în urma implementării designului inițial:

Tabel 12: Utilizarea resurselor FPGA de către modulul de conversie

Tip resursă	Utilizate	Disponibile	Procent utilizat
4-input LUT (4LUT)	3395	22956	14.79%
Flip-Flops (DFF)	3549	22956	15.46%
I/O Registers	0	108	0.00%
Logic Elements	4536	22956	19.76%
DPS Blocks (18x18 MACC)	4	68	5%

Implementarea inițială a conversiei pe FPGA a implicat utilizarea unui număr redus de resurse logice. Acest lucru reflectă o proiectare eficientă, deși în cadrul implementării au fost folosite operații de înmulțire, cunoscute pentru costul lor relativ ridicat în logica configurabilă FPGA. Deși operațiile de înmulțire pot consuma o cantitate semnificativă de LUT-uri și bistabile, s-a adoptat o strategie de optimizare care a permis reducerea impactului asupra resurselor:

- Evitarea operațiilor în virgulă mobilă: conversia YUV–RGB implică coeficienți cu valori reale (ex: 1.402, 0.344, etc.), care ar necesita unități de calcul cu virgulă mobilă — foarte costisitoare pe FPGA. S-au folosit, în schimb, coeficienți întregi obținuți din valorile inițiale extinse la 32 de biți și înmulțite cu 2^{10}, eliminându-se apoi partea zecimală.

- Shiftare pentru normalizare: după efectuarea înmulțirilor cu coeficienții întregi, rezultatele au fost ajustate printr-o operație logică de shiftare la dreapta cu 10 poziții (» 10). Această abordare oferă o bună aproximare a rezultatului original, menținând totodată un consum redus de resurse.

În acest mod, s-a redus numărul de LUT-uri utilizate, la un procent de doar 14.79%, folosite în mare parte la implementarea operațiilor aritmetice de conversie.

În ceea ce privește numărul de bistabile utilizate, acesta reflectă o utilizare moderată a registrelor. Bistabilele au fost utilizate în mare parte pentru gestionarea logicii FSM-ului, respectiv încărcarea și stocarea datelor în regiștrii pentru a putea fi transmise către modulul de conversie sau recepționate de la acesta.

Lipsa utilizării registrelor I/O (0 din 108 disponibile) indică faptul că semnalele de intrare și ieșire nu au fost înregistrate direct la marginea fizică a FPGA-ului. Această alegere este justificată, întrucât designul este destinat unei comunicări interne (cu procesorul prin magistrala APB), iar sincronizarea semnalelor nu este dependentă de cerințe de timp strict la nivelul interfețelor externe.

Având în vedere numărul de elemente logice utilizate, se poate observa că design-ul ajunge să ocupe doar un procent de aproape 20% din FPGA, un rezultat bun, mai ales având în vedere complexitatea operațiilor matematice.

Astfel, designul echilibrat, în care operațiile de înmulțire nu au fost înlocuite complet cu logica combinațională (ceea ce ar fi crescut complexitatea și lungimea traseelor), dar optimizările aplicate (eliminarea zecimalelor și folosirea shiftărilor) au dus la o implementare eficientă și scalabilă. Acest echilibru a permis o utilizare rezonabilă a resurselor FPGA, păstrând spațiu pentru eventuale extinderi ale funcționalității sau integrarea altor module hardware în același design.

8.1.2 Implementarea optimizată

În Tabelul 13 este prezentat un extras relevant din raportul de sinteză generat în urma implementării designului optimizat:

Tabel 13: Utilizarea resurselor FPGA de către modulul de conversie optimizată

Tip resursă	Utilizate	Disponibile	Procent utilizat
4-input LUT (4LUT)	6233	22956	27.15%
Flip-Flops (DFF)	3988	22956	17.37%
I/O Registers	0	108	0.00%
Logic Elements	7221	22956	31.46%
DPS Blocks (18x18 MACC)	4	68	5%

Comparând tabelul de resurse FPGA utilizate între implementarea optimizată și cea neoptimizată, se poate observa utilizare unui număr aproape dublu de 4-input LUT-uri față de implementarea inițială, având loc o creștere de 83,6%, o creștere de 12,37% a numărului de bistabile utilizate și o creștere de 59,2% a elementelor logice utilizate. Creșterea semnificativă reflectă complexitatea logică suplimentară introdusă de implementarea logicii pentru buffer-ul intern de stocare a valorilor RGB convertite până la finalizarea celor 16 scrieri pe magistrala APB.

Din punctul de vedere al utilizării resurselor, implementarea optimizată a condus la o creștere semnificativă a consumului resurselor FPGA logice (LUT-uri, elemente logice și într-o mai mică măsură bistabile), în timp ce resursele DSP au rămas neschimbate. Rezultatele reflectă un compromis tipic între performanța de calcul (timpul de execuție) și utilizarea resurselor. Conversia simultană cu primirea datelor permite prelucrarea unui volum mare de date într-un interval temporal scurt. Transmiterea grupată a datelor prin magistrala APB diminuează impactul frecvenței scăzute (125 MHz).

Astfel, varianta optimizată este superioară în termeni de eficiență temporală (performanță crescută), dar impune un cost semnificativ în termeni de resurse hardware suplimentare.

8.1.3 Implementarea cu DMA Controller

În Tabelul 14 este prezentat un extras relevant din raportul de sinteză generat în urma implementării designului cu DMA Controller:

Tabel 14: Utilizarea resurselor FPGA de către modulul de conversie cu DMA Controller

Tip resursă	Utilizate	Disponibile	Procent utilizat
4-input LUT (4LUT)	18231	22956	79.42%
Flip-Flops (DFF)	10179	22956	44.34%
I/O Registers	0	108	0.00%
Logic Elements	19925	22956	86.80%
DPS Blocks (18x18 MACC)	9	68	13%

Această implementare a adus creșteri semnificative în cazul resurselor FPGA utilizate, având loc triplarea LUT-urilor, a bistabilelor și a elementelor logice utilizate. Aceste majorări se datorează în mare parte implementării protocolului AXI care permite scrierea și citirea simultană a datelor din memoria LSRAM, utilizarea blocului de memorie LSRAM (având implementată propria logică internă a protocolului AXI), realizarea buffer-elor externe de stocare a valorilor citite, respectiv scrise, realizarea buffer-ului intern pentru adaptarea fluxului de date de ieșire la magistrala AXI de 64 de biți, plus utilizarea în paralel a celor 4 instanțe convertoare.

