UNIVERSITAEA TEHNICĂ „GHEORGHE ASACHI” DIN IAȘI FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

SPECIALIZAREA: Microelectronică, optoelectronică și nanotehnologii

Implementarea pe dispozitiv FPGA a unei arhitecturi de microprocesor

Coordonator științific:

Conf. dr. ing. Liviu Ţigăeru

Absolvent:

Stoica Andi-Ștefănel

Iași,2022

Cuprins

Motivație

Industria semiconductoarelor este piatra de temelie a companiilor ce se ocupa de proiectarea dispozitivelor integrate. Evoluția acestei industrii a dus mai departe la evoluția tuturor celorlalte industrii ce se bazeaza pe semiconductoare pentru a-și implementa cipurile fizic. În 2021, vanzarile de semiconductoare au atins un record, nu mai puțin de 555.9 miliarde dolari au fost cheltuiți în acest sector. Cel mai popular semiconductor este tranzistorul cu efect de câmp sau MOSFET, microprocesoarele moderne conțin câteva miliarde de astfel dispozitive.

În cazul procesoarelor există două mari companii ce domină piața: Intel si AMD. Deși liderul general a fost Intel, in ultimii ani AMD a cunoscut o evoluție infloritoare reușind să reducă dimensiunea tranzistoarelor la 7nm pentru procesoarele Ryzen 5, in timp ce Intel a rămas blocată la tehnologia de 14nm pentru o perioadă destul de lungă, astfel predand ștafeta de lider AMD-ului care dorește să-și consolideze poziția in fruntea producătorilor de procesoare utilizând în viitor tehnologia de 5nm.

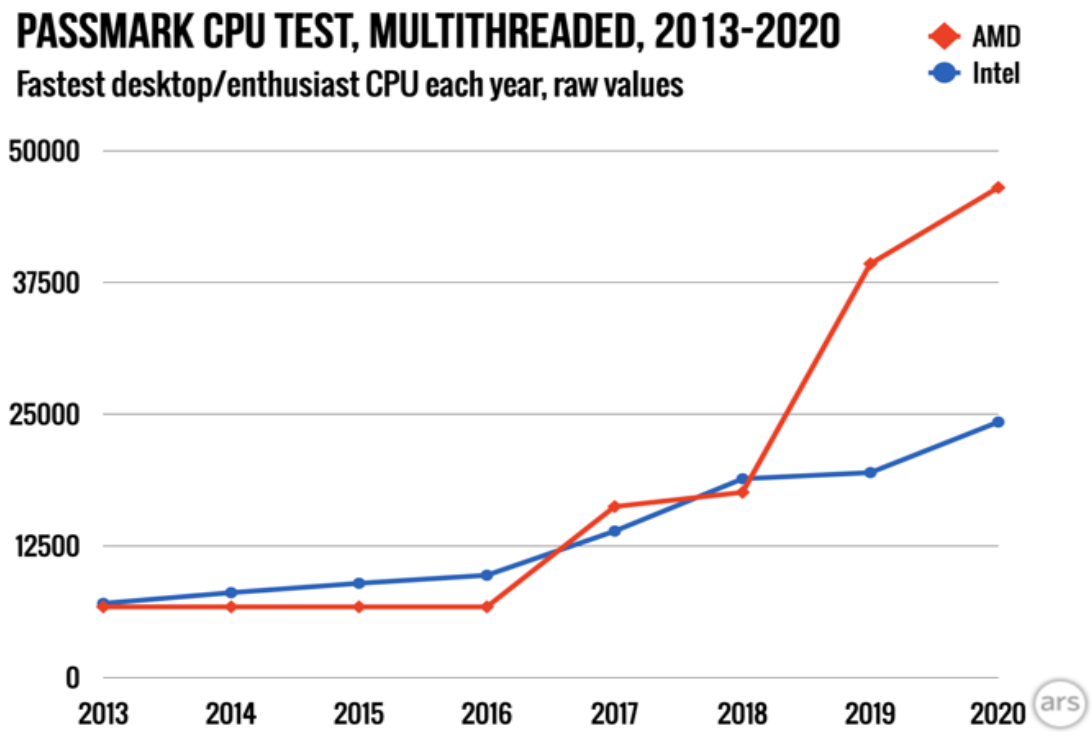


Figura 3. Evoluția AMD și Intel

Această întrecere între cei doi giganți nu se încheie cu victoriile recente ale AMD-ului, Intel dorește să-și recâștige poziția de lider, iar pentru asta propune o gamă de produse ce vor face parte din a 12-a generație Intel.

De asemenea această dezvoltare tehnologică implică un efort uman considerabil, cu fiecare nou produs ce dorește să revoluționeze piața trebuie noi idei, noi metode de implementare, pentru a obține o optimizare ce oferă o eficiență cât mai mare la un preț si consum cât mai redus. În spatele acestor produse, mii de ingineri, de la proiectanți la cei ce se ocupă de testare își unesc forțele pentru a crea ceva inovativ.

Evoluția tehnologică este inevitabilă, iar datorită acesteia, noi oportunități continuă să apară. În prezent facultățile din România nu reușesc să producă suficienți ingineri cât să acopere deficitul ce există deja pe piață. Domeniile proiectării și testării fiind o piață în continuă creștere necesită forță de muncă pentru a ocupa locurile deja vacante.

1. Introducere

Procesorul sau unitatea centrală de prelucrare a informației este circuitul electronic ce se ocupă de execuția instrucțiunilor cuprinse într-un program de calculator.

Forma, proiectarea și implementarea procesoarelor s-a schimbat de a lungul timpului, dar operațiile fundamentale au rămas aproape neschimbate. Principalele module ce formează un procesor sunt: unitatea aritmetică-logică cu rol de execuție a operațiilor aritmetice si logice, bancul de regiștri care oferă operanzi pentru unitatea aritmetică-logică, dar si stochează rezultatele acesteia și unitatea de control care dirijează modul în care datele sunt aduse din memorie, decodarea si execuția instrucțiunilor.

Cele mai moderne procesoare sunt de tipul microprocesoare si sunt implementate pe circuite integrate, cu una sau mai multe unități centrale de prelucrare a informației pe un singur cip integrat. Nucleul de procesare a informației poate executa pe mai multe fire creând astfel mai multe unități de procesare virtuale. Un circuit integrat ce conține o unitate centrală de prelucrare a informației poate de asemenea conține: memorie, interfețe periferice, dar și alte componente ale unui calculator.

Primele procesoare erau modele personalizate utilizate ca parte ale unui calculator mai mare cu caracteristici diferite față de alte calculatoare. Această metodă de proiectare a procesoarelor cu caracteristici diferite pentru compatibilitatea anumitor calculatoare a dus la dezvoltarea procesoarelor cu scopuri multiple ajungând ca acestea sa fie produse in masa.

Procesoarele tranzistoriale au fost primul mare pas dupa procesoarele ce foloseau ca elemente pentru comutație: tuburi vidate si relee. Această evoluție a fost datorată progresului tranzistoarelor ceea ce a permis construcția de procesoare mai complexe, implementate pe una sau mai multe plăci de circuite imprimate folosind componente discrete. Alte avantaje aduse de implementarea procesoarelor utilizand tranzistoare ar fi consumul redus, fiabilitatea si mărirea considerabilă a vitezei de operare.

In 1964 compania IBM introduce pe piață arhitectura IBM System/360 care a fost folosită intr-o serie de calculatoare capabile sa ruleze aceleași programe la diferite viteze si performanțe. Acest lucru a fost notabil la momentul respectiv deoarece majoritatea calculatoarelor nu erau compatibile intre ele, chiar și cele dezvoltate de același producător.

Pentru a facilita această imbunătățire, compania IBM a dezvoltat conceptul de microprogram, numit adesea si microcod care este intâlnit si in procesoarele moderne. Arhitectura System/360 s-a bucurat de un succes răsunator dominand piața pe parcursul a zeci de ani reușind de asemenea să inspire viitoarele produse dezvoltate de IBM.

Figura 1. IBM System/360 cip

De la introducerea primului microprocesor disponibil comercial, Intel 4004 în 1971, și a primului microprocesor utilizat pe scară largă, Intel 8080 în 1974, această clasă de procesoare a depășit aproape complet toate celelalte metode de implementare a unității centrale de procesare. Producătorii de minicalculatoare din acea vreme au lansat programe de dezvoltare a circuitelor integrate pentru a-și actualiza arhitecturile mai vechi, astfel au reușit să producă microprocesoare compatibile cu setul de instrucțiuni care nu erau compatibile până în acel punct.

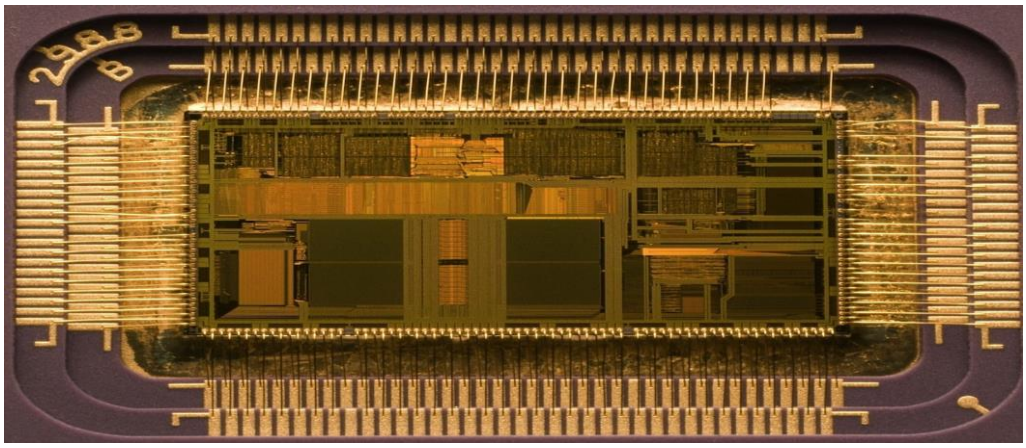


Figura 2. Vedere la microscop a microprocesorului Intel 80486DX2

1.1 Arhitecturi de microprocesoare

Când discutăm despre modul în care se accesează memoria la nivelul unității centrale de prelucrare a informației avem două modele pe care le putem lua în considerare: arhitectura Von Neumann si arhitectura Harvard.

Arhitectura von Neumann a fost făcută public prima dată de către John Von Neumann în 1945. Această arhitectură se bazează pe conceptul de memorie comună, în care datele pentru instrucțiuni și datele de program sunt stocate în aceeași memorie. Acest mod de proiectare este întâlnit și în zilele de azi în majoritatea calculatoarelor. Într-un sistem cu această arhitectură, instrucțiunile și datele fiind stocate în aceeași memorie, ele sunt aduse folosind aceeași cale de date, așadar un procesor nu poate simultan citi o instrucțiune și scrie sau citi date din memorie.

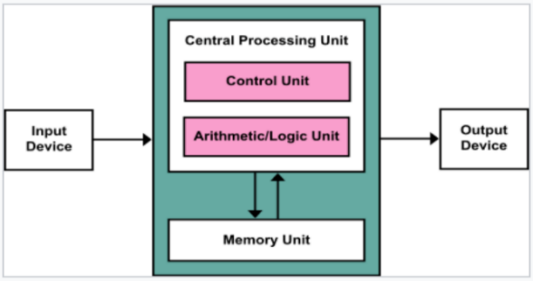


Figura 4. Arhitectura von Neumann

Arhitectura Harvard este arhitectura calculatorului digital al cărei proiectare se bazează pe conceptul de a exista stocare separată și magistrale separate pentru instrucțiuni și date. Această arhitectură a fost practic dezvoltată pentru a depăși neajunsurile arhitecturii von Neumann. Procesoarele moderne pentru utilizator par a fi mașinării von Neumann, având codul programului și datele stocate în memoria principală. Din motive de performanță, intern și neobservabil pentru utilizator majoritatea proiectărilor adoptă separarea cache-urilor din procesor pentru instrucțiuni și date, cu trasee separate în procesor pentru fiecare, astfel procesorul poate in paralel să citească instrucțiuni, dar și să facă accese la memorie. Aceasta este o formă a arhitecturii Harvard modificate ce permite accesarea instructiunilor ca fiind date.

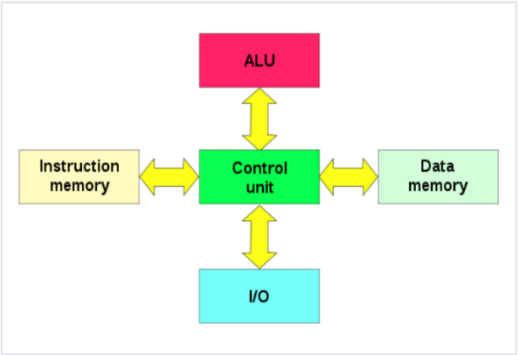


Figura 5. Arhitectura Harvard

De asemenea există două abordări prin care este compus setul de instrucțiuni, dar și modul cum este configurată partea hardware a sistemului de execuție: RISC și CISC.

Principalul scop al procesoarelor RISC este ca proiectarea părții hardware să fie simplificată folosind un set de instrucțiuni care este compus din câțiva pași simpli pentru: încărcare, evaluare și stocarea operațiilor. Nu este necesar să existe mai puține instrucțiuni, ceea ce primează este modul cum acestea sunt folosite. Prin această implementare numărul ciclilor pe instrucțiune executată este redus, dar numărul instrucțiunilor pe program va fi mai mare.

Principalul scop al procesoarelor CISC este ca o singură instrucțiune poate face intreaga parte de: încărcare, evaluare și stocarea operațiilor. Scopul abordării CISC este de a reduce numărul de instrucțiuni dintr-un program, dar această abordare va crește numărul de ciclii necesari execuției unei instrucțiuni.

RISC	CISC
-pune accent pe software	-pune accent pe hardware
-număr mare de instrucțiuni	-număr mic de instrucțiuni
-instrucțiuni simple, standartizate	-instrucțiuni complexe, dimensiuni variabile
-instrucțiuni executate pe o perioada de ceas	-instrucțiuni executate pe mai mulți cicli
-utilizare intensă a memoriei RAM	-utilizare eficientă a memoriei RAM
-puțini ciclii pe secundă, dar o dimensiune a codului mare	-dimensiune redusă a codului, dar număr crescut de cicli pe secundă

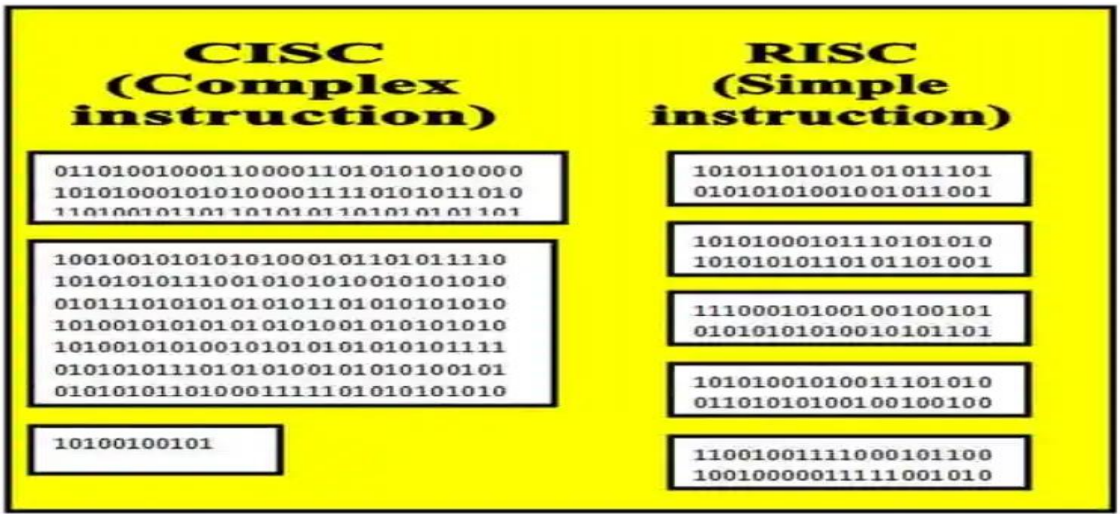


Figura 6. Seturi de instrucțiuni CISC și RISC, format binar

1.2 Descriere generală FPGA

FPGA(Field-programable gate array) este un circuit integrat proiectat să poată fi reconfigurat de către utilizator după fabricare, de aici vine si noțiunea de matrice programabilă. Configurarea dispozitivului FPGA se face folosind un limbaj de descriere hardware, similar folosit cu cel pentru un dispozitiv unic programabil(ASIC).

Dispozitivul FPGA conține o matrice de blocuri logice programabile și o ierarhie de interconexiuni reconfigurabile care permit ca blocurile să fie conectate împreună. Blocurile logice pot fi configurate să execute funcții combinaționale complexe sau să se poarte ca simple porți logice AND și XOR. În majoritatea dispozitivelor FPGA, blocurile logice includ de asemenea și elemente de memorie, care pot fi simple bistabile sau blocuri de memorie mai complexe.

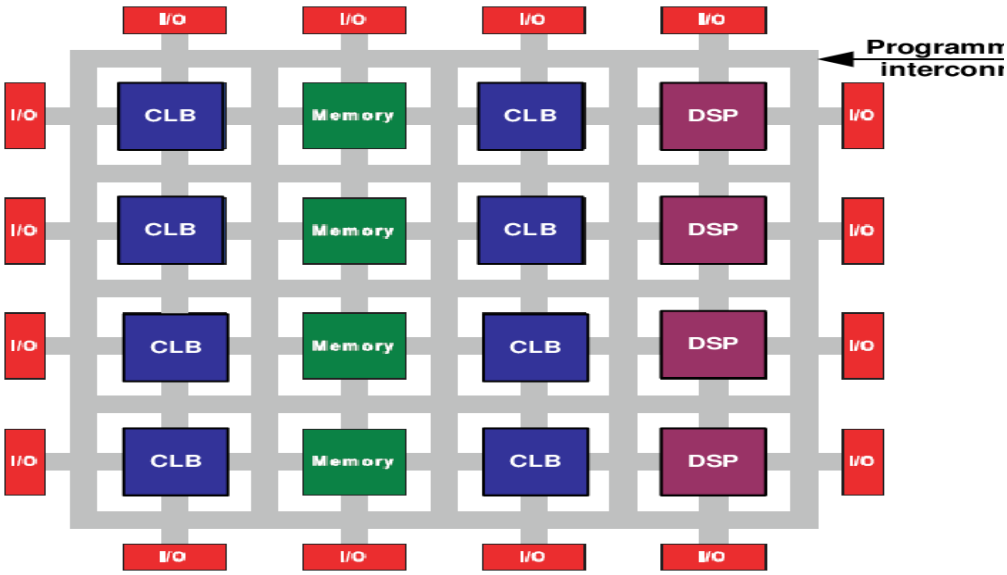


Figura 7. Structura internă a unui FPGA

CLB(configurable logic blocks) este un element fundamental al dispozitivului FPGA. Aceste blocuri pot fi configurate de către inginer pentru a oferi porți logice reconfigurabile.

Memoria flash este un loc de stocare electronic nevolatil a datelor care poate fi șters și reprogramat electric. Cele două tipuri de memorie flash sunt: NOR flash și NAND flash. Ambele folosesc același design de celulă, constând din MOSFET cu poartă flotantă.

DSP(digital signal processor) este un cip cu arhitectura optimizată pentru nevoile procesării digitale de semnal. Procesoarele pentru prelucrarea de semnal sunt fabricate din cipuri cu circuite integrate MOS. Scopul principal al unui DSP este de a măsura, filtra sau comprima semnale analogice continue din lumea reală.

Rivalii de lungă durată din industrie Xilinx(parte din AMD) și Altera(parte din Intel) sunt liderii pieței FPGA. În 2016 ei controlau 90% din piață.

Cateva familii de FPGA ce aparțin de Xilinx: Virtex, Spartan, Artix, Kintex, Zynq.

Cateva familii de FPGA ce aparțin de Altera: Stratix, Arria, Cyclone, Quartus, Enpirion.

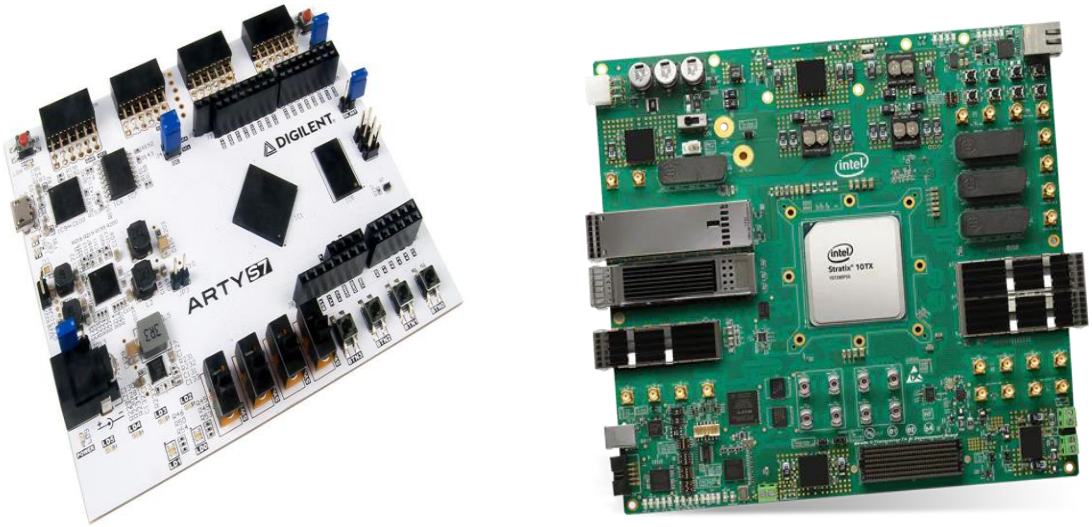


Figura 8. Familia Arty(Xilinx) contra familia Stratix(Intel)

Dispozitivele FPGA contemporane au mari resurse de porți logice și blocuri RAM pentru a putea implementa calcule digitale complexe. Cea mai mare provocare pentru proiectanți o constituie verificarea sincronizării corecte a datelor valide dintre timpul de setup și timpul de hold. FPGA poate fi folosit pentru implementarea oricărei funcții logice ce poate fi implementată pe un ASIC. Capacitatea de a actualiza funcționalitatea dispozitivului, reconfigurarea unei părți a designului, oferă avantaje pentru implementarea unei game largi de aplicații.

Un FPGA poate fi folosit pentru a rezolva orice problemă ce implică calcule. Acest lucru este dovedit de faptul că FPGA-urile pot fi folosite pentru a implementa un microprocesor.

1.3 Limbaje descriere hardware

În ingineria calculatoarelor, un limbaj de descriere hardware(HDL) este un limbaj specializat folosit pentru a descrie structura și comportamentul circuitelor electronice, cel mai frecvent al circuitelor logice digitale.

Un limbaj de descriere hardware permite o descriere precisă ce oferă posibilitatea pentru analiza și simularea circuitului electronic descris. De asemenea permite sinteza circuitului electronic prin specificarea componentelor electronice ce compun circuitul și modul cum aceste componente sunt conectate împreună. În final rezultând un set de măști pentru a produce circuitul integrat sub formă fizică.

Spre deosebire de un limbaj de programare, într-un limbaj de descriere hardware noțiunea de timp este explicit inclusă. Ca limbaje de descriere hardware avem: Abel, ISP, Verilog, VHDL.

Abel(Advanced Boolean Expresion Language) a fost dezvoltat în anii 1983 de către Data-I/O și este orientat spre circuitele logice programabile, neadecvat pentru descrieri complexe ce depășesc automatele cu stări finite.

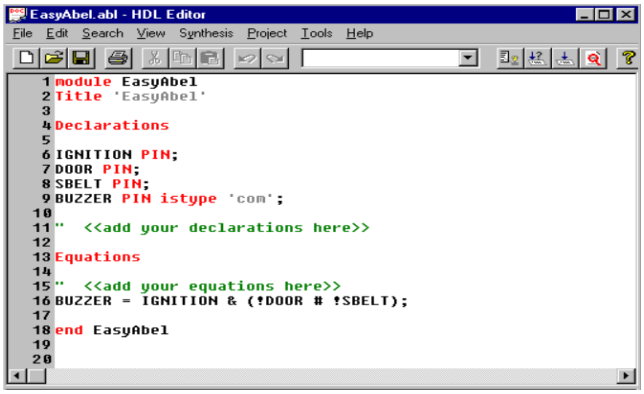


Figura 9. Sintaxa ABL

ISP(Instruction set processor) a fost dezvoltat în anii 1977 fiind un proiect de cercetare la CMU este utilizat pentru simulare, dar nu conține instrumente de sinteză. Au fost dezvoltate două noi implementări ale limbajului de bază ISP: ISPL și ISPS.

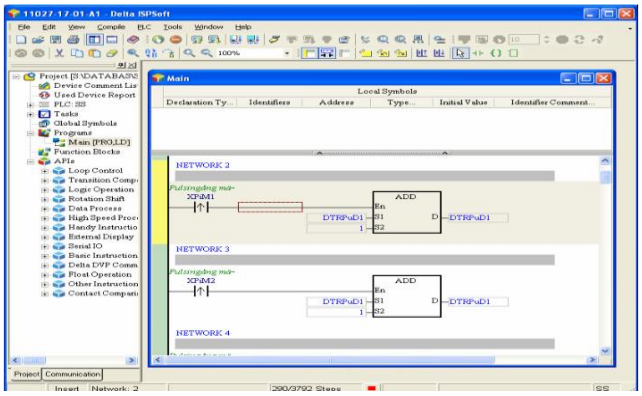


Figura 10. ISPSoft

Verilog a fost dezvoltat în anii 1985 de către compania Gateway ulterior preluată de Cadence, este similar cu limbajele Pascal si C, eficient și ușor de scris. Standardul IEEE pentru Verilog a apărut în 1995. Limbajul este utilizat în concepția asistată pe calculator a circuitelor integrate sau pentru configurarea FPGA-urilor.

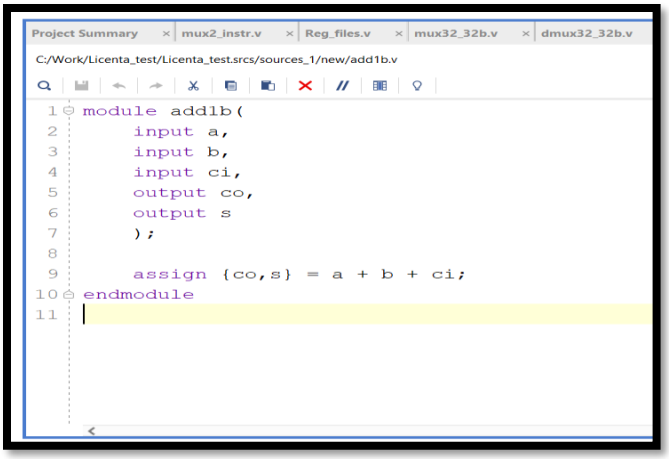


Figura 11. Sintaxa Verilog

VHDL(Very high-speed integrated circuit hardware description language) a fost dezvoltat în anii 1987 de către DoD(Department of Defense) este un limbaj orientat spre reutilizarea codului și o mentenanță ușoară cu semantică vizibilă, la fel ca Verilog este un standard IEEE.

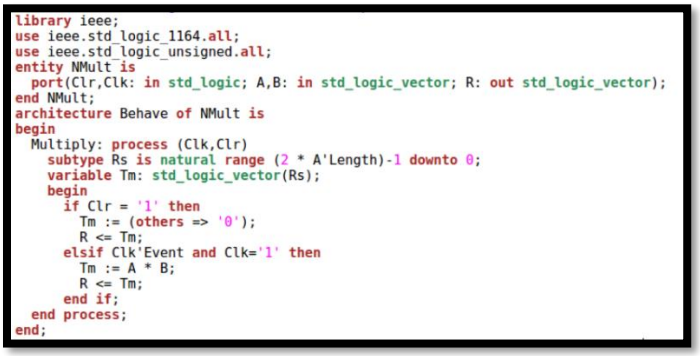


Figura 12. Sintaxa VHDL

Folosind limbajul de descriere hardware corezpunzător, un program numit sintetizator, poate transpune instrucțiunile din limbajul respectiv într-o listă de instanțe ce reproduce comportamentul descris. Sintetizatoarele logice digitale folosesc în general marginile semnalului de ceas ca modalitate de cronometrare a circuitului, ignorând orice construcție de sincronizare. Abilitatea de a avea capacitatea de sintetizare nu este o caracteristică întâlnită doar la limbajele de descriere hardware.

