UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICA ȘI CALCULATOARE

Calculator DLX

Îndrumător:

Gherman Filip-Marian

Student:

Ardelean Raluca-Nicoleta

Grupa: 30237

11.01.2023

CUPRINS

1. REZUMAT

Scopul lucrării este de a proiecta și implementa algoritmul de QuickSort pentru numere de 32 de biți pe Xilinx Basyx 3 FPGA. Designul se bazează pe o arhitectura DLX. Arhitectura DLX într-un nucleu RISC constă din Fetch, Decode, Execute, Acces la memorie și ciclu de scriere înapoi. Această arhitectură DLX este implementată atât ciclu unic, cât și pipeline pentru a realiza optimizarea algoritmului de sortare realizat în mod iterativ. Nucleul este proiectat folosind VHDL și au fost implementate operațiile de ADD, SUB, SLL, SRL, AND, OR, SGT, SLT, ADDI, LW, ANDI, SW, BEQZ și J la nivelul DLX. Pentru acest design, s-a folosit și un simulator atât pentru testarea programului final cât și pentru convertirea codului din asamblare în cod mașină.

2. INTRODUCERE

DLX (pronunțat „Deluxe”) este o arhitectură de procesor RISC proiectată de John L. Hennessy și David A. Patterson, principalii designeri ai designurilor Stanford MIPS și Berkeley RISC, cele două exemple de referință de design RISC (numite după designul Berkeley).

În arhitectura Stanford MIPS, una dintre metodele folosite pentru a câștiga performanță a fost de a forța toate instrucțiunile să se completeze într-un singur ciclu de ceas. Acest lucru a forțat compilatorii să insereze „no-ops” în cazurile în care instrucțiunea ar dura cu siguranță mai mult de un ciclu de ceas. Astfel, activitățile de intrare și de ieșire (cum ar fi accesele la memorie) au forțat în mod specific acest comportament, ducând la balonarea artificială a programului. În general, programele MIPS au fost forțate să aibă o mulțime de instrucțiuni NOP risipitoare, un comportament care a fost o consecință neintenționată. Arhitectura DLX nu forțează execuția unui singur ciclu de ceas și, prin urmare, este imună la această problemă.

În proiectarea DLX a fost folosită o abordare mai modernă a gestionării instrucțiunilor lungi: redirecționarea datelor și reordonarea instrucțiunilor. În acest caz, instrucțiunile mai lungi sunt „blocate” în unitățile lor funcționale și apoi reinserate în fluxul de instrucțiuni când se pot finaliza. În exterior, acest comportament de proiectare face să pară ca și cum execuția ar fi avut loc liniar.

DLX este în esență un procesor Stanford MIPS simplificat, curățat (și modernizat). DLX are o arhitectură simplă de încărcare/stocare pe 32 de biți, spre deosebire de CPU cu arhitectură MIPS modernă. Deoarece DLX a fost conceput în primul rând pentru scopuri didactice, designul DLX este utilizat pe scară largă în cursurile de arhitectură a computerelor la nivel universitar.

Există două implementări hardware „softcore” cunoscute: ASPIDA și VAMP. Proiectul ASPIDA a rezultat într-un nucleu cu multe caracteristici frumoase: este open source, acceptă Wishbone, are un design asincron, acceptă mai multe ISA și este dovedit ASIC. VAMP este o variantă DLX care a fost verificată matematic ca parte a proiectului Verisoft. A fost specificat cu PVS, implementat în Verilog și rulează pe un FPGA Xilinx. Pe el a fost construită o stivă completă de la compilator la kernel la TCP/IP.

Instrucțiunile DLX pot fi împărțite în trei tipuri, de tip R, de tip I și de tip J. Instrucțiunile de tip R sunt instrucțiuni de registru pur, cu trei referințe de registru conținute în cuvântul de 32 de biți. Instrucțiunile de tip I specifică două registre și folosesc 16 biți pentru a păstra o valoare imediată. În cele din urmă, instrucțiunile de tip J sunt salturi, care conțin o adresă de 26 de biți.

DLX poate suporta mai mult de 64 de instrucțiuni, atâta timp cât acele instrucțiuni funcționează exclusiv pe registre. Aceasta caracteristică este utilă pentru lucruri precum suportul FPU.

Arhitectura DLX, ca și cea MIPS, își bazează performanța pe utilizarea unui canal de instrucțiuni. În designul DLX, acesta este unul destul de simplu, RISC „clasic” în concept, conținând cinci etape:

- **IF (Instruction Fetch unit) -** Unitatea de extragere a instrucțiunilor
- **ID (Instruction Decode unit) -** Unitatea de decodificare a instrucțiunilor
- **EX (Execution unit) -** Unitatea de execuție a instrucțiunilor
- **MEM (Memory access unit) -** Unitatea de Memorie
- **WB (WriteBack unit) -** Unitatea de scriere a rezultatelor

2.1. OBIECTIVUL PROIECTULUI

Obiectivul principal al acestui proiect a fost elaborarea și simularea unui calculator DLX care este capabil cu ajutorul unor instrucțiuni de baza să sorteze numerele întregi aflate in memorie utilizând algoritmul QuickSort. Instrucțiunea curenta pe care o executa calculatorul DLX va fi afișată pe ecranul SSD-ului integrat pe placa de dezvoltare. Trecerea dintre instrucțiuni are loc la apăsarea unui buton. Pentru a optimiza programul, s-a realizat și implementarea pipeline, nu doar ciclu unic.

Simularea funcționalității proiectului, atât a celui cu ciclu unic cât și a celui pipeline, s-a realizat cu ajutorul unui simulator realizat în limbajul Java, care dispune de o interfață prietenoasă ce oferă o vizualizare atât a memoriei, cât și a registrelor, fiind de asemenea foarte utilă pentru depanarea eventualelor probleme.