Această implementare a fost realizată pentru a obține performanțe ridicate în termeni de viteză de procesare și disponibilitate a datelor prin magistrala AXI care operează cu date pe 64 de biți și la o frecvență dublă față de magistrala APB. Această variantă este ideală pentru aplicații unde latența redusă și debitul mare al datelor justifică investiția suplimentară în hardware logic.

Graficul comparativ din Figura 27 evidențiază evoluția consumului de resurse FPGA pentru cele trei variante de implementare ale conversiei YUV-RGB: inițială, optimizată prin magistrala APB și complexă cu DMA Controller și protocol AXI. Se observă o creștere graduală a utilizării resurselor odată cu complexitatea arhitecturii: varianta inițială are un consum redus și echilibrat, cea optimizată, dar încă folosind magistrala APB, introduce o dublare a LUT-urilor și o creștere moderată a logicii, în timp ce implementarea cu DMA Controller implică un consum cnsiderabil superior, reflectând complexitatea adăugată de protocolul AXI, FIFO-urile multiple și buffer-ul intern avansat. De asemenea, se remarcă o utilizare mai intensă a blocurilor DSP în varianta cu DMA, susținând paralelizarea extinsă a conversiei. Acest grafic subliniază clar compromisul între performanță și consumul hardware în proiectarea sistemelor embedded pe FPGA.

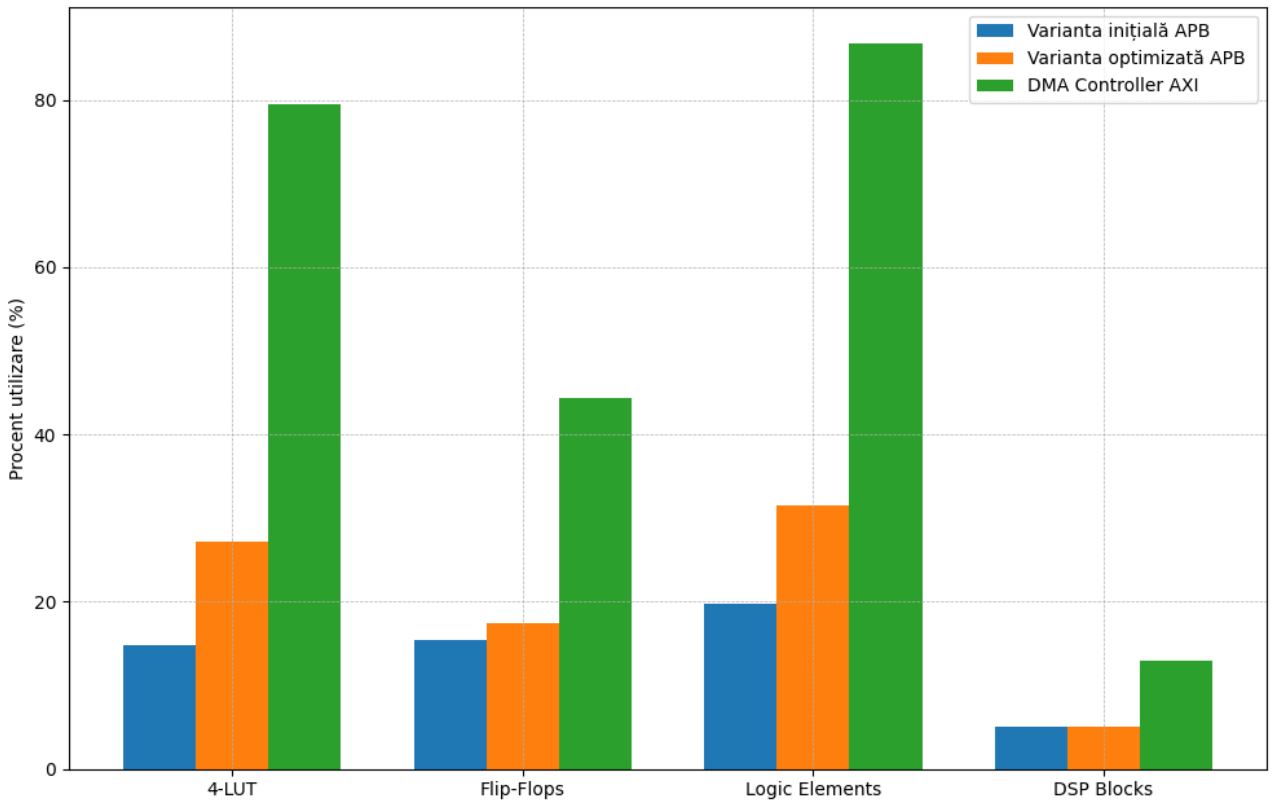


Figura 27: Compararea resurselor utilizate de cele 3 implementări FPGA

8.2 Compararea implementărilor în termeni de performanță de timp

Pentru a analiza performanța funcțională și temporală a implementării algoritmului de conversie din spațiul de culoare YUV 4:2:2 în RGB, se vor compara execuția acestuia pe procesorul sistemului cu execuția sa pe FPGA. Scopul principal este de a compara cele două variante în ceea ce privește timpul de execuție și utilizarea eficientă a resurselor, în contextul unor aplicații unde timpul de procesare este critic, precum prelucrarea imaginilor sau a semnalelor video în timp real.

Pentru o analiză obiectivă, au fost utilizate imagini de diferite dimensiuni, care acoperă o plajă variată de complexitate computațională. Fiecare imagine a fost procesată folosind aceeași metodologie, iar timpii de execuție au fost măsurați cu precizie atât pentru implementarea software pe procesor, cât și pentru cea hardware pe FPGA.