Ca rezultat al câștigurilor de eficiență realizate, majoritatea circuitelor moderne proiectate digital fac uz de limbajele HDL. Majoritatea proiectelor pornind de la un set de cerințe ce trebuie îndeplinite sau o diagramă arhitecturală de nivel înalt. Proiectanții folosesc adesea limbaje precum Perl pentru a genera automat structuri repetitive de circuite în limbaj HDl.

2. MIPS

MIPS(Microprocessor without Interlocked Pipelined Stages) este o familie de microprocesoare cu set redus de instrucțiuni. A fost dezvoltat de MIPS Computer Systems, companie cu sediul în S.U.A.

Arhitectura MIPS a dezvoltat pe parcursul anilor câteva extensii. MIPS-3D este un simplu set de instrucțiuni cu virgulă mobilă dedicat aplicațiilor 3D. MDMX este un set de instrucțiuni cu virgulă mobilă ce face uz de regiștri cu diumensiunea de 64 biți. MIPS16e care oferă posibilitatea comprimării setului de instrucțiuni oferind astfel mai mult spațiu. MIPS MT care oferă capacitatea de execuție a microprocesorului pe mai multe fire .

2.1 Prezentarea generală a arhitecturii MIPS

MIPS este o arhitectură cu un set de instrucțiuni divizat în două categorii: instrucțiuni cu acces la memorie și instrucțiuni de calcul(aritmetice logice). Exceptând instrucțiunile ce lucrează cu accese la memorie, celelalte instrucțiuni fac uz de regiștrii, exceptând instrucțiunea de salt care nu se încadrează in niciuna din categoriile menționate.

Arhitectura MIPS utilizează 32 de regiștri cu scop general, dintre care registrul 0 este legat la masă și nu se pot face scrieri la acesta. Program counter-ul folosește 32 de biți. Cei doi mai puțin semnificativi biți mereu vor conține zero deoarece instrucțiunile MIPS au lungimea de 32 de biți și sunt aliniate la limitele cuvântului. Microprocesoarele MIPS prezintă 5 nivele:

-nivelul IF (instruction fetch): se calculează adresa instrucțiunii ce trebuie citită din cache-ul de instrucțiuni sau din memoria principală și se aduce instrucțiunea.

-nivelul ID (instruction decode) - se decodifică instrucțiunea adusă și se citesc operanzii din setul de regiștri generali. În cazul instrucțiunilor de salt, pe parcursul acestei faze se calculează adresa de salt.

-nivelul UAL (execute address) - se execută operația UAL asupra operanzilor selectați în cazul instrucțiunilor aritmetico-logice; se calculează adresa de acces la memoria de date pentru instrucțiunile LOAD / STORE.

-nivelul MEM (memory acccess) - se accesează memoria cache de date sau memoria principală, numai pentru instrucțiunile LOAD/STORE. Acest nivel pe funcția de citire poate

pune probleme datorate neconcordanței între rata de procesare și timpul de acces la memoria principală.

-nivelul WB (write back) - se scrie rezultatul UAL sau data citită din memorie (în cazul unei instrucțiuni LOAD) în registrul destinație din setul de regiștri generali al microprocesorului.

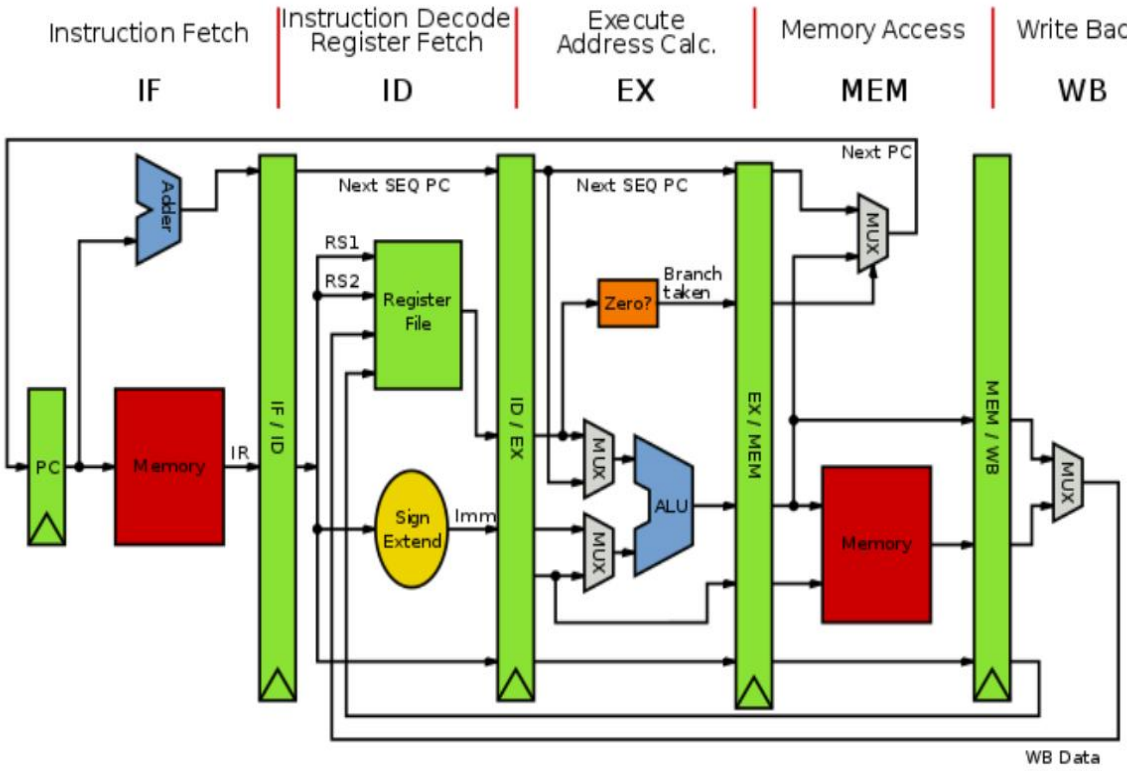


Figura 13. Nivele MIPS

2.2 Câmpurile MIPS

Instrucțiunile pot fi de trei tipuri: R(regiștri), I(valori imediate), J(salt). Fiecare instrucțiune incepe cu un câmp de 6 biți în interiorul căruia este codat în format binar tipul instrucțiunii. Restul câmpurilor sunt sub forma următoare:

Type	-31-	format (bits)				-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Figura 14. Formatul instrucțiunilor MIPS

MIPS are instrucțiuni ce incarcă sau stochează date cu dimensiune de 8 biți, 16 biți și 32 biți. Un singur mod de adresare este suportat: bază plus deplasare. Având în vedere că MIPS este o arhitectură pe 32 de biți, încărcarea datelor cu dimensiuni mai mici de 32 biți implică extensia de semn cu completare multiplicând bitul de semn până la dimensiunea de 32 biți. Operanzii fără semn vor avea parte de o extindere cu biți de zero, iar operanzii cu semn vor fi extinși cu biți de unu.

Instrucțiunile ce se ocupă cu accesele de memorie, folosesc o adresă formată prin adunarea unei baze ce este reprezentată de valoarea stocată intr-un registru și o valoare imediată. Este necesar ca toate accesele la memorie să fie aliniate la dimensiunea lor naturală(32 biți) altfel o excepție va fi semnalată.

Următoarele sunt cele trei formate pentru setul de instrucțiuni de bază:

Instrucțiunile de tip R folosesc doi operanzi reprezentați de regiștri de uz general(rs și rt) și scriu rezultatul într-un alt registru de uz general(rd), mai conțin un câmp pentru shiftarea rezultatului și un câmp pentru funcția dorită.

- op: codul instrucțiunii (opcode), 6 biți
- rs: primul operand sursă, 5 biți
- rt: al doilea operand sursă, 5 biți
- rd: operandul destinație, 5 biți
- shamt: shift amount, 5 biți
- funct: funcția operației (selectează varianta specifică de operație pentru un anumit opcode), 6 biți.

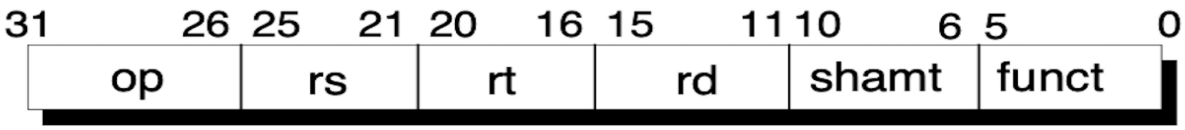


Figura 15. Format de tip R

În cazul instrucțiunilor de tip I formatul este ușor diferit față de cele de tip R, existând un câmp pentru operație, 2 câmpuri pentru regiștrii utilizați(rs și rt) și un câmp pentru valoarea imediată. Acest format este folosit de 3 categorii de instrucțiuni: instrucțiuni de acces la memorie, instrucțiuni de salt condiționat și instrucțiuni aritmetico-logice cu valori imediate.

- op: opcode, 6 biți
- rs: primul operand sau registru bază, 5 biți
- rt: al doilea operand sau registru destinație, 5 biți
- imm: adresă de memorie, de salt sau valoare imediată, 16 biți.

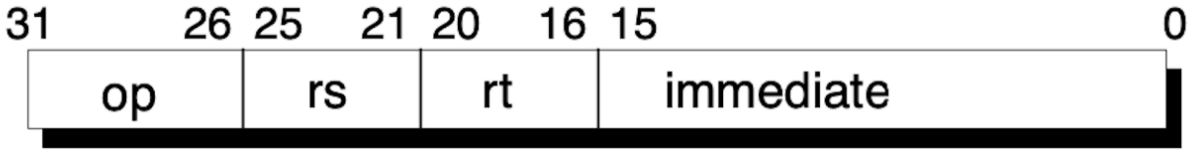


Figura 16. Format de tip I

Instrucțiunile de tip J sunt utilizate pentru salt necondiționat și fac uz de 2 câmpuri: unul pentru operație și unul pentru adresă.

- op: opcode, 6 biți
- adresă: adresa care este adăugată la PC+4 pentru a forma adresa de salt necondiționat, 26 biți.

Figura 17. Format de tip J

2.3 Setul de instrucțiuni propus spre implementare

Instrucțiunile și semnalele ce vor fi setate în funcție de codul operației și valoarea funcției sunt următoarele:

R-Type	Opcode	Function	RegDst	Jump	Branch	MemRead	MemtoReg	ALUop	MemWrite	ALUsrc	RegWrite	ALU Control
AND	6'b000000	6'b100100	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b000000
OR	6'b000000	6'b100101	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b000001
NOR	6'b000000	6'b100011	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b001100
ADD	6'b000000	6'b100000	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b000010
SUB	6'b000000	6'b100010	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b000110
SLT	6'b000000	6'b101010	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b000111
MULL	6'b000000	6'b101011	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b01xxxx
XOR	6'b000000	6'b110001	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b10xxxx
Sht_L	6'b000000	6'b110011	1'b1	1'b0	1'b0	1'b0	1'b0	3'b000	1'b0	1'b0	1'b1	6'b11xxxx
I-Type	Opcode	Function	RegDst	Jump	Branch	MemRead	MemtoReg	ALUop	MemWrite	ALUsrc	RegWrite	ALU Control
lw	6'b100011	6'bx	1'b0	1'b0	1'b0	1'b1	1'b1	3'b001	1'b0	1'b1	1'b1	6'b000010
sw	6'b101011	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b001	1'b1	1'b1	1'b0	6'b000010
addi	6'b001000	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b001	1'b0	1'b1	1'b1	6'b000010
beq	6'b000100	6'bx	1'bx	1'b0	1'b1	1'b0	1'b0	3'b010	1'b0	1'b1	1'b0	6'b000110
subi	6'b000001	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b011	1'b0	1'b1	1'b1	6'b000110
andi	6'b000011	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b100	1'b0	1'b1	1'b1	6'b000000
ori	6'b000111	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b101	1'b0	1'b1	1'b1	6'b000001
mulli	6'b001111	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b110	1'b0	1'b1	1'b1	6'b01xxxx
xori	6'b011111	6'bx	1'b0	1'b0	1'b0	1'b0	1'b0	3'b111	1'b0	1'b1	1'b1	6'b10xxxx
J-Type	Opcode	Function	RegDst	Jump	Branch	MemRead	MemtoReg	ALUop	MemWrite	ALUsrc	RegWrite	ALUControl
Jmp	6'b000010	6'bx	1'bx	1'b1	1'bx	1'bx	1'bx	3'bx	1'bx	1'bx	1'bx	6'bx

În cazul instrucțiunilor de tip R, acestea vor face uz în special de bancul de regiștri de unde vor fi utilizați 3 regiștri: 2 ca surse, 1 epntru stocarea rezultatului și unitatea aritmetică-logică ce va efectua operația decodată. Separat de operațiile cu date se va calcula următoarea adresă.

Albastru: calea de calcul a următoarei adrese.

Negru: semnale control setate.

Colorat: căile de prelucrare a datelor.

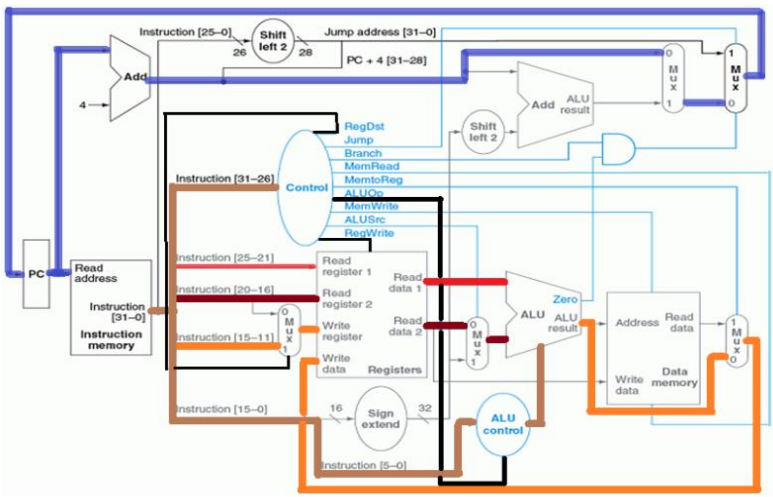


Figura 18. Căile utilizate de instrucțiunile de tip R

În cazul instrucțiunilor de stocare și încărcare din memorie avem următoarele căi și semnale utilizate:

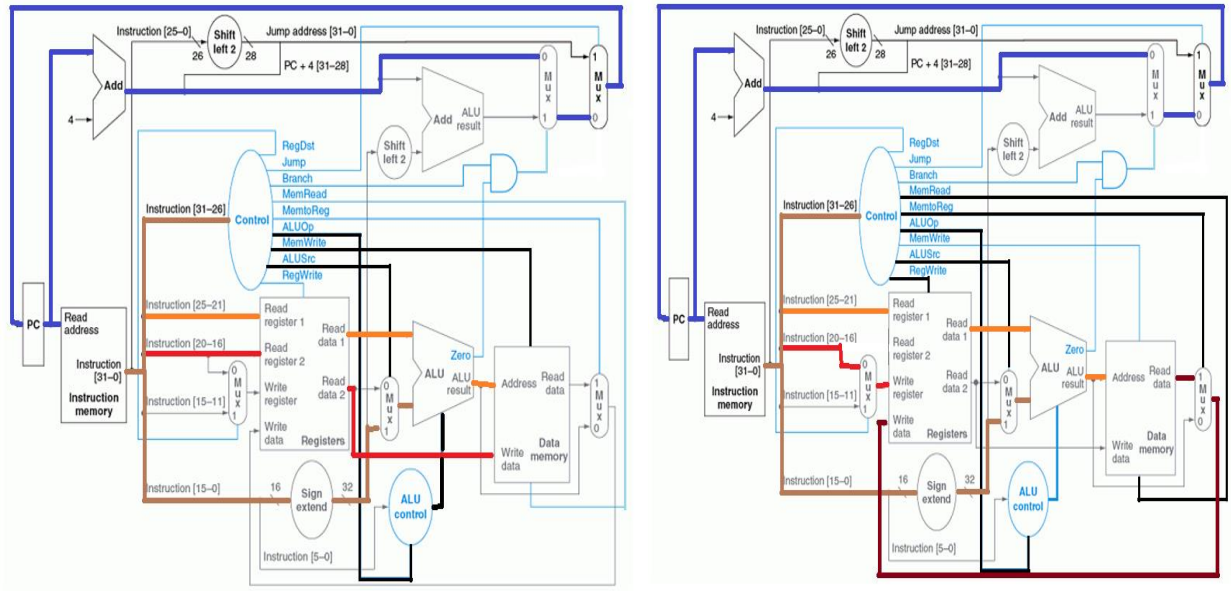


Figura 19. Căi utilizate de instrucțiunile SW(stânga) și LW(dreapta)

Albastru: calea de calcul a următoarei adrese.

Negru: semnale control setate.

Colorat: căile de prelucrare a datelor.

Pentru instrucțiunea SW, după decodificarea instrucțiunii vor fi utilizați 2 regiștri: unul stochează valoarea bazei adresei unde vor fi stocate datele, al doilea va conține datele pe care dorim să le stocăm. Calculul adresei se face cu ajutorul unității aritmetico-logice: bază+deplasare. În final data va fi scrisă în memoria pentru date.

Pentru instrucțiunea LW, după decodificare instrucțiunii vor fi utilizați 2 regiștri: unul stochează valoarea bazei adresei, iar în cel de al doilea vor fi încărcate datele din memorie. Calculul adresei se va face: bază+deplasare. De la adresa calculată se va accesa zona de memorie din memoria de date care va fi transmisă către bancul de regișstri unde va fi încărcată.

Cele 2 instrucțiuni sunt complementare, pentru a putea încărca din memorie în primă fază este nevoie ca memoria să fie scrisă, iar acest lucru se realizează cu instrucțiunea de SW.

Instrucțiunea condițională de salt condiționat utilizează următoarele căi:

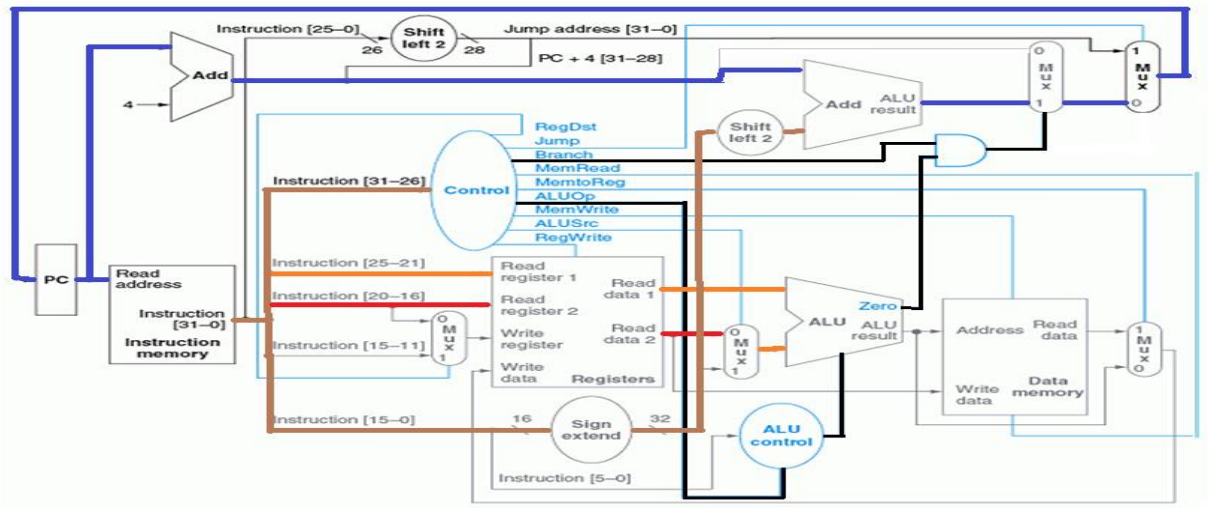


Figura 20. Căi utilizatede instrucțiunea BEQ

În cadrul instrucțiunii beq se compară valoarea din 2 regiștri sursă printr-o operație de scădere, dacă rezultatul este 0 inseamnă că operanzii sunt egali atunci condiția de salt este îndeplinită și se trece la calcularea adresei unde se va face saltul utilizând valoarea imediată, în caz contrar se trece la instrucțiunea următoare.

Instrucțiunea de salt necondiționat utilizează următoarele căi:

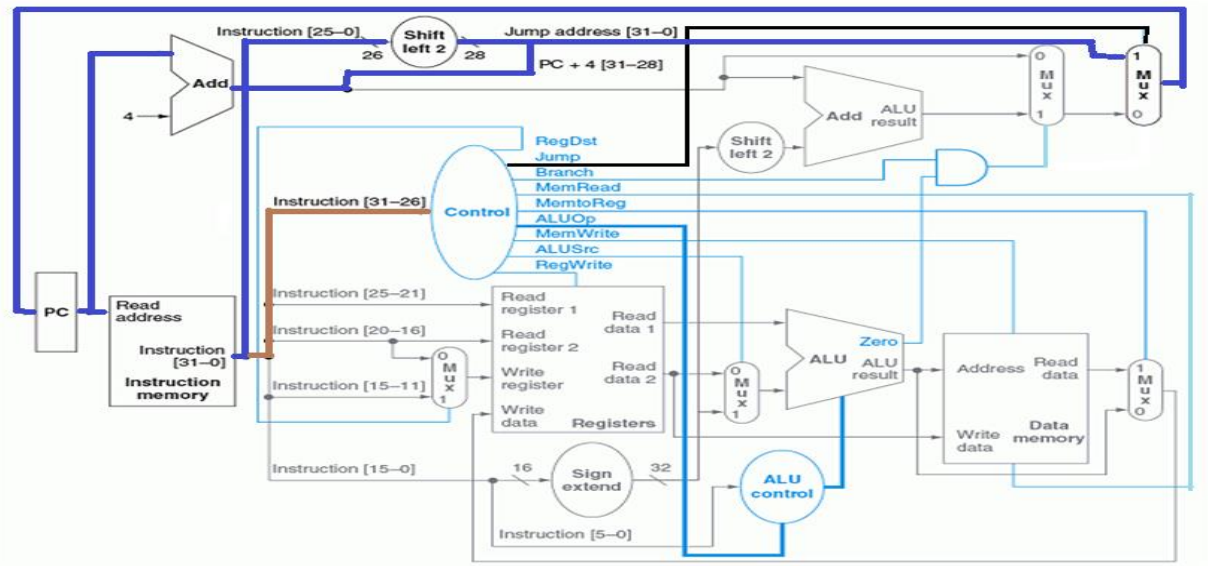


Figura 21. Căi utilizate de instrucțiunea Jmp

Adresa la care se face saltul provine de la valoarea de 26 biți din câmpul instrucțiunii.

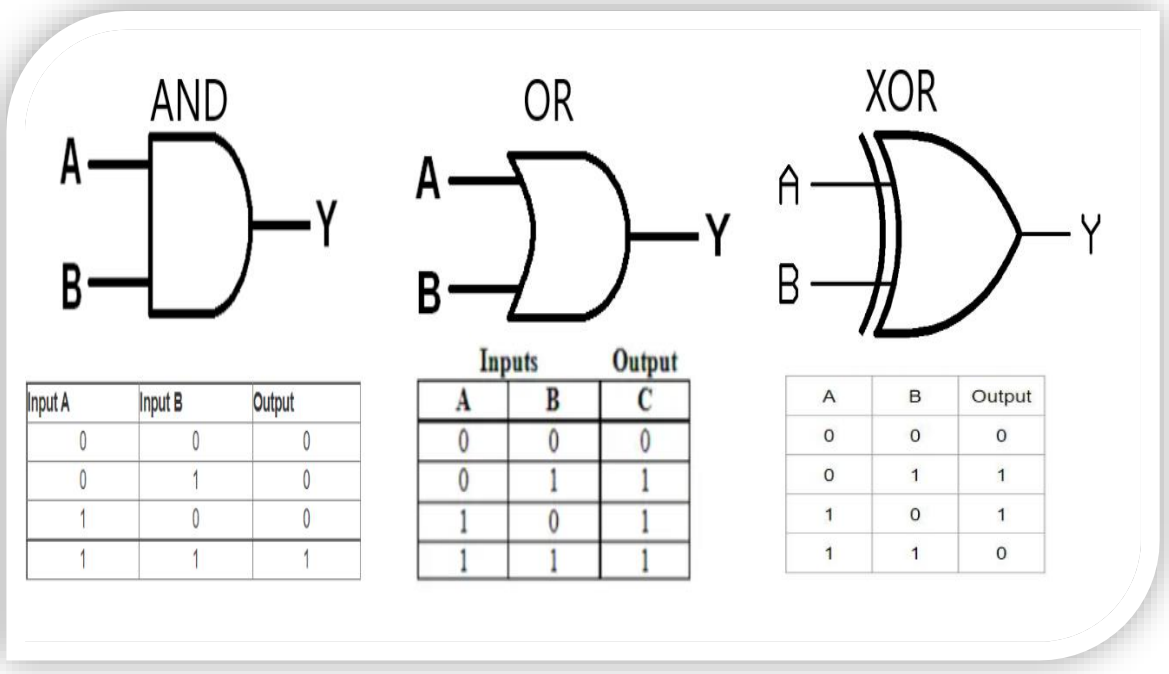
3. Implementarea procesorului

Procesorul va fi descris utilizând limbajul Verilog, iar codul rezultat va fi încărcat pe un dispozitiv FPGA.

3.1 Unitatea Aritmetică-logică

Pentru realizarea modulului TOP al unității aritmetică-logică s-a început de la circuite de bază: porți logice, celulă de adunare pe un bit, multiplexor cu 4 intrări. În continuare voi prezenta modul de funcționare al acestor circuite și realizarea legăturilor pentru obținerea unui UAL pe 1 bit.

Porțile logice utilizate sunt: AND, OR, XOR ale căror simboluri și tabele de adevăr sunt prezentate mai jos.



Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

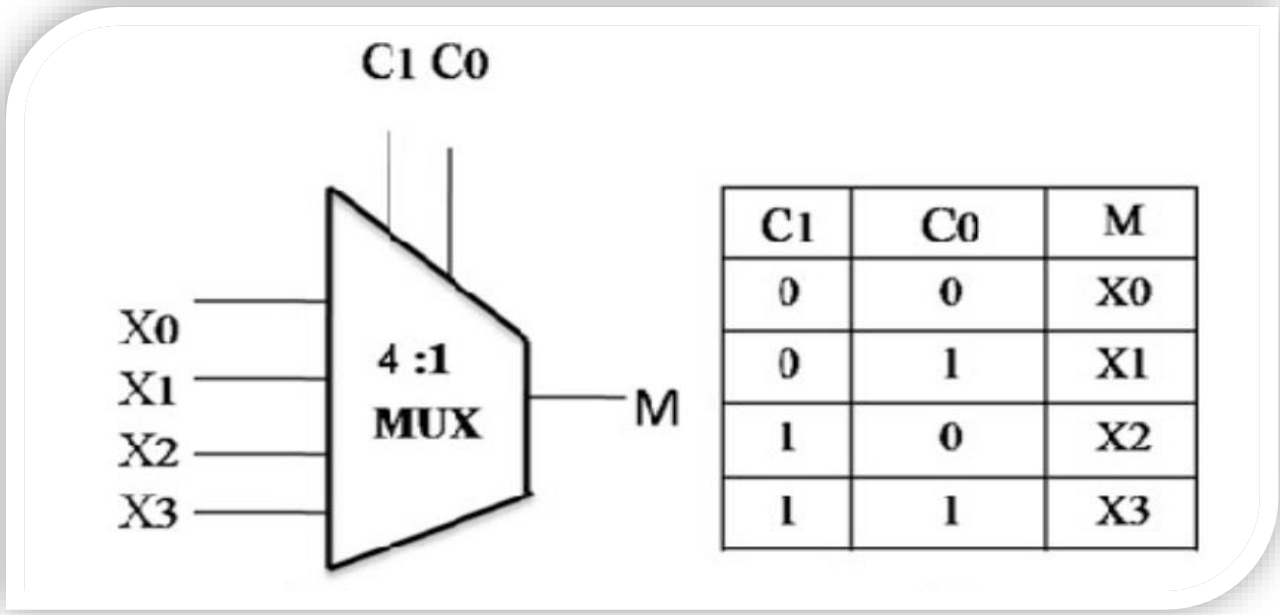
Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figura 22. Porți logice utilizate UAL 1 bit

Porțile elementare AND și OR vor fi folosite pentru operațiile logice. Poarta XOR va fi utilizată pentru negarea operanzilor în cazul instrucțiunii de scădere.