În capitolele următoarele sunt prezentate urmatoarele aspecte:

- Registre și tipuri de date
- Moduri de adresare
- Formatul instrucțiunilor
- Etapele de execuție
- Construirea căii de date
- Semnalele de control necesare căii de date
- Rezolvarea hazardurilor în DLX pipeline

3. FUNDAMENTARE TEORETICĂ

DLX respecta formatul Big Endian, spre deosebire de Little Endian, ceea ce înseamnă că o adresă DLX accesează mai întâi octetul in poziția cea mai semnificativa atunci când scoate un cuvânt din memorie. Memoria este adresabila pe octeți si poate efectua accese pe octet, jumătate de cuvânt sau cuvânt. Toate instrucțiunile sunt codificate în patru octeți.

Proiectarea unui procesor începe de la setul de instrucțiuni sau, mai exact, de la Arhitectura Setului de Instrucțiuni (ASI, eng. Instruction Set Architecture – ISA). Arhitectura setului de instrucțiuni reprezintă o colecție completă de specificații care permite scrierea de programe în cod mașină pentru procesor. ISA definește, de obicei, setul de instrucțiuni, cu formatul binar asociat, tipul operanzilor permiși în instrucțiune, registre, spațiul de memorie care poate fi accesat/adresat, modurile de adresare etc. ISA reprezintă un mod abstract de a descrie procesorul (nu se referă la implementarea fizică), același set de instrucțiuni putând avea mai multe implementări fizice (procesoare produse de companii diferite, cu implementări particulare, dar același ISA).

Există mai multe criterii pentru clasificarea ISA-urilor, dar probabil cel mai relevant este cel legat de complexitatea arhitecturii. Potrivit acestui criteriu, se pot identifica două clase de ISA-uri: RISC (Reduced Instruction Set Computer) și CISC (Complex Instruction Set Computer).

Arhitecturile RISC se bazează pe o complexitate de implementare (hardware) redusă, punând accent pe instrucțiunile de bază folosite frecvent. Astfel, se mută efortul pentru construirea de programe complexe spre compilator sau programator, dar simplitatea de implementare permite optimizări de tip pipeline (linie de asamblare).

Acest tip de arhitectură are următoarele caracteristici:

- accesul la memorie se face prin instrucțiuni dedicate de tip load/store care citesc/salvează date din/în memorie
- procesarea datelor se face folosind doar registre, din acest motiv existând un număr mai mare de registre
- număr redus de moduri de adresare (ce și cum se poate adresa folosind câmpurile din formatul de instrucțiune)
- instrucțiunile au un număr redus de formate, cu dimensiune fixă

3.1. REGISTRE ȘI TIPURI DE DATE

Mașina DLX este o mașină de registru de uz general (GPR) și, ca atare, are la bază o grămadă de registre. Cei mai relevanți sunt:

- Regiști întregi: R0, R1, . . . , R31 - GPR-urile
- Regiști în virgulă mobilă, F0, F1, . . . , F31 - FPR-urile

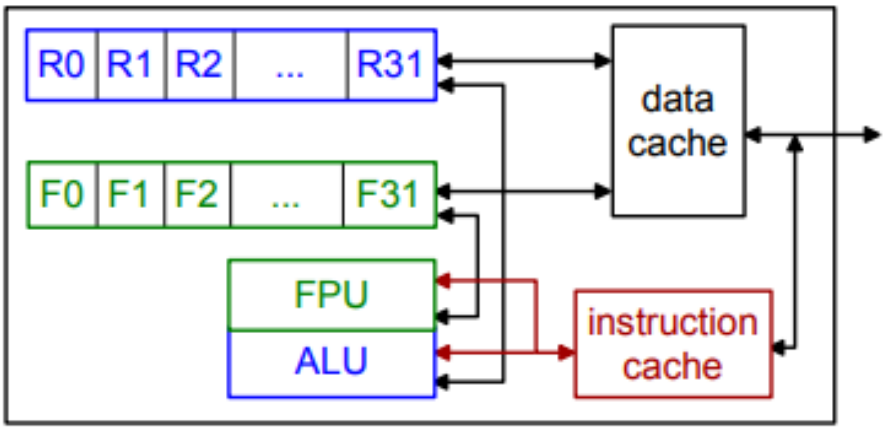


Fig.1

DLX gestionează numere în virgulă mobila, atât în precizie unică – 32 de biți sau un cuvânt cât și în precizie dublă – până la 64 de biți sau 2 cuvinte. Făcând analogie la alte limbaje de programare aceste tipuri de date sunt cunoscute ca “float” și “double”.

Pentru lucrul cu numere în precizie dublă, trebuie să se utilizeze 2 registre consecutive în virgulă mobilă, împachetate, începând cu unul care este numărat par și continuând cu unul care este numărat impar. Unele operații, în special înmulțirea și împărțirea, pot fi efectuate numai in registre FP, chiar și atunci când operanzii sunt numere întregi (mută datele din registrele întregi în regstrele FP, efectuează operațiile dorite, iar mai apoi mută datele înapoi.

Pe lângă cele 2 tipuri de date FP, DLX are 3 tipuri de date întregi:

- 8 biți – 1 octet
- 16 biți – 2 octeți sau jumătate de cuvânt
- 32 biți – 4 octeți sau un cuvânt

Un aspect confuz apare cand se dorește încărcarea unor date într-un registru care reprezintă mai puțin de un cuvânt, deorece trebuie să se aibă în vedere faptul că partea din registru care nu este ocupată de aceste date poate conține alte date care s-ar putea să producă rezultate greșite. De exemplu, într-un registru de 32 de biți au fost încărcați doar 8 biți, acest lucru implicând că ceilalți 24 de biți ce plutesc acolo pot avea valori nedorite. Soluția pentru această mică problemă este umplerea acestor biți cu 0.

