Proiectarea unei unități centrale care prelucrează instrucțiunile în regim pipeline (Arhitectura Intel 8086)

Student: Bîzgă Ana-Maria

Proiect Structura Sistemelor de Calcul

Universitatea Tehnică din Cluj-Napoca

Cuprins

Introducere

1.1 Context

Scopul acestui proiect este de a dezvolta o unitate centrală de procesare inspirată de Arhitectura Intel 8086, care va fi implementată în VHDL pentru testare și evaluare. Proiectul va implica simularea și verificarea funcționalității microprocesorului atât prin teste practice pe FPGA (Nexys A7 100T), cât și în Xilinx Vivado. Acest proces va oferi o oportunitate valoroasă de a înțelege arhitectura microprocesorului și modul în care funcționează.

1.2 Obiective

Proiectul urmărește proiectarea unei arhitecturi capabilă să execute operații aritmetice, logice, de manipulare a datelor, de salt condiționat și necondiționat, asigurând astfel flexibilitatea necesară unui microprocesor. Se dorește implementarea unui sistem pipeline care permite preluarea și execuția instrucțiunilor în mod eficient, ținând cont de posibilele hazarduri care pot apărea.

De asemenea, se vor crea test bench-uri pentru simularea funcționalității microprocesorului. Testele vor verifica corectitudinea execuției instrucțiunilor și gestionarea hazardelor. Se va utiliza o placă FPGA pentru testarea practică, fiind nevoie de componente suplimentare precum debouncer, afișaj cu 7 segmente.

Studiu bibliografic

2.1 Descrierea generală

Intel a lansat în 1978 primul microprocesor pe 16 biți, microprocesorul 8086. Acesta a semnificat o creștere semnificativă a performanței și a permis îndeplinirea unor cerințe specifice aplicațiilor pentru microcomputere. 8086 avea capacitatea de a gestiona seturi de instrucțiuni mai complexe decât versiunile anterioare, datorită arhitecturii sale avansate și a modurilor de adresare.

2.2 Arhitectura Intel 8086

2.2.1 Arhitectura Internă

Microprocesorul Intel 8086 prezintă o arhitectură alcătuită din două unități principale de procesare : Bus Interface Unit (BIU) și Execution Unit (EU). Cele două operează simultan, de unde și caracterul pipeline al acestui procesor.

Bus Interface Unit este responsabilă de extragerea instrucțiunilor din memorie, citirea și scrierea operanzilor. Stochează instrucțiunile extrase din memorie într-o coadă de instrucțiuni, care poate reține până la 6 bytes. Acest fapt permite stocarea eficientă a instrucțiunilor, permițând EU să execute simultan.

Execution Unit are ca scop execuția și decodificarea instrucțiunilor primite din coadă. ALU efectuează calculele aritmetice și logice, stocând rezultatele în registri sau în memorie.

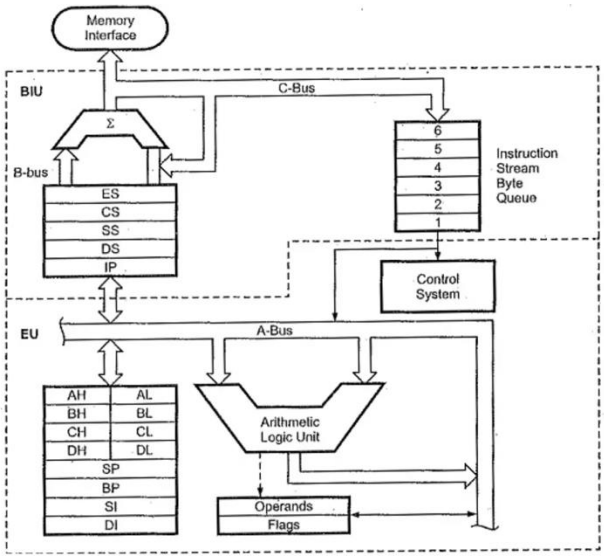


Figura 2.2 – Arhitectura microprocesorului 8086

2.2.2 Registrele procesorului

Microprocesorul 8086 prezintă în arhitectura sa mai multe tipuri de registre de 16 biți de care se folosește pentru a reține date : registre generale, de segment, de pointer, de index și de status.

Registrele generale :

- **AX (Accumulator)** – folosit pentru operații aritmetice și logice, rezultatele fiind adesea stocate aici
- **BX (Base Register)** – utilizat pentru operații de adresare indirectă
- **CX (Count Register)** – utilizat pentru a număra iterații în bucle
- **DX (Data Register)** – folosit în operații de ieșire/intrare sau pentru extinderea operanzilor

Registre de segment :

- **CS (Code Segment)** – indică adresa segmentului de cod curent
- **DS (Data Segment)** – indică adresa segmentului de date utilizat de program
- **SS (Stack Segment)** – indică adresa segmentului de stivă
- **ES (Extra Segment)**

Registre de Pointer și Index:

- **SP (Stack Pointer)** – pointează la vârful stivei
- **BP (Base Pointer)** – utilizat pentru a accesa variabilele locale și parametrii funcțiilor de pe stivă
- **SI (Source Index) și DI (Destination Index)** – folosite pentru a indica sursa și destinația în operațiile de copiere.

Registre de status :

- **SR (Status Register)** – 9 biți implementați ca flag-uri de status și control (Carry Flag, Parity Flag, Auxiliar Carry Flag, Zero Flag, Sign Flag, Trap Flag, Interrupt Enable Flag, Direction Flag, Overflow Flag)

2.2.3 Memoria

Arhitectura memoriei procesorului 8086 este organizată pe baza unei scheme segmentate care accesează 1MB de memorie, îmbunătățind gestionarea acesteia și eficiența accesării datelor. Este împărțită în 3 segmente : **Segmentul de cod** (conține instrucțiunile programului, accesat prin CS), **Segmentul de date** (stochează datele programului, accesat prin DS), **Segmentul de stivă** (folosit la apeluri de funcții sau variabile temporare, accesat prin SS).

2.3 Caracterul Pipeline

Arhitectura microprocesorului Intel 8086 este proiectată pentru a funcționa cu un model de execuție de tip pipeline, care permite suprapunerea execuției instrucțiunilor pentru a îmbunătății performanța generală a procesorului. Această abordare maximizează utilizarea resurselor.

Acest aspect de pipeline aduce totodată și hazarduri de date și de control. **Hazardurile de date** sunt întâlnite la folosirea instrucțiunilor de salt condiționat și necondiționat (precum jump) sau la instrucțiuni de ramificare (precum CALL). În cazul acestor instrucțiuni coada trebuie resetată la starea inițială (adică goală) și instrucțiunile trebuie extrase de la adresa curentă. **Hazardurile de date** sunt generate de instrucțiunile care depind de rezultatele unei instrucțiuni anterioare.

Analiză

3.1 Propunerea proiectului

Scopul acestui proiect este de a proiecta și implementa un microprocesor inspirat de Intel 8086 în Vhdl, cu o arhitectură de tip pipeline. Acest proiect va suporta un set specific de instrucțiuni, iar pentru verificarea corectitudinii acestora se va implementa un program de testare.

3.2 Analiza proiectului

3.2.1 Setul de instrucţiuni

Setul de instrucțiuni al microprocesorului Intel 8086 este unul complex și versatil, proiectat pentru a acoperi o gamă largă de operații, de la mișcarea datelor, la execuția de operații aritmetice și logice, până la controlul fluxului programului. Acest set de instrucțiuni permite accesul la diferite moduri de adresare, operanzi de dimensiuni variabile și interacțiunea cu memoria și registrele procesorului.

Aceste instrucțiuni sunt transformate în cod binar care urmează să fie procesat de microprocesor. Codurile binare au dimensiune variabilă, au minim 2 bytes, dar se poate ajunge la 6 bytes pentru o instrucțiuni mai complexă.

Codul unei instrucțiuni este împărțit în patru componente principale (după cum se poate observa și în figura 3.2.1.a).

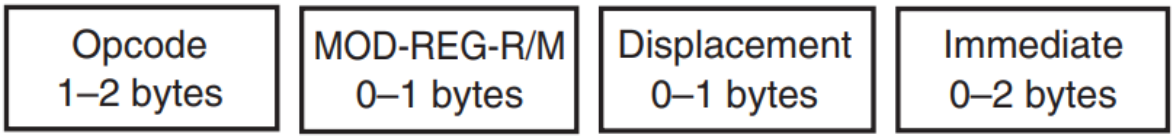


Figura 3.2.1.a – instrucțiuni pe 16 biti [2]

Opcode-ul este **prima componentă** și reprezintă codul de operație care va determina acțiunea ce trebuie executată de procesor (ex. adunare, scădere). Acesta poate ocupa unul sau doi bytes în funcție de complexitatea instrucțiunii, fiind acompaniat de încă doi biti denumiți **D** și **W**, care specifică direcția operației, respectiv dimensiunea operanzilor (W va fi 0 pentru byte, 1 pentru word).

Cea dea doua componentă este cea care definește modul de adresare și specifică dacă operandul este un registru sau o locație din memorie. Câmpul **Mod** specifică modul de adresare folosit : 00 – fără deplasament, 01 – deplasament de 8 biti, 10 – deplasament de 16 biti, 11 – operandul este registru. Câmpul **Reg** și **R/M** pot reprezenta amândouă câte un registru (figura 3.2.1.b) sau un registru și o locație din memorie.

A treia componentă este **Displacement** (Deplasament), care poate avea o lungime de 0, 1 sau 2 bytes, în funcție de modul de adresare utilizat. Deplasamentul este folosit în cazurile în care operandul se află într-o locație din memorie și specifică offset-ul față de o adresă de bază. Dacă operandul este un registru, acest câmp nu este folosit.

A patra componentă este **Immediate** (Valoare Imediată), care este o valoare constantă inclusă direct în instrucțiune. Acest câmp poate avea o lungime de 0 până la 2 bytes, în funcție de dimensiunea datelor operandului specificat (8 biți sau 16 biți).

Code	W = 0 *(Byte)*	W = 1 *(Word)*
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

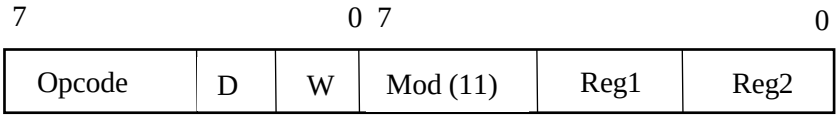
Figura 3.2.1.b – Codificarea pentru utilizare registrelor [2]

În ceea ce urmează vom explora setul de instrucțiuni al microprocesorului Intel 8086, concentrându-ne pe instrucțiunile esențiale pentru implementarea proiectului nostru. Pentru a simplifica analiza și implementarea, vom utiliza o formă simplificată a instrucțiunilor și codificării acestora. Această abordare ne va permite sa realizăm o implementare eficientă în VHDL. Vom selecta un subset de instrucțiuni reprezentative, care acoperă operațiile aritmetice, logice, de transfer de date și de salt condiționat sau necondiționat.

A. Instrucțiuni de transfer de date

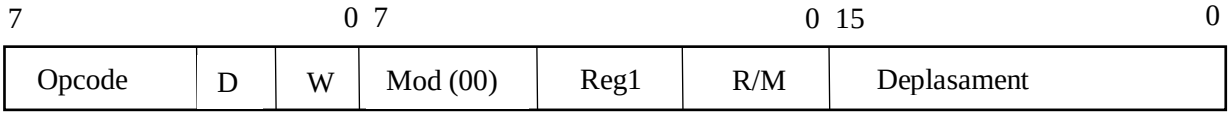
Instrucțiunea **MOV** este una dintre instrucțiunile fundamentale, fiind utilizată pentru a muta date între registre și memorie, între două registre sau pentru a încărca o valoare imediata într-un registru.

1. MOV între două registre



- Sintaxă: MOV dest, src
- OpCode: 100010
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  MOV AX, BX
  100010 0 1 11 000 011 (89C3)

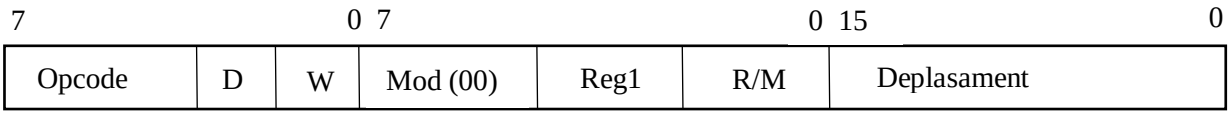
2. MOV din registru în memorie



- Sintaxă: MOV [mem], reg
- OpCode: 100010
- D: 0 (datele sunt mutate din registru în memorie)
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 00 (vom folosi adresarea directă)

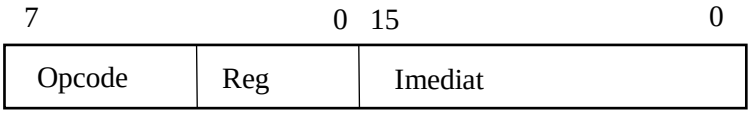
- R/M: 110 (adresare directă)
- Exemplu:
  MOV [100H], AX
  100010 0 1 00 000 110 00000001 00000000 (89060100)

3. MOV din memorie în registru



- Sintaxă: MOV reg, [mem]
- OpCode: 100010
- D: 1 (datele sunt mutate din memorie în registru)
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 00 (vom folosi adresarea directă)
- R/M: 110 (adresare directă)
- Exemplu:
  MOV AX, [100H]
  100010 1 1 00 000 110 00000001 00000000 (8B060100)

4. MOV cu valoare imediata în registru

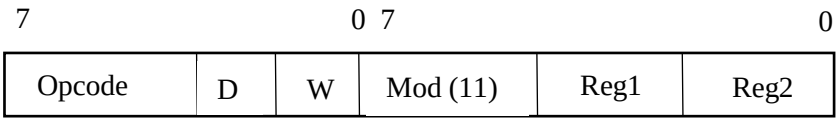


- Sintaxă: MOV reg, imediat
- OpCode: 10111
- Exemplu:
  MOV AX, 5
  10111 000 00000000 00000101 (B80005)

B. Instrucțiuni aritmetice

Instrucțiunile aritmetice din arhitectura Intel 8086 implică operații matematice precum adunarea și scăderea, fiind codificate cu un opcode specific pentru fiecare operație.