Multiplexorul utilizat va avea 4 intrări de date și o intrare de selecție, acesta va fi folosit pentru alegerea operației dorite prin intermediul semnalului de selecție.



C1	C0	M
0	0	X0
0	1	X1
1	0	X2
1	1	X3

Figura 23. Multiplexor 4:1

În continuare voi prezenta sumatorul pe 1 bit, acesta fiind utilizat pentru operațiile de: adunare, înmulțire, dar și scădere când unul dintre termeni va fi reprezentat in complement față de 2.

-A, B = operanzi

-Cin = transport intrare

-Cout = transport ieșire

-Sum = rezultat

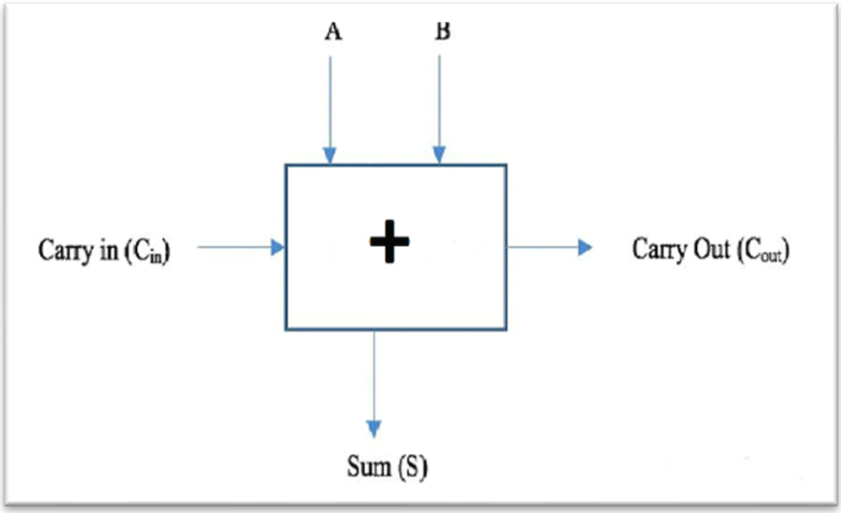


Figura 24. Sumator pe 1 bit

Utilizând componentele descrise mai sus se va implementa unitatea aritmetică-logică pe 1 bit ce va avea structura de forma următoare:

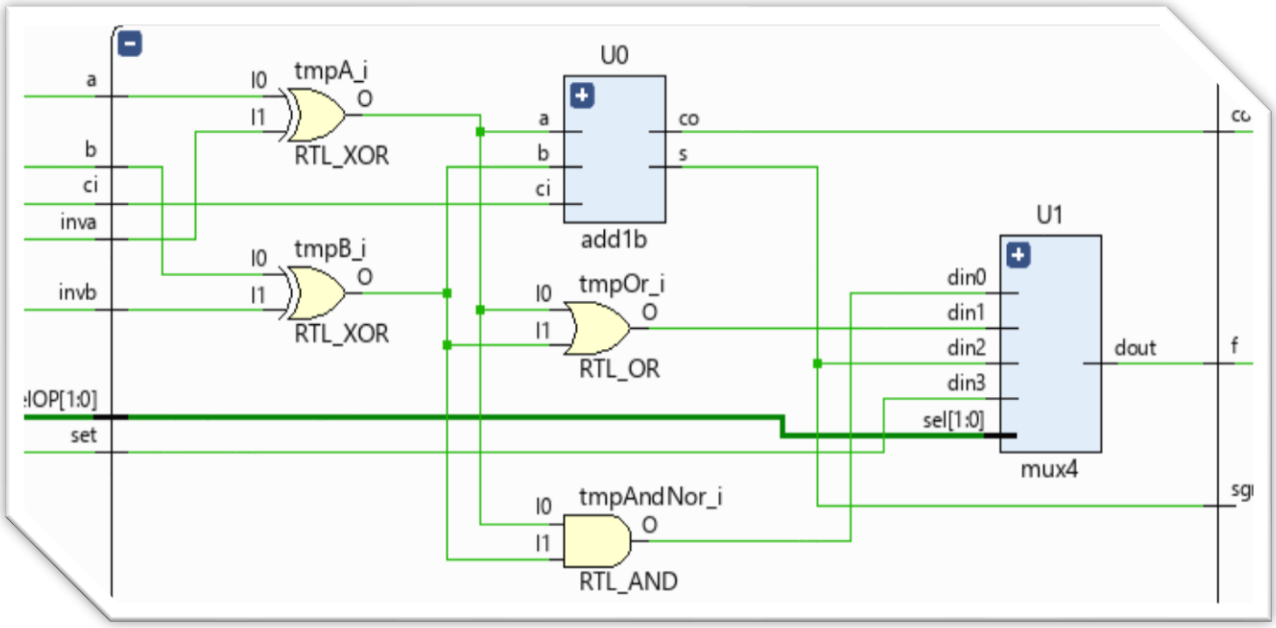


Figura 25. UAL 1 bit

Unitatea aritmetică logică pe 1 bit este capabilă să execute operațiile logice: AND, OR, NOR și operațiile aritmetice de adunare și scădere. În continuare plecând de la structura UAL pe 1 bit și folosind instrucțiunea pentru generarea instanțelor se va crea un nou modul pe 32 biți la care se va mai adauga o parte hardware pentru instrucțiunea SLT, instrucțiune utilizată pentru compararea operanzilor a și b.

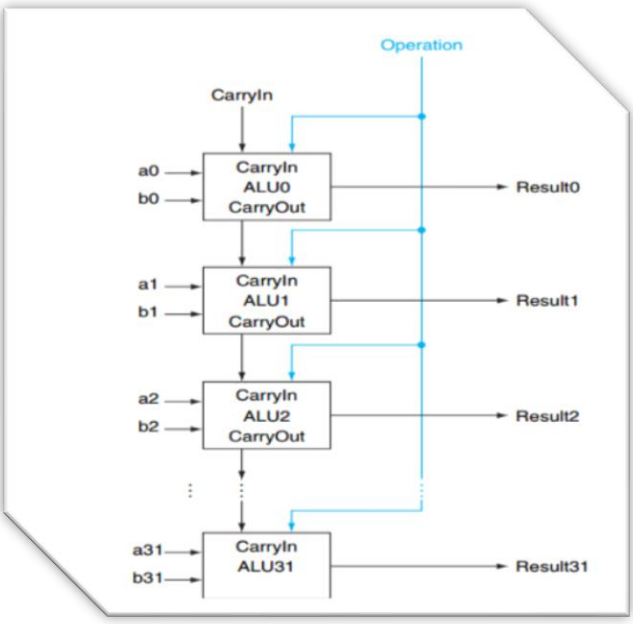


Figura 26. Structura de bază UAL 32 biți

Operația SLT setează bitul LSB al ieșirii Rez în funcție de valorile operanzilor ce se compară. Dacă operandul A<B ieșirea va avea valoarea Rez=1, în caz contrar ieșirea va avea valoarea Rez=0. În funcție de tipul operanzilor, întregi cu semn sau întregi fără semn condițiile pentru operația de comparare diferă.

Pentru operanzii cu semn, semnalul de interes pentru monitorizare este OV, în funcție de valoarea acestuia și valoarea semnalului sgn al al celulei MSB al UALului se va seta mai departe valoarea semnalului de set al celulei LSB a UALului.

Pentru operanzii fără semn, semnalul de interes este CO, valoarea sa negata va seta mai departe semnalul de set al celulei LSB a UALului.

Pentru a oferi o valoare corectă semnalului de set a celulei LSB a UALului va trebui utilizat un MUX2:1, al cărui semnal de selecție va alege intrarea corespunzătoare în funcțtie de tipul operanzilor utilizați.

În final UAL pe 32 biți va putea executa următoarele operații:

operaţie	opUAL[3]	opUAL[2]	opUAL[1:0]
and	0	0	00
or	0	0	01
nor	1	1	00
add	0	0	10
sub	0	1	10
slt*	0	1	11

Figura 27. Operații UAL32

Pe lângă operațiile deja prezentate care sunt executate de structura UAL32 ce face parte din unitatea UAL_TOP, am extins partea hardware a modulului TOP astfel permițând și execuția operațiilor de multiplicare fără semn, XOR și deplasare cu 2 poziții spre stânga.

Pentru operațiile de deplasare și XOR nu a fost nevoie de o utilizare prea mare de resurse.

Pentru implementarea multiplicatorului s-a pornit de la celula de adunare pe 1 bit, cu ea realizându-se în primă fază modulul TP, instanța realizată va fi multiplicată de 32 de ori folosind instrucțiunea generate pentru a realiza modulul TP_ROW, acest modul de asemenea va fi reprodus de 31 de ori. Bitul LSB din rezultat va fi obținut în urma operației AND dintre bitul

LSB al operandului Y și biții operandului X, obținând un rezultat pe 32 de biți, doar bitul de pe poziția 0 va fi utilizat pentru reprezentarea rezultatului final, ceilalți vor fi folosiți pentru calcularea următoarilor biți de rezultat final în celulele cascadate TP_ROW. După fiecare execuție a unei celule TP_ROW se va obține câte un bit pentru rezultatul final, mai puțin la ultima celulă unde se obțin 32 cei mai semnificativi biți.

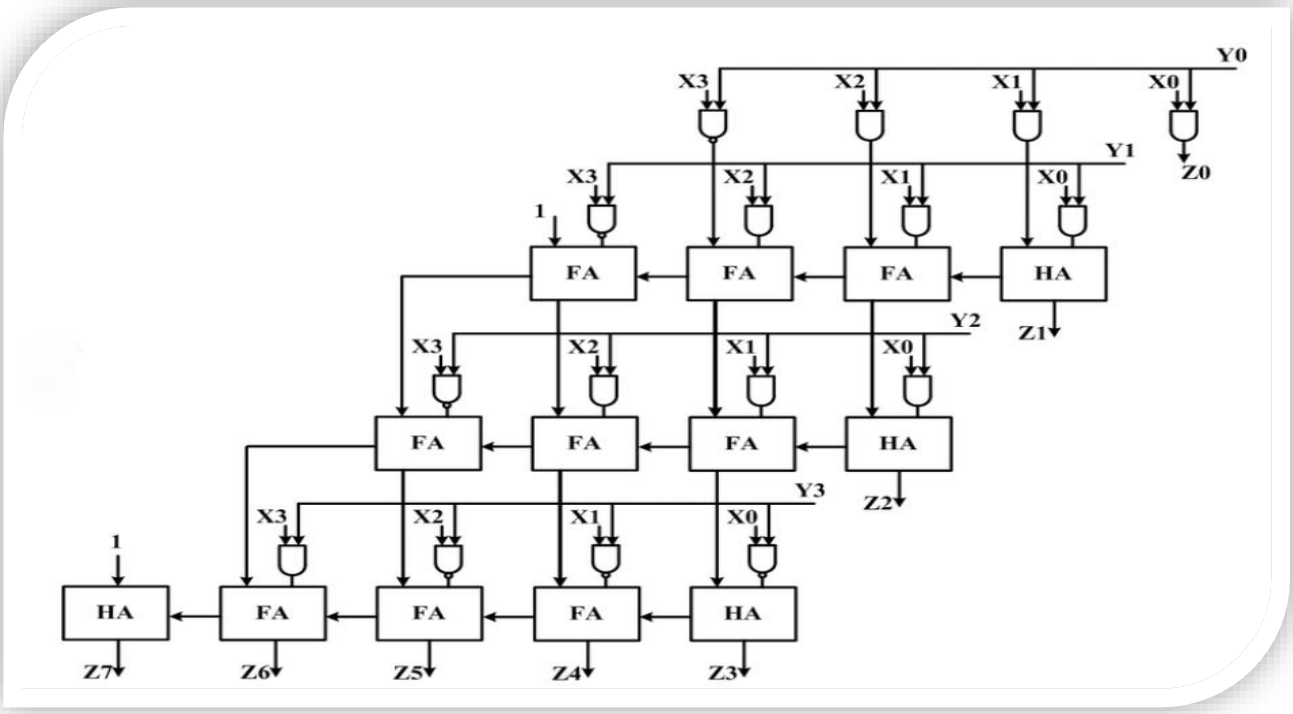


Figura 28. Structura unui multiplicator cu rezultat 8 biți

Setul final de operații pe care îl va putea executa UAL_TOP este următorul:

Operații	opUAL[5]	opUAL[4]	opUAL[3]	opUAL[2]	opUAL[1]	opUAL[0]
AND	0	0	0	0	0	0
OR	0	0	0	0	0	1
NOR	0	0	1	1	0	0
ADD	0	0	0	0	1	0
SUB	0	0	0	1	1	0
SLT	0	0	0	1	1	1
MULL	0	1	x	x	x	x
XOR	1	0	x	x	x	x
SHT_L	1	1	x	x	x	x

Figura 29. Set final operații UAL_TOP

Schema finală generată cu instrumentul pentru elaborare a designului oferit de programul Vivado este următoarea:

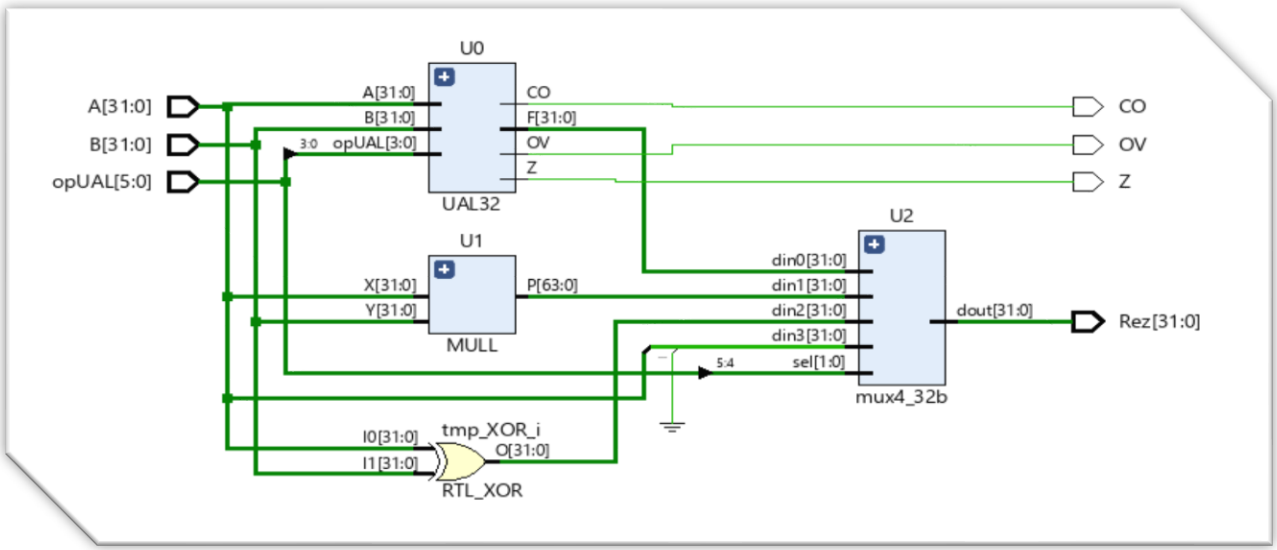
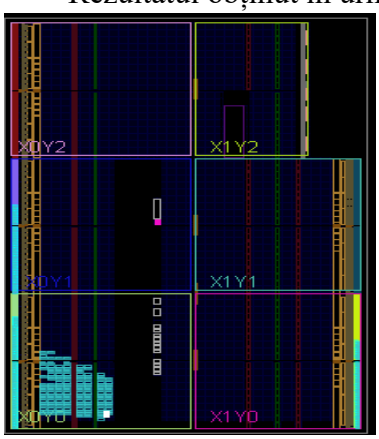


Figura 30. UAL_TOP

Rezultatul obținut în urma simulării implementării pe dispoziivul FPGA:



Se poate observa în imaginea din stânga zona din care au fost utilizate resursele dispozitivului FPGA, iar în mesajul de mai jos este semnalat că au fost utilizate pentru implementare 975 blocuri LUT.

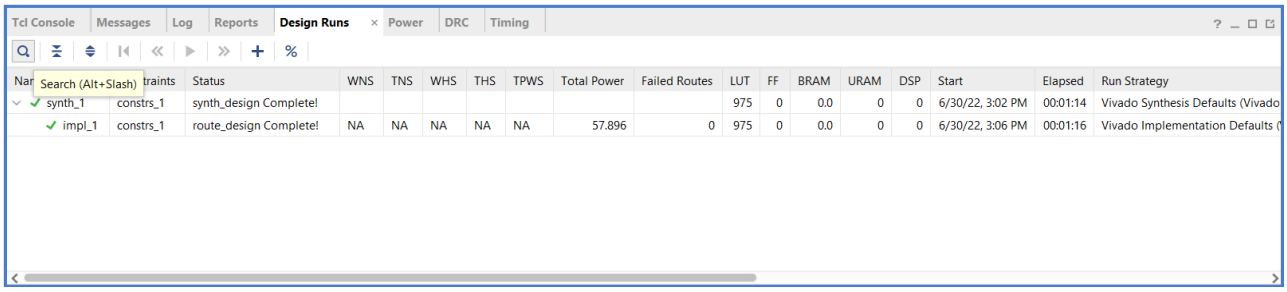


Figura 31. Resurse utilizate UAL_TOP

Codul și schemele aferente pentru UAL_TOP sunt prezentate la Anexa ...

3.2 Bancul de regiștri

Bancul de regiștri conține 32 de regiștri cu intrari și ieșiri paralele, dintre care primul registru va conține mereu valoarea 0, la acesta neputând a se face scrieri.

Pentru realizarea bancului de regiștri am pornit de la un registru paralel-paralel pe 32 biți. Folosind generatorul de instanțe(generate-endgenerate) am realizat o structură de 32 regiștri. Pentru a realiza funcțiile de citire din regiștrii s-au folosit 2 multiplexoare la ale căror intrări de selecție se va aplica numărul registrului care dorește să fie citit. Funcția de scriere în regiștri se realizează cu ajutorul unui demultiplexor la a cărui intrare de selecție se aplică o valoare ce va fi prelucrată de către un demultiplexor astfel încât va trimite un semnal ce va activa funcția de load a registrului dorit.

În arhitectura MIPS rolul regiștrilor diferă.

Numarul registrului	Numele conventional	Utilizare
$0	$zero	Conectat la 0
$1	$at	Rezervat pentru pseudoinstrucțiuni
$2 - $3	$v0, $v1	Memoreaza valorile returnate din functie
$4 - $7	$a0 - $a3	Argumente pentru funcții
$8 - $15	$t0 - $t7	Date temporare
$16 - $23	$s0 - $s7	Registre salvate, păstrate de subprograme
$24 - $25	$t8 - $t9	Mai multe registre temporare
$26 - $27	$k0 - $k1	Rezervat pentru nucleu,nu pot fi folosite.
$28	$gp	Global Area Pointer (baza segmentului global de date)
$29	$sp	Stack pointer
$30	$fp	Pointer cadru
$31	$ra	Stocheaza adresa de intoarcere din functie

Figura 32. Regiștri MIPS

În continuare se va elabora schema bancului de regiștri folosind programul Vivado, de asemena se va realiza și implementarea acestuia pe dispozitivul FPGA dorind să notez resursele utilizate.

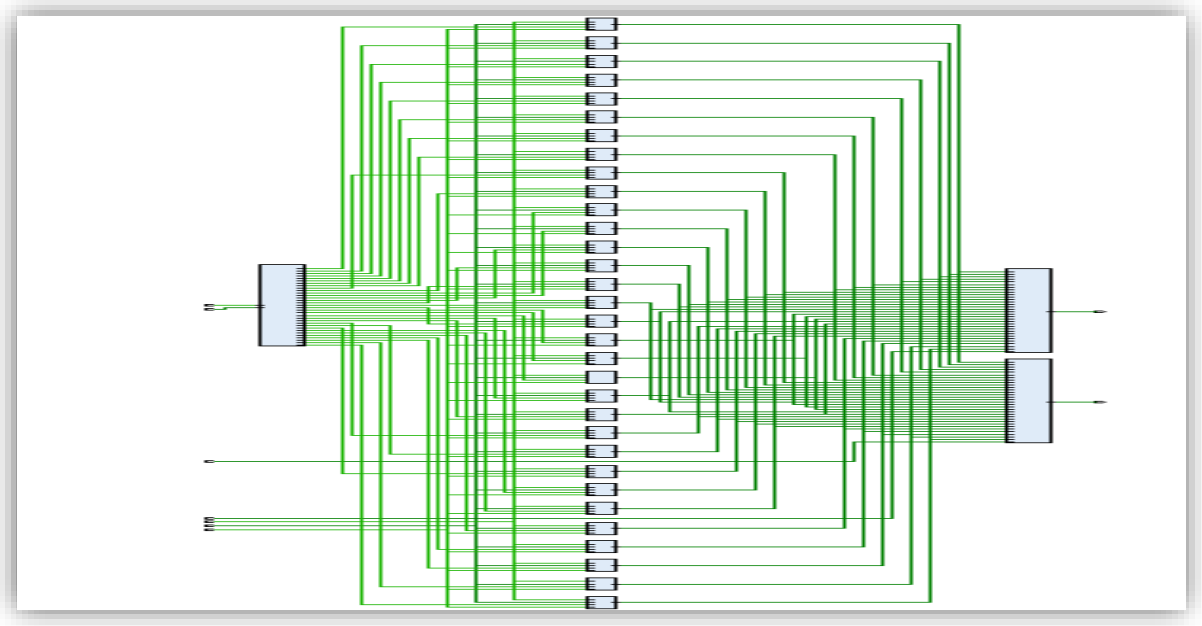
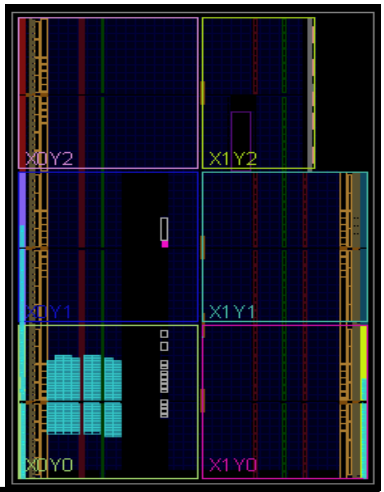
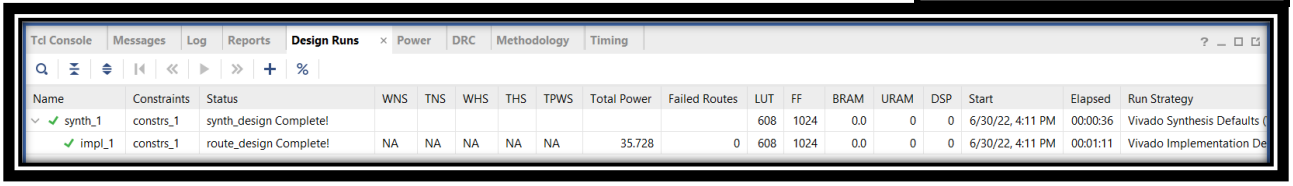


Figura 33. Banc de regiștri

Având în vedere că folosim regiștri în acest bloc component este de așteptat ca în urma implementării pe placa FPGA să se facă uz de bistabile. După cum se poate observa în rezultatele obținute se utilizează 608 blocuri LUT și 1024 bistabile.

În imaginea din dreapta se poate observa zona din care au fost utilizate resursele plăcii FPGA.





Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	synth_design Complete!								608	1024	0.0	0	0	6/30/22, 4:11 PM	00:00:36	Vivado Synthesis Defaults
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	35.728	0	608	1024	0.0	0	0	6/30/22, 4:11 PM	00:01:11	Vivado Implementation De

Figura 34. Resurse utilizate Banc regiștri

3.3 PC și Sign extend

PC(Program counter) este un modul cu rolul de a genera următoarea adresă de unde va fi preluată instrucțiunea ce va urma a fi executată. Acesta este un circuit secvențial, tranzițiile pozitive ale semnalelor de ceas și reset vor influența valoarea adresei viitoare. Program counterul are rolul de a limita viteza cu care se propagă următoarea adresă în sistem aceasta fiind recalculată la o perioadă de ceas.

Pentru calcularea adresei următoare se mai fac uz și de UAL4, ce va aduna la adresa pregedentă 4, valoare dată de modul cum sunt aliniate datele în memoria de instrucțiuni. O instrucțiune are dimensiunea de 32 biți, dar fiind aliniată la un byte este nevoie de unități parcurse pentru a trece la următoarea.

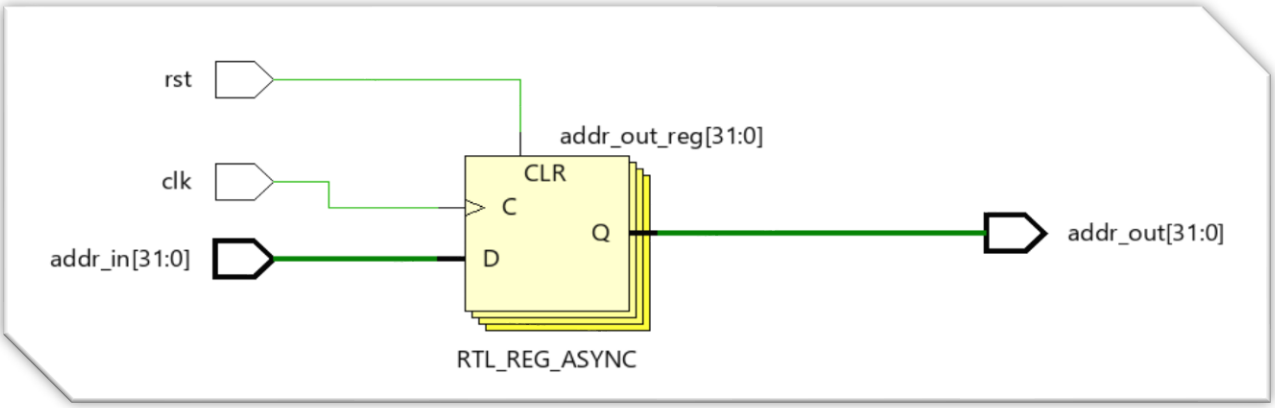


Figura 35. Program counter

Blocul sign extend are rolul de a mări o valoare de la dimensiunea de 16 biți la 32 biți cu extinderea bitului de semn pe pozițiile 31:17. Acesta este utilizat în cadrul instrucțiunilor de tip I unde este folosită valoarea imediată. Modulul pentru extinderea de semn este descris în felul următor:

```
module Sign_extend(
    input [15:0] word,
    output [31:0] d_word
    );

    assign d_word={{16{word[15]}},word};

dmodule
```

Figura 36. Cod Verilog modul Sign extend

3.4 Memorie pentru instrucțiuni

Memoria pentru instrucțiuni este unitatea ce se ocupă de stocarea instrucțiunilor ce urmează a fi apelate. Datele pentru instrucțiunile corespunzătoare sunt aliniate la 8 biți, instrucțiunile având dimensiunea de 32 biți. Schema generată de instrumentul de elaborare a designului a programului Vivado ve genera următoarea figură:

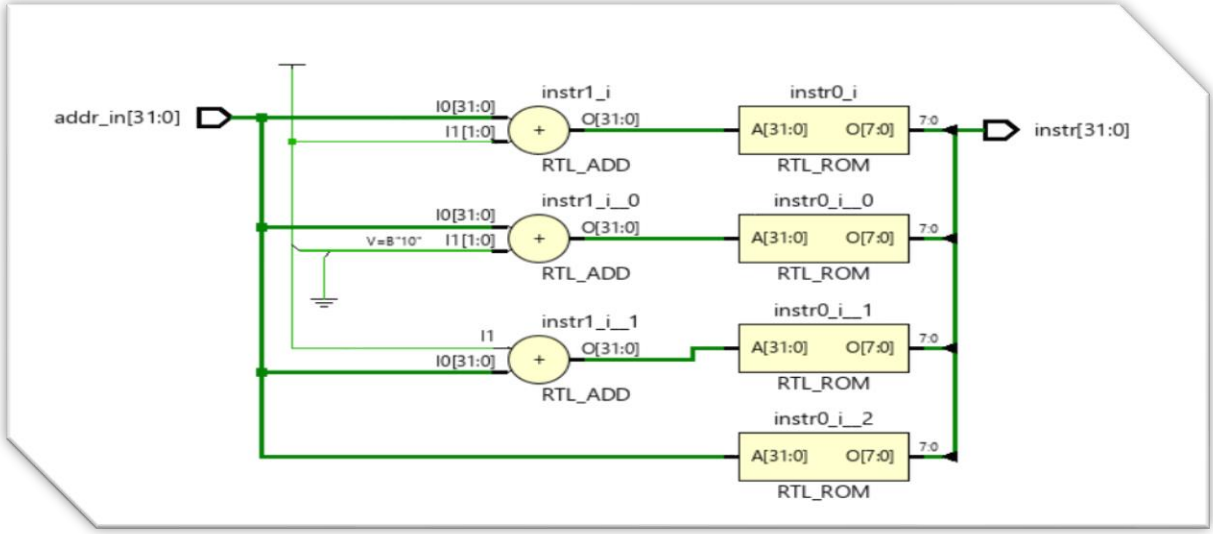
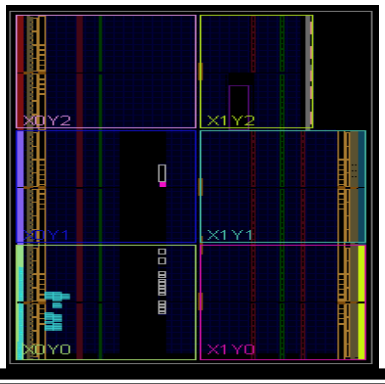


Figura 37. Memorie pentru instrucțiuni

Se poate observa în schema obținută cu instrumentul de design utilizarea memoriilor ROM în care se stochează instrucțiunile. Fiecare memorie va fi apelată cu o adresă diferită pentru a respecta alinierea datelor.