Deși FPGA-ul oferă avantaje evidente în ceea ce privește paralelismul și personalizarea arhitecturii, în cadrul platformei utilizate există limitări semnificative din punct de vedere al frecvenței de operare și al interfețelor de comunicație. În timp ce procesorul sistemului rulează la o frecvență de 625 MHz, ceea ce îi permite procesarea rapidă și accesul direct la memorie, logica FPGA funcționează la frecvențe considerabil mai mici. Magistrala APB (folosită pentru comunicarea procesor–FPGA) operează la doar 125 MHz, reprezentând un potențial factor încetinitor. Chiar și în cazul utilizării protocolului AXI, deși frecvența este dublă (250 MHz), aceasta rămâne inferioară celei a procesorului; totuși, AXI beneficiază de o lățime de date mai mare, permițând transmisia pe 64 de biți simultan, ceea ce reduce semnificativ impactul frecvenței mai mici.

Mai mult, comparând fluxul de procesare al datelor între cele două variante, implementarea pe procesor presupune succesiunea simplă: citire din memorie $\rightarrow$conversie $\rightarrow$scriere în memorie, toate operate la 625 MHz. În schimb, în cazul implementării pe FPGA, traseul este mai fragmentat și temporizat de interfețele de comunicație: citire din memorie (625 MHz) $\rightarrow$transfer spre FPGA (125 MHz APB/250 MHz AXI) $\rightarrow$conversie (125 MHz) $\rightarrow$transfer înapoi spre procesor (125 MHz) $\rightarrow$scriere în memorie (625 MHz). Astfel, chiar dacă procesarea pe FPGA poate fi paralelizată, timpii de acces și sincronizare între ceasuri impun un overhead de comunicare semnificativ, care trebuie compensat prin optimizări arhitecturale (conversii paralele, buffer intern, burst AXI).

8.2.1 Performanțe obținute de implementarea pe CPU

Pentru a evalua performanța implementării software a conversiei YUV-RGB pe procesor, au fost realizate două serii de teste, cu și fără paralelizare, utilizând OpenMP. Testele au fost efectuate pe procesorul RISC-V, care dispune de 4+1 nuclee, din cadrul SoC-ului, care funcționează la o frecvență de 625 MHz, folosind diferite niveluri de optimizare (-O0, -O1, -O2, -O3) ale compilatorului. Măsurătorile au fost efectuate pe același set de date (imagine YUV de dimensiune fixă) pentru a asigura comparabilitatea rezultatelor.

Tabel 15: Compararea timpilor de conversie realizată de procesor fără paralelizare

Rezoluție	**O0 [ms]**	**O1 [ms]**	**O2 [ms]**	**O3 [ms]**
640 x 360 (SD)	75,516	36,003	34,136	24,915
1280 x 720 (HD)	303,361	144,559	136,507	99,018
1920 x 1080 (Full HD)	1 132,072	324,551	308,342	223,163
3840 x 2160 (4K)	4 518,332	1 297,404	1 228,203	892,525

Tabelul 15 prezintă timpul de execuție în milisecunde pentru conversia completă YUV–RGB realizată secvențial pe procesor, fără utilizarea paralelizării. S-au testat patru niveluri de optimizare ale compilatorului (-O0 până la -O3) pentru cinci rezoluții reprezentative, de la 640 x 360 până la 4K. Se observă o scădere consistentă a timpului odată cu creșterea nivelului de optimizare, în special la rezoluții mari.

Tabel 16: Compararea timpilor de conversie realizată de procesor cu paralelizare

Rezoluție	**O0 [ms]**	**O1 [ms]**	**O2 [ms]**	**O3 [ms]**
640 x 360 (SD)	33,280	10,946	10,549	8,289
1280 x 720 (HD)	126,806	38,476	36,800	27,498
1920 x 1080 (Full HD)	304,055	83,997	80,671	77,683
3840 x 2160 (4K)	1 261,241	396,849	380,623	290,438

Tabelul 16 reflectă impactul utilizării paralelizării prin OpenMP asupra performanței conversiei YUV–RGB pe procesor. Testele au fost efectuate pe aceleași imagini și niveluri de optimizare ca în cazul secvențial. Se remarcă o reducere semnificativă a timpului de execuție, mai ales pentru rezoluțiile HD, Full HD și 4K, unde beneficiul paralelizării devine mai evident datorită volumului mare de date procesate.

În Figura 28 se poate observa de câte ori s-a majorat viteza de procesare folosind OpenMP ce permite utilizare a 4 nuclee în paralel, comparativ cu procesarea pe unul singur. Rezoluțiile mari (Full HD și 4K) beneficiază cel mai mult de paralelizare, obținând accelerări de peste 3×, în special pentru optimizările -O1 și -O2. -O0 (fără optimizări) permite uneori o accelerare aparent mai mare (3.72 ori

la Full HD), dar aceasta nu reflectă neapărat un cod eficient, timpul absolut de execuție rămânând mare. -O3, deși oferă cei mai mici timpi de execuție în absolut, prezintă o eficiență relativ mai scăzută a paralelizării, în special pentru rezoluții mici, unde optimizările agresive limitează beneficiile aduse de thread-uri multiple.