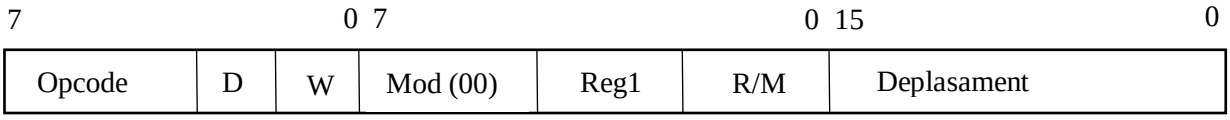
5. ADD între registre



- Sintaxă: ADD dest, src
- Dest ← Dest + Src
- OpCode: 000000
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  ADD AX, BX

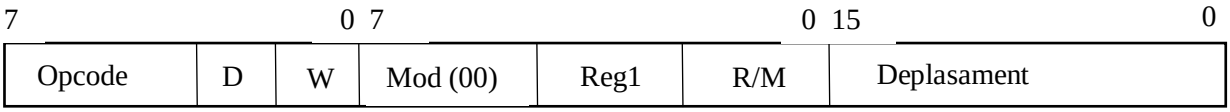
000000 0 1 11 000 011 (01C3)

6. ADD între un registru și memorie



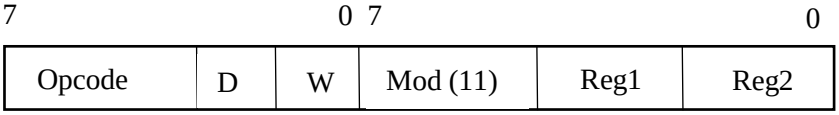
- Sintaxă: ADD reg, [mem]
- OpCode: 000000
- D: 0 (rezultatul se va pune în registru)
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 00 (vom folosi adresarea directă)
- R/M: 110 (adresare directă)
- Exemplu:
  ADD AX, [100H]
  000000 0 1 00 000 110 00000001 00000000 (03060100)

7. ADD între memorie și un registru



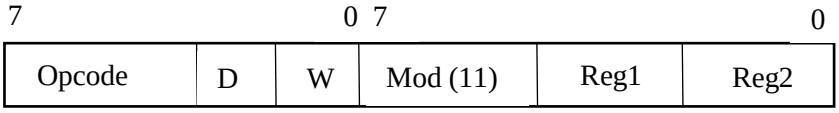
- Sintaxă: ADD [mem], reg
- OpCode: 000000
- D: 1 (rezultatul se va pune in memorie)
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 00 (vom folosi adresarea directă)
- R/M: 110 (adresare directă)
- Exemplu:
  ADD [100H], AX
  000000 1 1 00 000 110 00000001 00000000 (03060100)

8. SUB între registre



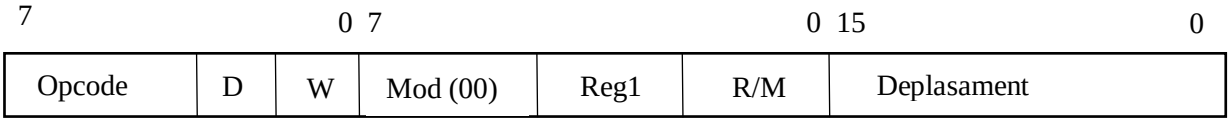
- Sintaxă: SUB dest, src
- Dest ← Dest - Src
- OpCode: 001010
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  SUB AX, BX
  001010 0 1 11 000 011 (29C3)

9. CMP între registre



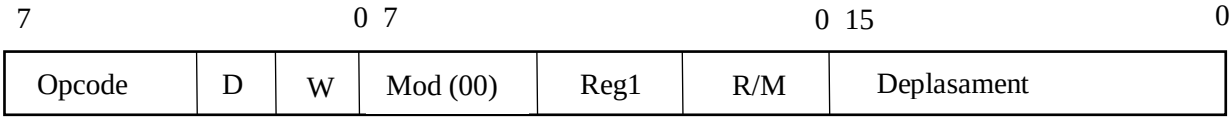
- Sintaxă: CMP dest, src
- OpCode: 001110
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  CMP AX, BX
  001110 0 1 11 000 011 (39C3)

10. ADD cu valoare imediată



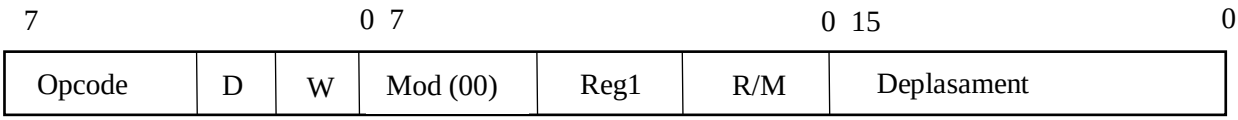
- Sintaxă: ADD dest, imediat
- Dest ← Dest + imediat
- OpCode: 100000
- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 000 (folosit ca un cod pentru adunare)
- Exemplu:
  ADD BX, 5
  100000 0 1 11 011 000 00000000 00000101 (81C30005)

11. SUB cu valoare imediată



- Sintaxă: SUB dest, imediat
- Dest ← Dest - imediat
- OpCode: 100000
- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 101 (folosit ca un cod pentru scădere)
- Exemplu:
  SUB BX, 5
  100000 0 1 11 011 101 00000000 00000101 (81EB0005)

12. CMP cu valoare imediată



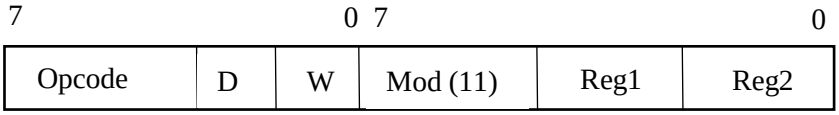
- Sintaxă: CMP dest, imediat
- OpCode: 100000
- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 111 (folosit ca un cod pentru comparație, adică tot scădere)
- Exemplu:
  CMP BX, 5
  100000 0 1 11 011 111 00000000 00000101 (81FB0005)

C. Instrucțiuni logice

Instrucțiuni logice implică operații bitwise precum AND, OR sau XOR. Acestea sunt utilizate pentru a manipula biți.

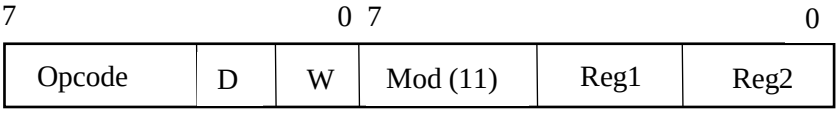
0

13. AND între registre



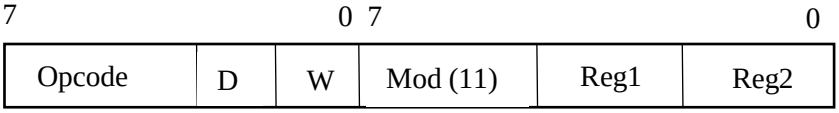
- Sintaxă: AND dest, src
- Dest ← Dest & Src
- OpCode: 001000
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  AND AX, BX
  001000 0 1 11 000 011 (21C3)

14. OR între registre



- Sintaxă: OR dest, src
- Dest ← Dest | Src
- OpCode: 000010
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  OR AX, BX
  000010 0 1 11 000 011 (9C3)

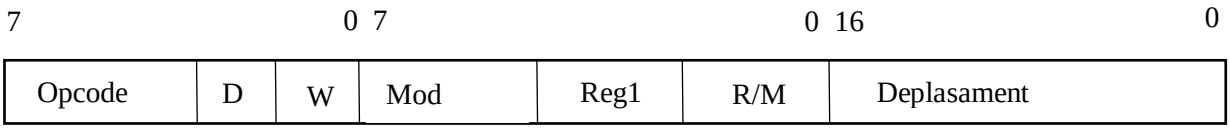
15. XOR între registre



- Sintaxă: XOR dest, src

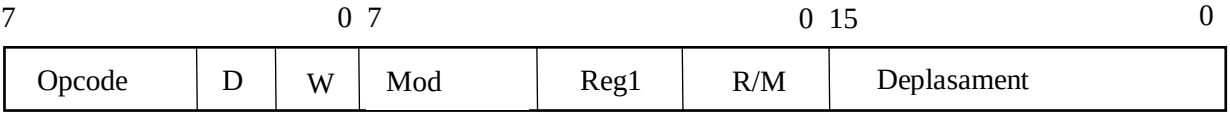
- Dest ← Dest ^ Src
- OpCode: 001100
- D: 0 în general
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- Mod: 11 (cel de-al doilea operand este de asemenea registru)
- Exemplu:
  XOR AX, BX
  001100 0 1 11 000 011 (31C3)

16. AND cu valoare imediată



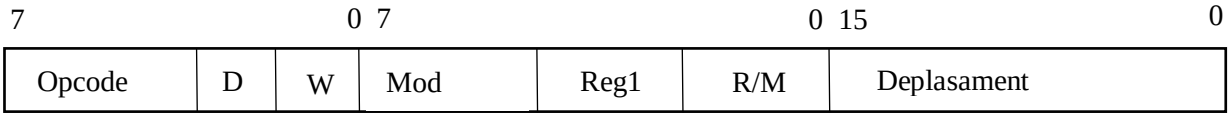
- Sintaxă: AND dest, imediat
- Dest ← Dest & imediat
- OpCode: 100000
- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 100 (folosit ca un cod pentru și logic)
- Exemplu:
  AND BX, 5
  100000 0 1 11 011 100 00000000 00000101 (81E30005)

17. OR cu valoare imediată



- Sintaxă: OR dest, imediat
- Dest ← Dest | imediat
- OpCode: 100000
- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 001 (folosit ca un cod pentru și logic)
- Exemplu:
  OR BX, 5
  100000 0 1 11 011 001 00000000 00000101 (81CB0005)

18. XOR cu valoare imediată



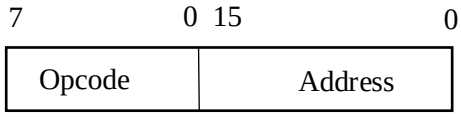
- Sintaxă: XOR dest, imediat
- Dest ← Dest ^ imediat
- OpCode: 100000

- D: 0
- W: 1 sau 0 (1 pentru operații pe word, 0 pentru operații pe byte)
- R/M: 110 (folosit ca un cod pentru și logic)
- Exemplu:
  XOR BX, 5
  100000 0 1 11 011 110 00000000 00000101 (81F30005)

D. Instrucțiuni de salt necondiționat

Instrucțiunile de salt modifică fluxul de execuție al programului.

19. JMP

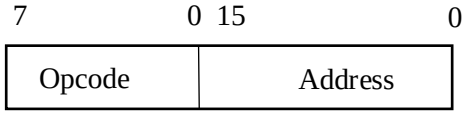


- Sintaxă: JMP address
- IP ← address
- OpCode: 11101001
- Exemplu:
  JMP 1
  11101001 00000000 00000001 (E90001)

E. Instrucțiuni de salt condiționat

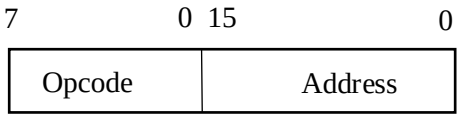
Instrucțiunile de salt modifică fluxul de execuție al programului în funcție de anumite condiții.

20. JE (jump equal)



- Sintaxă: JE address
- If ZeroFlag = 1 then IP ← address
- OpCode: 01110100
- Exemplu:
  JMP 1
  01110100 00000000 00000001 (E90001)

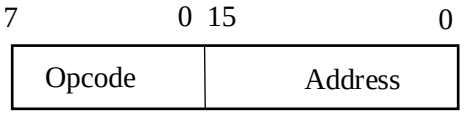
21. JNE (jump not equal)



- Sintaxă: JNE address
- If ZeroFlag = 0 then IP ← address
- OpCode: 01110101
- Exemplu:

JNE 1
01110101 00000000 00000001 (E90001)

22. JG (jump greater)



- Sintaxă: JG address
- If ZeroFlag = 0 and SignFlag = 0 then IP ← address
- OpCode: 01111111
- Exemplu:
  JG 1
  01111111 00000000 00000001 (E90001)

3.2.2 Etapele de execuţie

Prin împărțirea procesului de execuție între Bus Interface Unit și Execution Unit, microprocesorul Intel 8086 optimizează performanța printr-o abordare de lucru paralelă. BIU este responsabilă cu preluarea continuă a instrucțiunilor din memorie și pregătirea lor pentru execuție, în timp ce EU procesează instrucțiunea curentă. Astfel, BIU și EU colaborează pentru a realiza eficient etapele de execuție: Preluarea instrucțiunii, Decodificarea instrucțiunii, Execuția instrucțiunii, Accesul la memorie, Scrierea rezultatului.

A. Preluarea instrucțiunii (Instruction Fetch)

Primul pas în execuția unei instrucțiuni este preluarea acesteia din memorie. Procesorul accesează memoria la adresa specificată de **IP** (Instruction Pointer), iar instrucțiunea este încărcată în coada de instrucțiuni (Instruction Queue), pregătind-o pentru decodificare. La finalul acestei etape, **IP** este incrementat pentru a indica următoarea instrucțiune sau este actualizat la o adresă nouă, în cazul instrucțiunilor de salt.

B. Decodificarea instrucțiunii (Decode)

După ce instrucțiunea a fost preluată, **Unitatea de Control** (UC) o decodifică pentru a determina tipul de operație și operanzii necesari. UC identifică registrele și resursele implicate și generează semnalele de control corespunzătoare pentru a coordona etapele următoare. Această decodificare pregătește procesorul pentru accesul la memorie și execuția operației specificate de instrucțiune.