DLX are de asemenea si câteva registre speciale. Cel mai important este R0, deoarece valoarea lui este întotdeauna 0. Scopul acestui registru este de "a sintetiza o varietate de operații utile dintr-o instrucțiune simplă" [1]. Așadar, Este permisă scrierea în registrul 0, dar valoarea acestuia nu se modifică. Acest registru se poate folosi ca valoare de referință pentru 0, atunci când se fac comparații sau se dorește inițializarea altor registre cu valori constante (imediate).

Un alt registru important este registrul contor de program, Program Counter – PC, pe 32 de biți, care indică adresa instrucțiunii curente.

3.2. MODURI DE ADRESARE

Exista 3 tipuri de obiecte diferite pe care un CPU poate avea nevoie sa le acceseze:

- constante (cunoscute in DLX ca valori imediate)
- registre
- locatii de memorie

Este evident că orice program real de calculator va avea atât constante, cât și va reutiliza date, iar în DLX trebuie să încărcăm elementele de date în registrele din memorie înainte de a le putea procesa, ceea ce înseamnă că DLX se adresează la un moment dat celor 3 tipuri de obiecte. Un mod de adresare este pur si simplu modul în care se descrie obiectul care este adresat.

Așadar, DLX dispune de 3 moduri de adresare principale și 2 secundare, acestea din urmă fiind subtipuri ale unui tip și sunt simulate cu ajutorul registrului R0:

1) Adresare imediata – prin intermediul căreia se accesează constante
2) Adresare cu registre – utilizat la specificarea unui registru, fiind cel mai folosit
3) Adresare cu bază (indexată) a memoriei – adresa urmărită e dată de numărul din registru + valoarea deplasamentului specificat
    - Adresare indirectă – deplasamentul este 0
    - Adresare absolută – registrul este R0

3.3. FORMATUL INSTRUCTIUNILOR

Există trei formate pentru instrucțiunile DLX: tip I, tip R și tip J.

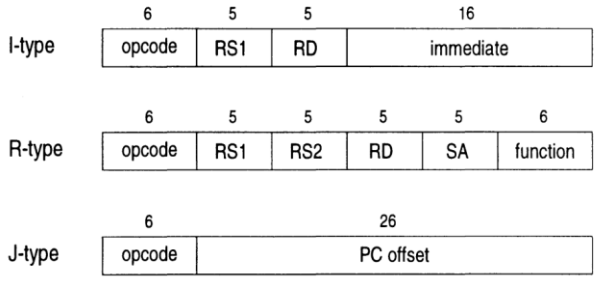


Fig. 2

Semnificația câmpurilor din formate este următoarea:

- opcode = codul de operație principal pe 6 biți, la tip R acesta este egal cu 0
- RS1 = indexul primului registru sursă, 5 biți
- RS2 = indexul celui de al doilea registru sursă / indexul registrului destinație, 5 biți
- RD = indexul registrului destinație, 5 biți
- SA = cantitatea de deplasare
- function = câmpul care indică operația efectivă pentru instrucțiuni de tip R, 6 biți
- immediate:
  - imediat/constantă pe 16 biți pentru operații aritmetice-logice
  - deplasamentul cu semn, în octeți, pentru operații cu memoria
  - deplasamentul cu semn, relativ la PC, în instrucțiuni, pentru instrucțiuni de salt condiționat
- PC offset = un număr pe 26 biți din care se va calcula adresa de salt necondiționat.

Instrucțiunile de tip R sunt instrucțiuni cu operanzi de tip registru (de unde se iau datele și unde se salvează rezultatul), care au următoarea sintaxă generală:

nume_instrucțiune RD, RS1, RS2

Aceste instrucțiuni sunt folosite, în special, pentru operații aritmetice / logice, cu operanzi de tip registru. Se face o anumită operație între două registre sursă (cu indecșii RS1 și RS2) din blocul de registre, și se salvează rezultatul în registrul destinație, cu indexul RD, din blocul de registre.

Instrucțiunile de tip I conțin o valoare imediată (semi-cuvânt pe 16 biți) în format, fiind de 3 subtipuri (toate au aceleași câmpuri în sintaxă). Prima variantă de instrucțiuni (subtip 1) de tip I este cea pentru operații aritmetice / logice cu valoare imediată (constantă numerică folosită în codul program):

subtip 1: nume_instrucțiune RS2, RS1, imm

sau

subtip 2: nume_instrucțiune RS1, imm

Se execută o operație între registrul sursă cu indexul rs și valoarea imediată imm, iar rezultatul este salvat în registrul reprezentat de RS2. Al doilea subtip are sintaxă similară, dar sunt instrucțiuni de salt condiționat: se compară valoarea din operandul de tip registru, RS1, și, în funcție de rezultatul comparației, se execută un salt relativ în program. Destinația saltului se calculează în funcție de valoarea imediată imm.
Al treilea subtip din tipul I este pentru instrucțiunile de lucru cu memoria de date. Sintaxa este un pic diferită, dar sunt aceleași câmpuri (offset este echivalent cu imm):

subtip 3: : nume_instrucțiune RS2, offset(RS1)

Pentru a accesa memoria, adresa se va forma din valoarea offset și RS1, iar RS2 va reprezenta valoarea care trebuie salvată în memorie (la scriere), respectiv unde se va salva valoarea citită din memorie.

Instrucțiunile de tip J sunt folosite pentru salturi necondiționate la adresa indicată de câmpul target_address, având următoarea sintaxă generală:

nume_instrucțiune PC_offeset

4. PROIECTARE ȘI IMPLEMENTARE

Pentru implementarea algoritmului de sortare QuickSort iterativ, primul pas a fost scrierea codului in limbaj de asamblare, iar mai apoi transformarea în cod mașină. Implementarea algoritmului a fost realizată atât pentru DLX ciclu unic cât și pentru DLX pipeline.

Pentru implementarea în simulator a fost nevoie de codul în limbaj de ansamblare, în simulator rulând doar varianta pipeline.