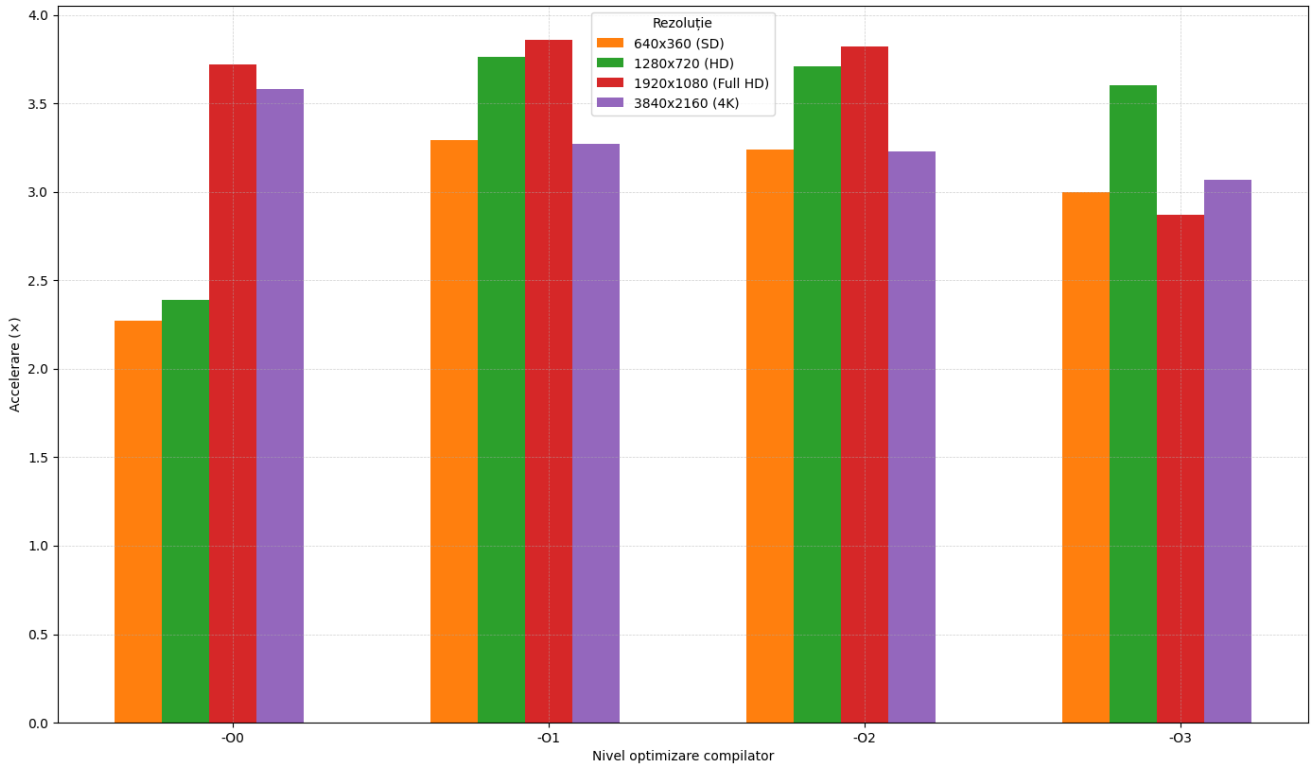


Figura 28: Grafic comparativ al creșterii vitezei conversiei pe procesor prin paralelizare

8.2.2 Performanțe obținute de implementările pe FPGA

Tabelul 17 prezintă timpul de execuție pentru conversia completă YUV–RGB pe FPGA, în cele două variante de implementare propuse folosind magistrala APB.

Tabel 17: Compararea timpilor de conversie

Rezoluție	FPGA inițial [ms]	FPGA optimizat [ms]
640 x 360 (SD)	105,008	67,736
1280 x 720 (HD)	420,532	270,659
1920 x 1080 (Full HD)	945,149	608,023
3840 x 2160 (4K)	3 780,008	2 425,578

Se observă o scădere semnificativă a timpilor de procesare în varianta optimizată pentru toate rezoluțiile analizate, de la cele reduse (640 x 360) până la 4K (3840 x 2160). Acest comportament confirmă beneficiile aduse de ajustările efectuate în arhitectura de comunicație internă și în modul de acces la resursele de memorie, chiar și în lipsa unei magistrale AXI sau a unui DMA controller.

Pe măsură ce dimensiunea datelor crește, beneficiile optimizării devin și mai evidente. La rezoluție HD (1280 x 720), timpul scade de la 420,532 ms la 270,659 ms (35,6% reducere), iar pentru Full HD (1920 x 1080), scăderea este de la 945,149 ms la 608,023 ms (35,7%). În cazul procesării unui cadru 4K, diferența devine considerabilă, cu un timp inițial de 3.780,008 ms redus la 2.425,578 ms, adică o scădere de 35,8%.

Această consistență a îmbunătățirii performanței (35% în medie) sugerează că optimizările introduse reduc semnificativ latențele interne și creșterea eficienței utilizării magistralei APB.