C. Citirea din memorie (Memory Read)

În cazul în care instrucțiunea necesită acces la memorie pentru a obține un operand, adresa este calculată și utilizată pentru a citi datele din memorie. Aceste date sunt transferate într-un registru temporar sau direct către ALU, pentru a fi folosite în etapa de execuție. Citirea din memorie are loc înaintea execuției, pentru a furniza ALU operanzii necesari.

D. Execuția instrucțiunii (Execute)

Etapa de execuție este momentul în care instrucțiunea este realizată efectiv. **ALU** (Unitatea Aritmetică și Logică) efectuează operația specificată de instrucțiune folosind

operanzii decodați. În această etapă, **ALU** generează rezultatul operației și actualizează anumite **flags** (indicatori de stare) din registrul de **Flags**. Acești indicatori semnalează condiții speciale, cum ar fi rezultat zero, semn negativ sau overflow.

E. Scrierea rezultatului (Memory Write)

Dacă instrucțiunea necesită scrierea rezultatului în memorie, adresa calculată anterior este utilizată pentru a stoca datele rezultate în locația de memorie corespunzătoare. Această etapă finalizează operațiile de acces la memorie pentru instrucțiunea curentă.

F. Scrierea rezultatului (Write Back)

Ultimul pas în execuția unei instrucțiuni este scrierea rezultatului în locația destinată. În funcție de instrucțiune, acest rezultat poate fi stocat într-un registru sau în memoria principală. Scrierea rezultatului finalizează execuția instrucțiunii, pregătind procesorul pentru următoarea instrucțiune din coadă.

Proiectare

4.1 Arhitectura Intel 8086

Structura internă a microprocesorului Intel 8086 este formată din mai multe componente interconectate, fiecare având un rol specific în fluxul de execuție al instrucțiunilor. Aceste componente lucrează împreună pentru a prelua, decodifica și executa instrucțiunile din memorie, gestionând atât datele, cât și condițiile necesare pentru o operație de salt.

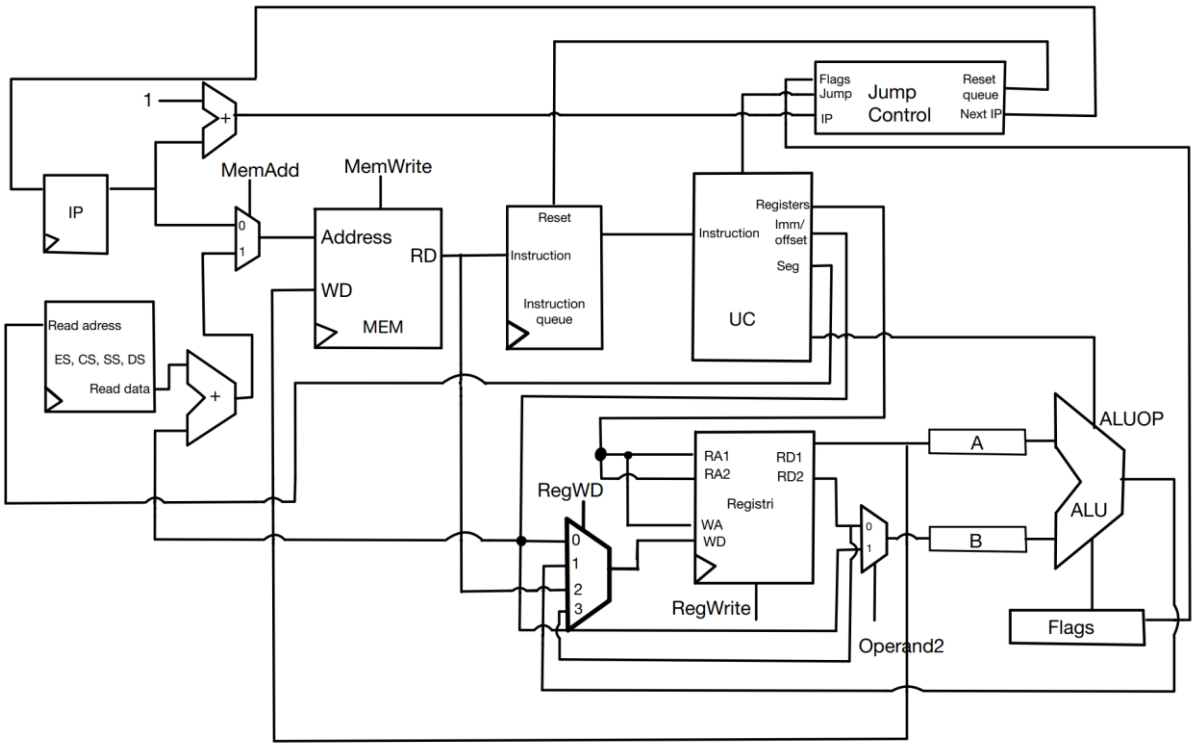


Figura 4.1.a – Design-ul microprocesorului Intel 8086

IP (Instruction Pointer) este un registru dedicat menținerii adresei următoarei instrucțiuni care urmează să fie executată. Acesta este incrementat automat după fiecare ciclu pentru a permite citirea fiecărei instrucțiuni secvențial. Este responsabil de menținerea fluxului de execuție, fiind influențat de instrucțiunile de salt. În cazul execuției unei astfel de instrucțiuni, IP va fi actualizat la adresa de salt.

Registrul de segmentare (ES, CS, SS, DS) va fi un registru folosit pentru calcularea adresei de unde vor fi extrase date din memorie sau unde se vor scrie date în memorie. Calculul acestei adrese se va realiza de un sumator, care are ca operanzi un registru de segment și offset-ul de memorie. Unitatea de Control va extrage offset-ul din instrucțiunea care este executată. Aceasta va seta de asemenea adresa corespunzătoare registrului de segment ce va fi utilizat.

MEM reprezintă memoria microprocesorului, folosită pentru a stoca atât instrucțiuni, cât și date. Este accesată pentru operații de scriere în memorie și de citire din memorie. MEM este activată în etapele de preluare a instrucțiunii și de citire/scriere a datelor. În funcție de semnalele de control, memoriei i se poate comanda să livreze instrucțiuni către Instruction Queue sau să primească/scrie date.

Coada de instrucțiuni este un buffer FIFO (First In, First Out) care permite procesorului să preia următoarele instrucțiuni înainte de a finaliza execuția instrucțiunii

curente. Aceasta furnizează pe rând câte o instrucțiune către Unitatea de Control.

Unitatea de Control este componenta centrală de comandă și coordonare a microprocesorului. Aceasta decodifică instrucțiunile, analizând fiecare câmp al acesteia și generează semnalele necesare pentru a controla celelalte componente (AluOp, MemAdd, RedWD, RegWrite, Operand2, JMP, JE, JNE, JG, etc), dar și datele necesare unei operații (de exemplu: adresa registrilor ce vor fi folosiți de ALU, imediatul sau offset-ul, registrul de segment).

Registrul ce conține registri imediați (AX, BX, CX, DX, SI, DI, BP, SP) va fi utilizat pentru stocarea temporară a datelor, în special a operanzilor care sunt folosiți de ALU pentru operații aritmetice și logice. Se pot citi doi registri simultan (RD1 și RD2), dar se și poate scrie o valoare (imediat, rezultatul din ALU sau o valoare din memorie) la adresa specificată pe intrarea WA (care va coincide cu RD1). Operația executată de acesta va fi decisă de Unitatea de Control prin semnalul RegWrite.

ALU (Unitatea aritmetico-logică) este responsabilă de efectuarea tuturor operațiilor matematice și logice din microprocesor. Aceasta va executa operații de adunare, scădere, și logic, sau logic, sau exclusiv, dar și comparații. Operațiile se pot face între doi registri sau între un registru și un imediat.

Registri A și B sunt folosiți pentru stocarea temporară a operanzilor pentru ALU.

Flag-urile vor fi de fapt, mai multe semnale (Carry Flag, Parity Flag, Auxiliar Carry Flag, Zero Flag, Sign Flag, Trap Flag, Interrupt Enable Flag, Direction Flag, Overflow Flag) folosiți pentru a indica anumite rezultate intermediare ale ALU. Au rol esențial în evaluarea instrucțiunilor de salt.

Jump Control este componenta care se va ocupa de gestionarea instrucțiunilor de salt condiționat sau necondiționat, în funcție de valorile unor anumite flag-uri. Vom folosi o serie de porți ȘI/SAU și multiplexoare pentru a decide care va fi valoarea adresei IP următoare, dar și pentru a controla reset-ul cozii de instrucțiuni (dacă se face un salt, aceasta trebuie golită).

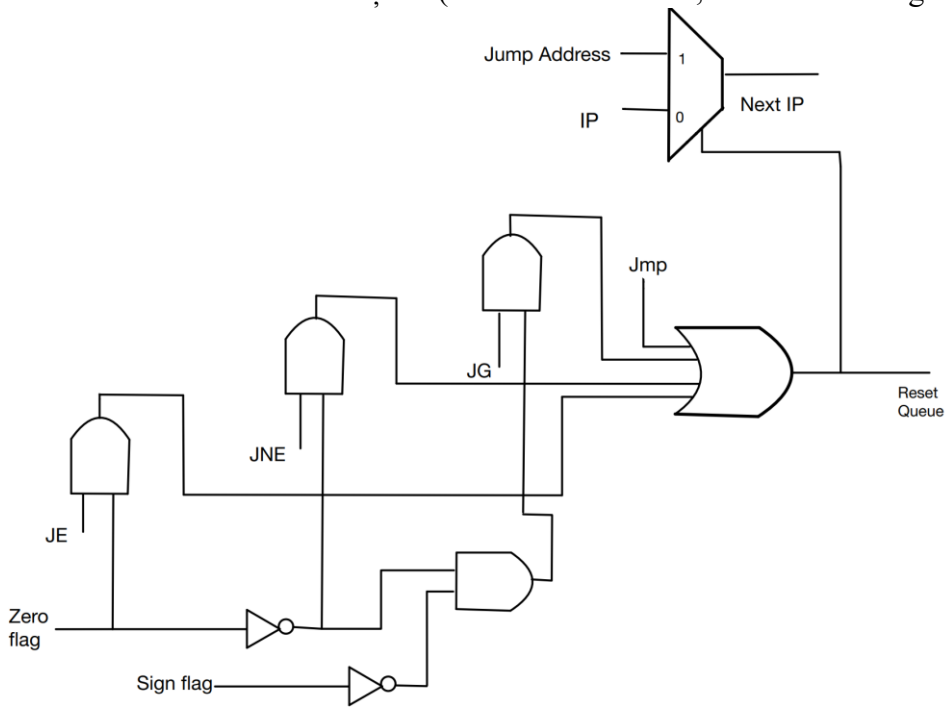


Figura 4.1.b – Jump Control explicit

4.2 Arhitectura Intel 8086 Pipeline

Pipeline-ul în această arhitectură permite execuția simultană a mai multor instrucțiuni prin împărțirea procesului într-o serie de etape distincte, fiecare responsabilă de o parte din execuția instrucțiunii. Astfel, în loc de o singură instrucțiune să fie executată complet înainte de a trece la următoarea, pipeline-ul permite mai multor instrucțiuni să fie procesate simultan. Registrele sunt esențiale pentru separarea etapelor și asigurarea fluxului de date între ele.

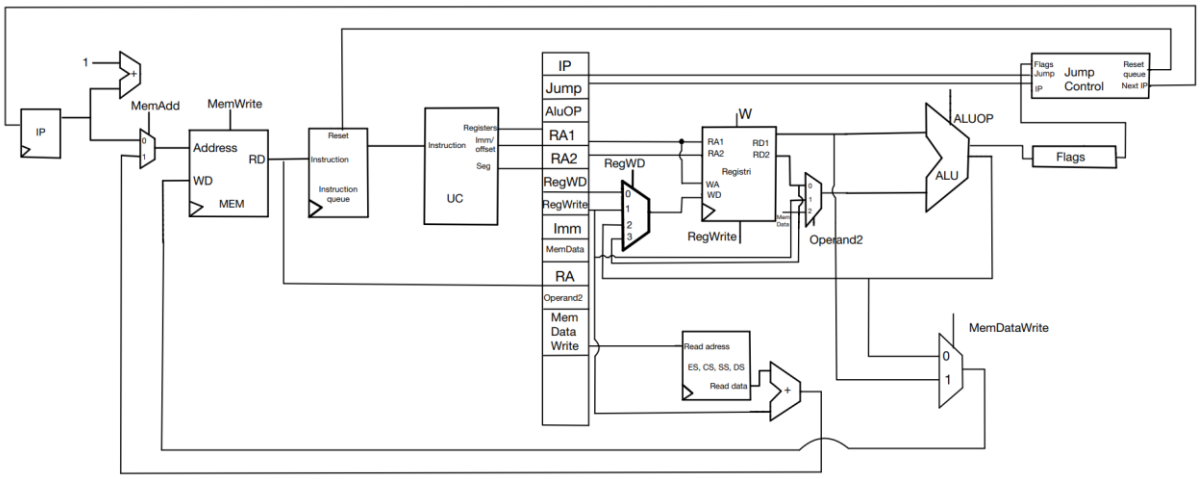


Figura 4.2 – Design-ul microprocesorului Intel 8086 Pipeline

4.3 Hazardurile în arhitectura pipeline

Pipeline-ul aduce un avantaj major prin suprapunerea execuției intruscțiunilor, dar introduce și o serie de hazarduri care pot afecta performanța și corectitudinea execuției.

A. Hazarduri de Date

Apar atunci când o instrucțiune depinde de rezultatul unei instrucțiuni anterioare care nu a fost încă executată complet. Acest tip de hazard poate duce la întârzieri, deoarece instrucțiunea următoare va trebui să aștepte până când datele necesare devin disponibile. O soluție pentru gestionarea hazardurilor de date este introducerea unor NoOp-uri (operații nule care nu afectează execuția programului) în pipeline, pentru a amâna execuția unei instrucțiuni până când datele necesare devin disponibile. Deși această metodă încetinește procesul de execuție, ea asigură corectitudinea execuției și previne erorile cauzate de lipsa datelor.