Instrucțiunile RTL implementate în cadrul calculatorului DLX atât ciclu unic cât și pipeline si cu ajutorul cărora s-a realizat algoritmului QuickSort iterativ, împărțite pe tipurile de instrucțiuni discutate anterior sunt:

- Tip R:
  - ADD - ADD RD, RS1, RS2 => R[RD] <- R[RS1] + R[RS2]
  - SUB - SUB RD, RS1, RS2 => R[RD] <- R[RS1] + R[RS2]
  - SLL – SLLI RD, RS1, RS2 => R[RD] <- R[RS1] sll SA, RS2 = R0
  - SRL - SRLI RD, RS1, RS2 => R[RD] <- R[RS1] srl SA
  - AND - AND RD, RS1, RS2 => R[RD] <- R[RS1] and R[RS2]
  - OR - OR RD, RS1, RS2 => R[RD] <- R[RS1] or R[RS2]
  - SGT - SGT RD, RS1, RS2 => if R[RS1] > R[RS2] then R[RD] <- 1 else R[RD] <- 0
  - SLT - SLT RD, RS1, RS2 => if R[RS1] < R[RS2] then R[RD] <- 1 else R[RD] <- 0

- TIP I:
  - ADDI - ADDI RD, RS1, IMM => R[RD] <- R[RS1]+IMM
  - LW - LW RD, IMM(RS1) => R[RD] <- M[R[RS1] + IMM]
  - ANDI - ANDI RD, RS1, IMM => R[RD] <- R[RS1] and IMM
  - SW - SW IMM(RD), RS1 => M[R[RD] + IMM] <- R[RS1]
  - BEQZ RS1, IMM - if (R[RS1] == 0) then PC <- PC + 4 + (S_Ext (IMM<<2))
  -
- TIP J:
  - J offset - PC <- ((PC +4) & 0xf0000000) | (offset << 2);

La aceste instrucțiuni se mai adaugă și operatia de nop care este reprezentată de adunarea registrului R0 cu 0 și stocarea rezultatului tot în R0 : ADDI R0,R0,0. Această operație de nop (no operation) este utilizată în cazul calculatorului DLX pipeline, pentru evitarea hazardurilor.

4.1. ETAPELE DE EXECUȚIE

Pentru a ușura procesul de proiectare, se va ține cont de etapele tipice de execuție pentru o instrucțiune. Aceste etape sunt construite din perspectiva fluxului de date, la execuția unei instrucțiuni, prin calea de date a procesorului: IF, ID, EX, MEM, WB amintite în capitolul Introducere.

- Extragerea instrucțiunii – IF
  Sunt necesare următoarele componente:
  - Memoria (unde se memorează instrucțiuni), 32 de linii de adresă (adresabilă pe octet), organizată pe cuvânt de 4 octeți, fiecare instrucțiune se va afla la adrese multiple de 4 octeți
  - Registrul contor de program PC, pe 32 de biți
  - Sumator, pe 32 de biți, pentru calculul adresei următoarei instrucțiuni (PC + 4), două intrări/o ieșire

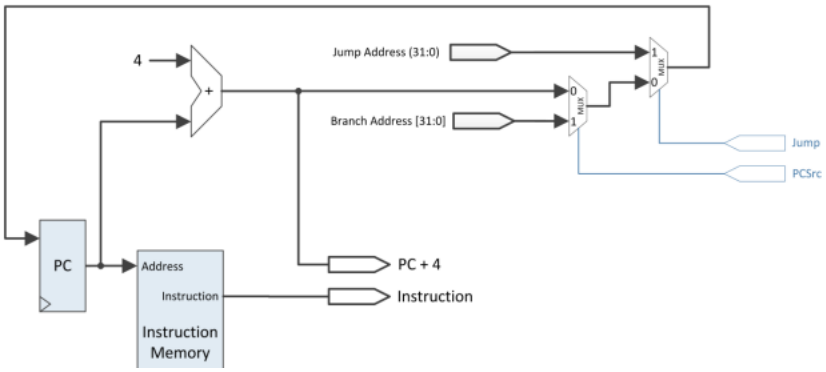


Fig 3. Calea de date pentru IF

- Interpretarea instrucțiunii – ID
  Fiind etapa de interpretare a codului binar al instrucțiunii, respectiv citirea operanzilor, în etapa de ID sunt necesare următoarele componente:
  - Blocul de registre RF
  - Unitate de extensie cu sau fără semn (pentru extinderea pe 32 biți a imm)
  - Unitatea de control

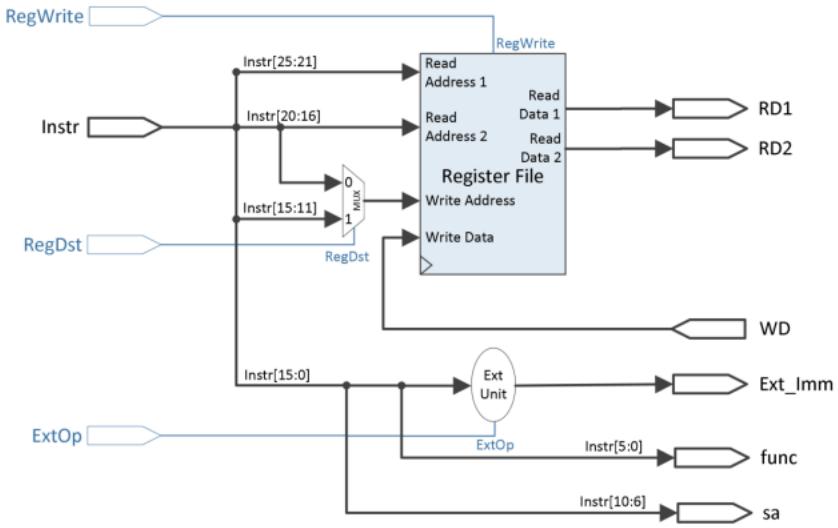


Fig 4. Calea de date pentru ID

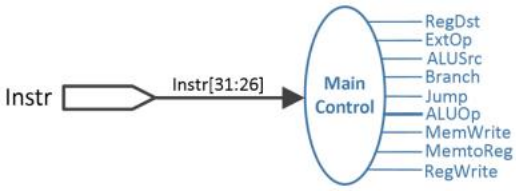


Fig 5. Unitatea de control

- Execuția instrucțiunii – EX
  În această etapă se analizează instrucțiunile individual (sau pe grupuri de instrucțiuni cu execuție similară). Pentru instrucțiunile de tip R este necesară o unitate aritmetică-logică ALU (se va folosi acronimul din engleză), pe 32 de biți, care să poată executa operații de adunare, scădere, SAU și ȘI logic pe biți, deplasare stânga/dreapta, testare mai mic/mai mare. Se va folosi ALU construită convenabil.
  Pentru instrucțiunile andi și addi se va folosi aceeași ALU pentru a executa and logic, respectiv adunare, între registrul selectat și imediatul extins fără semn / cu semn.