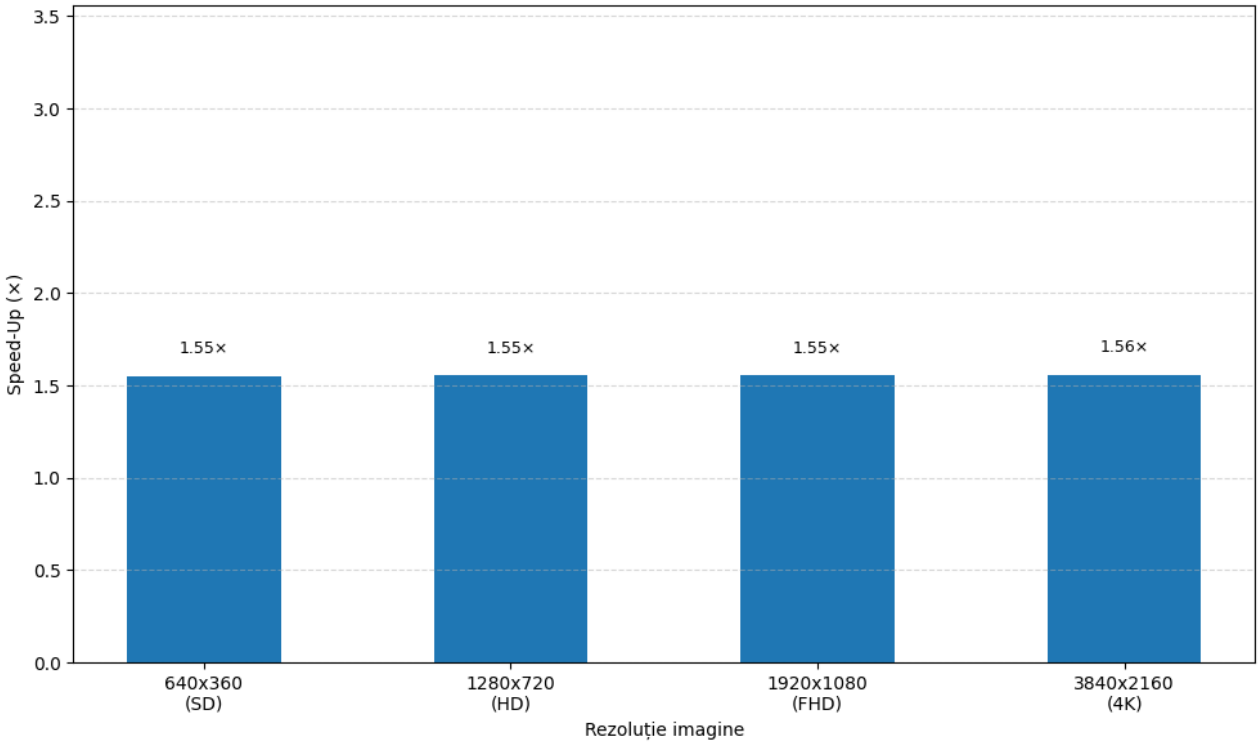


Figura 29: Grafic comparativ al creșterii vitezei conversiei pe FPGA

Compararea timpilor de conversie pentru arhitecturile hardware (FPGA inițială și optimizată) și software (CPU cu și fără paralelizare) din Figura 30 evidențiază clar diferențele de performanță în funcție de rezoluția imaginii și nivelul de optimizare aplicat.

Pentru rezoluții reduse, precum 640 x 360 (SD), procesorul paralelizat și compilat cu optimizări agresive (-O3) depășește semnificativ implementările FPGA. Diferențele devin și mai accentuate la rezoluția 640 x 360, unde CPU-ul cu OpenMP finalizează conversia în 8,289 ms, comparativ cu 67,736 ms pentru FPGA-ul optimizat. Acest rezultat subliniază eficiența procesorului în scenarii cu volum mic de date, unde overhead-ul introdus de logica de control și comunicație a FPGA-ului are o influență mai mare.

Pe măsură ce rezoluția crește, avantajul software-ului se păstrează, dar într-o măsură mai redusă. La rezoluția 1280 x 720 (HD), timpul de conversie pe FPGA optimizat este de 270,659 ms, în timp ce

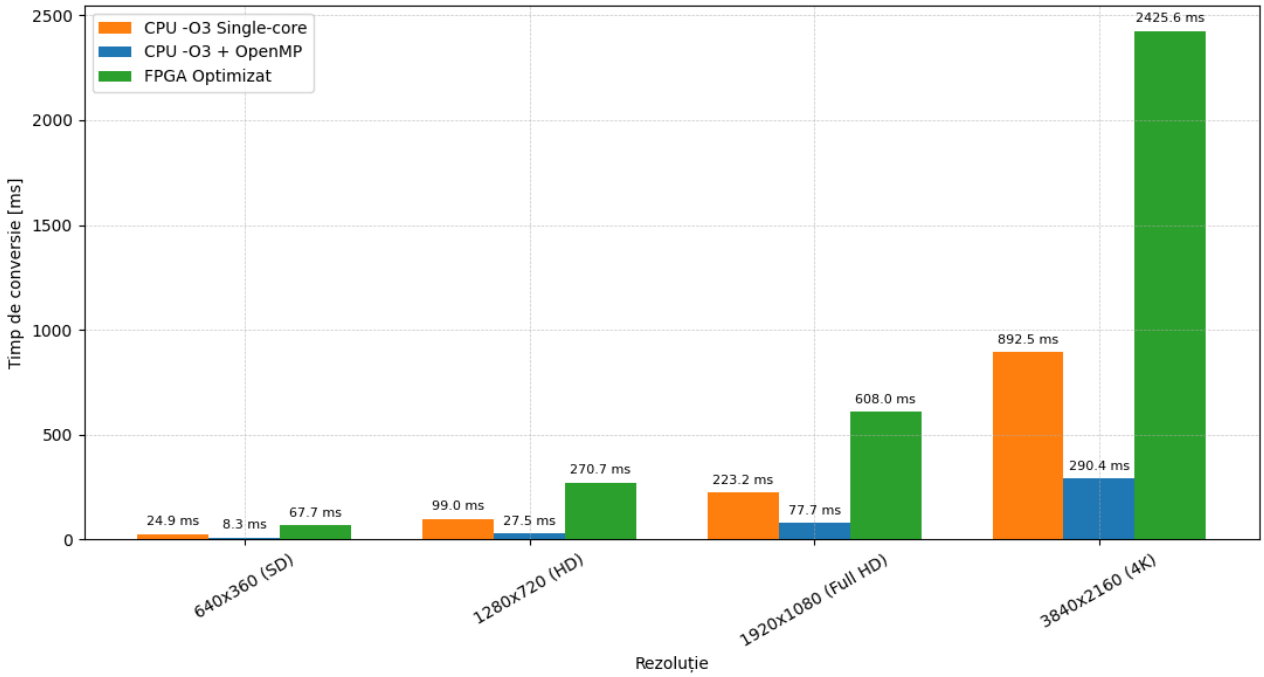


Figura 30: Comparație între timpii de conversie: CPU vs FPGA

CPU-ul paralelizat obține 27,498 ms – de aproximativ 9,8 ori mai rapid. Totuși, este de remarcat că implementarea CPU fără paralelizare (serială), chiar și cu optimizare -O3, înregistrează un timp de 99,018 ms, rămânând în continuare mai rapidă decât implementarea hardware, dar într-o proporție semnificativ mai mică.