În figura din dreapta se poate observa zona de unde sunt preluate resursele necesare implementării. În urma implementării se va face uz de 119 blocuri LUT.

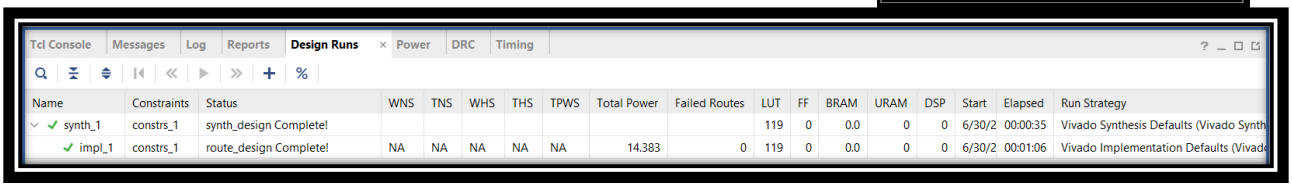


Figura 38. Resurse utilizate memorie de instrucțiuni

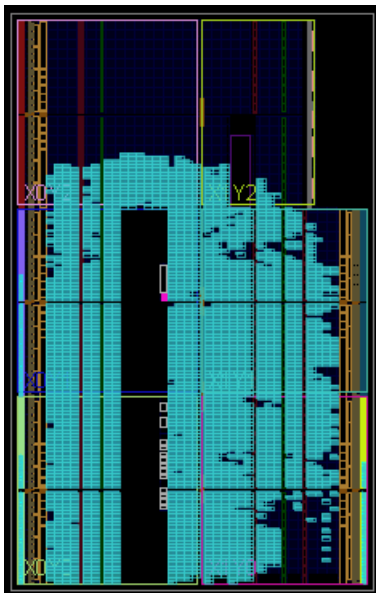
3.5 Memorie de date

Memoria de date spre deosebire de memoria de insrucțiuni stochează rezultate în urma execuției intrucțiunilor. Datele ce urmează să fie stocate provin din bancul de regiștri. Alinierea datelor se face tot la 1 byte la fel ca în memoria de instrucțiuni astfel păstrând criteriile de aliniere ale procesorului MPIS.

Această structură are forma unei matrici la care se pot face scrieri și citiri fiind o memorie de tip RAM. Singurele instrucțiuni ce pot accesa această memorie sunt LW și SW. LW penru încărcarea datelor din memorie în regiștri, iar SW pentru stocarea datelor din regiștri în memorie.

Se poate observa în figura din dreapta consumul mare de resurse, în urma implementării folosindu-se 17237 blocuri LUT.

Matricea utilizată ca zonă de stocare a datelor are dimensiuni mult prea mari și ar trebui restânsă.



Tcl Console | Messages | Log | Reports | Design Runs | Power | DRC | Methodology | Timing

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	synth_design Complete!								17237	0	0.0	0	0	6/30/22, 5:37 PM	00:12:23	Vivado Synthesis Defaults
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	229.236	0	17236	0	0.0	0	0	6/30/22, 5:50 PM	00:04:03	Vivado Implementation D

Figura 39. Resurse utilizate pentru memoria de date

3.6 Unitate de Control

Unitate de control se ocupă cu interpretarea operației dorite, în funcție de tipul operației se vor seta semnale diferite pentru selectarea componentelor hardware specifice operației ce dorește a se executa.

Semnificația și rolul semnalelor:

-op[5:0] = semnal de intrare ce va furniza tipul operației pentru care trebuie pregătită partea hardware;

-ALUSrc = semnal de selecție, în funcție de tipul instrucțiunii acesta va selecta: o valoare oferită de bancul de regiștri sau o valoare imediată;

-ALUop[2:0] = semnal de control pentru unitatea ALU_control;

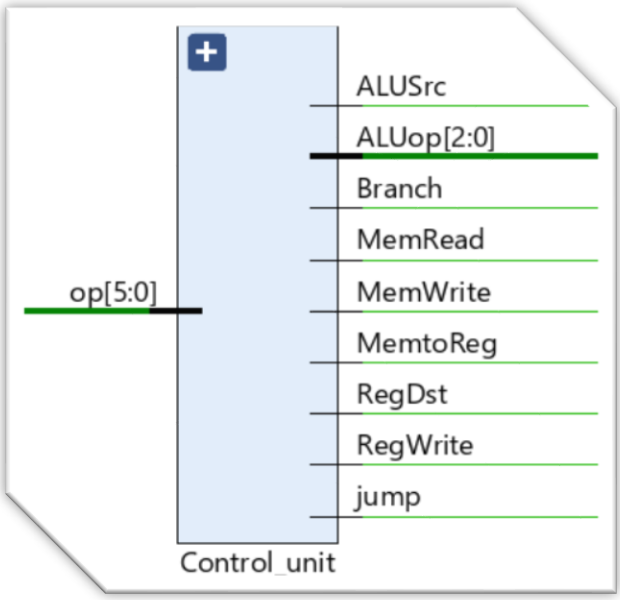


Figura 40. Unitate Control

-Branch = semnal ce va fi setat în cazul execuției instrucțiunii BEQ;

-MemRead = semnal ce va fi setat în cazul execuției instrucțiunii LW;

-MemWrite = semnal ce va fi setat în cazul execuției instrucțiunii SW;

-MemtoReg = semnal de selecție, în funcție de tipul instrucțiunii acesta va selecta: date din memorie sau date oferite direct de la UAL;

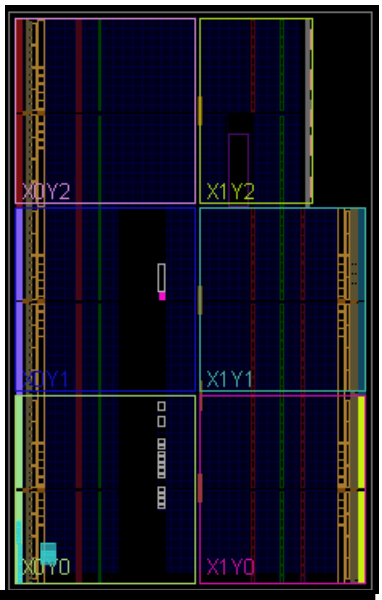
-RegDst = semnal de selecție, în funcție de tipul instrucțiunii acesta va selecta ce registru va fi scris;

-RegWrite = semnal ce va fi setat în cazul nevoii de a se scrie în regiștri;

-jump = semnal setat în cazul instrucțiunii de jump;

Implementarea acestui circuit face uz de puține resurse, funcționalitatea sa fiind descrisă printr-un simplu case-endcase, în funcție de tipul operației semnalele de la ieșire vor avea valori specifice.

Se poate observa că a fost folosită o cantitate redusă de blocuri LUT, mai exact 11.



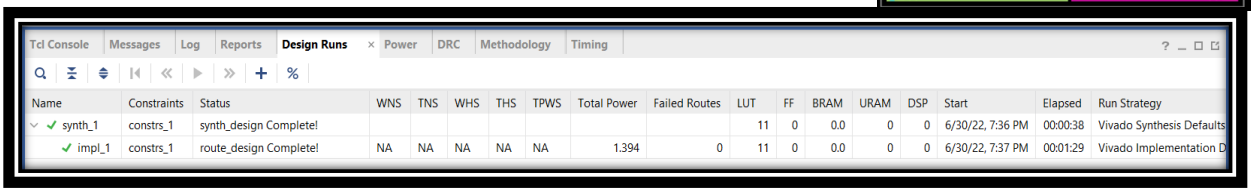


Figura 41. Resurse utilizate pentru Unitatea de Control

3.7 Unitate de control UAL

Unitatea aritmetică logică are nevoie de un bloc specializat care să preia informații de la unitatea de control, dar și informații obținute în urma decodării instrucțiunii, aici intervine unitatea de control a UALului. Acest modul îi oferă UALului valori pentru semnalul de selecție în funcție de datele primite.

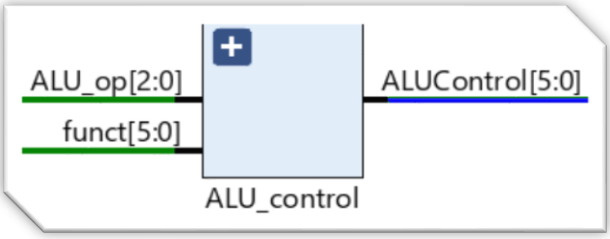


Figura 42. Unitate control UAL

-ALU_op[2:0] = semnal de intrare primit de la Unitatea de Control;

-funct[5:0] = funcția pentru instrucțiunile de tip R;

-ALUControl[5:0] = semnal de selecție pentru TOP_UAL;

3.8 Implementare pe FPGA a procesorului

În continuare se va folosi programul Vivado pentru o serie de simulări în cadrul unei posibile implementări pe un dispozitiv FPGA.

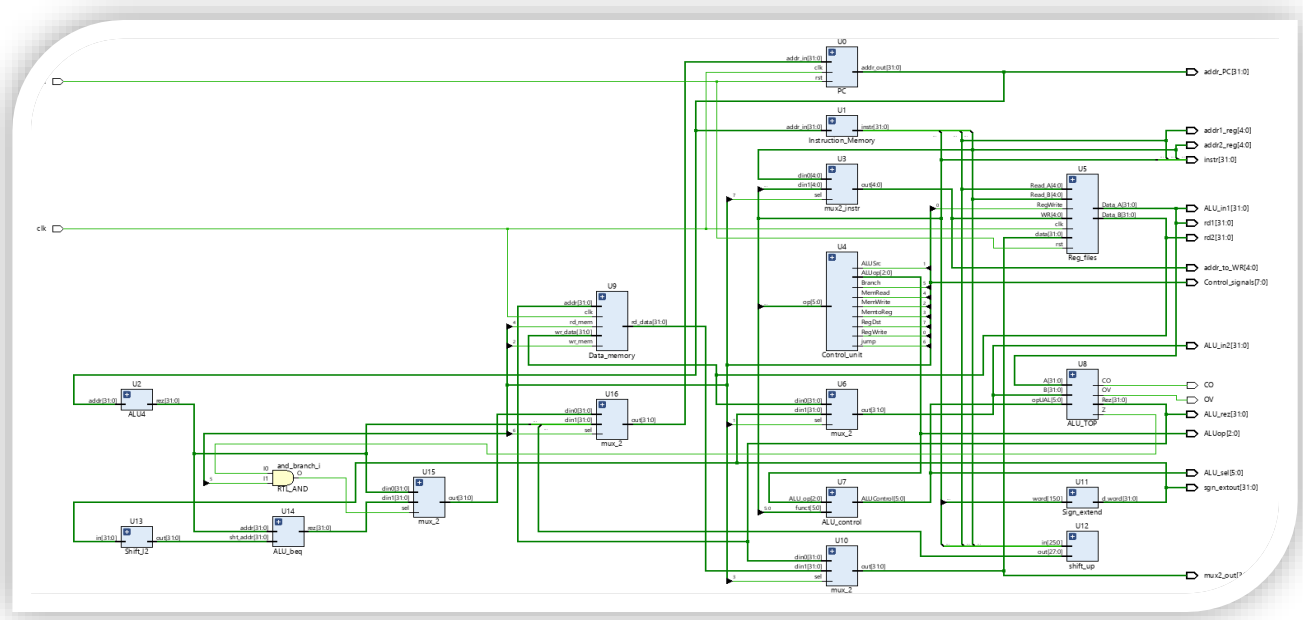


Figura 43. Arhitectura MIPS

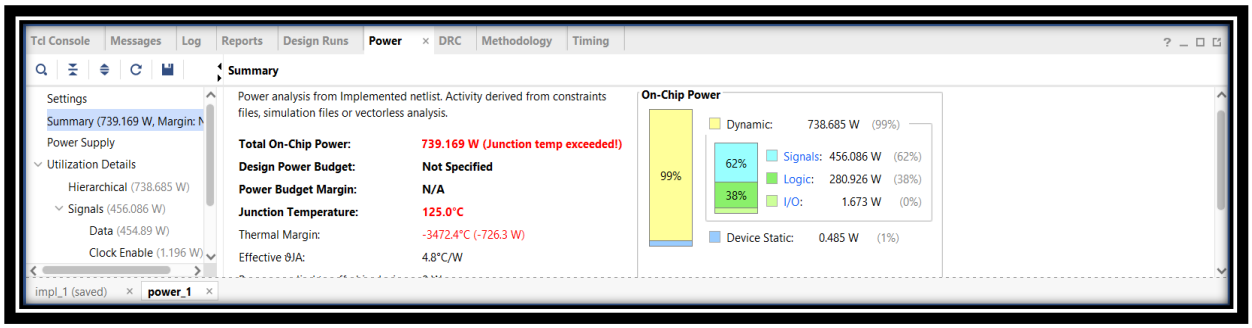
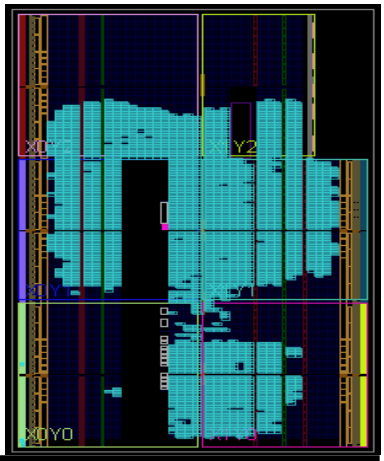


Figura 45. Consumul de putere

În urma sintezei și a implementării pe FPGA se obține un consum și o repartizare a resurselor sub forma umătoare(figura din dreapta):

În simularea finală se folosesc 12184 blocuri LUT și 5878 de bistabile.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	12184	0	20800	58.58
LUT as Logic	12184	0	20800	58.58
LUT as Memory	0	0	9600	0.00
Slice Registers	5878	0	41600	14.13
Register as Flip Flop	1031	0	41600	2.48
Register as Latch	4847	0	41600	11.65
F7 Muxes	2829	0	16300	17.36
F8 Muxes	1228	0	8150	15.07



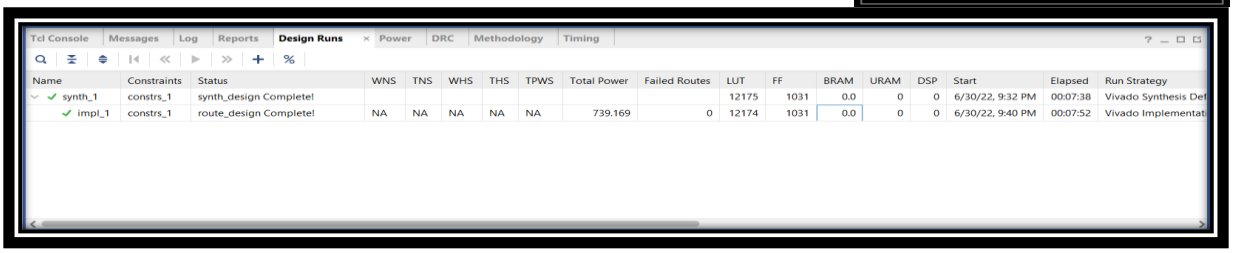
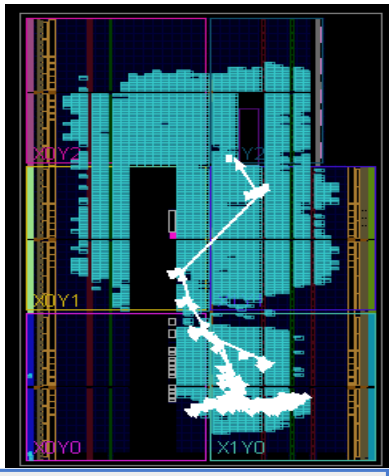


Figura 44. Resurse utilizate MIPS

În următoarea analiza am surprins căile critice, cele responsabile de frecvența maximă a semnalului de ceas. În figura din dreapta este prezentată cea mai lungă cale din întreg circuitul, iar mai jos este o listă cu topul căilor cu cel mai mare traseu din circuit.



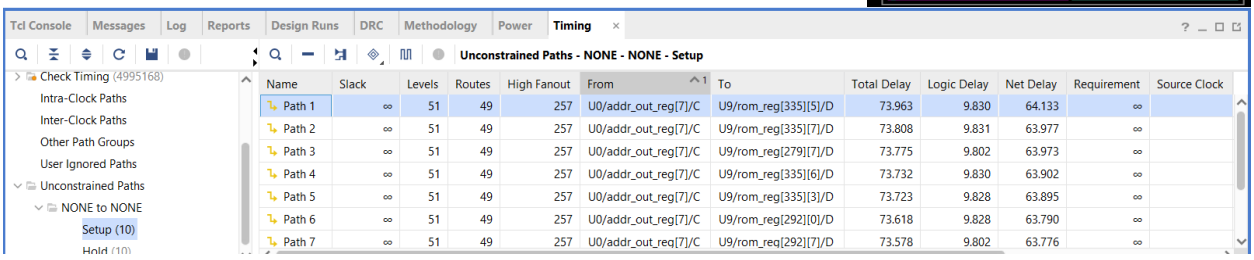


Figura 45. Drumul critic

4. Testarea blocurilor procesorului

Pentru a testa procesorul, un prim pas ar fi testarea individuală a blocurilor componente pentru a reduce timpul de debug în cazul apariției de erori. Astfel pentru fiecare bloc se va crea un test ce va avea ca scop acoperirea întreagii funcționalități, o acoperire cât mai bună detectează posibilele erori din timp și reduce timpul petrecut pentru debug considerabil.

4.1 Testbench Unitate aritmetică logică

Pentru testarea UALului testul va conține valori ce vor permite testarea simplă, dar și testare în cazul în care rezultatele vor prezenta depășire de format(semnalele CO sau OV fiind setate). Teste aferente operațiilor aritmetice și logice:

AND OR NOR XOR

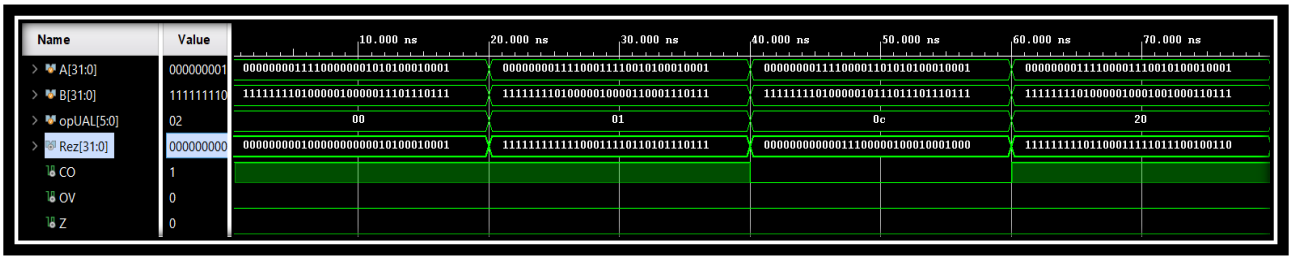
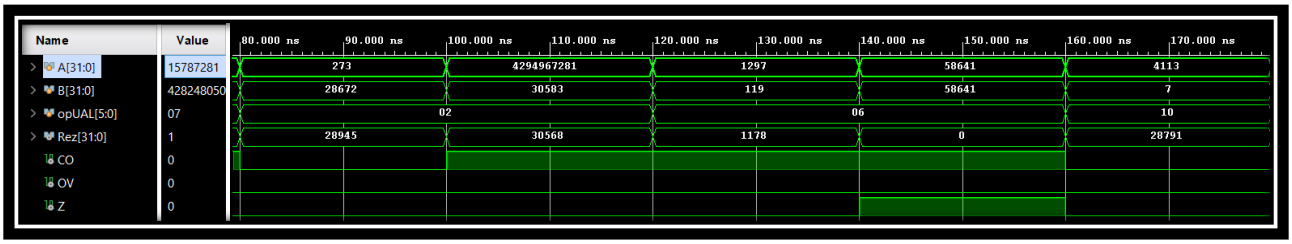


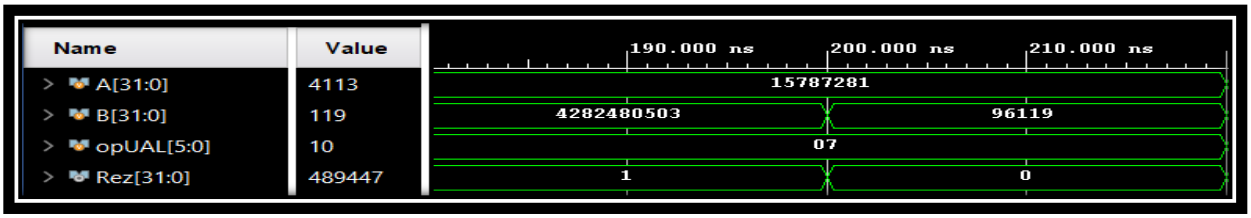
Figura 46. Testare operații logice

ADD ADD(CO) SUB SUB(Z) MULL



Figur 47. Testare operații aritmetice

SLT(A<B) SLT(A>B)



Figuria 48. Testare operații de comparare

4.2 Testbench Bancul de Regiștri

Pentru bancul de regiștri se va testa în primă fază dacă registrul 0 poate fi scris, iar mai apoi pentru ceilalți regiștri se vor face scrieri și citiri. Regiștri vor fi inițializați la început cu valoarea 0 prin semnalul de reset ce va fi activ pentru o perioadă scurtă.

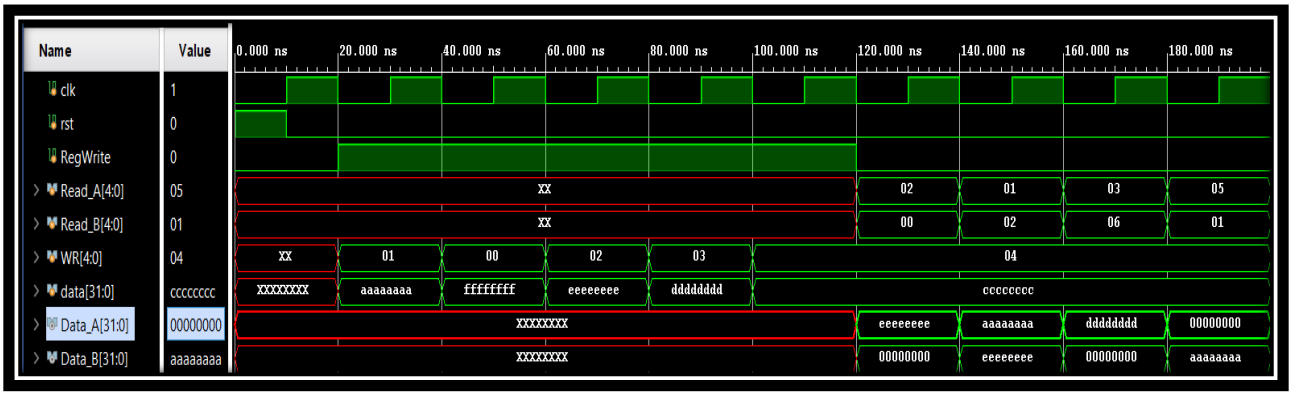


Figura 49. Simulare Banc de regiștri

Se poate observa cum s-a încercat încărcarea registrului 0 cu valoarea de FFFF_FFFF la perioada 40ns, dar m-ai târziu când s-a realizat citirea acestui registru la perioada 120ns pe portul Read_B, valoarea returnată este 0000_0000 ceea ce înseamnă că bancul este configurat corect și menține valoarea registrului 0 nulă.

Ceilalți regiștri sunt încărcați pe rând cu valori cât mai vizibile pentru a fi ușor de urmărit în cadrul testării. Încărcarea și citirea din regiștri se execută corect, regiștri care nu au fost scriși returnează valoarea 0 , astfel e clar că semnalul de reset a funcționat și întreg bancul a fost inițializat.

4.3 Testbench Memorie de Instrucțiuni

În cadrul testării memoriei de instrucțiuni se urmărește ca datele din memorie să fie aliniate la un byte. Astfel semnalul de intrare care este adresa de unde se vor prelua datele trebuie să aibă o valoare ce va fi multiplu de 4.

Datele vor fi preluate dintr-un fișier text pentru a verifica daca citirea de la adresa dorită se realizează conform specificațiilor.

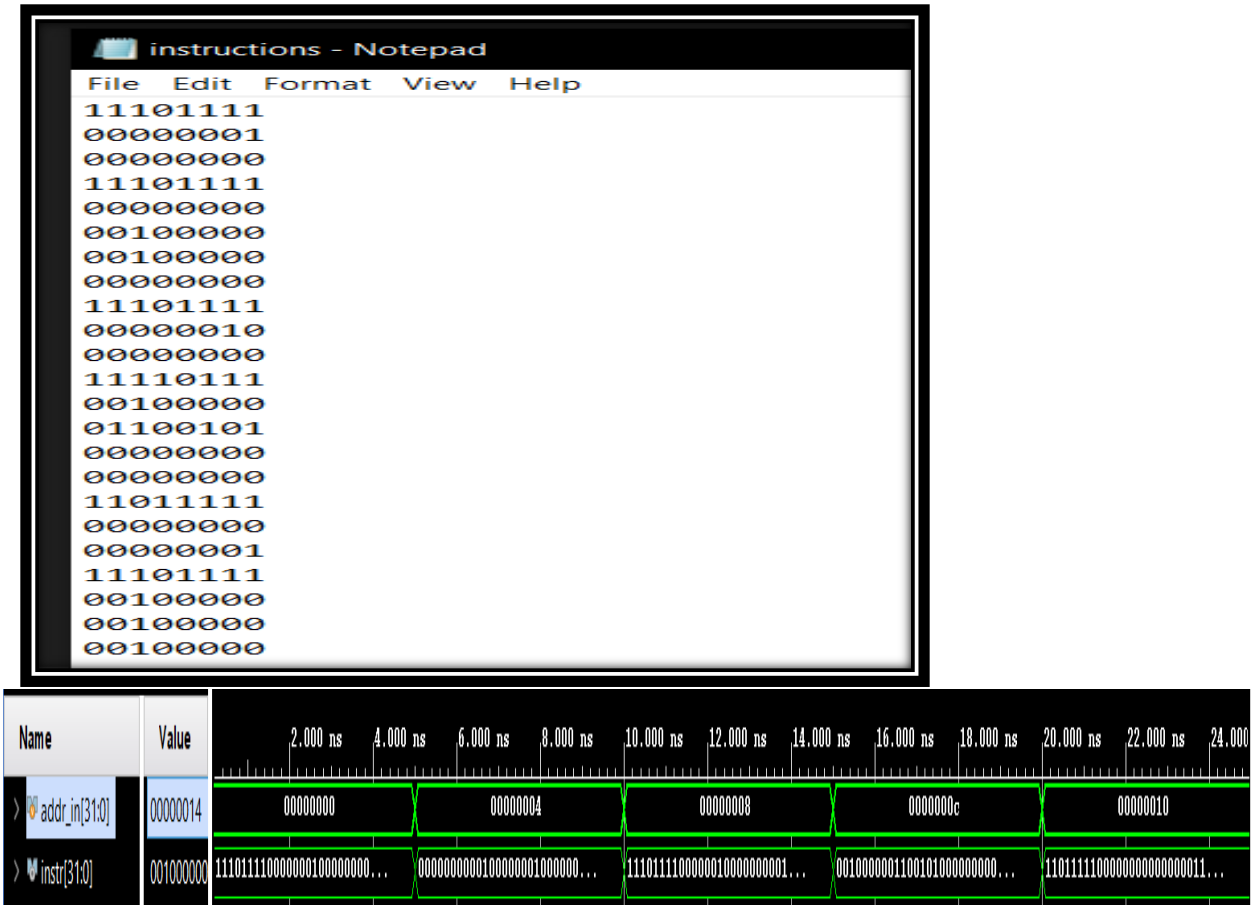


Figura 50. Simulare Memorie de Instrucțiuni

Comparând rezultatele din fișierul text cu cele din formele de undă se poate observa clar că s-a respectat alinierea, iar datele au fost citite corespunzător.