  În cazul instrucțiunilor de lucru cu memoria, este necesară calcularea adresei efective a locației de memorie. În RTL-ul instrucțiunilor lw/sw, acest calcul este reprezentat de operația de adunare care va fi executată tot de ALU

Instrucțiunea de salt condiționat beq necesită o analiză în detaliu. În RTL-ul instrucțiunii există două părți care necesită calcule: evaluarea condiției și calcularea adresei de salt. Prima este condiția de evaluat RF[RS1] == 0. În mod convențional, condițiile de salt în procesoare sunt evaluate de unitatea ALU. Unitatea ALU construită anterior are un semnal de stare denumit Zero prin care se semnalează prezența unui rezultat nul pe ieșirea ALU. Evaluarea condiției de egalitate se poate face printr-o operație de scădere între RS1 si RS2 setat pe 0. Rezultatul este nul în caz de egalitate și, implicit, se va activa semnalul de stare Zero. Pentru calculul adresei de salt se va folosi sumatorul menționat deja pentru IF (PC + 4), dar mai este necesară o adunare cu deplasamentul în octeți S_Ext(imm) <<2. Se adaugă un al doilea sumator pe 32 de biți și un circuit de deplasare la stânga cu 2 poziții.

Ultima instrucțiune de analizat este cea de salt necondiționat j. În mod normal, sunt necesari 32 de biți pentru a reprezenta adresa instrucțiunii țintă, dar formatul de instrucțiune permite doar 26 de biți pentru adresă (denumită și pseudo-adresă). Este necesară o soluție de compromis. Pentru a obține adresa pe 32 de biți se concatenează cei mai semnificativi 4 biți din adresa următoarei instrucțiuni după j, (PC+4)[31..28], cu cei 26 de biți și cu 2 biți de 0.

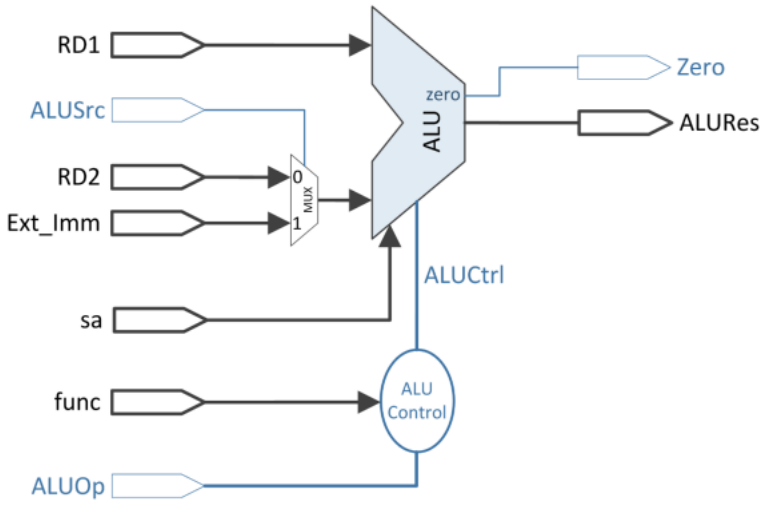


Fig 6. Calea de date pentru EX

- Operații cu memoria – MEM
  În cazul instrucțiunilor de lucru cu memoria, este necesar (evident) blocul de memorie. Este important de subliniat că, în această etapă de analiză, nu rezultă necesitatea folosirii a două blocuri distincte de memorie, pentru instrucțiuni, respectiv date. Memoria poate fi comună, reprezentată de un singur bloc.

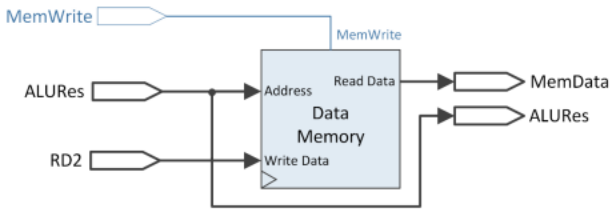


Fig 7. Calea de date pentru MEM

- Scrierea rezultatului – WB
  Rezultatul se scrie înapoi în blocul de registre RF, prin portul de scriere

4.2. CONSTRUIREA CĂII DE DATE

Odată ce etapele de execuție au fost finalizate, mai rămâne doar să se lege toate și astfel va rezulta calea de date pentru calculatorul DLX ciclu unic ce realizează algoritmul QuickSort iterativ.

La asamblarea căii de date e posibil să se identifice noi componente necesare. În mod particular, este vorba de multiplexoare, atunci când pe același semnal de intrare trebuie să ajungă valori din surse diferite în funcție de instrucțiune.