B. Hazarduri de Control

Acest tip de hazarduri apare atunci când pipeline-ul întâlnește instrucțiuni se salt, de exemplu JMP, JG, JE, JNE, etc. În arhitectura Intel 8086, aceste instrucțiuni pot modifica fluxul de execuție, iar procesorul trebuie să determine corect adresa următoarei instrucțiuni. Procesorul nu poate decide imediat care instrucțiune să preia atunci când întâlnește o instrucțiune de salt, deoarece rezultatul condiției de salt poate depinde de o operație anterioară care nu a fost încă finalizată. O soluție pentru această problemă ar fi golirea pipeline-ului. Dacă procesorul a preluat instrucțiuni greșite, acestea trebuie eliminate din pipeline, ceea ce poate introduce întârzieri și scădea performanța.

C. Hazarduri Structurale

Hazardurile structurale apar atunci când două sau mai multe instrucțiuni încearcă să acceseze aceeași resursă hardware simultan. Pe arhitectura Intel 8086, acest lucru se poate întâmpla dacă există conflicte pentru accesul la unități funcționale, cum ar fi memoria. Dacă

procesorul încearcă să preia o instrucțiune din memorie și, în același timp, o altă instrucțiune încearcă să scrie date în memorie sau să citească datele din memorie, poate apărea un conflict. Pentru a rezolva un astfel de conflict, putem introduce NoOp pentru a preveni accesul simultan la memorie.

4.4 Semnale de control

În arhitectura unui microprocesor, **semnalele de control** joacă un rol esențial în funcționarea corectă și eficientă a fiecărei componente. Ele sunt responsabile pentru sincronizarea și coordonarea tuturor etapelor de execuție ale instrucțiunilor, asigurând că datele sunt preluate, procesate și stocate la momentele potrivite. În contextul unui design cu pipeline, semnalele de control devin și mai importante, deoarece ele trebuie să gestioneze execuția simultană a mai multor instrucțiuni și să rezolve conflictele (hazardurile) care pot apărea între etapele pipeline-ului.

Instrucțiune	OpCode	ALUOP	MemDataWrite	MemAdd	MemWrite	LoadQ	ReadQ	RegWrite	RegWD	Operand2	Jmp	JE	JNE	JG	SegAddress
mov reg, reg	100010	x	x	0	0	1	1	1	11	x	0	0	0	0	00
mov mem, reg	100010	x	1	1	1	0	1	0	xx	x	0	0	0	0	01
mov reg, mem	100010	x	x	1	0	0	1	1	00	x	0	0	0	0	01
mov reg, imm	10111	x	x	0	0	1	1	1	01	x	0	0	0	0	00
add reg, reg	000000	000(+)	x	0	0	1	1	1	10	00	0	0	0	0	00
add reg, mem	000000	000(+)	x	1	0	0	1	1	10	10	0	0	0	0	01
add mem, reg	000000	000(+)	0	1	1	0	1	0	xx	10	0	0	0	0	01
sub reg, reg	001010	001(-)	x	0	0	1	1	1	10	00	0	0	0	0	00
and reg, reg	001000	010(&)	x	0	0	1	1	1	10	00	0	0	0	0	00
or reg, reg	000010	011(|)	x	0	0	1	1	1	10	00	0	0	0	0	00
xor reg, reg	001100	100(^)	x	0	0	1	1	1	10	00	0	0	0	0	00
cmp reg, reg	001110	001(-)	x	0	0	1	1	0	xx	00	0	0	0	0	00
add reg, imm	100000	000(+)	x	0	0	1	1	1	10	01	0	0	0	0	00
sub reg, imm	100000	001(-)	x	0	0	1	1	1	10	01	0	0	0	0	00
and reg, imm	100000	010(&)	x	0	0	1	1	1	10	01	0	0	0	0	00
or reg, imm	100000	011(|)	x	0	0	1	1	1	10	01	0	0	0	0	00
xor reg, imm	100000	100(^)	x	0	0	1	1	1	10	01	0	0	0	0	00
cmp reg, imm	100000	001(-)	x	0	0	1	1	0	xx	01	0	0	0	0	00
jump add	11101001	x	x	0	0	1	1	0	xx	x	1	0	0	0	00
je add	01110100	x	x	0	0	1	1	0	xx	x	0	1	0	0	00
jne add	01110101	x	x	0	0	1	1	0	xx	x	0	0	1	0	00
jg add	01111111	x	x	0	0	1	1	0	xx	x	0	0	0	1	00

Implementare

În acest capitol, vom prezenta cum a fost realizată, pas cu pas, implementarea microprocesorului. Vom explica modul în care fiecare componentă teoretică a fost transformată într-un design hardware funcțional, descriind detaliat componentele principale, cum ar fi unitatea de control, registrele de pipeline și ALU. Scopul acestui capitol este de a arăta cum ideile teoretice au fost puse în practică pentru a construi un sistem eficient și funcțional.

5.1 Componente

5.1.1 Memoria

Memoria microprocesorului este utilizată atât pentru stocarea instrucțiunilor, cât și a datelor. Aceasta este accesată pentru operațiuni de citire și scriere. În funcție de semnalele de control, memoria poate fi configurată pentru a furniza instrucțiuni către coada de instrucțiuni sau pentru a accepta și stoca date.

Entitatea pentru componenta „InstructionDataFetch”

```
entity InstructionDataFetch is
Port (clk: in std_logic;
   ip: in std_logic_vector(15 downto 0);
   en: in std_logic;
   codsegment: in std_logic_vector(16 downto 0);
   memAddress: in std_logic_vector(15 downto 0);
   wd: in std_logic_vector(15 downto 0);
   memWrite: in std_logic;
   memAdd: in std_logic;
   instruction: out std_logic_vector(31 downto 0);
   data: out std_logic_vector(15 downto 0));
end InstructionDataFetch;
```

Declararea memoriei de date și de instrucțiuni

```
type mem_array is array(0 to 65535) of std_logic_vector(31 downto 0);
signal memory : mem_array:= (others => X"00000000");
```

Componenta utilizează un bloc de memorie de tip RAM pentru stocarea instrucțiunilor și datelor, accesând fie o adresă specificată de contorul de instrucțiuni (**ip**), fie o adresă furnizată extern (**memAddress**). Adresa activă este determinată de semnalul **memAdd**. Dacă

este 0, se folosește contorul de instrucțiuni (**ip**), altfel se folosește **memAddress**. Datele din **wd** sunt scrise în memoria internă la adresa activă, atunci când semnalul **memWrite** este activ și ceasul (clk) detectează o tranziție de tip rising edge. Instrucția completă de 32 de biți este citită și transmisă pe ieșirea **instruction**, iar partea inferioară de 16 biți este transmisă pe ieșirea **data**. **NextIp** este setat să fie adresa următoarei instrucțiuni, calculată prin incrementarea valorii curente a lui ip.

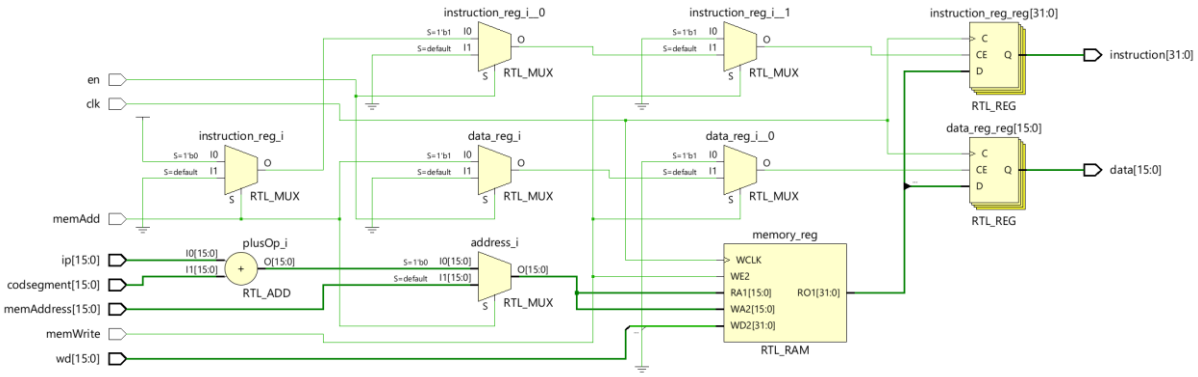


Figura 5.1.1 – Schematic InstructionDataFetch

5.1.2 Coada de instrucţiuni (Instruction Queue)

Scopul acestei componente este de a oferi un mecanism de stocare temporară a instrucțiunilor într-o coadă circulară, care poate adăuga instrucțiuni la sfârșit și le poate citi în ordine de la început, asigurându-se că coada nu se umple sau golește peste limita sa de dimensiune.

Entitate InstructionQueue

```
entity InstructionQueue is
Port (clk: in std_logic;
   reset: in std_logic;
   loadQ: in std_logic;
   read_enable: in std_logic;
   instruction: in std_logic_vector(31 downto 0);
   queueFull: out std_logic;
   queueEmpty: out std_logic;
   instructionOut: out std_logic_vector(31 downto 0) );
end InstructionQueue;
```

Declararea cozii

```
constant QUEUE_SIZE : integer := 6;
type queue_array is array(0 to QUEUE_SIZE-1) of std_logic_vector(31 downto 0);
signal queue : queue_array := (others => (others => '0'));
```

InstructionQueue implementează o coadă circulară pentru stocarea instrucțiunilor în procesorul. La semnalul de reset activat (reset = '1'), indicii **head** și **tail** ale cozii sunt inițializați, iar numărul de elemente din coadă este setat la 0. Când **enable** este activat și coada nu este plină (**count < QUEUE_SIZE**), instrucțiunea primită pe intrare este adăugată la coadă, iar tail este actualizat într-un mod circular. Când **read_enable** este activat și coada nu este goală (**count > 0**), instrucțiunea din capul cozii (**head**) este citită și trimisă pe ieșirea **instructionOut**, iar **head** este actualizat circular. Când numărul de elemente din coadă ajunge la dimensiunea maximă (QUEUE_SIZE), semnalul **queueFull** va fi activat ('1'). Când nu sunt elemente în coadă (count = 0), semnalul **queueEmpty** va fi activat ('1').

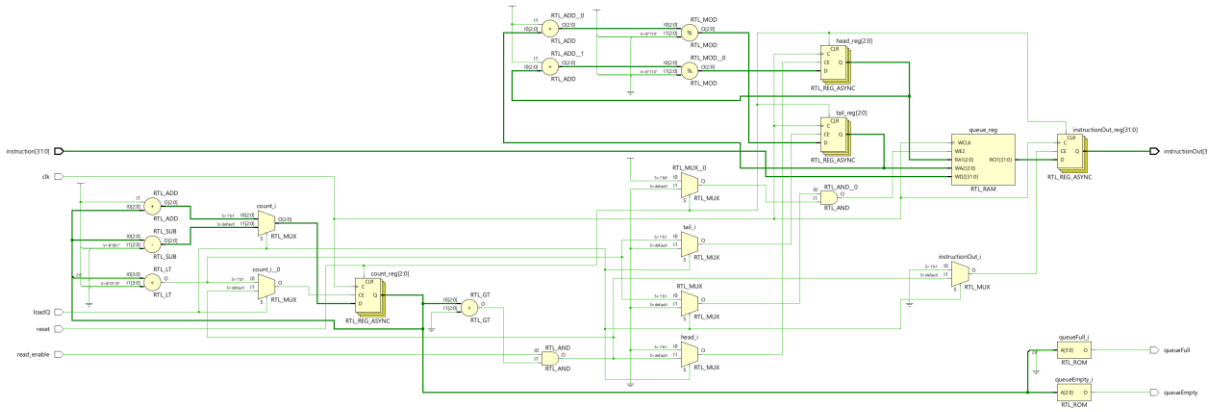


Figura 5.1.2 – Schematic InstructionQueue

5.1.3 Unitatea de control (UC)

Implementarea **Unității de Control (UC)** pentru procesor are scopul de a decoda o instrucțiune binară de 32 de biți și de a genera semnalele de control necesare pentru execuția acesteia. Instrucțiunea de intrare este analizată printr-un proces care interpretează câmpul **OpCode** și, pe baza acestuia, configurează semnalele pentru operații aritmetice, logice, acces la memorie și salturi condiționate sau necondiționate. UC utilizează o mașină de stări finite (FSM) cu stări precum **FETCH, DECODE, EXECUTE, MEMORY_READ, MEMORY_WRITE și WRITE_BACK** pentru a controla fiecare pas al execuției. În starea **FETCH**, instrucțiunea este citită din memorie, iar în starea **DECODE** este interpretată pentru a determina tipul operației și sursa datelor. Stările **EXECUTE** și MEMORY_**READ** sunt utilizate pentru procesarea instrucțiunilor aritmetice, logice sau de acces la memorie, iar starea **WRITE_BACK** asigură scrierea rezultatelor în registre. Controlul ALU este realizat prin semnalul **aluOp**, iar operandul secund este selectat prin **operand2**, care poate fi un registru, o valoare imediată sau o locație din memorie. Accesul la memorie este gestionat prin semnalele

memDataWrite, **memAdd** și **memWrite**, iar registrele sursă și destinație sunt determinate prin reg1 și reg2, cu scrierea controlată de **regWrite** și **regWD**. Codul acoperă instrucțiuni de bază precum MOV, ADD, SUB, CMP, operații logice (AND, OR, XOR) și salturi condiționate (JE, JNE, JG) sau necondiționate (JMP). De asemenea, este inclusă gestionarea valorilor imediate și a offset-urilor pentru accesul la memorie, ceea ce asigură o execuție completă și flexibilă a instrucțiunilor procesorului.