Pentru rezoluții înalte precum 1920 × 1080 (Full HD) și 3840 x 2160 (4K), diferențele încep să se tempereze, iar implementările hardware devin mai competitive. Deși CPU-ul paralelizat păstrează un avantaj clar (77,683 ms față de 608,023 ms pentru Full HD și 290,438 ms față de 2.425,578 ms pentru 4K), arhitectura FPGA rămâne stabilă și scalabilă, cu o creștere aproximativ liniară a timpului de procesare în funcție de dimensiunea imaginii. În același timp, CPU-ul serial compilat cu -O3 obține 223,163 ms pentru Full HD și 892,525 ms pentru 4K, ceea ce înseamnă că FPGA-ul optimizat începe să se apropie de performanțele acestuia la rezoluții mari.

În acest context, rezultatele experimentale sunt perfect justificate: CPU-ul paralelizat (OpenMP + -O3) oferă timpi de execuție superiori în majoritatea cazurilor testate, profitând de o frecvență mai mare, acces rapid la memorie și optimizări software avansate. Totuși, în cazul în care se ia în considerare faptul că magistrala APB nu este folosită exclusiv de către sistemul de conversie, iar în timpul cronometrării se realizează și transferul datelor de la procesor la FPGA cu o frecvență scăzută, atunci sunt justificate rezultatele obținute. În schimb dacă se scade din timpul obținut, timpul de transfer al datelor și se folosesc la capacitate maximă blocurile DSP disponibile, este posibilă obținerea unor performanțe net superioare celor ale procesorului.

9 Concluzii

În cadrul acestui proiect a fost implementat un sistem hardware-software capabil să efectueze conversia imaginilor din format YUV 4:2:2 în format RGB 8:8:8, utilizând placa de dezvoltare BeagleV®-Fire bazată pe un procesor RISC-V și logică reconfigurabilă FPGA. Implementarea a inclus atât o variantă de conversie pe procesor (CPU), cât și 3 versiuni implementate hardware, folosind structura FPGA.

Implementarea software a algoritmului de conversie YUV la RGB utilizează tehnici de paralelizare OpenMP pentru accelerarea execuției pe procesor, fiind însă limitată în mod intrinsec de natura secvențială a instrucțiunilor. Deși optimizată prin operații pe numere întregi și utilizarea paralelismului la nivel de bucle, această implementare se confruntă cu constrângeri fundamentale ale arhitecturii CPU, precum latențele mari la accesul memoriei și utilizarea ineficientă a resurselor de calcul pentru procesarea intensivă a pixelilor. Aceste limitări fac ca abordarea hardware să devină clar preferabilă pentru aplicațiile care necesită performanță în timp real.

În ceea ce privește conversia pe FPGA, au fost propuse trei variante de implementări care să asigure eficiență atât din punct de vedere al timpului de execuție, cât și din punct de vedere al resurselor utilizate. Aceste implementări urmăresc o evoluție graduală, de la o complexitate redusă, la care se aduc optimizări pentru îmbunătățirea performanței, la complexitate ridicată adusă de utilizare a două protocoale de comunicație concomitent (AXI și APB), realizare accesului direct la memorie și implementarea unui buffer pentru care scrierile și citirile nu sunt sincronizate.

Implementarea inițială a presupus realizarea a 2 scrieri pe magistrala APB de la procesor către FPGA, urmată de conversie și apoi citirea datelor convertite. În urma simulărilor și a testării pe placă, această implementare s-a dovedit a fi ineficientă din punct de vedere al timpului de execuție, necesitând pentru o astfel de tranzacție aproximativ 20 de tacte de ceas.

Implementarea optimizată a presupus utilizarea aceleiași magistrale APB pentru transferul datelor de 32 de biți, însă de această dată s-au realizat conversii concomitent cu scrieri, iar rezultatul conversiilor s-a stocat într-un buffer intern pentru a facilita transferul intercalat a valorilor RGB.

Cea de-a treia implementare a constat în dezvoltarea unui DMA Controller care permite accesul direct la memoria LSRAM a modulului de conversie pentru a prelua date în mod continuu, a le converti și a le transfera înapoi, iar la finalizarea procesului transmițând o întrerupere către procesor pentru ca acesta să poată copia imaginea din memorie în fișierul de ieșire.

Compararea celor trei variante de implementare – CPU serial, CPU paralelizat și FPGA optimizat – a evidențiat diferențe importante în ceea ce privește eficiența temporală, utilizarea resurselor și

scalabilitatea. Variantele pe CPU, în special cea paralelizată cu OpenMP și optimizată cu -O3, s-au dovedit semnificativ mai rapide. Acest avantaj se explică prin frecvența ridicată de operare a procesorului (625 MHz), accesul direct și rapid la memorie, precum și prin posibilitatea de paralelizare software eficientă. În schimb, implementarea pe FPGA, deși teoretic superioară din punct de vedere al paralelismului hardware, este afectată de frecvențele mai reduse (125 MHz) și de latențele introduse de magistrala de comunicare (APB) dintre procesor și logica programabilă.

Concluzionând, deși implementările software pe CPU oferă un timp de execuție mai mic în multe cazuri datorită frecvenței și paralelizării, arhitectura FPGA poate aduce performanțe mai bune dacă nu se ia în considerare timpul de transfer al datelor între procesor și FPGA.

Acknowledgements

This work was supported by grants of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI - UEFISCDI, project numbers PN-IV-P8-8.1-PME-2024-0022 and PN-IV-P8-8.1-PME 2024-0025 within PNCDI IV.

The ISOLDE project, nr. 101112274 is supported by the Chips Joint Undertaking and its members Austria, Czechia, France, Germany, Italy, Romania, Spain, Sweden, Switzerland.

Bibliografie

[1] S. L. Harris, D. Chaver, L. Piñuel, *et al.*, "RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 145–150. **DOI**: 10.1109/FPL53798.2021.00032.

[2] J. Ledin and D. Farley, *Modern Computer Architecture and Organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers*. Packt Publishing Ltd, 2022.