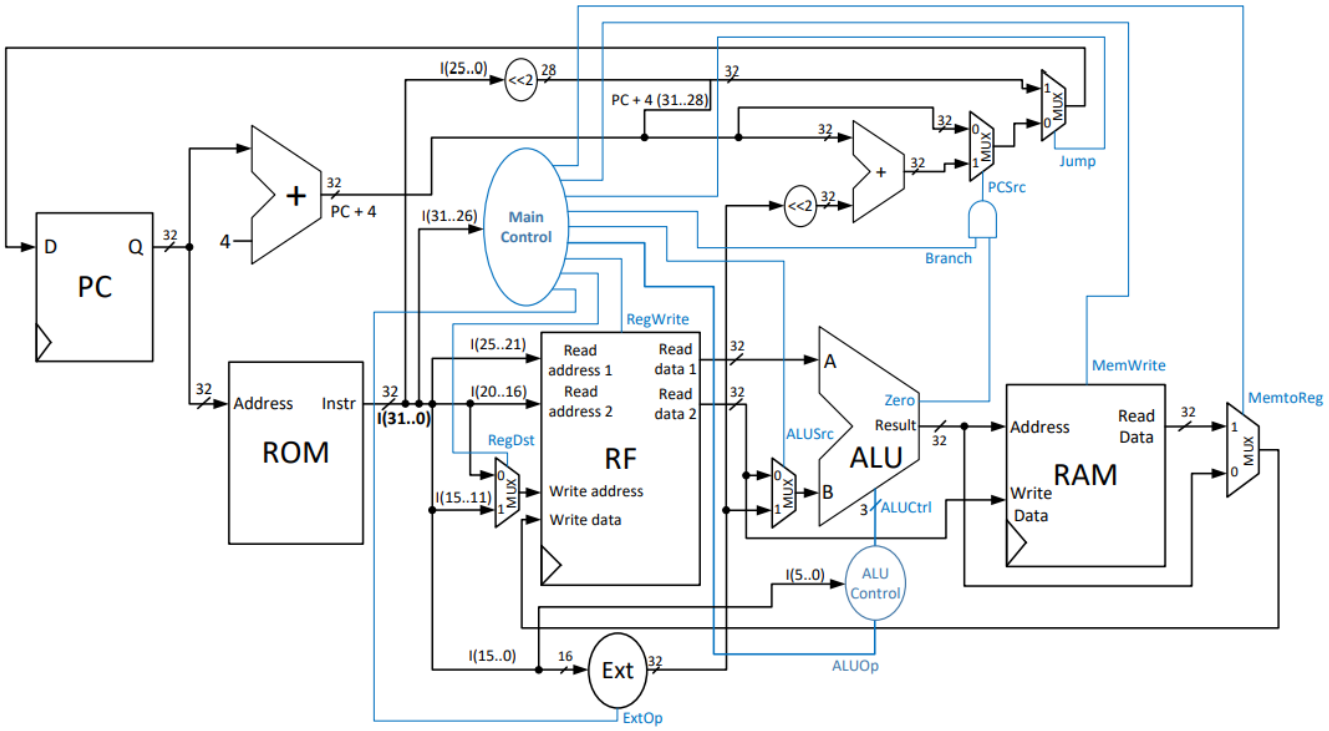


Fig 8. Calea de date pentru DLX

Pentru a reduce durata ciclului de ceas, soluția aleasă a fost secționarea căii critice cu elemente de stocare sincrone (de tip registru). În versiunea pipeline se pot executa până la 5 instrucțiuni consecutive din program, fiecare aflându-se într-una din cele 5 faze de execuție, în funcție de ordinea lor în program.

Așadar, schema pentru DLX pipeline este foarte similara:

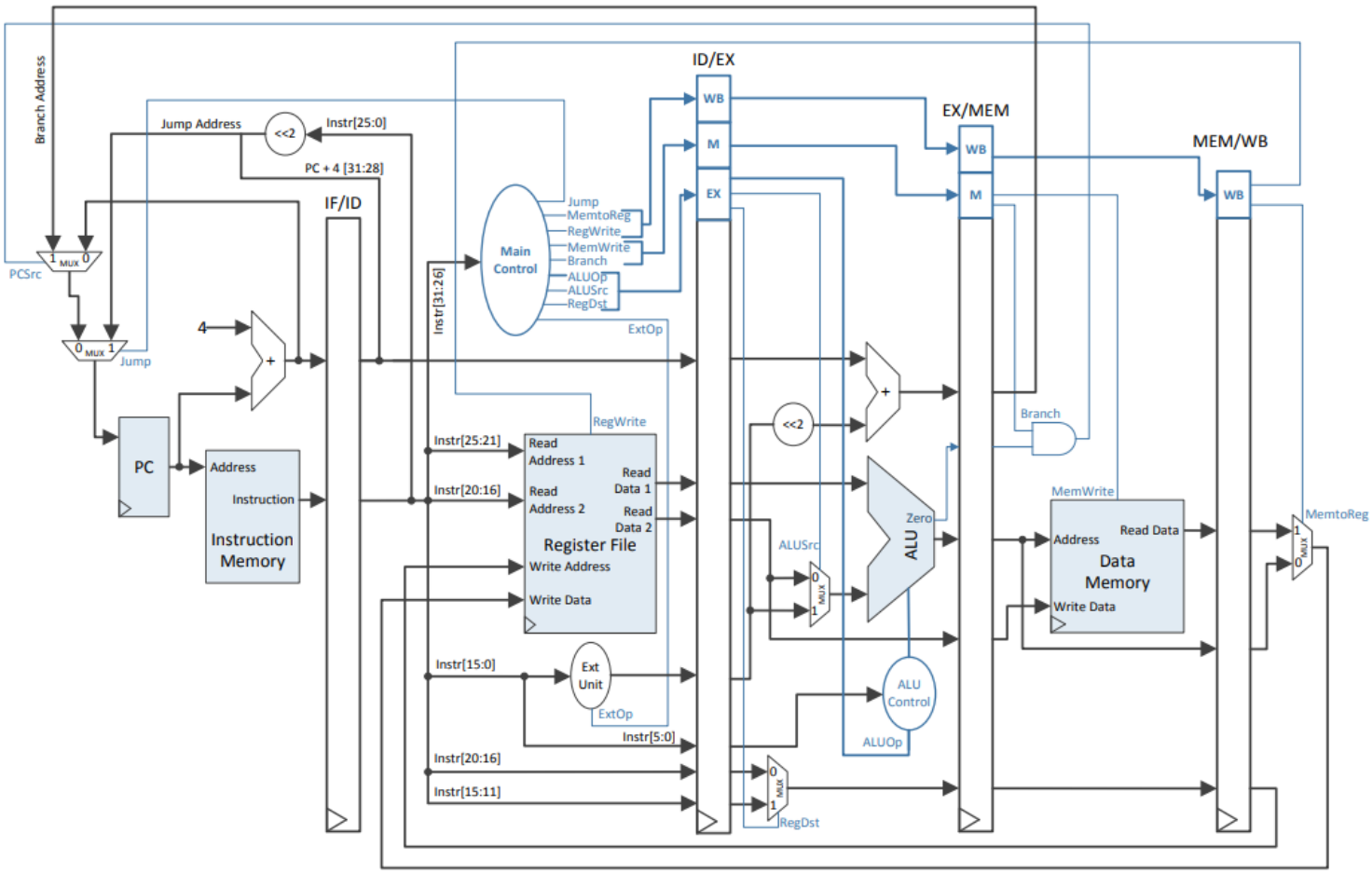


Fig 9. Calea de date pentru DLX pipeline

4.3. SEMNALELE DE CONTROL

Semnalele de control pe un bit, menționate pe schemele anterioare, și efectul lor în calea de date sunt prezentate în continuare:

- **RegDst** – semnal de selecție pentru mux-ul de la adresa de scriere Write address a RF. Dacă are valoarea 0, atunci trece mai departe valoarea câmpului rt. Dacă are valoarea 1, atunci trece mai departe valoarea câmpului rd
- **RegWrite** – semnalul de validare a scrierii în RF. Dacă are valoarea 0, nu se execută scriere, altfel, dacă e 1, se scrie în blocul de registre valoarea de pe Write data la adresa de pe Write address
- **ExtOp** – controlează funcționarea unității Ext. Dacă are valoarea 0, imediatul se va extinde fără semn, cu 0. Dacă are valoarea 1, atunci se face extensie cu semn
- **ALUSrc** – semnal de selecție pentru mux-ul de la intrarea B a ALU. Dacă are valoarea 0, atunci pe intrarea B a ALU ajunge valoarea de la ieșirea Read data 2 a RF. Dacă are valoarea 1, atunci va ajunge valoarea de la ieșirea circuitului Ext
- **PCSrc** – semnal de selecție pentru mux-ul de salt condiționat. Dacă are valoarea 0, atunci trece mai departe valoarea PC + 4. Altfel, dacă are valoarea 1, va trece adresa de salt (pentru beq)
- **Jump** – semnal de selecție pentru mux-ul de salt necondiționat. Dacă are valoarea 0, atunci spre intrarea PC va trece valoarea prezentă pe ieșirea mux-ului precedent (PCSrc). Dacă are valoarea 1, atunci spre intrarea PC va trece valoarea adresei absolute de salt (pentru j)
- **MemWrite** – semnalul de validare a scrierii în memoria RAM de date. Dacă are valoarea 0 nu se execută scriere, altfel, dacă e 1, se scrie în memorie valoarea de pe Write Data la adresa de pe Address
- **MemtoReg** – semnal de selecție pentru mux-ul prin care se scrie înapoi rezultatul. Dacă are valoarea 0, atunci spre intrarea Write data a RF va trece rezultatul de la ieșirea ALU. Dacă are valoarea 1, atunci va trece valoarea citită din memoria de date.

4.4. REZOLVAREA HAZARDURILOR ÎN DLX-PIPELINE

Hazardurile sunt situații în care o instrucțiune următoare nu poate fi executată corect în următoarea perioadă de ceas. Există trei tipuri de hazard, în funcție de cauza lor:

Hazard structural (dependență de resurse) = Două instrucțiuni încearcă să folosească simultan o resursă, în scopuri diferite -> constrângeri de resurse

Hazardul structural apare atunci când, pe același ciclu de ceas, instrucțiuni aflate etaje diferite din pipeline încearcă să folosească aceeași componentă din procesor. De interes pentru implementarea calculatorului DLX este hazardul structural care apare la utilizarea blocului de registre, pentru instrucțiuni situate la distanță de 3. Soluția cea mai simplă este introducerea unei instrucțiuni nop.

Hazard de date (dependență de date) = Încercarea de a folosi valori înainte să fie actualizate de instrucțiunile anterioare din program. Pentru o instrucțiune care a ajuns în etajul ID, operanzii sursă sunt în curs de calculare în etajele următoare de pipeline (de către instrucțiuni anterioare).

Hazardul de date (tip Read After Write, sau Load data hazard) apare la acele instrucțiuni care folosesc ca operanzi sursă valori care sunt în curs de calculare de către instrucțiuni anterioare, nefiind actualizate încă.

Hazard de control/comandă (dependență de condiții, control) - Decizia pentru un salt condiționat nu se știe înainte de evaluarea condiției saltului și calcularea adresei de ramificare pentru PC. Apare la trecerea prin pipeline a instrucțiunilor care influențează PC (ramificări, salturi).

Hazardurile de control apar la instrucțiuni de salt, unde instrucțiunile care urmează după instrucțiunea de salt intră implicit în execuție. În cazul instrucțiunilor de salt condiționat, în forma de bază a procesorului DLX pipeline, vor intra implicit în execuție următoarele 3 instrucțiuni.