Entitatea pentru componenta UC

```
entity UC is
Port (clk: in std_logic;
   reset: in std_logic;
   enable: in std_logic;
   instruction: in std_logic_vector(31 downto 0);
   aluOp: out std_logic_vector(2 downto 0);
   memDataWrite: out std_logic;
   memAdd: out std_logic;
   memWrite: out std_logic;
   enableNextIP: out std_logic;
   enableRead: out std_logic;
   enableFlags: out std_logic;
   loadQ: out std_logic;
   readQ: out std_logic;
   regWD: out std_logic_vector(1 downto 0);
   regWrite: out std_logic;
   operand2: out std_logic_vector(1 downto 0);
   jmp: out std_logic;
   je: out std_logic;
   jg: out std_logic;
   jne: out std_logic;
   reg1: out std_logic_vector(2 downto 0);
   reg2: out std_logic_vector(2 downto 0);
   seg: out std_logic_vector(1 downto 0);
   imm_offset: out std_logic_vector(15 downto 0);
   d: out std_logic;
   w: out std_logic;
   nextStateDebug: out std_logic_vector(2 downto 0);
);
end UC;
```

5.1.4 Registrii imediati (Reg)

Registrul care gestionează registrele imediate (AX, BX, CX, DX, SI, DI, BP, SP) servește ca spațiu temporar de stocare pentru date, în special operanzi utilizați de ALU în calcule aritmetice și logice. Acesta permite acces simultan pentru citirea a două registre (RD1

și RD2) și oferă flexibilitatea de a scrie o valoare specificată, fie că este un operand imediat, un rezultat al ALU sau o valoare din memorie.

Entitatea pentru componenta „RegFile”

```
entity RegFile is
Port (clk: in std_logic;
   en: in std_logic;
   reg1: in std_logic_vector(2 downto 0);
   reg2: in std_logic_vector(2 downto 0);
   w: in std_logic;
   regWrite: in std_logic;
   regWD: in std_logic_vector(1 downto 0);
   imm: in std_logic_vector(15 downto 0);
   memData: in std_logic_vector(15 downto 0);
   aluRes: in std_logic_vector(15 downto 0);
   readreg2: in std_logic_vector(15 downto 0);
   rd1: out std_logic_vector(15 downto 0);
   rd2: out std_logic_vector(15 downto 0) );
end RegFile;
```

Declararea registrilor

```
type reg_array is array(0 to 7) of std_logic_vector(15 downto 0);
signal reg_file : reg_array := (others => X"0000");
-- 0 - AX, 1 - CX, 2 - DX, 3 - BX, 4 - SP, 5 - BP, 6 - SI, 7 - DI
```

Arhitectura modulului **RegFile** este organizată în jurul unui array de registre, care stochează și gestionează datele utilizate de procesor. Acest array, denumit reg_file, conține 8 registre a câte 16 biți fiecare, corespunzătoare registrelor din arhitectura 8086: AX, CX, DX, BX, SP, BP, SI și DI. Funcționarea modulului se bazează pe trei procese principale care coordonează citirea datelor, selecția surselor pentru scriere și actualizarea conținutului registrelor.

În primul proces, citirea din registre este determinată de semnalul w, care indică dacă operațiunea se va face pe 16 biți sau pe 8 biți. Dacă **w** este 1, valorile complete ale registrelor selectate de reg1 și reg2 sunt citite direct din array-ul **reg_file**. Aceste valori sunt atribuite semnalelor de ieșire **rd1** și **rd2**, care reprezintă datele citite. Dacă **w** este 0, operațiunea este realizată pe byte. Pentru fiecare registru selectat de reg1 sau reg2, se determină fie byte-ul inferior, fie byte-ul superior. Această decizie este controlată de codul binar al lui reg1 sau reg2 și de semnalul auxiliar **low_high**, care indică partea activă a registrului. Datele citite sunt ajustate pentru a include zero în partea neutilizată, oferind o ieșire consistentă de 16 biți. În

timpul acestui proces, adresa registrului activ este stocată în semnalul **wa**, care este utilizat ulterior pentru scriere.

Al doilea proces determină sursa datelor care vor fi scrise în registru. Această decizie este guvernată de semnalul **regWD**, care selectează una dintre următoarele surse: memorie (**memData**), valoare imediată (**imm**), rezultatul ALU (**aluRes**), alt registru (**readreg2**). Valoarea selectată este atribuită semnalului **wd**, care conține efectiv datele destinate pentru scriere.

Ultimul proces actualizează conținutul registrelor pe baza datelor din **wd**. Acest proces este sincronizat la frontul ascendent al semnalului de ceas (clk) și este condiționat de activarea semnalului **regWrite**. Dacă **w** este 1, întregul cuvânt de 16 biți este scris direct în registrul corespunzător adresei **wa**. Dacă **w** este 0, se scrie doar byte-ul selectat (superior sau inferior). Această selecție este controlată de semnalul **low_high**, astfel: dacă low_high este 0, byte-ul inferior al registrului este înlocuit cu partea inferioară din **wd**, iar dacă low_high este 1, byte-ul superior este actualizat cu partea superioară a semnalului **wd**.

5.1.5 Registrii de segment (ES, CS, DS, SS)

Registrul de segmentare este utilizat pentru a calcula adresa de memorie, adunând valoarea unui registru de segment cu un offset. Unitatea de control extrage offset-ul din instrucțiune și selectează registrul de segment corespunzător. Adresa finală calculată va fi adresa de memorie la care se vor efectua operații de scriere sau de citire. Implementarea utilizează un tip de memorie ROM pentru stocarea fiecărui registru de segment. (valorile din rom sunt provizorii)

Entitatea pentru componenta „Segment Registers”

```
entity SegmentRegisters is
Port (segAddress: in std_logic_vector(1 downto 0);
   offset: in std_logic_vector(15 downto 0);
   memAddress: out std_logic_vector(15 downto 0);
   segmentValue: out std_logic_vector(15 downto 0) );
end SegmentRegisters;
```

Declararea memoriei Rom

```
type rom_type is array (0 to 3) of std_logic_vector(15 downto 0);
signal rom : rom_type := ( 0 => x"0000", -- CS
                           1 => x"0040", -- DS
                           2 => x"0100", -- SS
                           3 => x"1000"); -- ES
```

Logica pentru calcularea adresei din memorie

```
segOffset <= rom(conv_integer(segAddress));
memAddress <= segOffset + offset;
segmentValue <= segOffset;
```

Acest cod VHDL definește componenta numită **SegmentRegisters** care calculează adresa de memorie, folosind adresa specifică unui registru de segment și offset-ul extras din instrucțiune. Se implementează un mecanism simplu de adresare segmentată: segAddress va selecta registrul de segment corespunzător din ROM și va fi ulterior combinat cu offset-ul pentru a rezulta adresa de memorie finală.

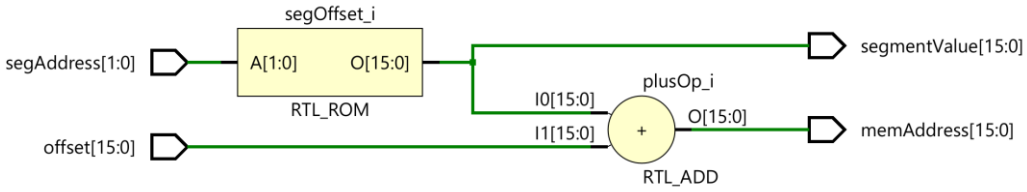


Figura 5.1.5 – Schematic SegmentRegisters

5.1.6 Unitatea Aritmetico-Logică (ALU)

În implementarea microprocesorului, **ALU** un rol deosebit de important, fiind responsabilă pentru realizarea tuturor operațiilor matematice și logice. ALU a fost proiectată să execute operații precum adunare, scădere, și operații logice, cum ar fi „și” (AND), „sau” (OR), „sau exclusiv” (XOR), precum și comparații. Aceste operații pot fi efectuate fie între doi registri, fie între un registru și o valoare imediată, fie între un registru și o valoare din memorie.

Entitatea componentei ALU

```
entity ALU is
Port (clk: in std_logic;
   enable: in std_logic;
   reset: in std_logic;
   rd1: in std_logic_vector(15 downto 0);
   rd2: in std_logic_vector(15 downto 0);
   imm: in std_logic_vector(15 downto 0);
   memData: in std_logic_vector(15 downto 0);
   operand2: in std_logic_vector(1 downto 0);
   aluOp: in std_logic_vector(2 downto 0);
   memDataWrite: in std_logic;
   dir: in std_logic;
   result: out std_logic_vector(15 downto 0);
   flags: out std_logic_vector(8 downto 0);
   wdMem: out std_logic_vector(15 downto 0));
end ALU;
```

Arhitectura modulului **ALU** implementează o unitate aritmetică și logică în care sunt gestionate semnale și operații între doi operanzi, rezultând valori calculate și flaguri asociate. Principalele semnale interne includ trei vectori pe 16 biți: A, B și C. Semnalul A este atribuit direct de la primul operand (**rd1**), în timp ce B este selectat printr-un proces de multiplexare care alege între mai multe surse pe baza semnalului de control **operand2**. Aceste surse pot fi al doilea operand (**rd2**), un offset imediat (**imm**) sau datele de memorie (**memData**).

Operațiile efectuate de ALU sunt controlate printr-un alt proces care interpretează semnalul **aluOp**, combinat cu un indicator de direcție (**dir**) pentru operațiile de scădere. Funcțiile implementate includ adunare, scădere (cu suport pentru direcția operandului), operații logice precum AND, OR și XOR, precum și un caz implicit pentru rezultate nedeterminate. Rezultatul fiecărei operații este stocat în semnalul C, care este ulterior utilizat pentru ieșirea finală **result**.

Pe lângă calculul operațiilor, arhitectura include logica pentru generarea registrilor de flag, determinată de semnalul enable. Flagurile principale sunt: **ZeroFlag**, activat dacă rezultatul este zero; **ParityFlag**, bazat pe cel mai puțin semnificativ bit al rezultatului; și **SignFlag**, determinat de cel mai semnificativ bit al rezultatului. Restul flagurilor sunt rezervate și setate la zero. În ceea ce privește datele de scriere în memorie, acestea sunt selectate printr-un semnal de control (**memDataWrite**), care alege între rezultatul operației (C) și primul operand (rd1). În acest mod, arhitectura integrează atât operațiile aritmetice și logice, cât și semnalele asociate, pentru a gestiona ieșirile într-o manieră controlată.

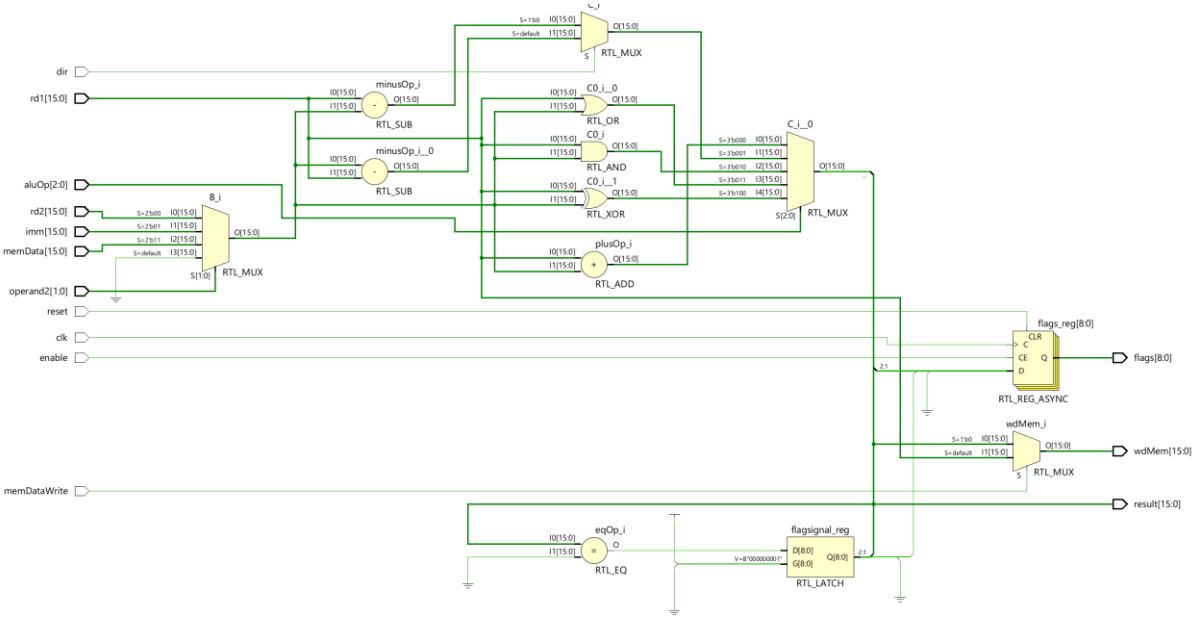


Figura 5.1.6 – Schematic ALU

5.1.7 Jump Control

Componenta Jump Control se ocupă de gestionarea instrucțiunilor de salt condiționat și necondiționat, folosind flag-uri precum Zero (ZF) sau Sign (SF) pentru a determina condițiile de salt. Logica este implementată prin porți AND și OR, care evaluează aceste flag-uri, și multiplexoare care selectează între adresa de instrucțiune următoare și adresa de salt, în funcție de semnalele de control generate. În cazul săriturilor necondiționate, adresa de salt este preluată direct din instrucțiune, fără a verifica flag-urile.

Entitatea componentei JumpControl

```
entity JumpControl is
Port (ip: in std_logic_vector(15 downto 0);
   jump: in std_logic;
   je: in std_logic;
   jne: in std_logic;
   jg: in std_logic;
   flags: in std_logic_vector(8 downto 0);
   jumpAddress: in std_logic_vector(15 downto 0);
   nextIp: out std_logic_vector(15 downto 0);
   resetqueue: out std_logic );
end JumpControl;
```

Arhitectura modulului **JumpControl** implementează logica de control pentru gestionarea săriturilor condiționate într-un program, folosind diverse semnale de intrare și flaguri pentru a determina dacă execuția ar trebui să continue secvențial sau să sară la o adresă specifică (**jumpAddress**).