Din memoria de instrucțiuni se pot face doar citiri fiind o memorie de tip ROM.

4.4 Tesbench Memorie de date

Testarea va urmări scrierea și citirea datelor de la adrese ce vor respecta alinierea de 1 byte pentru valori de 32 biți. Memoria de date funcționează pe același principiu ca memoria pentru instrucțiuni, singura diferență fiind faptul că la memoria de instrucțiuni nu se pot face scrieri.

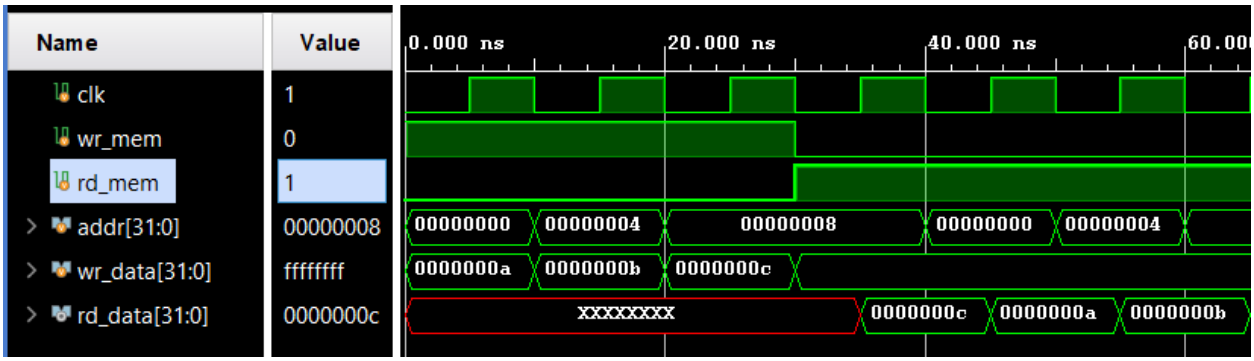


Figura 51. Simulare Memorie de date

Se poate observa cum prima dată se face scriere la adresele: 0, 4 și 8 ca mai apoi să se facă citire de la aceste adrese.

4.5 Observații simulări

Simulările au fost realizate doar pe o parte din blocurile componente ale procesorului. Decizia de a simula doar aceste blocuri a fost luată pe criterii precum dificultatea implementării unui bloc sau erori pe care le-am întâmpinat pe parcurs. Simulările sunt pentru a confirma înțelegerea funcționării acestor componente, dar și pentru a fixa problemele întâmpinate.

Restul circuitelor prezintă o logică relativ simplă fiind circuite ce au rol de decodare cum sunt de exemplu: Unitatea de Control, Unitatea de Control UAL ce au rol de a decoda tipul operației și tipul funcției. Circuite de deplasare, folosite pentru calculul viitoarei adrese. Multiplexoare utilizate pentru selecția diferitelor valori în funcție de tipul operațiilor. Sumatoare simple cu rol în calcularea viitoarei adrese. Unitatea de extindere a semnului folosită pentru a obține din valori pe 16 biți, valori pe 32 biți.

5. Testarea procesorului

Pasul următor în obținerea unui procesor cu arhitectura MIPS funcțional îl reprezintă testarea modulului TOP. Testarea va fi realizată cu ajutorul unui script scris în limbajul C++ .

5.1 Script generare instrucțiuni

Programul creat are ca rol generarea unui set de instrucțiuni: aleatorii sau specific alese.

În cadrul programului sunt create 2 clase, una pentru instrucțiunile de tip R și una pentru instrucțiunile de tip I. Fiecare clasa va conține 2 constructori: unul declarat implicit ce va crea valori aleatorii pentru operații, regiștri utilizați și funcții, iar cel de al doilea constructor va fi apelat cu informațiile dorite legat de operația pe care doresc să o utilizez, regiștri pe care doresc să îi folosesc și funcția aleasă.

În cadrul constructorului pentru instrucțiuni de tip R există o singură operație pe care am definit-o și o serie de funcției ale căror valori le-am stocat intr-o matrice. Mai departe constructorul urmărește să aleagă operația, funcția dorită din matrice; funcția poate fi aleasă aleatoriu pentru constructorul implicit generând o valoare cu funct=rand() % 8; numărul funcției va fi stocată într-o variabilă, iar mai departe se va alege funcția din matricea mai sus definită. În cele ce urmează se vor alege regiștri utilizați la fel generându-se numere aleatorii într 0 și 31 deoarece bancul de regiștri este de 32, valorile se vor transforma în format binar. Utilizând valorile în binar ale: operației, regiștrilor, funcției se va crea instrucțiunea ce va fi stocată temporar într-un vector cu dimensiunea de 32 biți. Acest vector va fi scris pe rând, prima dată într-un fișier ce stochează instrucțiunile în format binar cu aliniere la 1 byte, instrucțiuni ce vor fi folosite mai departe de memoria de instrucțiuni în cadrul testării procesorului și intr-un fișier ce va traduce din cod mașină(format binar) în cod assembly care este utilizat pentru urmărirea mai ușoară a ordinii instrucțiunilor.

În cadrul constructorului pentru instrucțiuni de tip I procesul este în mare parte asemănător, dar cu mici diferențe. Se va defini o matrice pentru operații, funcții neexistând în cadrul instrucțiunilor de tip I, se alege operația printr-un proces de randomizare operation=rand() % 8; operația având valoarea corespondentă în matricea definită mai sus în ierarhia codului. Se vor genera valori pentru regiștri care urmează a fi utilizați, iar aceste valori

vor fi traduse în format binar fiind salvate temporar în vectori. Pentru ultimul câmp ce face parte din instrucțiuniea de tip I și anume imm(16 biți) se va genera o valoare aleatorie cuprinsă între 0-65535(FFFF), valoarea va fi tradusă în format binar și va fi stocată temporar într-un vector. Valorile în format binar pentru operație, numărul regiștrilor și valoarea imediată vor fi stocate într-un vector temporar de unde vor fi scrise în fișierul ce stochează instrucțiunile în format binar și traduse în fișierul cu cod assembly.

Prezentare unui mic test de creare instrucțiuni aleatorii:

```
493         for(i=0; i<10; i++)
494         {
495             R_type();
496             I_type();
497         }
498         return 0;
499     }
```

instructions.txt

```
1   00000010
2   10110101
3   00100000
4   00100011
5   00011100
6   11000001
7   01000101
8   01101001
9   00000001
10  01000000
11  11001000
12  00110001
13  10101111
14  00011101
15  00011001
16  11000101
17  00000000
18  00110110
19  00011000
20  00100011
21  00100010
22  10010110
23  00111100
24  11010111
```

asm.txt

```
1   NOR Rd(4)=Rs(21) NOR Rt(21);
2   ORI Rt(1)=Rs(6) ORI imm(17769);
3   XOR Rd(25)=Rs(10) XOR Rt(0);
4   SW [Rt](29)<-Rs(24) + imm(6597);
5   NOR Rd(3)=Rs(1) NOR Rt(22);
6   ADDI Rt(22)=Rs(20) + imm(15575);
7   NOR Rd(12)=Rs(31) NOR Rt(11);
8   SW [Rt](25)<-Rs(24) + imm(17317);
9   SLT Rd(3)=Rs(27) SLT Rt(27);
10  LW Rt(21)<-Rs(21) + imm(9277);
11  AND Rd(23)=Rs(0) AND Rt(5);
12  BEQ [imm](15407)<-Rs(15) - Rt(30);
13  NOR Rd(0)=Rs(13) NOR Rt(10);
14  ORI Rt(11)=Rs(1) ORI imm(26384);
15  OR Rd(22)=Rs(21) OR Rt(19);
16  MULLI Rt(24)=Rs(23) MULLI imm(9464);
17  SUB Rd(23)=Rs(31) SUB Rt(8);
18  MULLI Rt(4)=Rs(18) MULLI imm(20741);
19  XOR Rd(6)=Rs(7) XOR Rt(26);
20  ANDI Rt(20)=Rs(8) ANDI imm(26582);
```

Figura 52. Instrucțiuni create aleatoriu

După cum se poate observa în fișierul din stânga sunt instrucțiunile în format binar, iar în dreapta cod assembly pentru a putea urmări cât mai ușor desfășurarea operațiilor.

Pentru a facilita modul simplu de inserare al instrucțiunilor în fișierul text am decis să mai includ o clasă pentru instrucțiunea jump, instrucțiune ce va conține doar 2 câmpuri: unul pentru operație și unul pentru adresa la care se va face saltul. Hardwareul ce se ocupă de partea de salt în arhitectura MIPS folosește cel de al doilea câmp din instrucțiunea jump pentru a preciza numărul de operații peste care se face saltul, am modificat acest lucru pentru a face saltul la o adresă direct precizată astfel putând să implementez bucle de calcul, deoarece cu arhitectura veche se putea face salt doar înainte.

5.2 Generare instrucțiuni și testare

În cele ce urmează voi crea câteva seturi de operații cu ajutorul scriptului descris mai sus astfel testând funcționalitatea procesorului. În primă fază voi folosi o generare aleatorie a instrucțiunilor pentru a observa cum scriptul oferă operații valide indiferent de gradul de randomizare.

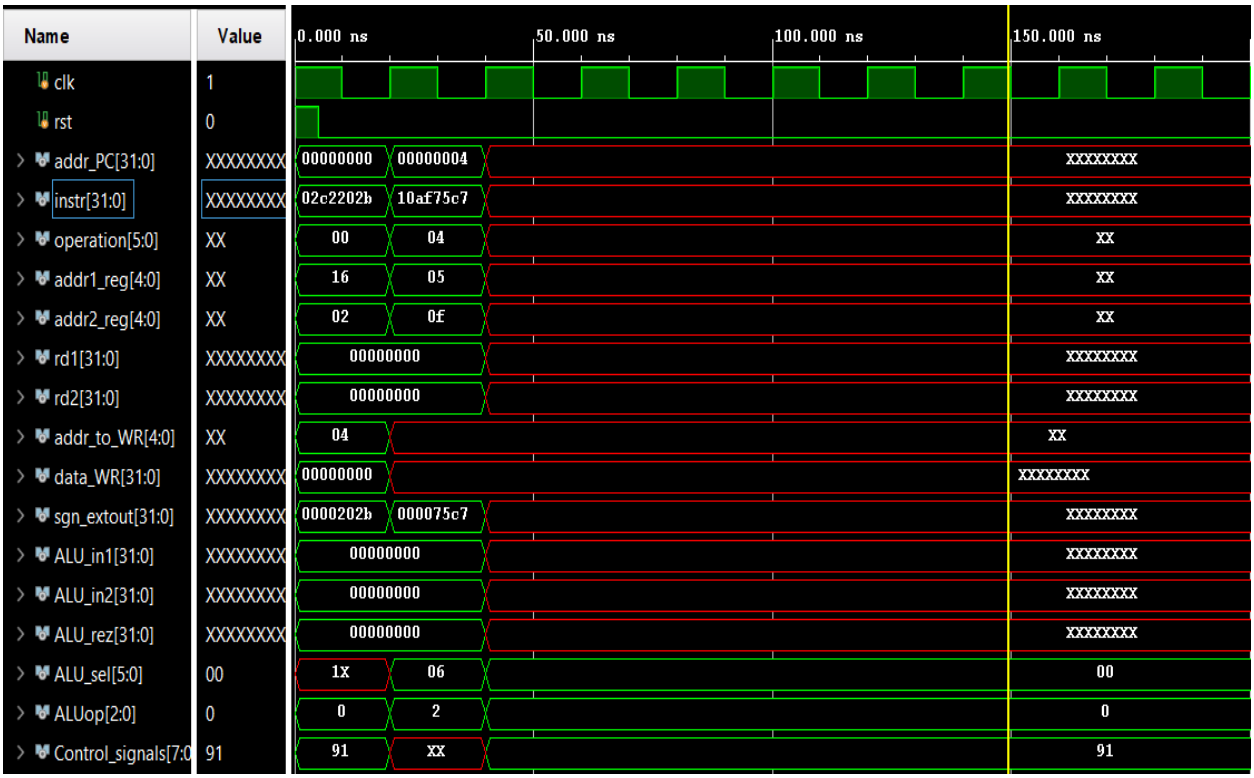


Figura 53. Test aleatoriu 1(waves)

Am generat 20 de instrucțiuni dintre care 10 de tip R și 10 de tip I. Se poate observa în figura 53 că după execuția celei de a doua instrucțiune apar doar valori de indiferență. Verificând fișierul în care se regăsește codul assembly observ că a doua instrucțiune este BEQ ce verifică dacă regiștri 5 și 15 sunt egali, iar dacă da va face un salt la adresa imediată 30151. Acest salt se face într-o zonă de memorie unde nu se află date despre nicio operație, așadar testul funcționează corespunzător returnând aceste valori de indiferență.

În figura din dreapta se pot observa instrucțiunile generate în urma execuției scriptului pentru generarea de operații.

asm.txt

```
1    MULL Rd(4)=Rs(22) MULL Rt(2);
2    BEQ [imm](30151)<-Rs(5) - Rt(15);
3    OR Rd(22)=Rs(27) OR Rt(29);
4    MULLI Rt(24)=Rs(16) MULLI imm(435);
5    OR Rd(19)=Rs(21) OR Rt(1);
6    ANDI Rt(12)=Rs(12) ANDI imm(11055);
7    MULL Rd(17)=Rs(29) MULL Rt(27);
8    SUBI Rt(30)=Rs(6) SUBI imm(5399);
9    SUB Rd(29)=Rs(19) SUB Rt(18);
10   SUBI Rt(17)=Rs(14) SUBI imm(19384);
11   NOR Rd(20)=Rs(25) NOR Rt(28);
12   ANDI Rt(31)=Rs(25) ANDI imm(341);
13   SUB Rd(11)=Rs(21) SUB Rt(6);
14   ORI Rt(22)=Rs(0) ORI imm(6819);
15   NOR Rd(8)=Rs(19) NOR Rt(17);
16   MULLI Rt(22)=Rs(30) MULLI imm(2893);
17   MULL Rd(6)=Rs(0) MULL Rt(16);
18   MULLI Rt(20)=Rs(22) MULLI imm(625);
19   NOR Rd(21)=Rs(16) NOR Rt(11);
20   LW Rt(26)<-Rs(10) + imm(26400);
21
```

Figura 54. Test aleatoriu 1(assembly)

În continuare voi genera un test ce va avea ca scop în primă fază inițializarea bancului de regiștri, iar mai apoi operații cu rol de a accesa memoria: scrieri și citiri din memoria de date.

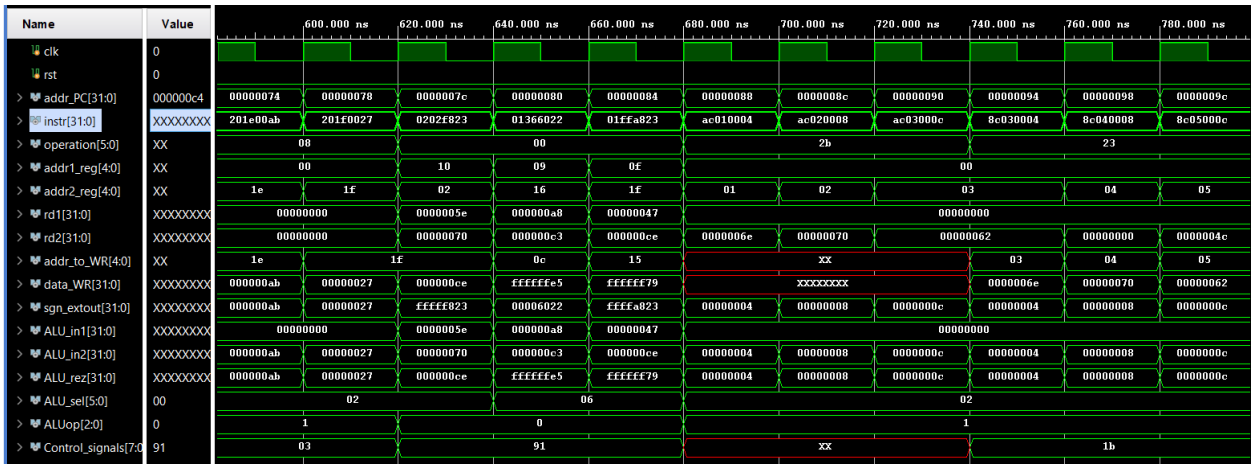


Figura 55. Test 2(waves)

În figura de mai sus se observă o ultimă parte a inițializării regiștrilor, urmat de 3 operații de tip R și anume: NOR, SUB, NOR și în final operații ce lucrează cu memoria, 3 scrieri la memorie urmat de 3 citiri din memorie de la aceleași adrese pentru a se observa datele mai ușor.

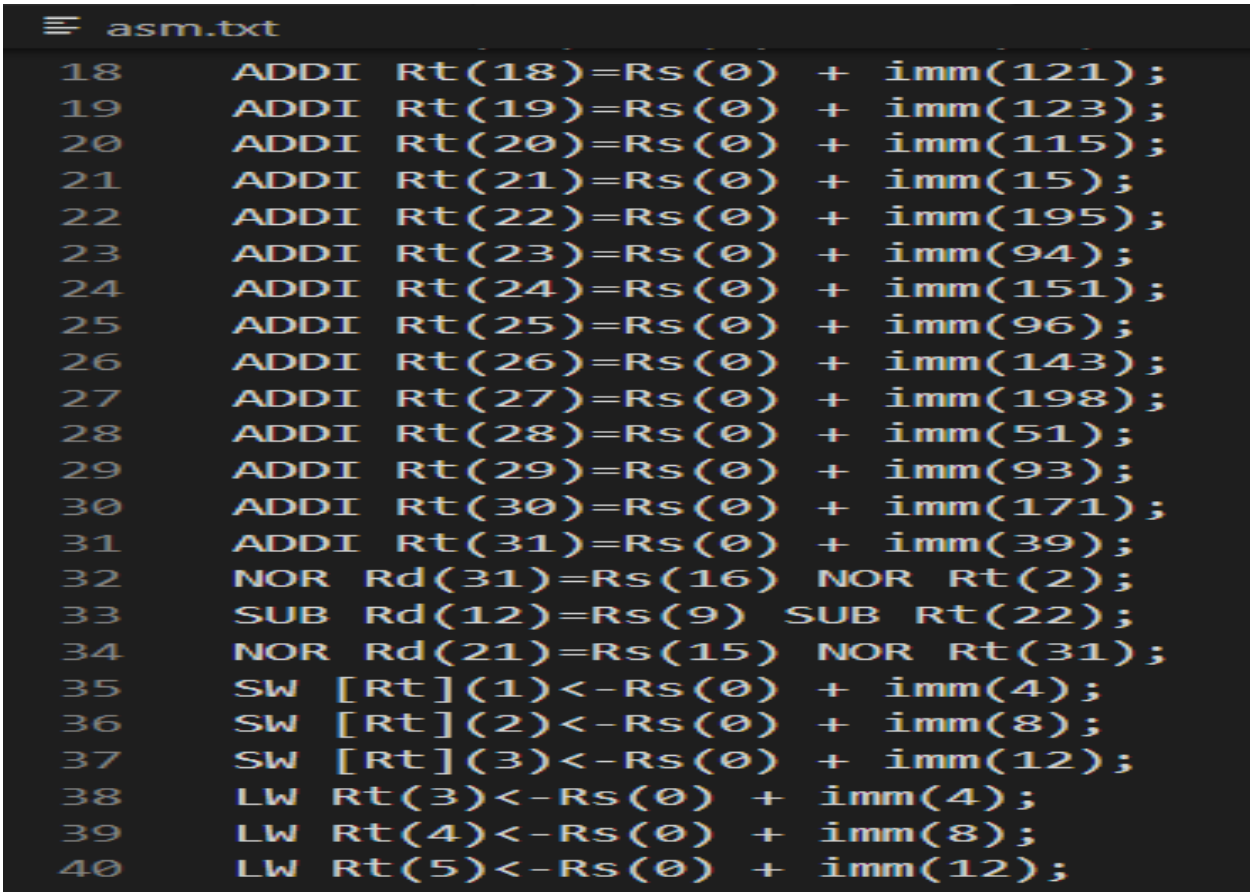


Figura 56. Test 2(assembly)

În figura de mai sus se observă valorile cu care sunt inițializați regiștri și adresele la care se face scriere și de la care se face citire.

5.3 Testare în cadrul unor aplicații

Aplicațiile cu care doresc să testez funcționalitatea procesorului și a operațiilor propuse sunt: calcularea termenilor din șirul lui Fibonacci, determinarea numerelor pare sau impare dintr-un șir.

Șirul lui Fibonacci, în cadrul acestei aplicații se va calcula numerele ce compun acest șir, de exemplu:0, 1, 1, 2, 3, 5, 8, 13. Se va face uz de 2 regiștri pentru calcularea următorului termen al șirului, un registru în care se va stoca o valoare utilizată pentru ieșirea din bucla de calcul, 2 regiștri pentru calculul viitoarei adrese deoarece după calcularea unui termen acesta va fi stocat în memoria de date, iar la final pentru o reprezentare grafică succesivă a termenilor calculați aceștia vor fi încărcați din memorie folosind instrucțiunea LW.

asm.txt

```
1   ADDI Rt(6)=Rs(0) + imm(10);
2   SW [Rt](1)<-Rs(5) + imm(0);
3   ADDI Rt(2)=Rs(1) + imm(1);
4   SW [Rt](2)<-Rs(0) + imm(4);
5   ADDI Rt(5)=Rs(0) + imm(4);
6   ADD Rd(1)=Rs(2) ADD Rt(1);
7   ADDI Rt(8)=Rs(5) + imm(4);
8   ADDI Rt(5)=Rs(8) + imm(0);
9   SW [Rt](1)<-Rs(5) + imm(0);
10  ADDI Rt(1)=Rs(2) + imm(0);
11  LW Rt(2)<-Rs(5) + imm(0);
12  SLT Rd(9)=Rs(6) SLT Rt(2);
13  BEQ [imm](20)<-Rs(0) - Rt(9);
14  LW Rt(10)<-Rs(11) + imm(0);
15  LW Rt(10)<-Rs(11) + imm(4);
16  LW Rt(10)<-Rs(11) + imm(8);
17  LW Rt(10)<-Rs(11) + imm(12);
18  LW Rt(10)<-Rs(11) + imm(16);
19  LW Rt(10)<-Rs(11) + imm(20);
20  LW Rt(10)<-Rs(11) + imm(24);
21  LW Rt(10)<-Rs(11) + imm(28);
22
```

```
I_type(2, 6, 0, 10); //ADDI
I_type(1, 1, 5, 0);  //SW
I_type(2, 2, 1, 1);  //ADDi
I_type(1, 2, 0, 4);  //SW
I_type(2, 5, 0, 4);  //ADDI
R_type(1, 1, 2, 3);  //ADD
I_type(2, 8, 5, 4);  //ADDI
I_type(2, 5, 8, 0);  //ADDI
I_type(1, 1, 5, 0);  //SW
I_type(2, 1, 2, 0);  //ADDI
I_type(0, 2, 5, 0);  //LW
R_type(2, 9, 6, 5);  //SLT
I_type(3, 9, 0, 20); //BEQ

for(i=0; i<8; i++)
{
    I_type(0, 10, 11, 0+i*4);
}
```

Figura 57. Cod asm generat(stânga), mod apelare constructori(dreapta)

În cele ce urmează voi prezenta rezultatele testului creat utilizând formele de undă generate.

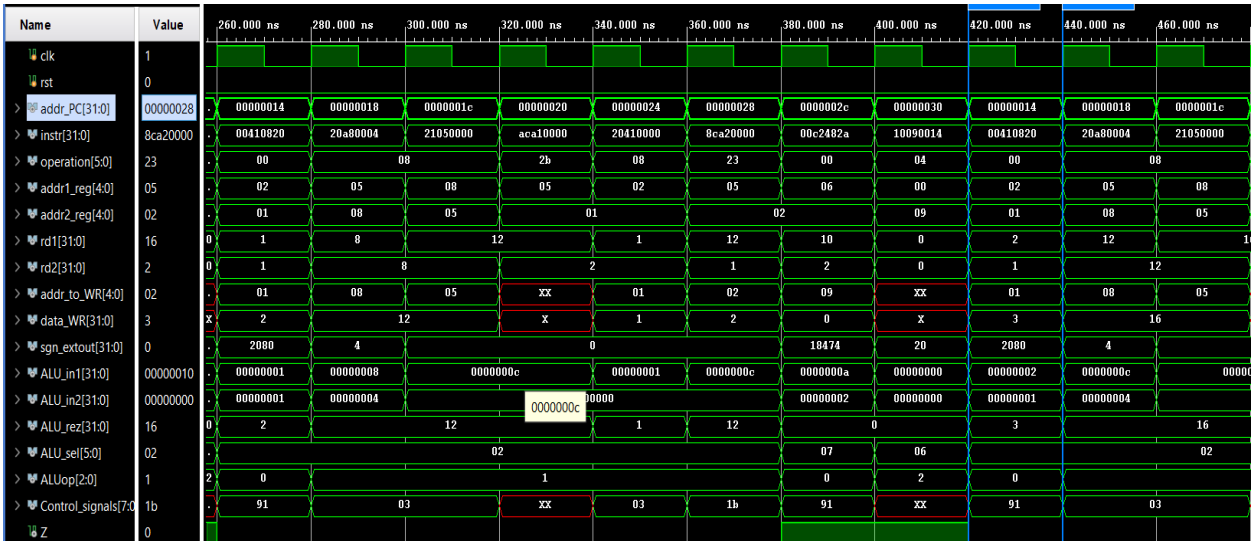


Figura 58. Secvență de calcul termen Fibonacci

În figura de mai sus am selectat una din secvențele ce se repetă pentru a calcula un termen din șirul lui Fibonacci, în acest caz valoarea calculată este 3(valoare evidențiată de markere), semnalul ALU_rez.

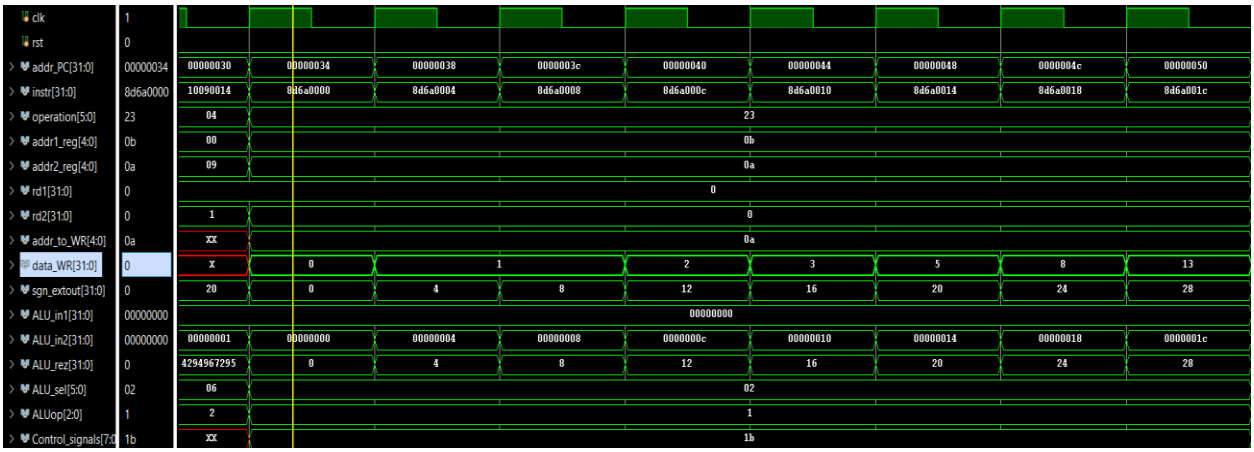


Figura 59. Termenii șirului Fibonacci

În figura de mai sus urmărind semnalul data_WR se pot observa primii 8 termeni din șirul lui Fibonacci.