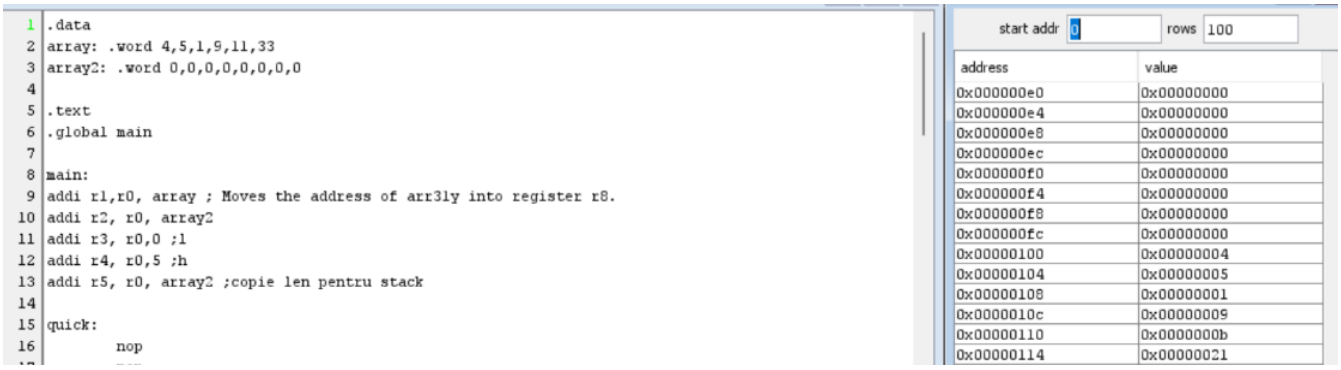
În cazul instrucțiunilor de salt necondiționat j, aceste instrucțiuni se finalizează (execută saltul) când sunt în etajul ID. Rezultă că intră în execuție doar instrucțiunea imediat următoare. Soluția simplă este să se insereze un nop după instrucțiunea j.

Pentru rezolvarea hazardurilor în simulator s-a folosit o unitate de forwarding preimplementată în programul Java, astfel încât după fiecare instrucțiune de salt necondiționat trebuie adăugate 2 instrucțiuni de nop.

5. REZULTATE EXPERIMENTALE

Pentru verificarea corectitudinii implementării, am testat în primul rând în simulator, adăugând in memorie atât numere pozitive cât și numere negative, iar în urma rulării s-a putut observa că datele din memorie, de la aceleași adrese sunt sortate.

Inainte de rulare:



```
.data
array: .word 4,5,1,9,11,33
array2: .word 0,0,0,0,0,0,0,0

.text
.global main

main:
addi r1,r0, array ; Moves the address of arr3ly into register r8.
addi r2, r0, array2
addi r3, r0,0 ;l
addi r4, r0,5 ;h
addi r5, r0, array2 ;copie len pentru stack

quick:
        nop
```

start addr 0 rows 100

address	value
0x000000e0	0x00000000
0x000000e4	0x00000000
0x000000e8	0x00000000
0x000000ec	0x00000000
0x000000f0	0x00000000
0x000000f4	0x00000000
0x000000f8	0x00000000
0x000000fc	0x00000000
0x00000100	0x00000004
0x00000104	0x00000005
0x00000108	0x00000001
0x0000010c	0x00000009
0x00000110	0x0000000b
0x00000114	0x00000021

Fig 10. Memoria înainte de sortare

După rulare:

```
.data
array: .word 4,5,1,9,11,33
array2: .word 0,0,0,0,0,0,0,0

.text
.global main

main:
addi r1,r0, array ; Moves the address of arr3ly into register r8.
addi r2, r0, array2
addi r3, r0,0 ;l
addi r4, r0,5 ;h
addi r5, r0, array2 ;copie len pentru stack

quick:
        nop
```

start addr 0 rows 100

address	value
0x000000e0	0x00000000
0x000000e4	0x00000000
0x000000e8	0x00000000
0x000000ec	0x00000000
0x000000f0	0x00000000
0x000000f4	0x00000000
0x000000f8	0x00000000
0x000000fc	0x00000000
0x00000100	0x00000001
0x00000104	0x00000004
0x00000108	0x00000005
0x0000010c	0x00000009
0x00000110	0x0000000b
0x00000114	0x00000021

Fig 11. Memoria înainte de sortare

De asemenea, am testat programul și pe o plăcuță FPGA Basys3. S-au adăugat în memorie 3 numere pentru a fi ușor de testat întrucât trebuie să se parcurgă fiecare instrucțiune până a se ajunge la finalul programului. Când s-a ajuns la final, s-au realizat operatii de load pentru cele 3 numere, iar pe switch-urile de pe placută s-a adăugat o comandă care să afișeze pentru fiecare instrucțiune ce aduce ea din memorie, dacă aduce ceva. Astfel, activând aceasta comandă când s-a ajuns la finalul programului se vor putea vedea pe SSD – ul plăcuței numerele sortate.

6. CONCLUZII

Calculatorul DLX implementat atât ciclu unic, cât și pipeline, care realizează algoritmul de sortare QuickSort iterativ a fost realizat cu succes. Succesul a fost dovedit prin testarea procesorului pe FPGA Basys 3 și în Simulator. Testarea pe plăcută a fost realizată pentru ambele arhitecturi – ciclu unic și pipeline, iar rezultatele au fost cele corecte. În ceea ce privește testarea în simulator, s-a realizat doar testarea arhitecturii pipeline, întrucât cea ciclu unic nu este suportată de simulatorul ales.

Ca viitoare dezvoltări, procesorul s-ar putea îmbunătății prin implementarea algoritmului QuickSort recursiv, întrucât la nivel de optimizare rezultatele cu siguranță ar fi furnizate mai repede. Tot ca viitoare dezvoltare ar fi utilă implementarea mai multor instrucțiuni astfel încât să fie posibilă realizarea mai multor programe pe care procesorul DLX să le ruleze.

7. BIBLIOGRAFIE

https://users.utcluj.ro/~baruch/resources/DLX/DLX-Instruction-Set.htm#InstrA

https://link.springer.com/content/pdf/bbm:978-3-662-04267-0/1.pdf

https://users.utcluj.ro/~baruch/resources/DLX/DLX-For-Neophytes.htm

https://sourceforge.net/projects/opendlx/

https://en.wikipedia.org/wiki/DLX