Logica internă extrage flagurile relevante, zero și sign, din vectorul flags. Acestea sunt utilizate pentru a calcula trei condiții distincte: **resultJE**, activată dacă săritura este

condiționată de egalitate și flagul zero este activ; **resultJNE**, activată dacă condiția este de

Logica pentru Jump Control

```
zero <= flags(0);
sign <= flags(1);

resultJE <= je and zero;
resultJNE <= jne and (not zero);
resultJG <= jg and ( (not zero) and (not sign) );

resultOR <= jump or resultJE or resultJNE or resultJG;

nextIp <= ip when resultOR = '0' else jumpAddress;
resetqueue <= resultOR;
```

inegalitate și flagul zero este inactiv; și **resultJG**, activată dacă valoarea este mai mare decât zero, ceea ce implică atât dezactivarea flagului zero, cât și a flagului de semn. Aceste condiții sunt combinate logic cu semnalul de săritură necondiționată jump pentru a genera **resultOR**, care decide dacă săritura trebuie efectuată. În funcție de rezultatul lui **resultOR**, ieșirea **nextIp** este setată fie la valoarea curentă a pointerului de instrucțiuni (**ip**), dacă nu are loc o săritură, fie la adresa de săritură (**jumpAddress**), dacă una dintre condiții este adevărată. În paralel, ieșirea **resetqueue** este utilizată pentru a reseta coada de instrucțiuni, sincronizând execuția cu noua locație a programului, dacă are loc o săritură.

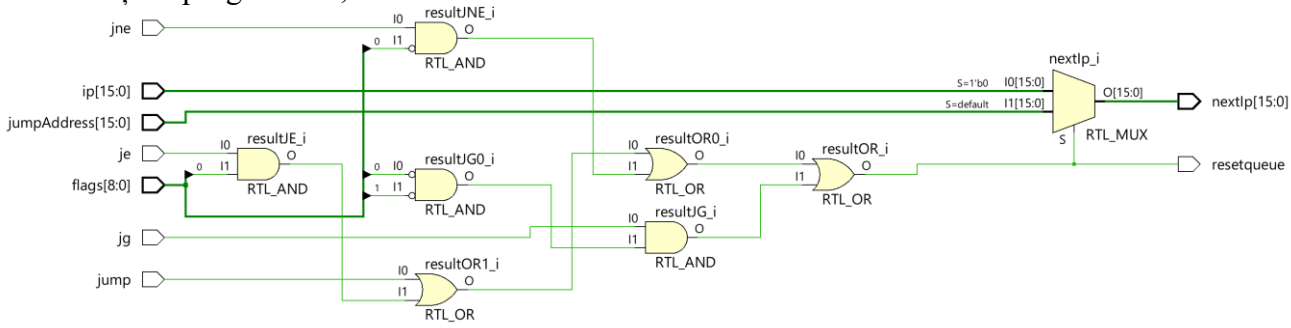


Figura 5.1.7 – Schematic Jump Control

5.1.8 Microprocesorul Pipeline complet

Acest cod implementează un modul de nivel superior (topLevel) pentru procesor, care integrează și coordonează mai multe componente esențiale, inclusiv controlul instrucțiunilor, registrele, ALU-ul și mecanismele de salt. Modulul este structurat pentru a executa o instrucțiune în mai multe etape, asigurând o execuție coordonată și funcțională.

Componente Integrare

1. **Instruction Data Fetch (IF):** Citește instrucțiunile din memorie pe baza adresei IP și furnizează datele pentru procesare ulterioară.
2. **Segment Registers:** Gestionează adresarea segmentată, combinând valorile de segment și offset pentru a calcula adresele efective.
3. **Instruction Queue:** Stochează instrucțiunile pentru execuție ulterioară, controlată prin

semnalele de încărcare (loadQ) și citire (readQ).

4. **Unitatea de Control (UC):** Decodifică instrucțiunea și generează semnalele de control necesare pentru fiecare etapă de execuție.
5. **RegFile:** Gestionează registrele procesorului, permițând citirea și scrierea datelor.
6. **ALU:** Realizează operațiile aritmetice și logice pe baza instrucțiunii curente.
7. **Jump Control:** Gestionează salturile condiționate și necondiționate, actualizând IP-ul pe baza rezultatului ALU și a steagurilor (flags).

Funcționalitate

- **Fluxul Instrucțiunii:** Modulul începe cu citirea unei instrucțiuni din memorie (InstructionDataFetch), care este stocată în coada de instrucțiuni (InstructionQueue). Instrucțiunea este apoi decodificată de UC, care generează semnalele necesare pentru ALU, accesul la memorie sau registre.
- **Operații ALU:** ALU procesează datele din registre, memoria de date sau valori imediate, rezultatul fiind scris fie în registre, fie în memorie.
- **Salturi:** Salturile condiționate sunt realizate de JumpControl, care decide actualizarea IP-ului în funcție de condițiile specificate și steagurile ALU.
- **Controlul General:** UC sincronizează execuția instrucțiunilor pe baza unei mașini de stări finite.

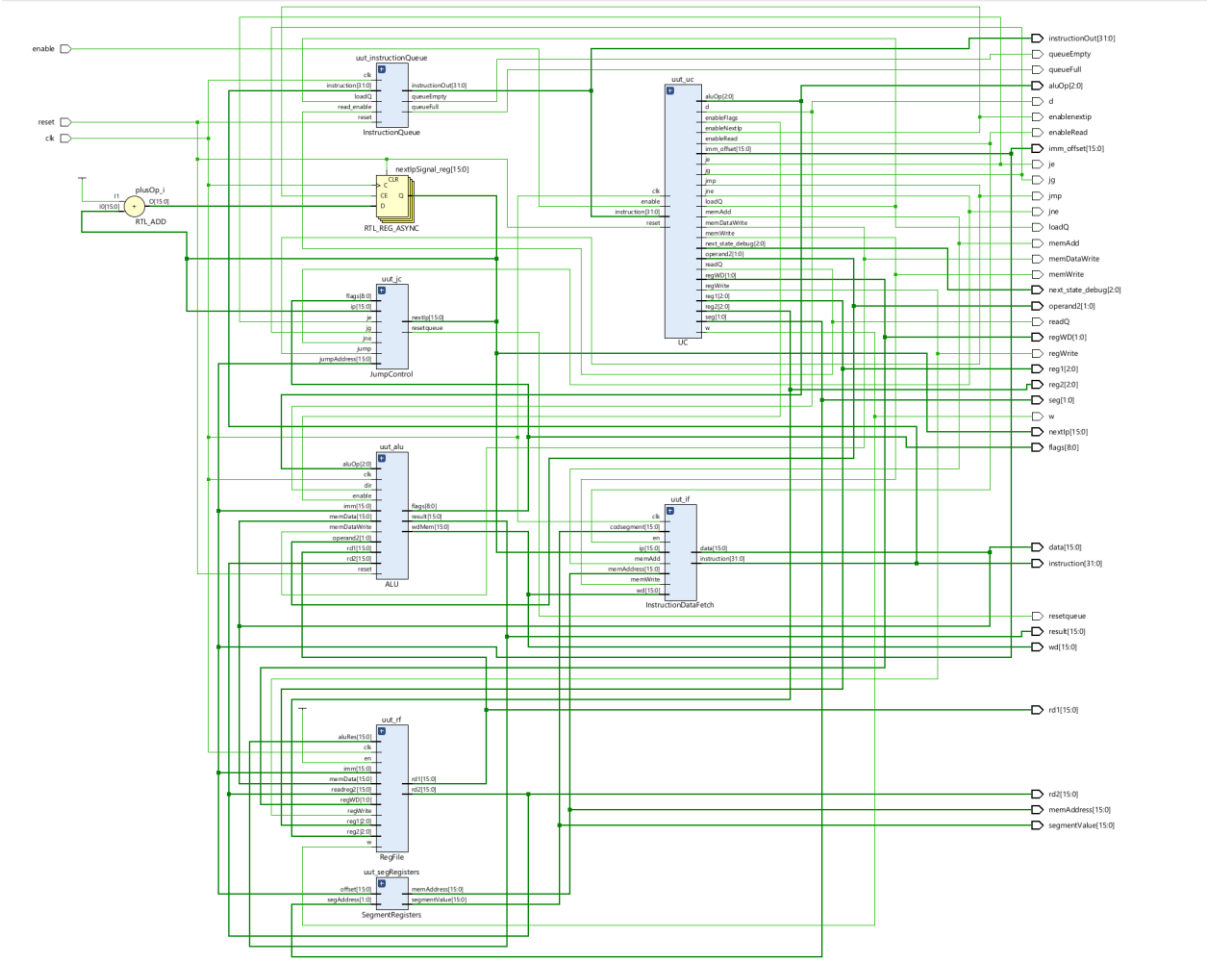


Figura 5.1.8 – Schematic TopLevel

5.2 Cod program Assembly x86

Pentru a demonstra funcționalitatea procesorului Intel 8086 implementat în VHDL, acest capitol prezintă un program de testare scris în limbaj de asamblare. Programul utilizează un set divers de instrucțiuni fundamentale ale arhitecturii 8086, incluzând operații aritmetice, logice, de comparare, precum și salturi condiționate și necondiționate. Scopul acestui program este de a valida funcționarea corectă a unității de control și a unității aritmetico-logice (ALU), precum și de a verifica generarea semnalelor de control pentru fiecare tip de instrucțiune. Prin rularea acestui cod, se evaluează comportamentul procesorului într-o gamă largă de scenarii, asigurând conformitatea cu specificațiile arhitecturii originale și identificarea eventualelor erori în implementare.

Program in assembly x86

```
section .data
   value1 dw 10
   value2 dw 5
   result dw 0

section .text
   mov AX, value1      ; Încărcăm value1 în AX
   mov BX, value2      ; Încărcăm value2 în BX
   mov CX, 4
   mov DX, 2

   add AX, BX          ; AX = AX + BX (10 + 5 = 15)
   mov [result], AX
   sub CX, DX          ; CX = CX - DX (4 - 2 = 2)
   cmp CX, DX          ; Comparăm CX și DX
   je equal_label      ; Salt la equal_label dacă sunt egale (ZeroFlag = 1)
   jne not_equal_label  ; Salt la not_equal_label (ZeroFlag = 0)

equal_label:
   add BX, 5           ; BX = BX + 5 (5 + 5 = 10)

not_equal_label:
   sub BX, 3           ; BX = BX - 3
   and AX, CX          ; AX = AX & CX
   or CX, DX           ; CX = CX | DX
   jmp continue_label   ; Salt la continue_label

reset_label:
    mov AX, 0
    jmp end_label

continue_label:
   xor AX, BX          ; AX = AX ^ BX
   and BH, 5           ; BH = BH & 5
   or AL, 3            ; AL = AL | 3
   xor BX, 2           ; BX = BX ^ 2
   add AX, value1
   cmp AX, 1
   jg reset_label    ; Salt la reset_label daca AX > 1 (ZeroFlag = 0 && SignFlag = 0)
end_label:
```

Pentru a asigura funcționarea corectă a procesorului Intel 8086 implementat, a fost necesară identificarea și rezolvarea hazardurilor care ar putea afecta execuția instrucțiunilor. În codul prezentat anterior, au fost identificate hazarduri de date și de control.

Hazardurile de date apar atunci când o instrucțiunre utilizează un registru al cărui conținut este modificat de o instrucțiune anterioară care încă nu a fost executată complet. Câteva exemple relevante extrase din codul prezentat sunt :

1. add AX, BX
   mov result, AX
2. add AX, value1
   cmp AX, 1

Hazardurile de control apar în cazul salturilor condiționate, atunci când procesorul trebuie să ia decizii pe baza flag-urilor de stare, iar aceste informații nu sunt disponibile imediat. În codul prezentat se regăsesc câteva situații similare :

1. cmp CX, DX
   je equal_label
2. cmp AX, 1
   jg reset_label

Pentru a rezolva aceste tipuri de hazarduri, se vor introduce instrucțiuni de tip NoOp, oferind timp pipeline-ului să finalizeze procesarea. O instrucțiune este considerată NoOp dacă nu modifică starea procesorului, a registrelor, a memoriei sau a semnalelor de control. În cazul arhitecturii Intel 8086, puntem considera o NoOp instrucțiunea **add AX, 0** (codificarea în binar: 100000 0 1 11 000 000 0000000000000000). De asemenea, unele hazarduri au fost rezolvate prin rearanjarea instrucțiunilor pentru a permite procesorului să continue execuția în mod eficient, evitând conflictele.

Program in assembly x86 cu hazardurile rezolvate

```
section .data
   value1 dw 10
   value2 dw 5
   result dw 0

section .text
   mov AX, value1      ; Încărcăm value1 în AX
   mov BX, value2      ; Încărcăm value2 în BX
   mov CX, 4
   mov DX, 2

   add AX, BX          ; AX = AX + BX (10 + 5 = 15)
   sub CX, DX          ; CX = CX - DX (4 - 2 = 2)
   mov [result], AX
   cmp CX, DX          ; Comparăm CX și DX
   noop
   je equal_label      ; Salt la equal_label dacă sunt egale (ZeroFlag = 1)
   jne not_equal_label  ; Salt la not_equal_label (ZeroFlag = 0)
```

```
equal_label:
   add BX, 5          ; BX = BX + 5 (5 + 5 = 10)

not_equal_label:
   and AX, CX         ; AX = AX & CX
   sub BX, 3          ; BX = BX - 3
   or CX, DX          ; CX = CX | DX
   jmp continue_label   ; Salt la continue_label

reset_label:
    mov AX, 0
    jmp end_label

continue_label:
   xor AX, BX         ; AX = AX ^ BX
   and BH, 5          ; BH = BH & 5
   or AL, 3           ; AL = AL | 3
   xor BX, 2          ; BX = BX ^ 2
   add AX, value1
   noop
   cmp AX, 1
   noop
   jg reset_label    ; Salt la reset_label daca AX > 1 (ZeroFlag = 0 && SignFlag = 0)
end_label:
```

Testare și validare

Capitolul de Testare și Validare descrie procesul prin care funcționalitatea și performanța microprocesorului au fost verificate pentru a asigura că designul respectă specificațiile și funcționează corect în toate condițiile.

6.1 Memoria (Mem)

Testbench-ul validează funcționalitatea modulului **InstructionDataFetch**, testând citirea instrucțiunilor din memorie și scrierea/citirea unei valori la o anumită adresă.