Cea de a doua aplicație am secționat-o în 4 părți: partea de inițializare, identificarea numărului ca fiind par sau impar, încărcarea numărului într-un registru, pentru numere pare acest registru va fi reprezentat de R30, iar pentru numere impare va fi reprezentat de R20.

În partea de inițializare se va folosi o mască încărcată în R1 cu rol de a verifica cel mai puțin semnificativ bit al numărului testat, de asemena regiștri R2 și R3 vor fi inițializati cu valorile 1 respectiv 0. Mai departe vor fi generate numere aleatorii și încărcate în memoria de date fiind folosite mai târziu.

Identificarea numărului, în primă fază se calculează adresa de la care se dorește să se preia numărul din memorie urmând ca acesta să fie stocat în registrul R4, se aplică masca pentru identificarea valorii celui mai puțin semnificativ bit. După identificarea valorii acestuia se verifică, dacă este egal cu 0 atunci se va face jump în rutina pentru număr par, iar dacă este egal cu 1 se va face jump în rutina pentru număr impar.

Valoarea testată va fi încărcată în registrul specific: R30 pentru număr par, R20 pentru număr impar, la finalul fiecărei rutine de încărcare este apelată instrucțiunea jump ce va face salt înapoi pentru a se verifica următorul număr din memorie.

asm.txt

```
ADDI Rt(1)=Rs(0) + imm(1);
ADDI Rt(2)=Rs(0) + imm(1);
ADDI Rt(10)=Rs(0) + imm(143);
SW [Rt](0)<-Rs(10) + imm(0);
ADDI Rt(11)=Rs(0) + imm(112);
SW [Rt](0)<-Rs(11) + imm(4);
ADDI Rt(12)=Rs(0) + imm(36);
SW [Rt](0)<-Rs(12) + imm(8);
ADDI Rt(13)=Rs(0) + imm(304);
SW [Rt](0)<-Rs(13) + imm(12);
ADDI Rt(14)=Rs(0) + imm(399);
SW [Rt](0)<-Rs(14) + imm(16);
ADDI Rt(7)=Rs(6) + imm(4);
ADDI Rt(6)=Rs(7) + imm(0);
LW Rt(4)<-Rs(6) + imm(0);
AND Rd(5)=Rs(1) AND Rt(4);
BEQ [imm](72)<-Rs(2) - Rt(5);
BEQ [imm](84)<-Rs(3) - Rt(5);

ADDI Rt(20)=Rs(19) + imm(0);  //rutină număr impar
AND Rd(20)=Rs(0) AND Rt(4);
J  address(52);

ADDI Rt(30)=Rs(19) + imm(0);  //rutină număr par
AND Rd(30)=Rs(0) AND Rt(4);
J  address(52);

```

```
 int val;

 I_type(2, 1, 0, 1);
 I_type(2, 2, 0, 1);

//start
srand ((unsigned)time(NULL));
for(i=0;i<5;i++)
{
 val=rand() % 500;
 I_type(2, i+10, 0, val);  //rand val
 I_type(1, 0, i+10, i*4);  //addr to store
}

//start
I_type(2, 7, 6, 4); //next addr
I_type(2, 6, 7, 0); //next addr
I_type(0, 4, 6, 0); //LW
R_type(4, 5, 1, 0); //AND
I_type(3, 5, 2, 72);  //jmp numar impar
I_type(3, 5, 3, 84);  //jmp numar par

//impar
I_type(2, 20, 19, 0);
R_type(4, 20, 0, 0);
J_type(52);

//par
I_type(2, 30, 19, 0);
R_type(4, 30, 0, 0);
J_type(52);
```

Figura 60. Test generat(stânga), mod generare test(dreapta)

6.Concluzie

După terminarea lucrării de licență mă pot considera mulțumit de produsul final obținut. Sunt bucuros că am reușit să descriu o arhitectură de procesor și să o testez utilizând cunoștințele acumulate pe parcursul celor 4 ani de studii. A fost un drum plin de provocări, deși la începutul elaborării lucrării de licență am pornit cu un set de cunoștințe pe care eu îl consideram solid mi-am dat seama că la fiecare pas cu care înaintam mai am multe de învățat. După fiecare greșeală o nouă lecție mi-a fost predată astfel fiecare nouă etapă din desfășurarea proiectului fiind mai ușor de parcurs decât cea precedentă.

Din punctul meu de vedere arhitectura MIPS deși este învechită prezintă multe din caracteristicile procesoarelor moderne, ideile de bază rămânând aceleași. Înțelegând modul de funcționare și orientarea blocurilor componente conferă o bază solidă în înțelegerea arhitecturilor moderne.

Procesorul MIPS având o arhitectură de bază cu un set restrâns de instrucțiuni poate fi îmbunătățit în multe moduri.

În cadrul acestui proiect poate fi implementat pipelineul astfel crescând semnificativ viteza de execuție a instrucțiunilor.

O altă îmbunătățire poate fi reprezentată de extinderea părții hardware astfel realizând un set mai mare de instrucțiuni ce va ajuta la simplificarea codului.

În final aș dori să mulțumesc tuturor oamenilor ce au contribuit la dezvoltarea mea pe parcursul acestor 4 ani de studii: profesori, prieteni și colegi. Mereu am avut oameni în jur de la care am avut și am ce invăța, după acești 4 ani de studenție simt că aventura în domeniul microelectronicii abia începe.

7. Bibliografie

[ ] Digital Design and Co - autori David Harris, Sarah Harris

[ ] Computer Organization And Design The Hardware Software Interface - autor Patterson

[ ] https://en.wikipedia.org/wiki/Central_processing_unit

[ ] https://en.wikipedia.org/wiki/Semiconductor_industry

[ ] https://arstechnica.com/gadgets/2020/11/a-history-of-intel-vs-amd-desktop-performance-with-cpu-charts-galore/

[ ] https://en.wikipedia.org/wiki/Harvard_architecture

[ ] https://en.wikipedia.org/wiki/Von_Neumann_architecture

[ ] https://www.per-international.com/news-and-insights/risc-vs-cisc-architecture-which-is-better

[ ] https://en.wikipedia.org/wiki/Field-programmable_gate_array

[ ] https://en.wikipedia.org/wiki/Hardware_description_language

[ ] https://cloud.kyme32.ro/userfiles/_Anul_3/LMDSH/Curs/LMDSHNoteCurs.pdf

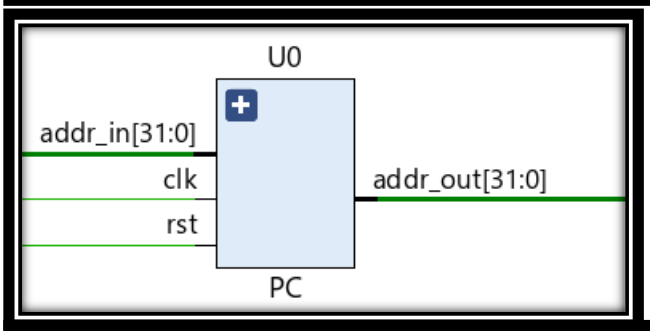
[ ] https://www.youtube.com/watch?v=YGSAWqQy9bI&ab_channel=ScottMoore

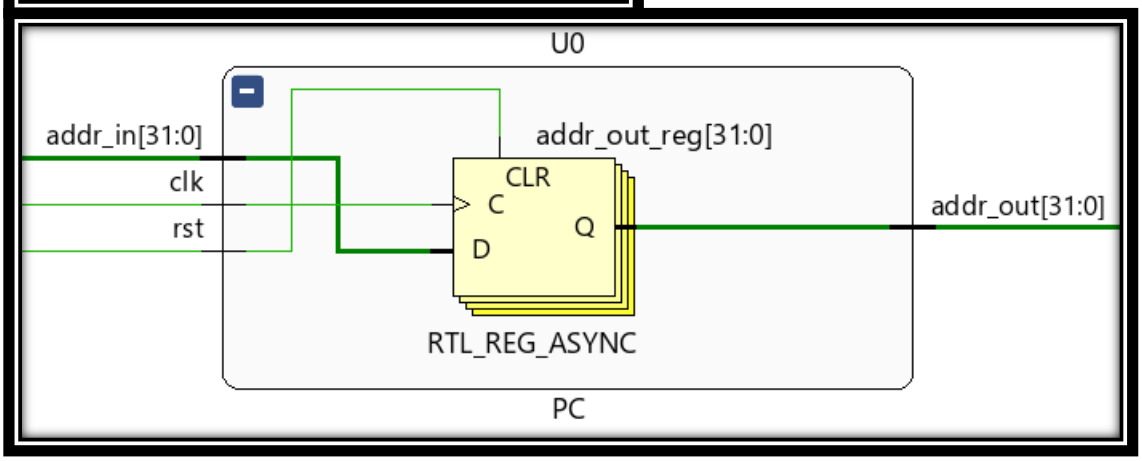
[]https://www.youtube.com/watch?v=oETOwVBzu1s&t=14s&ab_channel=ProgressiveLearningPlatform

8. Anexe

Anexa 1: Program Counter

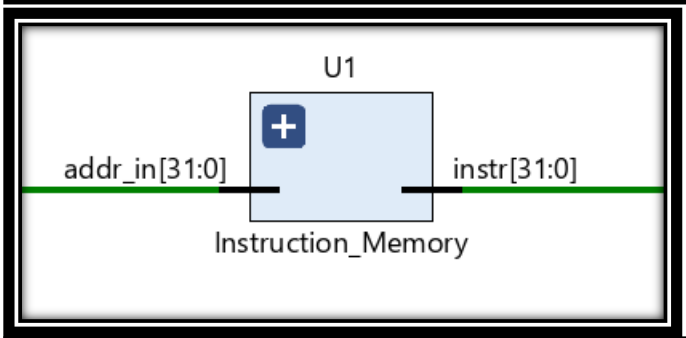
```
module PC(
    input [31:0] addr_in,
    output reg [31:0] addr_out,
    input clk,
    input rst
    );

    always@(posedge clk, posedge rst)
        begin
            if(rst)
                addr_out<=32'h0000_0000;
            else
                addr_out<=addr_in;
        end
endmodule
```

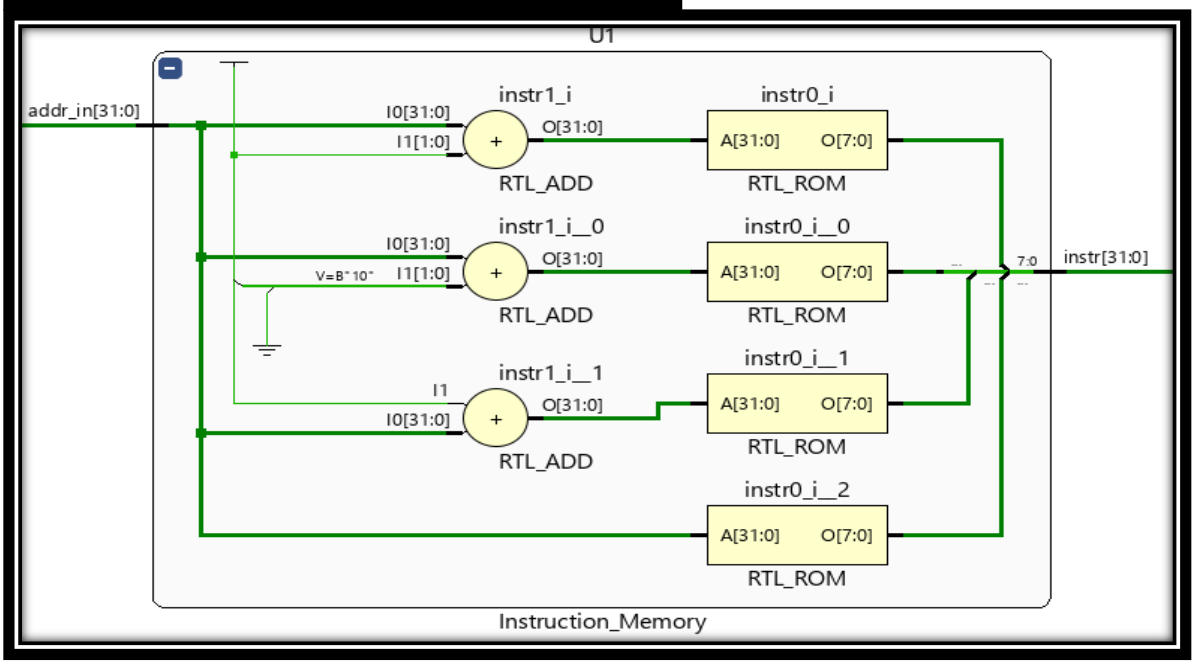




Anexa 2: Instruction memory

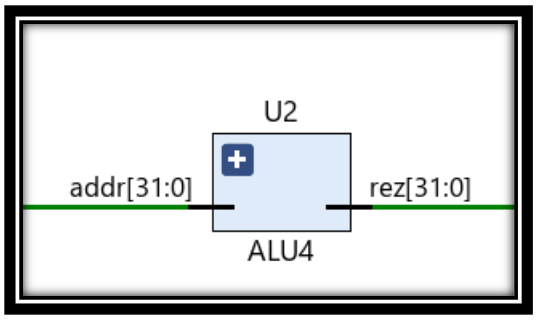
```
module Instruction_Memory(
    input  [31:0] addr_in,
    output [31:0] instr
    );

    reg [0:7] data [420:0];

    initial
        begin
        $readmemb("C:\\Work\\Licenta\\instructions.txt", data);
        end
    //adauga fisier cu instructiuni
    assign instr= {data[addr_in],data[addr_in+1],data[addr_in+2],data
    //0<->4 avem o instructiune de 32 biti, instructiunile in fisier
endmodule
```

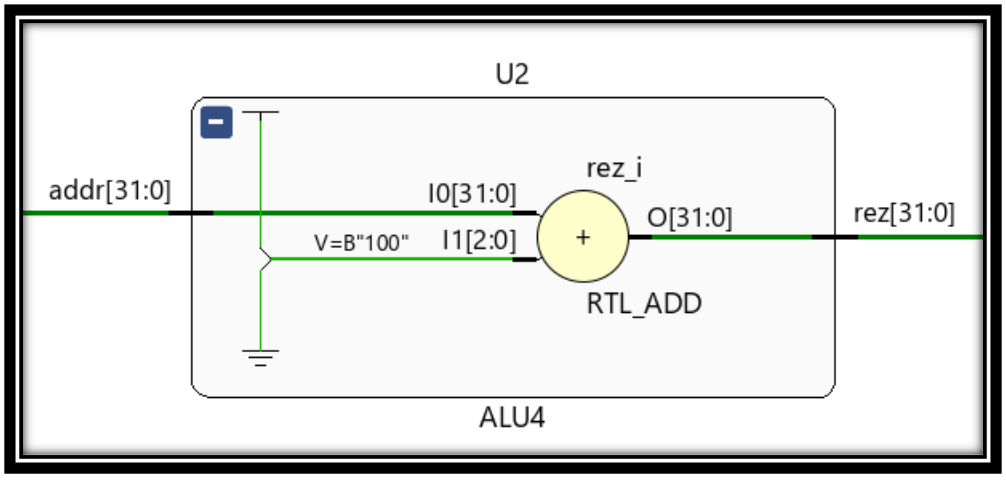




Anexa 3: ALU4

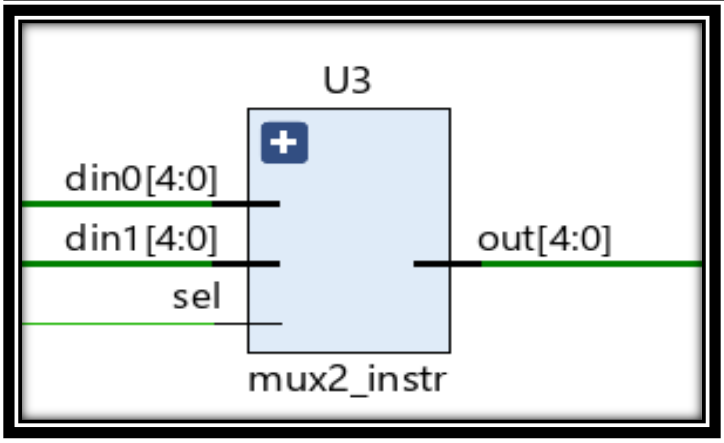
```
module ALU4(
    input [31:0] addr,
    output [31:0] rez
    );

    assign rez=addr+32'd4;

endmodule
```

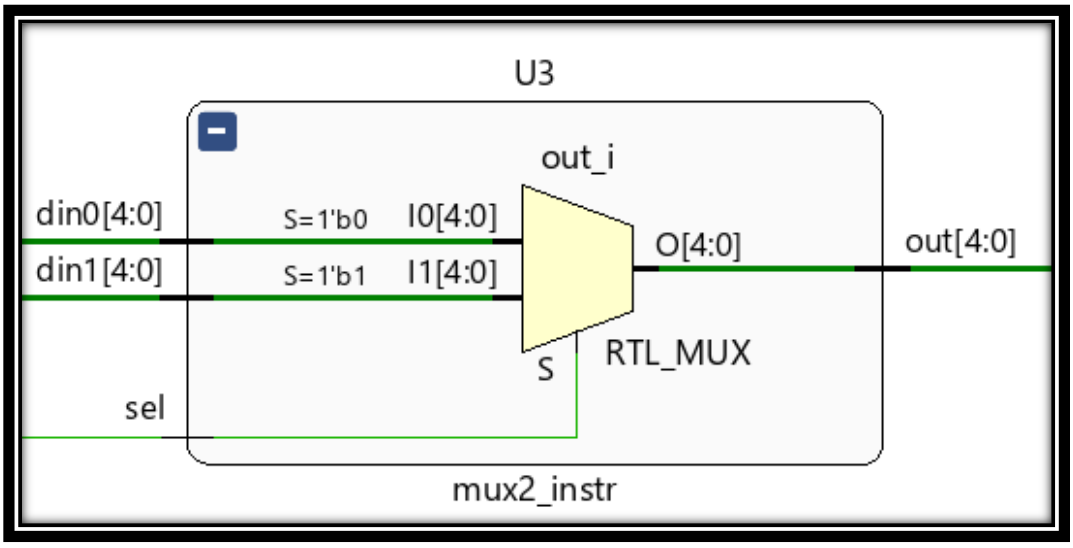




Anexa 4: MUX2_instr

```
module mux2_instr(
    input [4:0] din0,
    input [4:0] din1,
    input sel,
    output reg [4:0] out
    );

    always@*
        case(sel)
            1'b0: out=din0;
            1'b1: out=din1;
            default: out=5'bx;
        endcase
endmodule
```





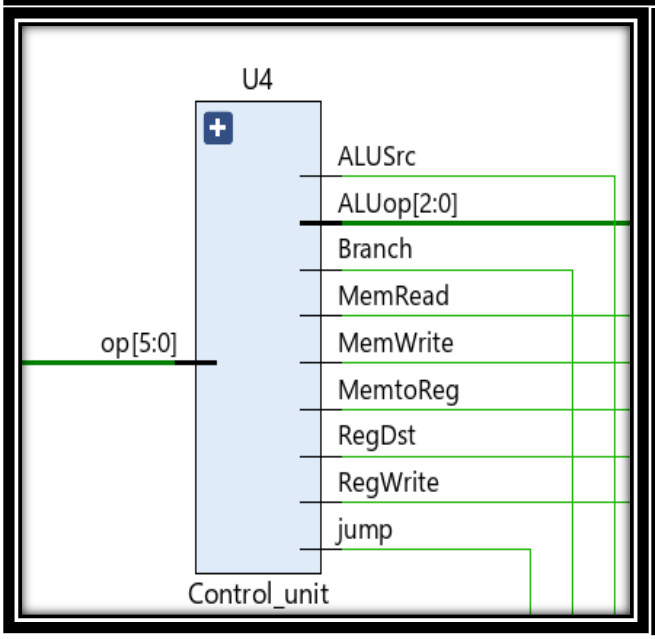
Anexa 5: Control Unit

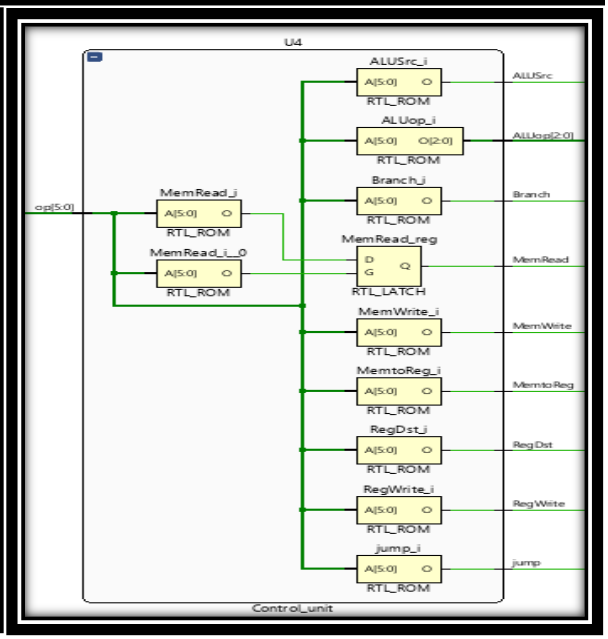
```
module Control_unit(
    input [5:0] op,
    output reg RegDst, jump, Branch, MemtoReg, ALUSrc, RegWrite, MemWrite, MemRead,
    output reg [2:0] ALUop
    );


    always@(op)
    begin
        casex(op)
            6'b000000: begin  //R Type instruction
            RegDst =1'b1; jump = 1'b0; ALUop = 3'b000; Branch = 1'b0; MemRead=1'b1;
            MemtoReg = 1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b0;
            end

            6'b100011: begin  //I Type lw instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b001; Branch =1'b0; MemRead=1'b1;
            MemtoReg =1'b1; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end
```

```
            6'b101011: begin  //I type sw instruction
            RegDst = 1'bx; jump =1'b0; ALUop = 3'b001; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'dX; RegWrite =1'b0; MemWrite =1'b1; ALUSrc=1'b1;
            end

            6'b001000: begin  //I type addi instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b001; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end

            6'b000100: begin  //I type beq instruction
            RegDst = 1'bX; jump =1'b0; ALUop = 3'b010; Branch =1'b1; MemRead=1'b0;
            MemtoReg = 1'bX; RegWrite =1'b0; MemWrite =1'b0; ALUSrc =1'b0;
            end

            6'b000001: begin  //I type subi instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b011; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end
```

```
            6'b000011: begin  //I type andi instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b100; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end

            6'b000111: begin  //I type ori instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b101; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end

            6'b001111: begin  //I type mulli instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b110; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end

            6'b011111: begin  //I type xori instruction
            RegDst =1'b0; jump =1'b0; ALUop = 3'b111; Branch =1'b0; MemRead=1'b0;
            MemtoReg =1'b0; RegWrite =1'b1; MemWrite =1'b0; ALUSrc =1'b1;
            end
```

```
            6'b000010: begin  //J type jmp instruction
            RegDst = 1'bX; jump =1'b1; ALUop = 2'bXX; Branch = 1'bX; MemRead=1'bx;
            MemtoReg = 1'bX; RegWrite =1'b0; MemWrite =1'b0;  ALUSrc = 1'bX;
            end

            default: begin
            RegDst =1'b0; jump =1'b0; ALUop =2'b00; Branch =1'b0; MemtoReg =1'b0; RegWrite =1'b0;
            MemWrite =1'b0; ALUSrc =1'b0;
            end
        endcase
    end
endmodule
```

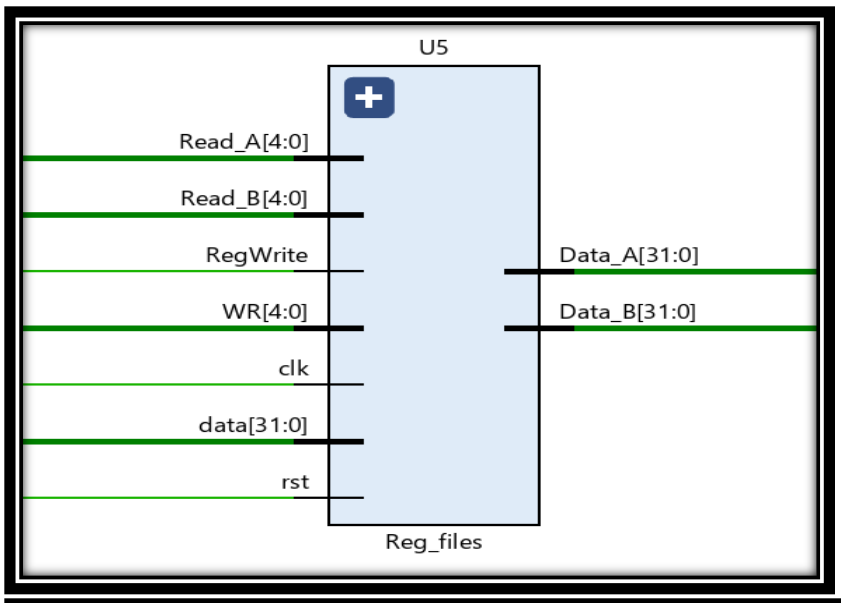


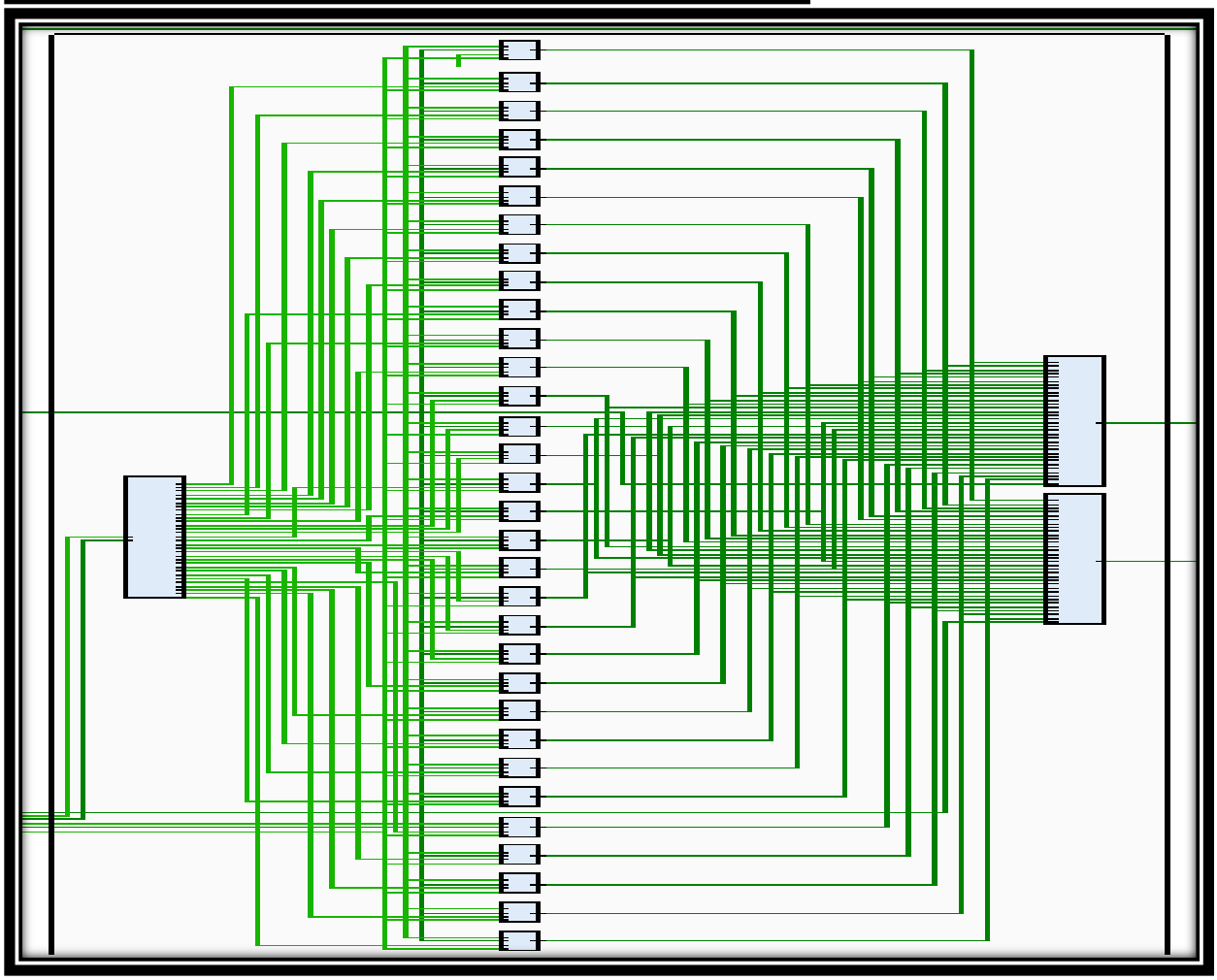


Anexa 6: Reg_files

```
module rpp32(
    input clk,
    input rst,
    input ld,
    input [31:0] d,
    output [31:0] q
    );

    reg [31:0] tmp;

    always@(posedge clk, posedge rst)
        begin
            if(rst)
              tmp <= 31'b0;
            else if(ld)
              tmp <= d;
        end

    assign q = tmp;
endmodule
```