[3] N. Simson, A. Tahigara, and W. Ecker, "A Comparative Analysis of ARM and RISC-V ISAs for Deeply Embedded Systems," in *MBMV 2024; 27. Workshop*, 2024, pp. 110–119.

[4] Mezger, Benjamin W. and Santos, Douglas A. and Dilillo, Luigi and Zeferino, Cesar A. and Melo, Douglas R., "A Survey of the RISC-V Architecture Software Support," *IEEE Access*, vol. 10, pp. 51 394–51 411, 2022. **DOI**: 10.1109/ACCESS.2022.3174125.

[5] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, Computer Science Division, EECS Department, University of California, Berkeley, 2019.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd. Morgan Kaufmann, 2021, chapter 2, chapter 4, **ISBN**: 978-0-12-820331-6.

[7] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023. **DOI**: 10.1109/ACCESS.2023.3246491.

[8] S. Ahmed and A. B. M. Harun-Ur-Rashid, "Design, Implementation and Verification of Five Stage Pipeline RISC-V Core (RV32I ISA)," in *2025 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, 2025, pp. 1–6. **DOI**: 10.1109/ECCE64574.2025.11013913.

[9] T. Jose and D. Shankar, in *2023 IEEE Space Computing Conference (SCC), title=System Model Evaluation of RISC-V Cores for improved performance and fault tolerance*, 2023, pp. 86–91. **DOI**: 10.1109/SCC57168.2023.00022.

[10] R. C. Gonzalez and R. E. Woods, "Digital Image Processing," in 3rd ed. Upper Saddle River, New Jersey: Pearson Prentice Hall, 2008, ch. 6, pp. 394–425, Chapter on Color Image Processing, **ISBN**: 9780131687288.

[11] E. Gencer. "Newton and the Science of Color." Accessed: 2025-04-24. (2020), [Online]. Available: %7Bhttps://www.thecolumbiasciencereview.com/blog/newton-and-the-science-of-color%7D.

[12] Khan Academy. "Electromagnetic spectrum." Accessed: 2025-04-24. (2025), [Online]. Available: %7Bhttps://www.khanacademy.org/science/electromagnetism/x4352f0cb3cc997f5:the-remaining-maxwell-s-equation-and-understanding-light/x4352f0cb3cc997f5:properties-of-em-waves/a/light-and-the-electromagnetic-spectrum%7D.

[13] K. N. Plataniotis and A. N. Venetsanopoulos, *Color Image Processing and Applications*. Berlin, Heidelberg: Springer, 2000, Accessed: 2025-04-24, **ISBN**: 978-3-540-66953-3. [Online]. Available: %7Bhttps://www.researchgate.net/publication/243766531_Color_Image_Processing_and_Applications%7D.

[14] Commission Internationale de l'Éclairage, *Commission Internationale de l'Éclairage Proceedings, 1931*. Cambridge: Cambridge University Press, 1932, CIE (1932).

[15] C. Abraham. "A Beginner's Guide to Colorimetry." Accessed: 2025-04-24. (2020), [Online]. Available: %7Bhttps://medium.com/hipster-color-science/a-beginners-guide-to-colorimetry-401f1830b65a%7D.

[16] M. Azimi, R. Boitard, P. Nasiopoulos, and M. T. Pourazad, in *2017 25th European Signal Processing Conference (EUSIPCO), title=Visual color difference evaluation of standard color pixel representations for high dynamic range video compression*, 2017, pp. 1480–1484. **DOI**: 10.23919/EUSIPCO.2017.8081455.

[17] B. Ahirwal, M. Khadtare, and R. Mehta, "FPGA based system for color space transformation RGB to YIQ and YCbCr," in *2007 International Conference on Intelligent and Advanced Systems*, 2007, pp. 1345–1349. **DOI**: 10.1109/ICIAS.2007.4658603.

[18] Microchip Technology Inc., *Color Space Conversion User Guide*, Accessed: 2025-04-24, Microchip Technology, 2022. [Online]. Available: %7Bhttps://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/UserGuides/ip_cores/directcores/color_space_conversion_ug.pdf%7D.

[19] R. Ji, B. Kong, F. Zheng, and J. Gao, "Color Edge Detection Based on YUV Space and Minimal Spanning Tree," in *2006 IEEE International Conference on Information Acquisition*, 2006, pp. 941–945. **DOI**: 10.1109/ICIA.2006.305862.

[20] GotoChrome. "YUV (Color Space)." Accessed: 2025-05-20. (2024), [Online]. Available: %7Bhttps://gotochrome.com/yuv-color-space/%7D.

[21] Microsoft, *Recommended 8-Bit YUV Formats for Video Rendering*, Accesat: 2025-05-20, 2023. [Online]. Available: %7Bhttps://learn.microsoft.com/en-us/windows/win32/medfound/recommended-8-bit-yuv-formats-for-video-rendering%7D.

[22] International Telecommunication Union, "Recommendation T.871: Information technology - Digital compression and coding of continuous-tone still images - JPEG File Interchange Format (JFIF)," International Telecommunication Union, Tech. Rep. T-REC-T.871-201105-I, 2011. [Online]. Available: https://www.itu.int/rec/T-REC-T.871-201105-I/en.

[23] ITU-R, *ITU-R Recommendation BT.601-5: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*, Available at: https://www.itu.int/rec/R-REC-BT.601-5-199508-I/en, 1995.

[24] B. Foundation. "Design Specifications – BeagleV-Fire." Accesat la 20 mai 2025. (2025), [Online]. Available: %7Bhttps://docs.beagleboard.org/boards/beaglev/fire/03-design.html%7D.

[25] Microchip Technology Inc., *PolarFire and PolarFire SoC FPGA Fabric User Guide*, Document Number: UG0880, 2021.

[26] R. C. Nelson, *Balenaetcher: Known good versions*, https://forum.beagleboard.org/t/balenaetcher-known-good-versions/41655, BeagleBoard Forum, Mar. 2025. (visited on 05/27/2025).