Scenarii de testare

1. **Citirea primei instrucțiuni din memorie:**
   - **Intrare:** ip = x"0000", codsegment = x"0000", en = '1', memAdd = '0'.
   - **Rezultat așteptat:** Ieșirea instruction să conțină instrucțiunea din locația specificată.
2. **Citirea unei valori din memorie:**
   - **Intrare:** codsegment = x"0040", memAdd = '1'.
   - **Rezultat așteptat:** Ieșirea data să conțină valoarea din locația memoriei adresată.
3. **Scrierea unei valori în memorie:**
   - **Intrare:** memWrite = '1', wd = x"1234", memAddress = x"0000".
   - **Rezultat așteptat:** Memoria să stocheze valoarea la locația specificată.
4. **Verificarea valorii scrise:**
   - **Intrare:** memAdd = '1', memWrite = '0'.
   - **Rezultat așteptat:** Ieșirea data să conțină valoarea x"1234" scrisă anterior

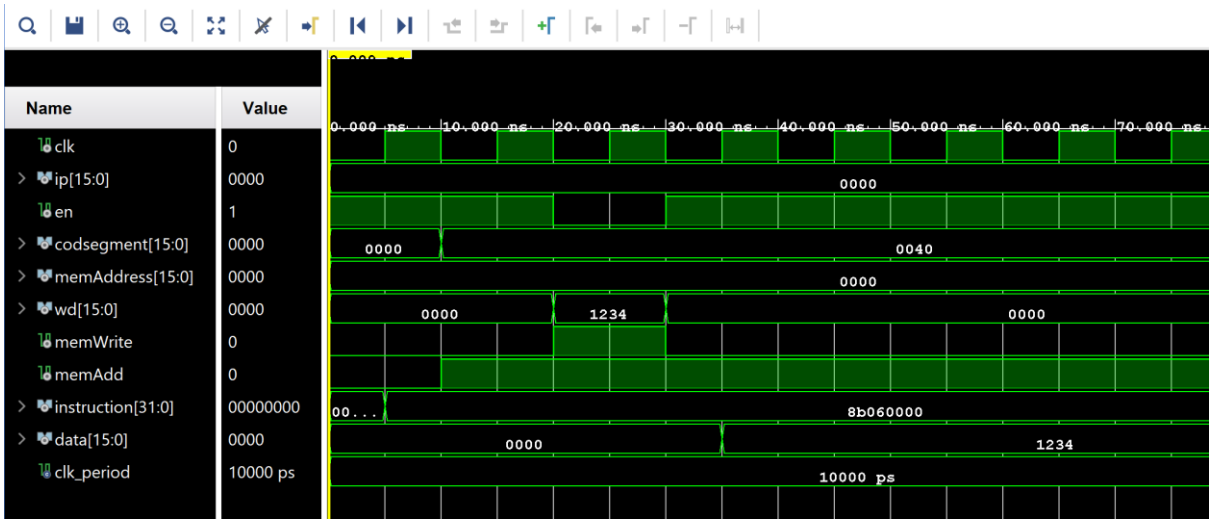


Figura 6.1 – Simulare InstructionDataFetch

6.2 Coada de instrucțiuni (Instruction Queue)

Acest Testbench validează funcționalitatea modulului **InstructionQueue** prin resetarea inițială a cozii, încărcarea instrucțiunilor și citirea acestora. Modulul este testat pentru comportamentul corect la activarea semnalelor de control **loadQ** și **read_enable**, verificându-se atât introducerea instrucțiunilor în coadă, cât și extragerea acestora. De asemenea, se observă semanelele **queueFull** și **queueEmpty** pentru a confirma gestionarea corectă a stărilor de

coadă plină și goală.

Scenarii de testare

1. **Resetare:** Coada este resetată (semnalele interne sunt inițializate).
2. **Încărcare:** Se încarcă șase instrucțiuni consecutive în coadă.
3. **Verificare queueFull:** După ce coada este plină, se verifică semnalul queueFull.
4. **Citire:** Se citesc toate cele șase instrucțiuni din coadă.
5. **Verificare queueEmpty:** După ce coada este goală, se verifică semnalul queueEmpty.

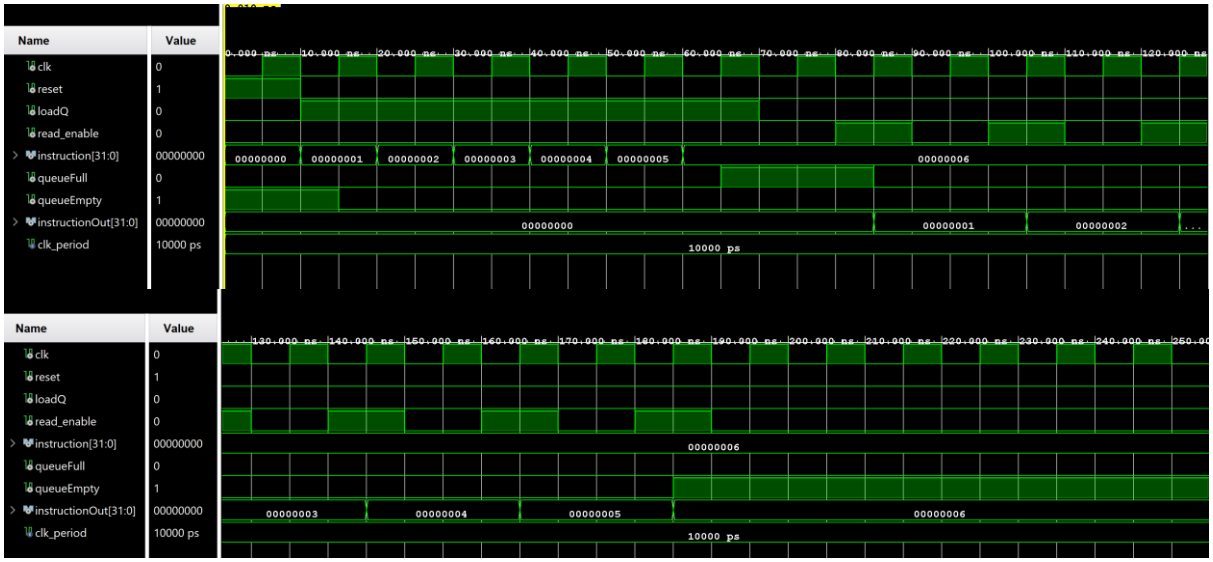
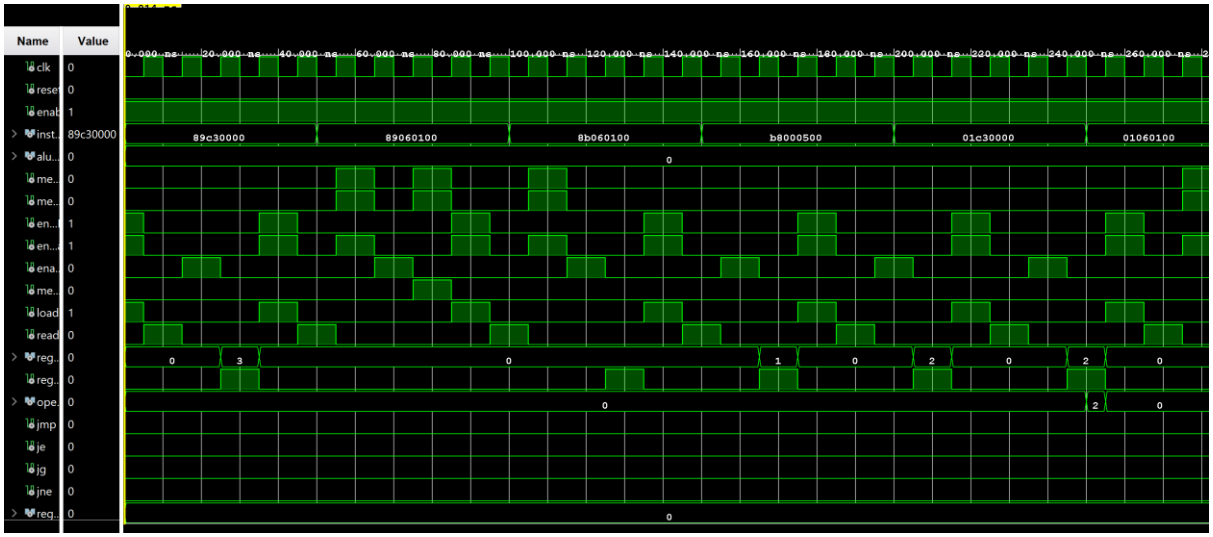


Figura 6.2 – Simulare InstructionQueue

6.3 Unitatea de control (UC)

Testbench-ul verifică funcționalitatea modulului **UC** prin aplicarea unui set de instrucțiuni codificate și monitorizarea semnalelor de control generate la ieșire. Instrucțiunile testate includ operații precum transfer de date între registre și memorie (**mov**), operații aritmetice (**add**, **sub**), operații logice (**and**, **or**, **xor**) și instrucțiuni de salt condiționat (**jmp**, **je**, **jne**, **jg**). Semnalele de ieșire sunt analizate pentru a valida decodificarea corectă a instrucțiunilor și generarea semnalelor de control corespunzătoare fiecărei operații.



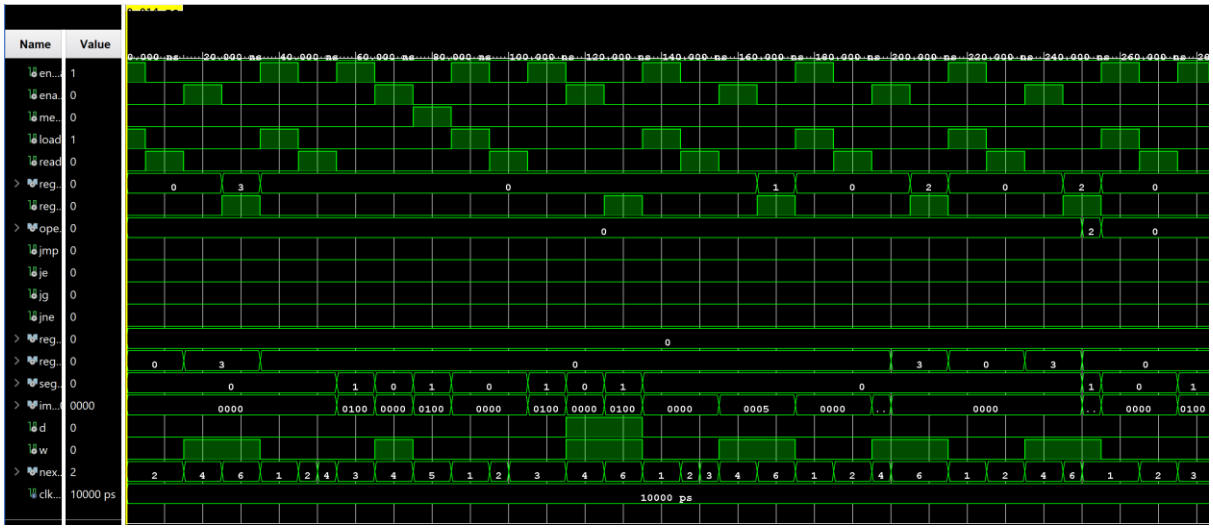


Figura 6.3 – Simulare Unitate de Control

6.4 Registrii imediati (Reg)

Testbench-ul validează funcționalitatea modulului **RegFile** prin testarea operațiilor de scriere și citire în registre utilizând diferite surse de date: **memData**, **imm** și **aluRes**. Se scriu valori în registrele **AX, BX, AH** și **BL**, iar citirile ulterioare confirmă actualizarea corectă a registrelor. Semnalul **regWrite** controlează scrierea, iar combinațiile semnalelor **regWD** și w selectează sursa datelor și partea registrului accesată. Rezultatele obținute pe ieșirile **rd1** și **rd2** validează comportamentul corect al modulului pentru toate scenariile testate.

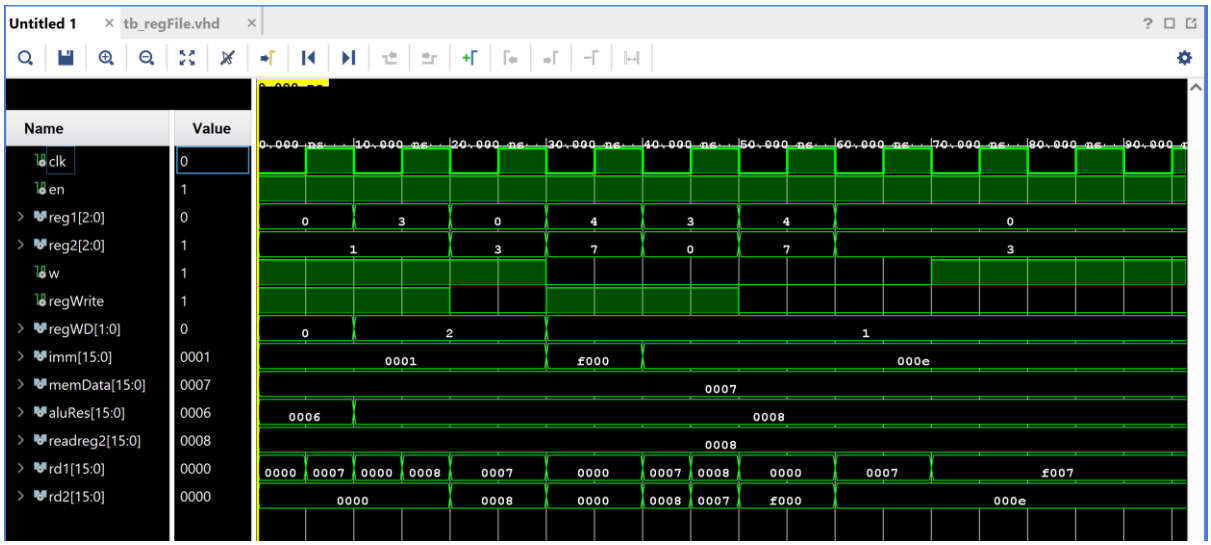


Figura 6.4 – Simulare RegFile

6.5 Registrii de segment (ES, CS, DS, SS)

Testbench-ul verifică funcționalitatea modulului **SegmentRegisters**, testând corectitudinea calculului adresei de memorie (**memAddress**) și a valorii segmentului (**segmentValue**) pentru diferite valori ale semnalului **segAddress**. Se aplică aceleași valori de offset (0x1234) pentru toate combinațiile posibile ale segmentului (00, 01, 10, 11), iar ieșirile

sunt monitorizate pentru a confirma generarea corectă a rezultatelor.