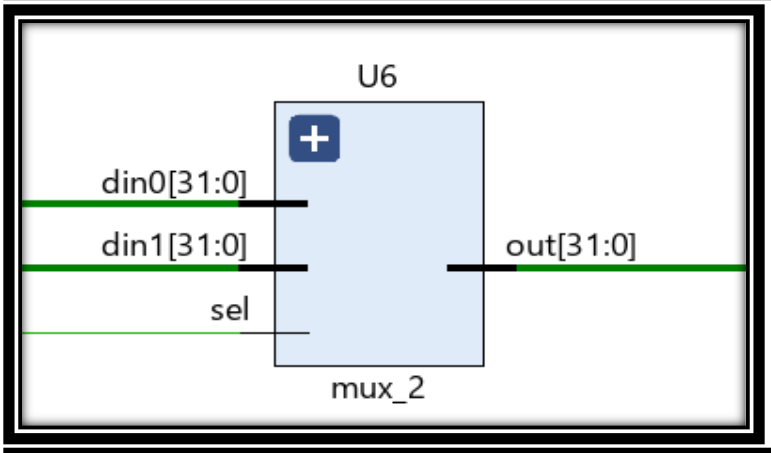
```
module Reg_files(
    input clk,
    input rst,
    input RegWrite,
    input [4:0] WR,
    input [4:0] Read_A,
    input [4:0] Read_B,
    output [31:0] Data_A,
    output [31:0] Data_B,
    input [31:0] data
    );
    wire [31:0] tmpLD;
    wire [31:0] tmpD [31:0];
    assign tmpLD[0] = 1'b0;
    generate
        genvar i;
            for(i=0; i<32; i=i+1)
            begin: struct_reg
                rpp32 Ux(.clk(clk), .rst(rst), .ld(tmpLD[i]), .d(data), .q(tmpD[i]));
            end
    endgenerate
```

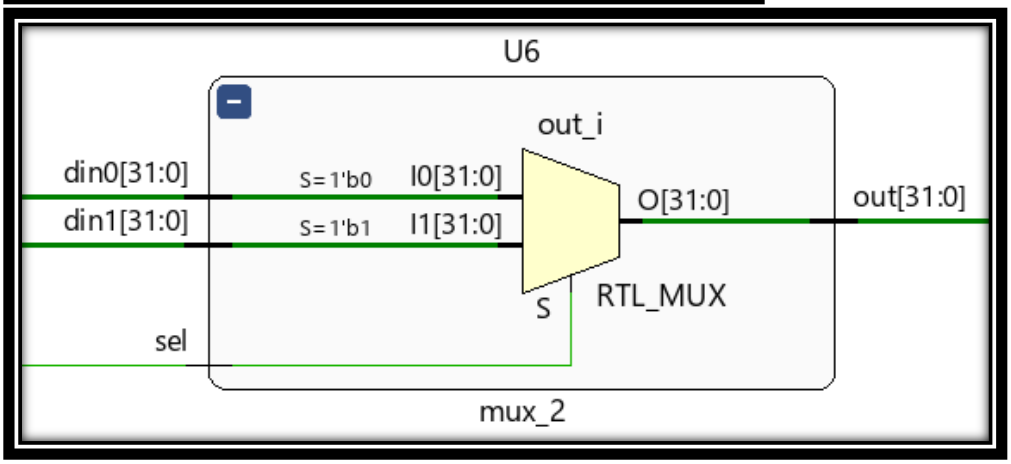

U5
Read_A[4:0]
Read_B[4:0]
RegWrite
WR[4:0]
clk
data[31:0]
rst
Data_A[31:0]
Data_B[31:0]
Reg_files



Anexa 7: MUX2:1

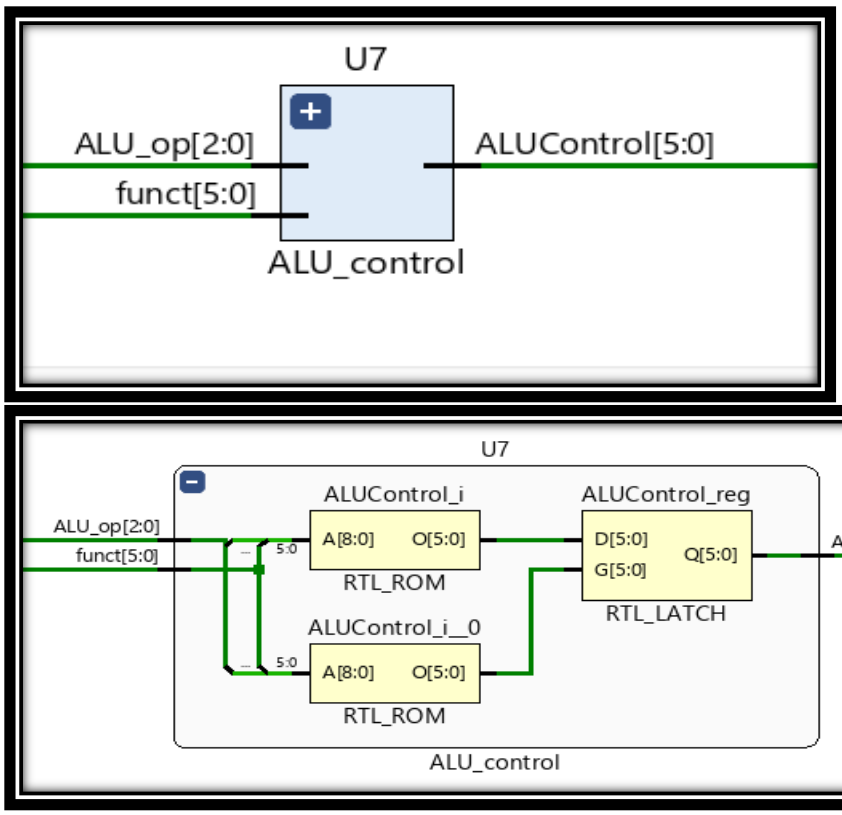
```
module mux_2(
    input [31:0] din0,
    input [31:0] din1,
    input sel,
    output reg [31:0] out
    );

    always@*
        case(sel)
            1'b0: out=din0;
            1'b1: out=din1;
            default: out=32'bx;
         endcase
endmodule
```





Anexa 8: ALU_Control

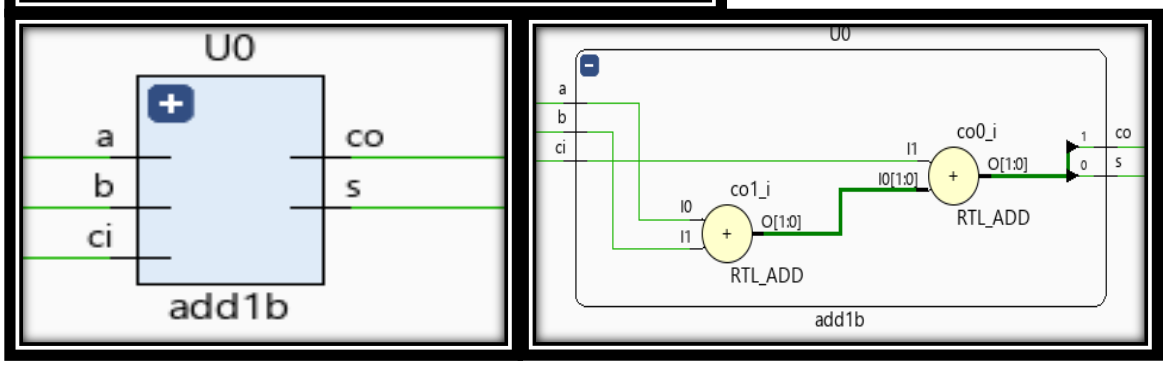
```
module ALU_control(
    input [5:0] funct,
    input [2:0] ALU_op,
    output reg [5:0] ALUControl
    );

     always @(ALU_op, funct) begin
      casex ({ALU_op , funct})
        9'b001_xxxxxx : ALUControl = 6'b000010; //lw/sw/addi
        9'b010_xxxxxx : ALUControl = 6'b000110; //beq
        9'b011_xxxxxx : ALUControl = 6'b000110; //subi
        9'b100_xxxxxx : ALUControl = 6'b000000; //andi
        9'b101_xxxxxx : ALUControl = 6'b000001; //ori
        9'b110_xxxxxx : ALUControl = 6'b01xxxx; //mulli
        9'b111_xxxxxx : ALUControl = 6'b10xxxx; //xori

        9'b000_100100 : ALUControl = 6'b000000;//and
        9'b000_100101 : ALUControl = 6'b000001;//or
        9'b000_000000 : ALUControl = 6'b001100;//nor
        9'b000_100000 : ALUControl = 6'b000010;//add
        9'b000_100010 : ALUControl = 6'b000110;//subtract
        9'b000_101010 : ALUControl = 6'b000111;//set less than
        9'b000_101011 : ALUControl = 6'b01xxxx;//mull
        9'b000_110001 : ALUControl = 6'b10xxxx;//XOR
        9'b000_110011 : ALUControl = 6'b11xxxx;//Sht_L


      endcase
    end
endmodule
```



Anexa 9: ALU_TOP

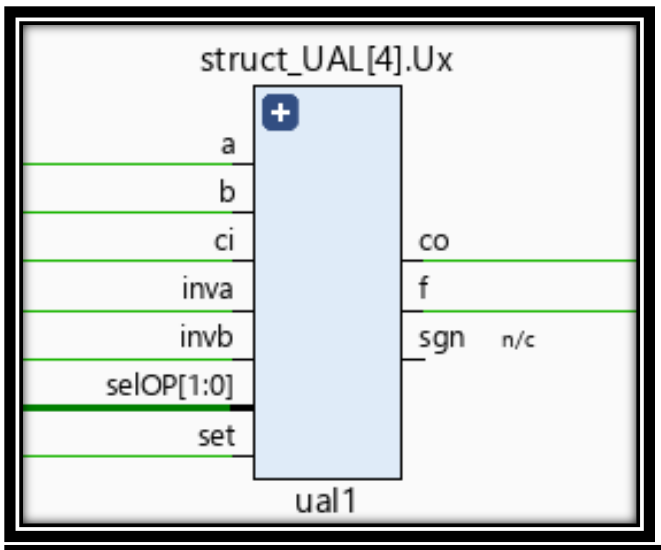
```
module add1b(
    input a,
    input b,
    input ci,
    output co,
    output s
    );

    assign {co,s} = a + b + ci;
endmodule
```

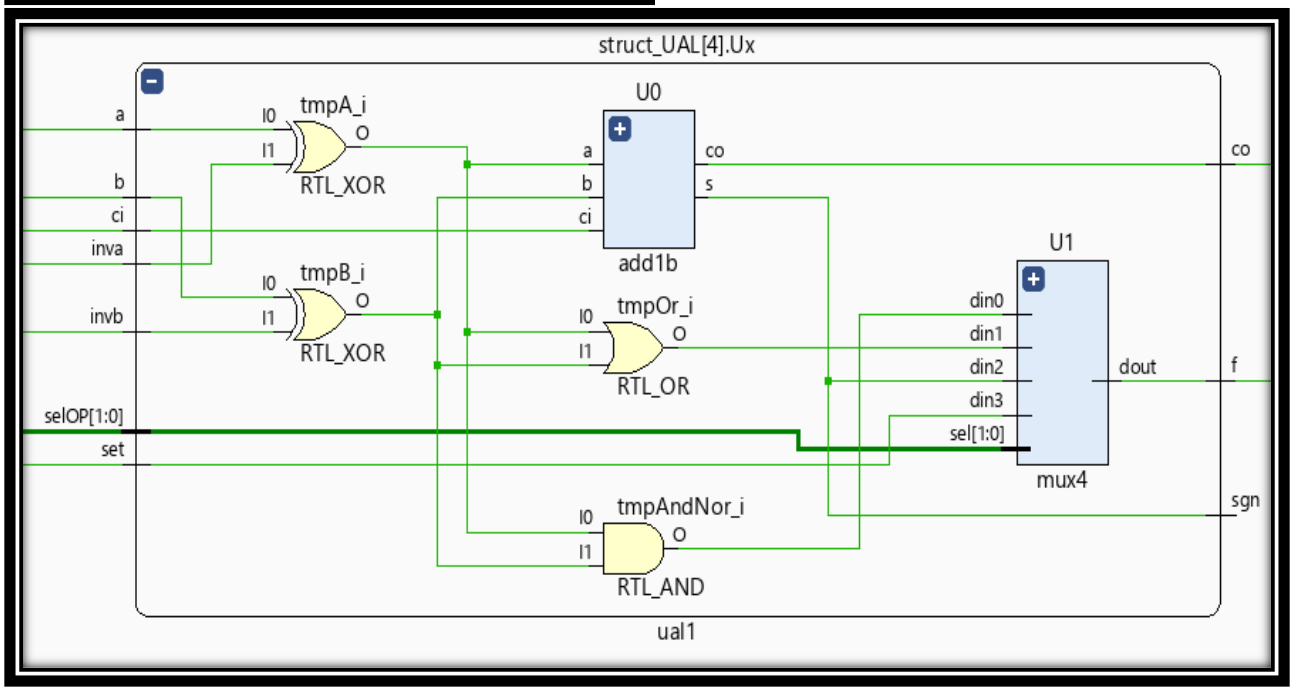
ADD1b



UAL1

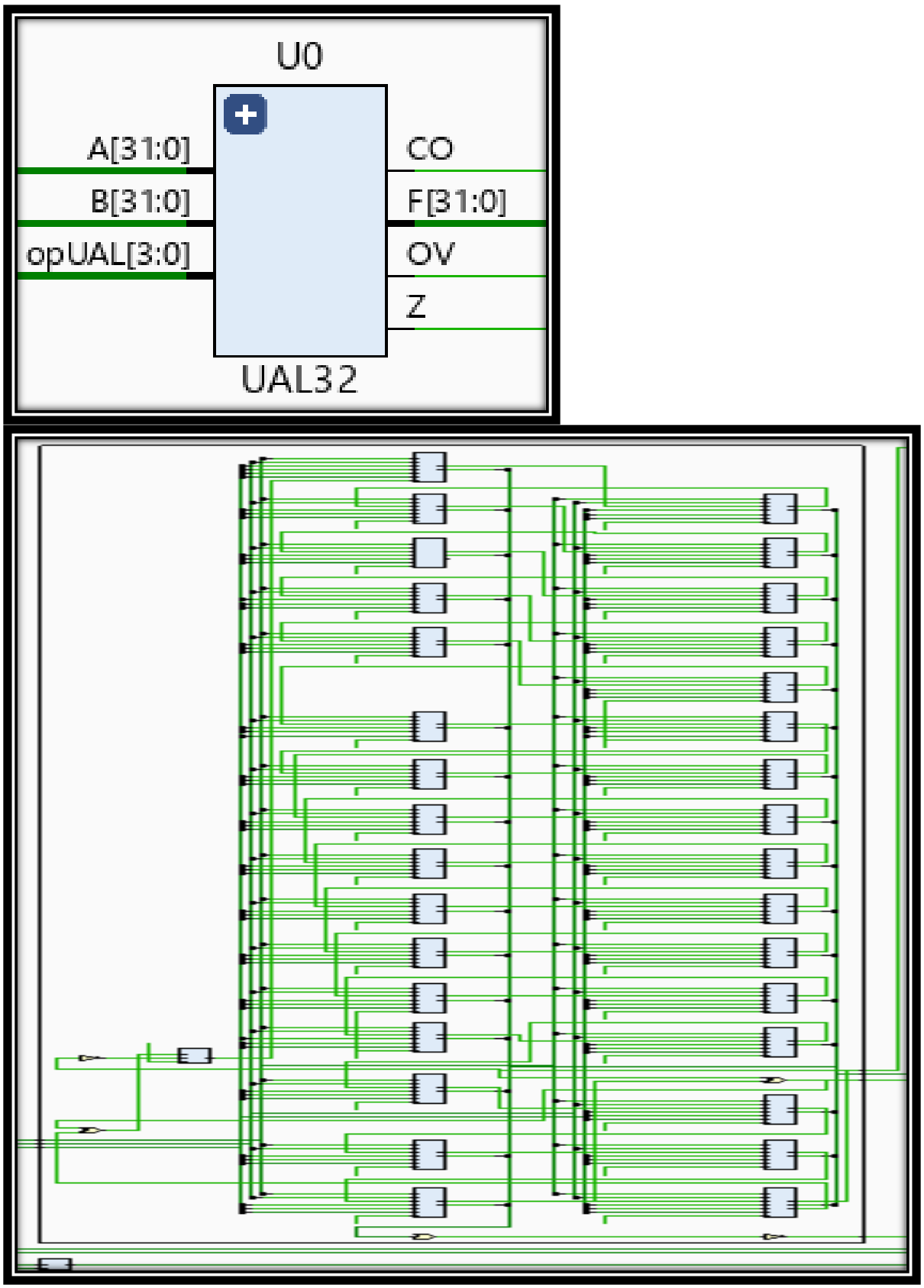
```
module ual1(
    input a,
    input b,
    input set,
    input inva,
    input invb,
    input ci,
    input [1:0] selOP,
    output co,
    output f,
    output sgn
    );


    wire tmpA, tmpB;
    wire tmpAndNor, tmpOR, tmpAddSub;

    assign tmpA      = a ^ inva;
    assign tmpB      = b ^ invb;
    assign tmpAndNor = tmpA & tmpB;
    assign tmpOr     = tmpA | tmpB;
    assign sgn = tmpAddSub;

    add1b U0(
    .a(tmpA),
    .b(tmpB),
    .ci(ci),
    .s(tmpAddSub),
    .co(co)
    );

    mux4 U1(
    .din0(tmpAndNor),
    .din1(tmpOr),
    .din2(tmpAddSub),
    .din3(set),
    .sel(selOP),
    .dout(f)
    );
endmodule
```

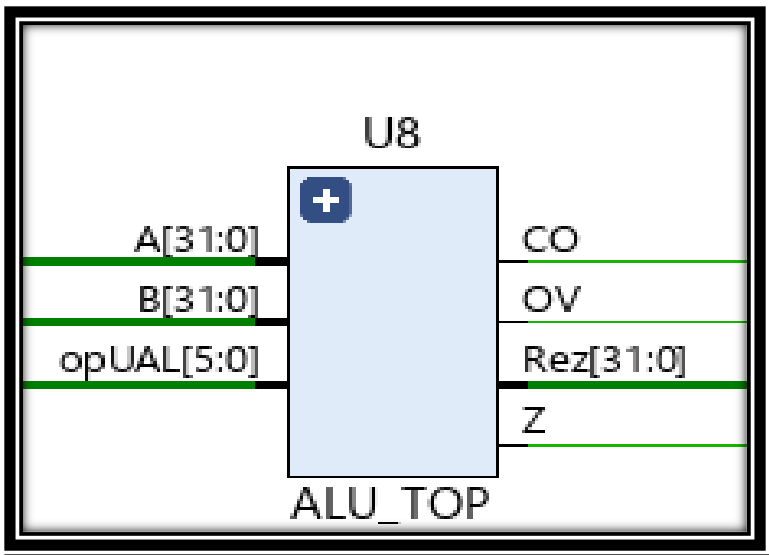

struct_UAL[4].Ux
a
b
ci
inva
invb
selOP[1:0]
set
co
f
sgn
n/c
ual1

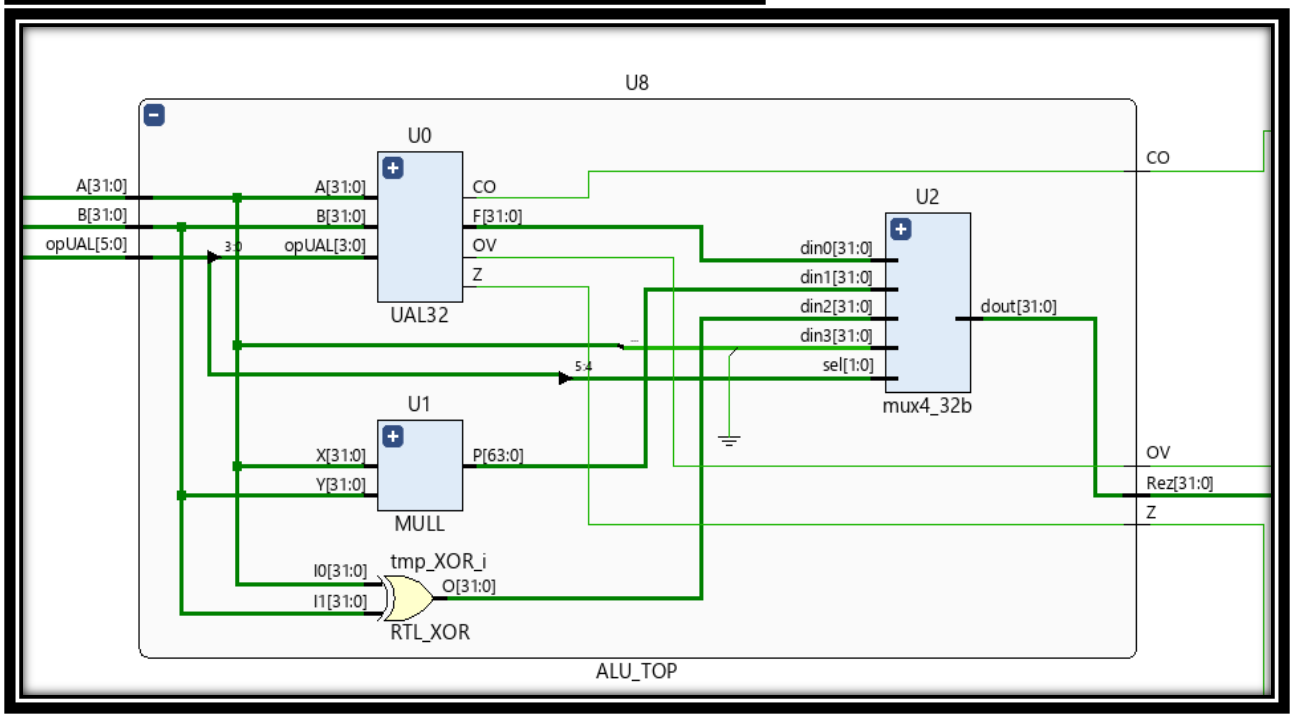

struct_UAL[4].Ux
a
b
ci
inva
invb
selOP[1:0]
set
tmpA_i
I0
I1
O
RTL_XOR
tmpB_i
I0
I1
O
RTL_XOR
U0
a
b
ci
co
s
add1b
tmpOr_i
I0
I1
O
RTL_OR
tmpAndNor_i
I0
I1
O
RTL_AND
U1
din0
din1
din2
din3
sel[1:0]
dout
mux4
co
f
sgn
ual1

UAL32

```
module UAL32(
    input [31:0] A,
    input [31:0] B,
    input [3:0] opUAL,
    output CO,
    output OV,
    output Z,
    output [31:0] F
    );

    wire [31:0] tmpSet, tmpSgn, tmpF;
    wire [32:0] tmpC;
    wire tmpOV, tmpSetS, tmpSetU, tmpSetFLSB;
    assign tmpC[0] = opUAL[2];

    generate
        genvar i;
        for(i=0; i<=31; i=i+1)
            begin: struct_UAL
            ual1 Ux(
            .invb(opUAL[2]),
            .inva(opUAL[3]),
            .ci(tmpC[i]),
            .f(tmpF[i]),
            .sgn(tmpSgn[i]),
            .a(A[i]), .b(B[i]),
            .set(tmpSet[i]),
            .co(tmpC[i+1]),
            .selOP(opUAL[1:0])
            );

            end

    endgenerate

    mux2 Uy(.din0(tmpSetS), .din1(tmpSetU), .sel(1'b1), .dout(tmpSe

    assign tmpSetU = ~tmpC[32];
    assign tmpOV = tmpC[32]^tmpC[31];
    assign tmpSetS = tmpOV ^ tmpSgn[31];
    assign CO = tmpC[32];
    assign OV = tmpOV;
    assign tmpSet[0]=tmpSetFLSB;
    assign tmpSet[31:1]=31'b0;
    assign F=tmpF;
    assign Z=~(|tmpF);

endmodule
```

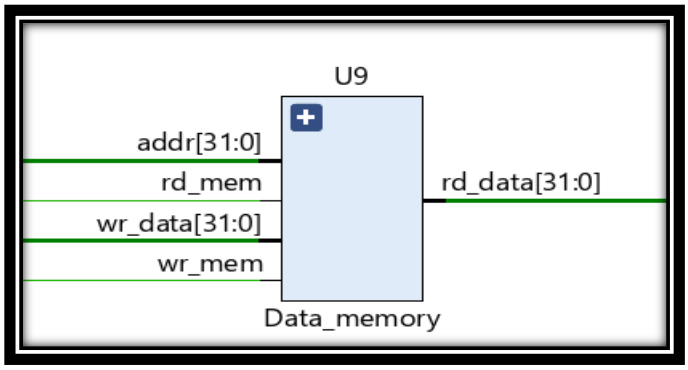

U0
A[31:0]
B[31:0]
opUAL[3:0]
CO
F[31:0]
OV
Z
UAL32


U8
A[31:0]
B[31:0]
opUAL[5:0]
CO
OV
Rez[31:0]
Z
ALU_TOP


U8
U0
A[31:0]
B[31:0]
opUAL[5:0]
A[31:0]
B[31:0]
opUAL[3:0]
CO
F[31:0]
OV
Z
UAL32
U2
din0[31:0]
din1[31:0]
din2[31:0]
din3[31:0]
sel[1:0]
dout[31:0]
mux4_32b
U1
X[31:0]
Y[31:0]
P[63:0]
MULL
tmp_XOR_i
I0[31:0]
I1[31:0]
O[31:0]
RTL_XOR
CO
OV
Rez[31:0]
Z
ALU_TOP

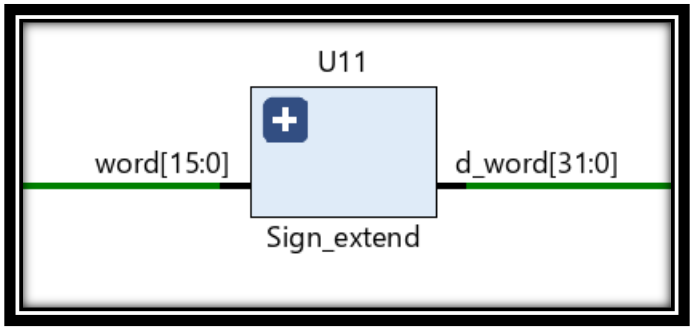
Anexa 10: Data memory

```
module Data_memory(
    input wr_mem,
    input rd_mem,
    input [31:0] addr,
    input [31:0] wr_data,
    output reg [31:0] rd_data
    );

    reg [7:0] rom [256:0];

    always@(addr)
        begin
            if(wr_mem)
                begin
                rom[addr]= wr_data[31:24];
                rom[addr+1]=wr_data[23:16];
                rom[addr+2]=wr_data[15:8];
                rom[addr+3]=wr_data[7:0];
                end
            else if(rd_mem)
                 begin
                 rd_data[31:24]= rom[addr];
                 rd_data[23:16]= rom[addr+1];
                 rd_data[15:8]= rom[addr+2];
                 rd_data[7:0]= rom[addr+3];
                 end
         end

endmodule
```