[27] BeagleBoard.org, *BeagleV-Fire-ubuntu - Jobs*, Accesat: 27 mai 2025, 2025. [Online]. Available: %5Curl%7Bhttps://git.beagleboard.org/beaglev-fire/BeagleV-Fire-ubuntu/-/jobs%7D.

[28] Luciana Mitu, *testing-gateware — Gateware development/testing for BeagleBoard (OpenBeagle)*, https://openbeagle.org/Luciana.lm11/testing-gateware, [Online; accesat în iunie 2025], 2025.

[29] Intel Corporation. "ModelSim-Intel® FPGA Standard Edition Software Version 18.1." Accessed: 2025-05-30. (2018), [Online]. Available: %7Bhttps://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html%7D.

[30] Microchip Technology Inc. "Libero SoC Design Suite v2024.2." Accessed: 2025-05-30. (2024), [Online]. Available: %7Bhttps://www.microchip.com/en-us/about/media-center/blog/2024/libero-soc-design-suite-v2024-2%7D.

[31] Microchip Technology Inc., *PolarFire SoC FPGA MSS Technical Reference Manual*, DS60001702Q, page 131, May 2025. [Online]. Available: %5Curl%7Bhttps://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/ReferenceManuals/PolarFire_SoC_FPGA_MSS_Technical_Reference_Manual_VC.pdf%7D.

[32] Microchip Technology Inc., *PolarFire® SoC Datasheet*, page 25, Microchip Technology Inc., Chandler, Arizona, USA, Jun. 2025. [Online]. Available: %7Bhttps://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/DataSheets/PolarFire-SoC-Datasheet-DS00004248.pdf%7D.

[33] Arm Limited, *AMBA AXI and ACE Protocol Specification*, ARM IHI 0022E, EAC 0 release of version E, Mar. 2013.

[34] *update-gateware.sh Script*, https://github.com/polarfire-soc/polarfire-soc-linux-examples/blob/master/gateware/update-gateware.sh, Accesat: 18.06.2025, Microchip PolarFire SoC Team, 2025.

[35] Codechaser, Iranders. "Boot failure after image update - BeagleV-Fire," BeagleBoard Forum. (2024), [Online]. Available: %7Bhttps://forum.beagleboard.org/t/boot-failure-after-image-update/40992/5%7D.

[36] Microchip Technology Inc., *hart-software-services: Hardware Abstraction Run-Time software stack for RISC-V HARTs*, https://github.com/polarfire-soc/hart-software-services, [Online; accesat în aprilie 2025], 2024.

REZUMAT

În cadrul acestui proiect a fost implementat și analizat un sistem hardware-software destinat conversiei imaginilor din formatul YUV 4:2:2 în RGB 8:8:8. Sistemul a fost dezvoltat folosind placa BeagleV®-Fire, echipată cu un procesor cu arhitectură RISC-V și logică reconfigurabilă FPGA.

Au fost abordate și comparate patru implementări distincte ale algoritmului de conversie: o variantă software executată exclusiv pe procesor și trei variante hardware implementate pe FPGA. Implementările hardware au fost dezvoltate gradual, începând cu o versiune inițială simplificată bazată pe protocolul APB, urmată de o variantă optimizată cu buffer intern și o implementare avansată ce utilizează un DMA Controller pentru acces direct și continuu la memoria FPGA, folosind concomitent protocoalele AXI și APB.

Implementarea execuției pe procesor s-a dovedit mai rapidă în contextul platformei utilizate, însă acest avantaj nu reflectă neapărat superioritatea arhitecturii software, ci mai degrabă limitele impuse de resursele de pe placa de dezvoltare, în special de lățimea și frecvența magistralei de comunicare între procesor și FPGA. În testele efectuate pe placă, transferul datelor prin APB la 125 MHz introduce un overhead semnificativ, care favorizează soluția software, unde procesorul RISC-V la 625 MHz poate prelua, converti şi scrie rapid datele în memorie fără penalizările de comunicare hardware.

Proiectul subliniază importanța logicii reconfigurabile în accelerarea sarcinilor computaționale intensive, atâta timp cât platforma hardware dispune de resurse pertinente în ceea ce privește procesarea paralelă, lățimea de bandă și comunicarea procesor-FPGA. De asemenea, proiectul accentuează relevanța și importanța explorării arhitecturilor RISC-V în mediul academic, contribuind la dezvoltarea competențelor și la aprofundarea interacțiunilor hardware-software în sisteme embedded complexe.

ABSTRACT

This project involved the implementation and analysis of a hardware-software system designed to convert images from the YUV 4:2:2 format into the RGB 8:8:8 format. The system was developed using the BeagleV®-Fire development board, featuring a RISC-V processor and reconfigurable FPGA logic.

Four distinct implementations of the conversion algorithm were approached and compared: a purely software approach executed on the processor and three hardware approaches implemented on the FPGA. The hardware implementations evolved gradually, starting with a simplified initial version based on the APB protocol, followed by an optimized version with an internal buffer, and culminating with an advanced implementation that utilizes a DMA Controller for direct and continuous memory access on the FPGA, concurrently employing AXI and APB protocols.

The implementation of execution on the processor proved faster in the context of the used platform. However, this advantage does not necessarily reflect the superiority of the software architecture but rather the limitations imposed by the development board's resources, particularly the bandwidth and clock frequency of the processor-FPGA communication bus. In on-board testing, data transfer via APB at 125 MHz introduces significant overhead, favoring the software solution where the RISC-V processor at 625 MHz can quickly fetch, convert, and write data to memory without the penalties of hardware communication.

The project highlights the importance of reconfigurable logic in accelerating computationally intensive tasks, provided that the hardware platform has relevant resources in terms of parallel processing, bandwidth, and processor-FPGA communication. Additionally, the project emphasizes the relevance and importance of exploring RISC-V architectures in the academic environment, contributing to skill development and a deeper understanding of hardware-software interactions in complex embedded systems.