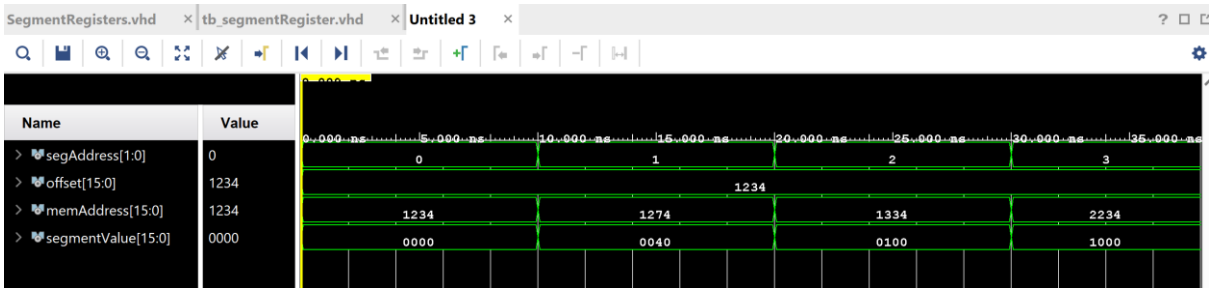


Figura 6.5 – Simulare SegmentRegisters

6.6 Unitatea Aritmetico-Logică (ALU)

Testbench-ul validează funcționalitatea modulului **ALU**, testând operațiile de **adunare**, **scădere**, **AND**, **OR** și **XOR** pe valori din registre (**rd1** și **rd2**), operanzi imediat (**imm**) și date din memorie (**memData**). Se verifică selectarea corectă a operandului secund în funcție de semnalul **operand2** și activarea semnalului **memDataWrite** pentru datele din memorie. Rezultatele obținute sunt monitorizate prin ieșirile **result**, **flags** și **wdMem**, confirmând funcționarea corectă a ALU pentru toate cazurile de testare.

Cazuri de Testare

1. **ADD (A + B):**
   - **Intrări:** rd1 = x"0005", rd2 = x"0003", aluOp = "000", operand2 = "00".
   - **Rezultat așteptat:** result = x"0008", steagurile setate corect.
2. **SUB (A - B):**
   - **Intrări:** rd1 = x"0005", rd2 = x"0003", aluOp = "001", dir = '0'.
   - **Rezultat așteptat:** result = x"0002", steagurile setate corect.
3. **AND (A AND B):**
   - **Intrări:** rd1 = x"0005", rd2 = x"0003", aluOp = "010".
   - **Rezultat așteptat:** result = x"0001" (5 AND 3 = 1).
4. **OR (A OR B):**
   - **Intrări:** rd1 = x"0005", rd2 = x"0003", aluOp = "011".
   - **Rezultat așteptat:** result = x"0007" (5 OR 3 = 7).
5. **XOR (A XOR B):**
   - **Intrări:** rd1 = x"0005", rd2 = x"0003", aluOp = "100".
   - **Rezultat așteptat:** result = x"0006" (5 XOR 3 = 6).
6. **ADD cu imm:**
   - **Intrări:** rd1 = x"0005", imm = x"0002", aluOp = "000", operand2 = "01".
   - **Rezultat așteptat:** result = x"0007".
7. **ADD cu memData:**
   - **Intrări:** rd1 = x"0005", memData = x"0004", aluOp = "000", operand2 = "11".
   - **Rezultat așteptat:** result = x"0009".

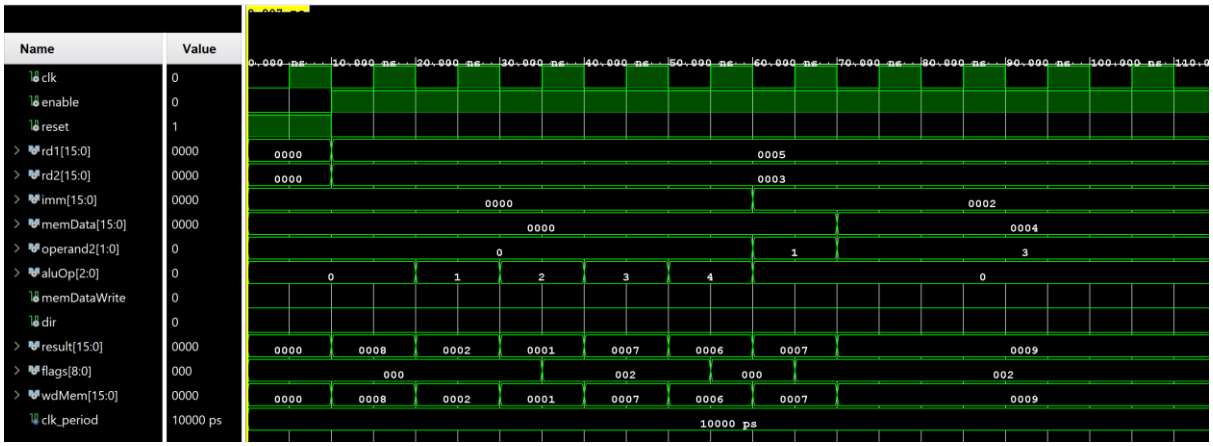


Figura 6.6 – Simulare ALU

6.7 Jump Control

Testbench-ul verifică modulul **JumpControl** prin testarea salturilor necondiționate și condiționate (**je, jne, jg**) în funcție de semnalul **flags**. Se monitorizează ieșirea **nextIp** și **resetQueue** pentru a confirma actualizarea corectă a adresei în funcție de condițiile de salt.

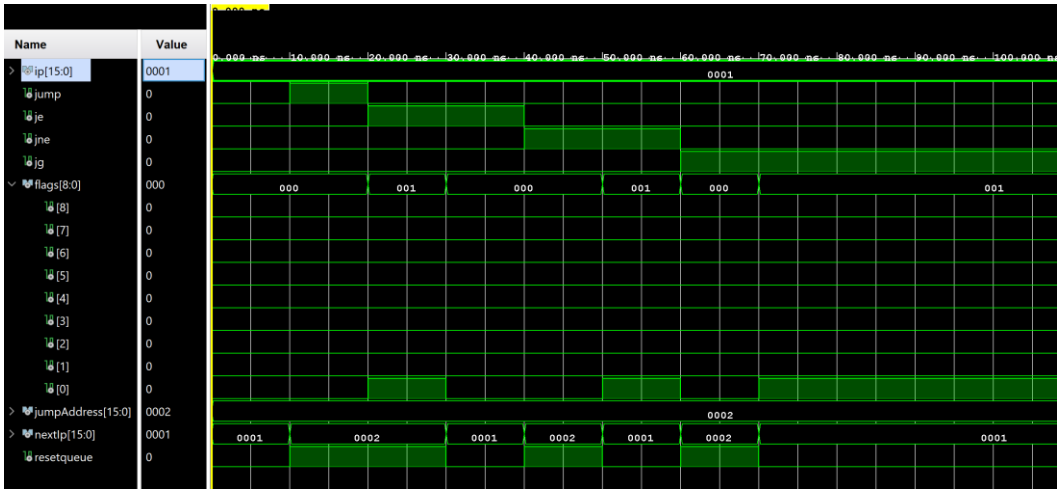


Figura 6.7 – Simulare Jump Control

6.8 Microprocesorul Pipeline complet

Testbench-ul pentru modulul topLevel simulează funcționarea unui procesor simplu bazat pe instrucțiuni pre-încărcate în memorie. Acesta testează execuția instrucțiunilor, actualizarea adresei IP, interacțiunea cu memoria, și controlul registrele și ALU.

Structura Testbench-ului

1. **Instanțierea Modulului topLevel:**
   - Modulul de nivel superior (topLevel) este instanțiat în testbench, conectând toate porturile acestuia la semnale locale definite în testbench.
2. **Generarea Semnalului de Ceas (clk):**
   - Un proces creează un semnal de ceas periodic cu o perioadă de 20 ns (10 ns pentru fiecare jumătate de ciclu).
3. **Procesul de Stimulare:**
   - Activează semnalul enable pentru a iniția execuția instrucțiilor.

- Instrucțiunile sunt deja în memorie, astfel execuția începe direct, controlată de logica internă a modulului topLevel.

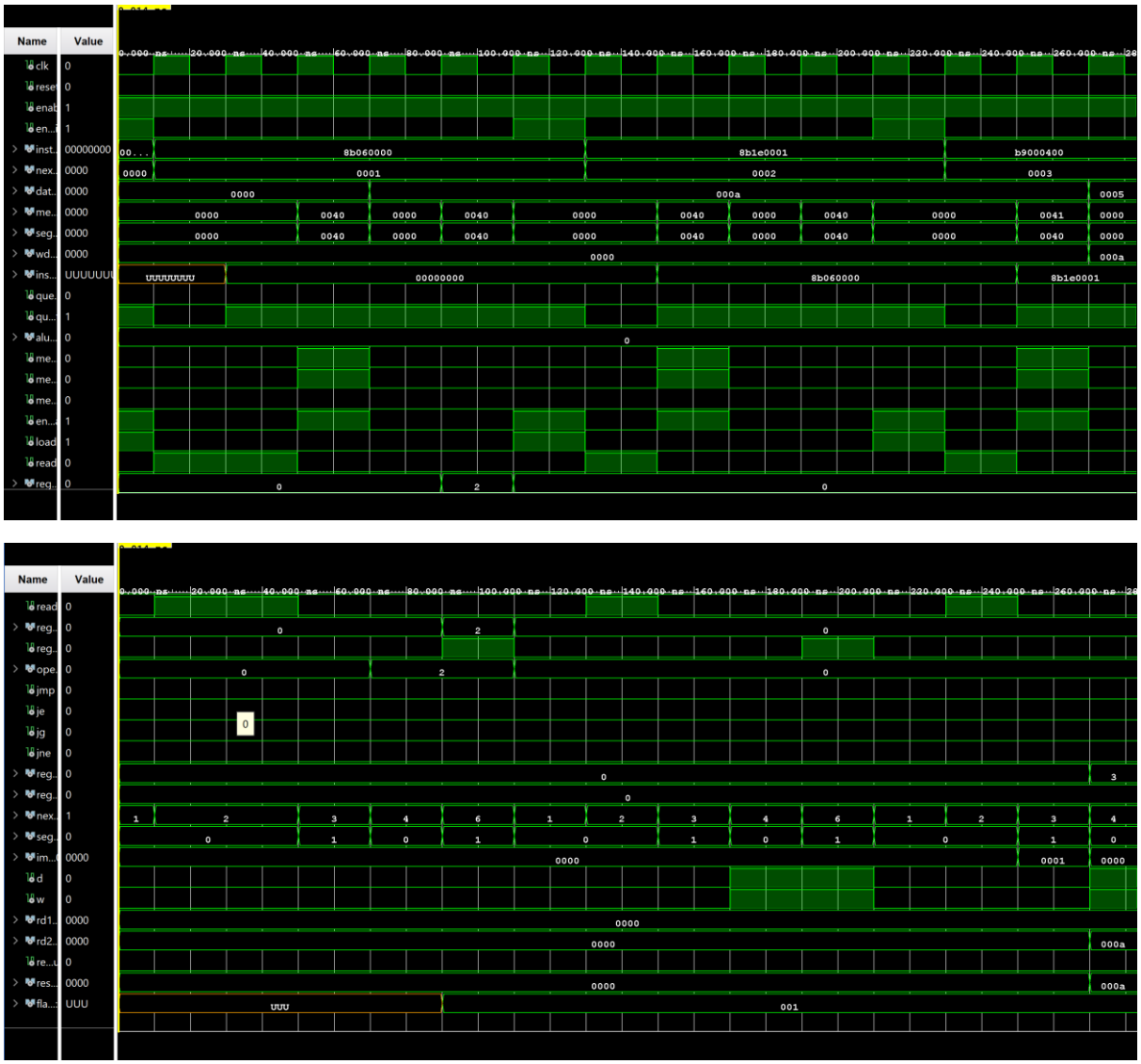


Figura 6.7 – Simulare Modul Top Level

Concluzii

Proiectul de implementare a procesorului Intel 8086 în VHDL reprezintă un pas important în înțelegerea arhitecturii și funcționării unui procesor clasic, având aplicații educaționale și practice. Prin utilizarea limbajului VHDL, a fost posibilă modelarea componentelor principale, precum Unitatea de Control (UC), ALU, registrele, și mecanismele de salt, oferind o replică digitală fidelă a procesorului 8086. Implementarea și simularea procesorului au permis validarea execuției instrucțiunilor x86 de bază, cum ar fi MOV, ADD, SUB, CMP, și salturile condiționate (JE, JNE, JG), confirmând corectitudinea designului. Integrarea cu un testbench dedicat a permis testarea sistematică a comportamentului procesorului, simulând un flux complet de instrucțiuni și validând interacțiunile dintre componente.

Proiectul poate fi extins prin implementarea unor caracteristici suplimentare, cum ar fi suportul pentru instrucțiuni mai complexe. În concluzie, această realizare nu doar că îmbunătățește înțelegerea arhitecturii Intel 8086, dar servește și ca punct de plecare pentru explorări mai avansate în domeniul designului hardware și simulării digitale.

Bibliografie

[1] Walter A. Triebel, Avtar Singh, „The 8088 and 8086 Microprocessors Programming, Interfacing, Hardware" 2014

[2] Barry B. Brey „The Intel Microprocessors Arhitecture, Programming, and Interfacing", 2009