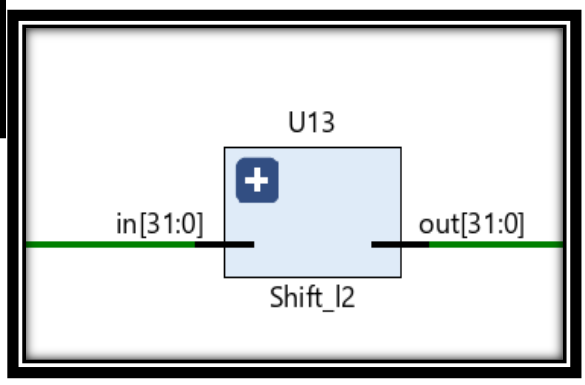
Anexa 11: Sign Extend

```
module Sign_extend(
    input [15:0] word,
    output [31:0] d_word
    );

    assign d_word={{16{word[15]}},word};

endmodule
```



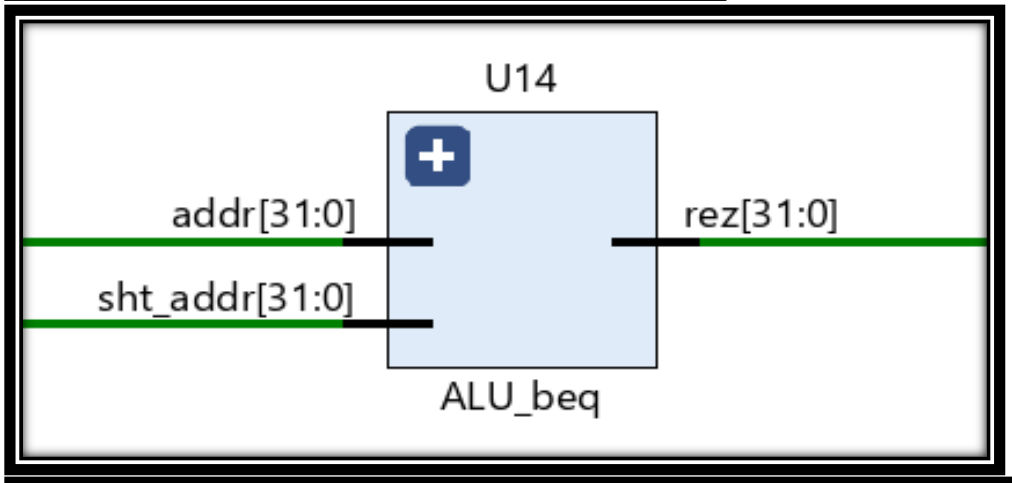
Anexa 12: Shift_l2

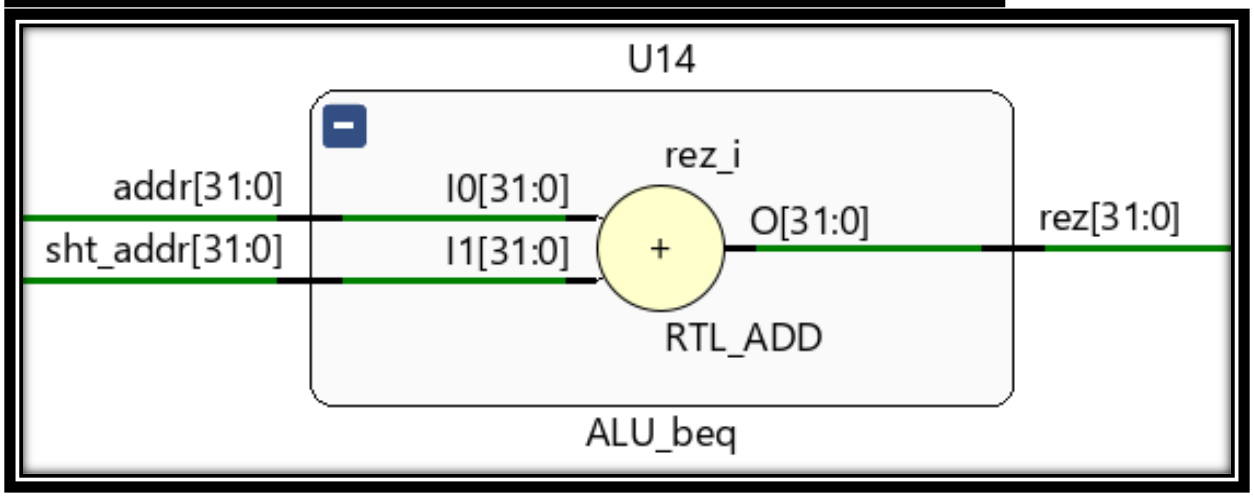
```
module Shift_l2(
    input [31:0] in,
    output reg [31:0] out
    );

    always@*
        begin
            out={in[29:0],2'b00};
        end
endmodule
```



Anexa 13: ALU_beq

```
module ALU_beq(
    input [31:0] addr,
    input [31:0] sht_addr,
    output [31:0] rez
    );

    assign rez=addr+sht_addr;
endmodule
```

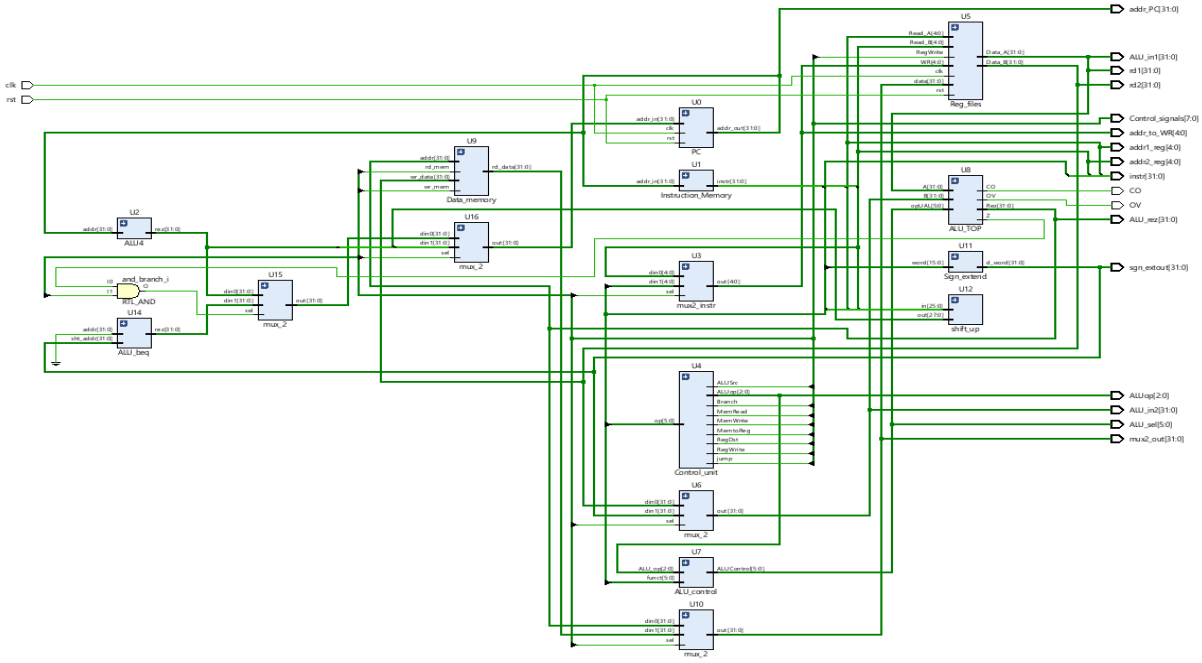




Anexa 14:MIPS

```
module MIPS(
        input clk,
        input rst,
       output [31:0] rd1, rd2, mux2_out, addr_PC, instr,
       output [7:0] Control_signals,
       output [2:0] ALUop,
       output [5:0] ALU_sel,
       output [4:0] addr1_reg, addr2_reg, addr_to_WR,
       output [31:0] ALU_rez,
       output [31:0] ALU_in1, ALU_in2,
       output [31:0] sgn_extout,
        output CO, OV
);


    wire [31:0] tmp_addr, tmp_addr1, tmp_alu4, tmp_instr;  //PC to Instr memory signals

    wire tmp_RegDst, tmp_Jump, tmp_Branch, tmp_MemRead, tmp_MemtoReg, tmp_MemWrite, tmp_ALUSrc, tmp_RegW
    wire [2:0] tmp_ALUOp; //signal Control Unit<=> ALU
    wire [4:0] tmp_WR; //signal Control Unit <=> Reg File
    wire [31:0] data_to_RF;  //signal MUX_2 <=> reg File
     wire [31:0] data_to_RF;  //signal MUX_2 <=> reg File
     wire [31:0] tmp_RD1, tmp_RD2; //signal Reg File <=> ALU
     wire [31:0] tmp_sgn_ext;  // signal Sign Extend <=> mux_2, shift_l2
     wire [31:0] tmp_ALU_in2;  // signal mux_2 <=> ALU
     wire [5:0]  tmp_ALU_ctrl; // signal ALU Control <=> ALU32
     wire [31:0] tmp_ALU_rez;  // signal ALU <=> Data memory
     wire tmp_zero;            // signal ALU <=> AND gate used for branch
     wire [31:0] tmp_data_mem; //signal Data memory <=> Mux2
     wire [27:0] tmp_shift; //signal out shift up
     wire [31:0] tmp_jmp_addr;  // {tmp_alu4[31:28],tmp_shift}
     wire [31:0] tmp_rez_beq;   //ALU beq <=> Mux 2
     wire and_branch;           //AND gate <=> Mux 2
     wire [31:0] mux_to_mux;    //Mux 2 <=> Mux 2

     assign Control_signals={tmp_RegDst, tmp_Jump, tmp_Branch, tmp_MemRead, tmp_MemtoReg, tmp_MemWrite, tr

     //Instruction manipulating
     PC U0(.addr_in(tmp_addr), .addr_out(tmp_addr1), .clk(clk), .rst(rst));
    ALU4 U2(.addr(tmp_addr1), .rez(tmp_alu4));  //Next addr calculation

    //Decoding address
    mux2_instr U3(.din0(tmp_instr[20:16]), .din1(tmp_instr[15:11]), .sel(tmp_RegDst), .out(tmp_WR));

    Control_unit U4(.op(tmp_instr[31:26]), .RegDst(tmp_RegDst), .jump(tmp_Jump),
                    .Branch(tmp_Branch), .MemRead(tmp_MemRead), .MemtoReg(tmp_MemtoReg), .ALUSrc(tmp_ALU
                    .RegWrite(tmp_RegWrite), .MemWrite(tmp_MemWrite), .ALUop(tmp_ALUOp));


    //Data calling
    Reg_files U5(.clk(clk), .rst(rst), .Read_A(tmp_instr[25:21]), .Read_B(tmp_instr[20:16]),
                    .WR(tmp_WR), .data(data_to_RF), .Data_A(tmp_RD1), .Data_B(tmp_RD2),
                    .RegWrite(tmp_RegWrite));

    //Execution
    mux_2 U6(.din0(tmp_RD2), .din1(tmp_sgn_ext), .sel(tmp_ALUSrc), .out(tmp_ALU_in2));

    ALU_control U7(.funct(tmp_instr[5:0]), .ALU_op(tmp_ALUOp), .ALUControl(tmp_ALU_ctrl));

    ALU_TOP U8(.A(tmp_RD1), .B(tmp_ALU_in2), .opUAL(tmp_ALU_ctrl), .Rez(tmp_ALU_rez), .Z(tmp_zero), .CO(
```

```
    Data_memory U9(.addr(tmp_ALU_rez), .wr_data(tmp_RD2), .rd_data(tmp_data_mem),
                        .wr_mem(tmp_MemWrite), .rd_mem(tmp_MemRead));

    mux_2 U10(.din0(tmp_ALU_rez), .din1(tmp_data_mem), .sel(tmp_MemtoReg), .out(data_to_RF));

    //Next address calculation
    Sign_extend U11(.word(tmp_instr[15:0]), .d_word(tmp_sgn_ext));

    shift_up U12(.in(tmp_instr[25:0]), .out(tmp_shift));

    assign tmp_jmp_addr={tmp_alu4[31:28],tmp_shift};

    ALU_beq U14(.addr(1'b0), .sht_addr(tmp_sgn_ext), .rez(tmp_rez_beq));

    assign and_branch=tmp_zero&tmp_Branch;
    mux_2 U15(.din0(tmp_alu4), .din1(tmp_rez_beq), .sel(and_branch), .out(mux_to_mux));

    mux_2 U16(.din0(mux_to_mux), .din1(tmp_jmp_addr), .sel(tmp_Jump), .out(tmp_addr));


    assign rd1=tmp_RD1;   //lines only for test
    assign rd2=tmp_RD2;
    assign mux2_out=data_to_RF;
    assign addr_PC=tmp_addr1;
    assign instr=tmp_instr;
    /*assign RegDst=tmp_RegDst;
    assign RegWrite=tmp_RegWrite;
    assign ALUop=tmp_ALUOp;
    assign MemRead=tmp_MemRead;*/
    assign addr1_reg=tmp_instr[25:21];
    assign addr2_reg=tmp_instr[20:16];
    assign addr_to_WR=tmp_WR;
    assign ALU_rez=tmp_ALU_rez;
    assign ALUop=tmp_ALUOp;
    assign ALU_sel=tmp_ALU_ctrl;
    assign ALU_in1=tmp_RD1;
    assign ALU_in2=tmp_ALU_in2;
    assign sgn_extout=tmp_sgn_ext;
endmodule
```



Anexa 15: Script testare

```
#include <iostream>
#include <time.h>
#include <fstream>
using namespace std;

class R_type{
   public:
   int op[6]={0,0,0,0,0,0};  //R-type operation

   int functions_matrix[9][6]={{1,0,0,1,0,0}, {1,0,0,1,0,1}, {1,0,0,0,1,1}, {1,0,0,0,0,0}, //AND
//OR //NOR //ADD
                        {1,0,0,0,1,0}, {1,0,1,0,1,0}, {1,0,1,0,1,1}, {1,1,0,0,0,1}, {1,1,0,0,1,1}};
//SUB  //SLT  //MULL  //XOR  //SHT_L

   int rd, rs, rt;  //addresses for my registers
   int vrd[5]={0,0,0,0,0}, vrt[5]={0,0,0,0,0}, vrs[5]={0,0,0,0,0};
   int shmt[5]={0,0,0,0,0};

   R_type(){

      int i=0;  //auxiliar variables to calculate 32 bit instruction
      rd=rand() % 32;  //randomizing a number between 0->32 to get an address for my
destination register

      cout<<"Register generator:"<<endl;
      cout<<"RD="<<rd<<" ";

      int tmp_rd;  tmp_rd=rd;   // saving variable into an temporary variable to use it later at
asm.txt file

      while(rd!=0){   // conversion to binary
            vrd[i]=rd%2;      // binary value after division
      rd=rd/2;
      i++;
            }

      for(i=4; i>=0; i--){  //reading value in binary
         cout<<vrd[i]<<" ";
       }

      cout<<"\n";
      i=0;  //auxiliar variables to calculate 32 bit instruction
```

```
	rs=rand() % 32;  //randomizing a number between 0->32 to get an address for my
destination register
	cout<<"RS="<<rs<<" ";

	int tmp_rs;  tmp_rs=rs;   // saving variable into an temporary variable to use it later at
asm.txt file

	while(rs!=0){   // conversion to binary
		vrs[i]=rs%2;      // binary value after division
	rs=rs/2;
	i++;
		}

	for(i=4; i>=0; i--){  //reading value in binary
	   cout<<vrs[i]<<" ";
	}

	cout<<endl;

	i=0;  //auxiliar variables to calculate 32 bit instruction
	rt=rand() % 32;  //randomizing a number between 0->32 to get an address for my
destination register
	cout<<"RT="<<rt<<" ";

	int tmp_rt;  tmp_rt=rt;   // saving variable into an temporary variable to use it later at asm.txt
file

	while(rt!=0){   // conversion to binary
			vrt[i]=rt%2;      // binary value after division
	   rt=rt/2;
	   i++;
		}

	for(i=4; i>=0; i--){  //reading value in binary
	   cout<<vrt[i]<<" ";
	}

	cout<<endl;
	int instruction[31];
	int funct;
	funct=rand() % 8;  //randomizing a number between 0->8 to pick an function
	cout<<"Functia NR="<<funct<<" ";
	cout<<endl;

	cout<<"Functia=";
	for(i=0; i<6; i++){
```

```
        cout<<functions_matrix[funct][i]<<" ";  //display function
    }

    for(i=0; i<6; i++){
        instruction[i+26]=functions_matrix[funct][i];  //writing function and operation in
instruction vector
        instruction[i]=op[i];
    }

    int j=0;
    for(i=0; i<=4; i++)
    {
        instruction[i+6]=vrs[4-j];      //writing register addresses in instruction vector
        instruction[i+11]=vrt[4-j];
        instruction[i+16]=vrd[4-j];
        instruction[i+21]=shmt[4-j];   //shamt value also is written here
        j++;
    }
    cout<<endl;
    cout<<"Instructiune=";
    for(i=0; i<32; i++){
        cout<<instruction[i]<<" ";  //display vector
    }

    ofstream file;
    file.open("instructions.txt", ofstream::app);  // open file and continue writing in that
    for(i=1; i<=32; i++){
        file<<instruction[i-1];
        if(i%8==0){
            file<<endl;
        }
    }

    file.close();

    //funct->numarul functiei 0->AND 1->OR 2->NOR 3->ADD 4->SUB  5->SLT  6->MULL
7->XOR  8->SHT_L
    ofstream fila;
    fila.open("asm.txt", ofstream::app);  //asembly file creation

        switch(funct){
            case 0:
                fila<<"AND Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") AND
Rt("<<tmp_rt<<");"<<endl;
            break;
```

```
            case 1:
               fila<<"OR Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") OR Rt("<<tmp_rt<<");"<<endl;
            break;

            case 2:
               fila<<"NOR Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") NOR
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 3:
               fila<<"ADD Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") ADD
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 4:
               fila<<"SUB Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") SUB
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 5:
               fila<<"SLT Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") SLT
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 6:
               fila<<"MULL Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") MULL
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 7:
               fila<<"XOR Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") XOR
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 8:
               fila<<"Sht_L Rd("<<tmp_rd<<")=Rs("<<tmp_rs<<") SHT_L
Rt("<<tmp_rt<<");"<<endl;
            break;

            default:
               fila<<"Nothing"<<endl;
         }
         fila.close();

   }
```

```
};

class I_type{
   public:
   int op[9][6]={{1,0,0,0,1,1}, {1,0,1,0,1,1}, {0,0,1,0,0,0}, {0,0,0,1,0,0}, {0,0,0,0,0,1},
{0,0,0,0,1,1}, //LW  //SW  //ADDI  //BEQ  //SUBI  //ANDI
              {0,0,0,1,1,1}, {0,0,1,1,1,1}, {0,1,1,1,1,1}};  //ORI  //MULLI  //XORI

   int vrt[5]={0,0,0,0,0}, vrs[5]={0,0,0,0,0}, imm[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   int rs, rt, im;  //addresses for my registers and the immediate value

   I_type(){
      int i=0;  //auxiliar variables to calculate 32 bit instruction
      rt=rand() % 32;  //randomizing a number between 0->32 to get an address for my
destination register

      cout<<"Register generator:"<<endl;
      cout<<"Rt="<<rt<<" ";

      int tmp_rt;  tmp_rt=rt;   // saving reg address an temporary variable to use it later at asm.txt
file

      while(rt!=0){   // conversion to binary
            vrt[i]=rt%2;      // binary value after division
      rt=rt/2;
      i++;
            }

      for(i=4; i>=0; i--){  //reading value in binary
         cout<<vrt[i]<<" ";
      }
      cout<<endl;

      i=0;
      rs=rand() % 32;
      int tmp_rs;  tmp_rs=rs;   // saving reg address into an temporary variable to use it later at
asm.txt file
      cout<<"Rs="<<rs<<" ";
      while(rs!=0){   // conversion to binary
            vrs[i]=rs%2;      // binary value after division
      rs=rs/2;
      i++;
            }

      for(i=4; i>=0; i--){  //reading value in binary
         cout<<vrs[i]<<" ";
```

```
}
cout<<endl;

i=0;
im=rand() % 65534;
int tmp_im; tmp_im=im;
cout<<"Imm="<<im<<" ";
while(im!=0){   // conversion to binary
      imm[i]=im%2;      // binary value after division
im=im/2;
i++;
      }

for(i=15; i>=0; i--){  //reading value in binary
   cout<<imm[i]<<" ";
}
cout<<endl;

int instruction[31];
int operation;
operation=rand() % 8;  //randomizing a number between 0->8 to pick an operation
cout<<"Operatia NR="<<operation<<" ";
cout<<endl;

cout<<"Operatia=";
for(i=0; i<6; i++){
   cout<<op[operation][i]<<" ";  //display function
}


for(i=0; i<6; i++){
   instruction[i]=op[operation][i];
}

int j=0;
for(i=0; i<=4; i++)
{
   instruction[i+6]=vrs[4-j];      //writing register addresses in instruction vector
   instruction[i+11]=vrt[4-j];
   j++;
}

j=0;
for(i=0; i<=15; i++){
   instruction[i+16]=imm[15-j];
   j++;
```

```
    }
    cout<<endl;

    cout<<"Instruction= ";
    for(i=0; i<32; i++){
       cout<<instruction[i]<<" ";
    }

    ofstream file;
    file.open("instructions.txt", ofstream::app);  // open file and continue writing in that
    for(i=1; i<=32; i++){
       file<<instruction[i-1];
       if(i%8==0){
          file<<endl;
       }
    }

    file.close();

    //funct->numarul functiei 0->AND 1->OR 2->NOR 3->ADD 4->SUB  5->SLT  6->MULL
7->XOR  8->SHT_L
    ofstream fila;
    fila.open("asm.txt", ofstream::app);  //asembly file creation

       switch(operation){  //LW  //SW  //ADDI  //BEQ  //SUBI  //ANDI   //ORI  //MULLI
//XORI
          case 0:
             fila<<"LW Rt("<<tmp_rt<<")<-Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
          break;

          case 1:
             fila<<"SW [Rt]("<<tmp_rt<<")<-Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
          break;

          case 2:
             fila<<"ADDI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
          break;

          case 3:
             fila<<"BEQ [imm]("<<tmp_im<<")<-Rs("<<tmp_rs<<") -
Rt("<<tmp_rt<<");"<<endl;
          break;
```

```
            case 4:
               fila<<"SUBI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") SUBI
imm("<<tmp_im<<");"<<endl;
            break;

            case 5:
               fila<<"ANDI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") ANDI
imm("<<tmp_im<<");"<<endl;
            break;

            case 6:
               fila<<"ORI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") ORI
imm("<<tmp_im<<");"<<endl;
            break;

            case 7:
               fila<<"MULLI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") MULLI
imm("<<tmp_im<<");"<<endl;
            break;

            case 8:
               fila<<"XORI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") XORI
imm("<<tmp_im<<");"<<endl;
            break;

            default:
               fila<<"Nothing"<<endl;
         }
         fila.close();
   }

   I_type(int operation, int rt, int rs, int im){        //constructor de forma: tipul operatiei, registrii
cu val<32, val imm<65535
      int i=0;  //auxiliar variables to calculate 32 bit instruction

      if((rt>32)||(rs>32)||(operation>8)||(im>65500))
      {
         exit(1);
      }
      cout<<"Register generator:"<<endl;
      cout<<"Rt="<<rt<<" ";

      int tmp_rt;  tmp_rt=rt;   // saving reg address an temporary variable to use it later at asm.txt
file

      while(rt!=0){   // conversion to binary
```

```
            vrt[i]=rt%2;      // binary value after division
        rt=rt/2;
        i++;
           }

    for(i=4; i>=0; i--){  //reading value in binary
        cout<<vrt[i]<<" ";
    }
    cout<<endl;

    i=0;
    int tmp_rs;  tmp_rs=rs;   // saving reg address into an temporary variable to use it later at
asm.txt file
    cout<<"Rs="<<rs<<" ";
    while(rs!=0){   // conversion to binary
            vrs[i]=rs%2;      // binary value after division
        rs=rs/2;
        i++;
           }

    for(i=4; i>=0; i--){  //reading value in binary
        cout<<vrs[i]<<" ";
    }
    cout<<endl;

    i=0;
    int tmp_im; tmp_im=im;
    cout<<"Imm="<<im<<" ";
    while(im!=0){   // conversion to binary
            imm[i]=im%2;      // binary value after division
        im=im/2;
        i++;
           }

    for(i=15; i>=0; i--){  //reading value in binary
        cout<<imm[i]<<" ";
    }
    cout<<endl;

    int instruction[31];

    cout<<"Operatia NR="<<operation<<" ";
    cout<<endl;

    cout<<"Operatia=";
    for(i=0; i<6; i++){
```

```
        cout<<op[operation][i]<<" ";  //display function
    }


    for(i=0; i<6; i++){
        instruction[i]=op[operation][i];
    }

    int j=0;
    for(i=0; i<=4; i++)
    {
        instruction[i+6]=vrs[4-j];      //writing register addresses in instruction vector
        instruction[i+11]=vrt[4-j];
        j++;
    }

    j=0;
    for(i=0; i<=15; i++){
        instruction[i+16]=imm[15-j];
        j++;
    }
    cout<<endl;

    cout<<"Instruction= ";
    for(i=0; i<32; i++){
        cout<<instruction[i]<<" ";
    }

    ofstream file;
    file.open("instructions.txt", ofstream::app);  // open file and continue writing in that
    for(i=1; i<=32; i++){
        file<<instruction[i-1];
        if(i%8==0){
            file<<endl;
        }
    }

    file.close();

    //funct->numarul operatia 0->LW 1->SW 2->ADDI 3->BEQ 4->SUBI  5->ANDI  6->ORI
7->MULLI  8->XORI
    ofstream fila;
    fila.open("asm.txt", ofstream::app);  //asembly file creation

        switch(operation){  //LW  //SW  //ADDI  //BEQ  //SUBI  //ANDI   //ORI  //MULLI
//XORI
```

```
            case 0:
               fila<<"LW Rt("<<tmp_rt<<")<-Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
            break;

            case 1:
               fila<<"SW [Rt]("<<tmp_rt<<")<-Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
            break;

            case 2:
               fila<<"ADDI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") +
imm("<<tmp_im<<");"<<endl;
            break;

            case 3:
               fila<<"BEQ [imm]("<<tmp_im<<")<-Rs("<<tmp_rs<<") -
Rt("<<tmp_rt<<");"<<endl;
            break;

            case 4:
               fila<<"SUBI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") SUBI
imm("<<tmp_im<<");"<<endl;
            break;

            case 5:
               fila<<"ANDI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") ANDI
imm("<<tmp_im<<");"<<endl;
            break;

            case 6:
               fila<<"ORI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") ORI
imm("<<tmp_im<<");"<<endl;
            break;

            case 7:
               fila<<"MULLI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") MULLI
imm("<<tmp_im<<");"<<endl;
            break;

            case 8:
               fila<<"XORI Rt("<<tmp_rt<<")=Rs("<<tmp_rs<<") XORI
imm("<<tmp_im<<");"<<endl;
            break;

            default:
```

```
            fila<<"Nothing"<<endl;
        }
        fila.close();
    }


};

class J_type{
    public:
    int op[6]={0,0,0,0,1,0};
    int imm[26]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int instr[31];
    J_type();
    J_type(int im){
        int i=0;
        int tmp_im; tmp_im=im;
            while(im!=0){   // conversion to binary
                imm[i]=im%2;      // binary value after division
            im=im/2;
            i++;
                }

            for(i=0;i<26;i++)
            {
               cout<<imm[i]<<" ";
            }
        cout<<endl;
        for(i=0; i<6; i++)
        {
            instr[i]=op[i];
        }

        int j=25;

        for(i=6; i<32; i++){
            instr[i]=imm[j];
            j--;

        }

        cout<<"Instructiunea: ";
        for(i=0; i<32; i++){
            cout<<instr[i];
        }
```

```
        ofstream file;
        file.open("instructions.txt", ofstream::app);  // open file and continue writing in that
        for(i=1; i<=32; i++){
            file<<instr[i-1];
            if(i%8==0){
                file<<endl;
            }
        }

         ofstream fila;
        fila.open("asm.txt", ofstream::app);  //asembly file creation
            fila<<"J  address("<<tmp_im<<");"<<endl;
        fila.close();
    };

};

Int main(){

//Apelare constructori

return 0;

}
```