Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare

Documentație

Proiectarea unei unități pentru detectarea si evitarea unor situații de hazard in arhitecturile pipeline

Structura Sistemelor de Calcul

Student: Maria-Magdalena Creț

Grupa 30223

Anul Universitar 2024-2025

Cuprins

1 Introducere

1.1 Context

Se dorește realizarea unei unități de detectare și evitare a hazardurilor pentru arhitectura pipeline MIPS (Microprocessor without Interlocked Pipeline Stages). Utilizarea pipelining-ului în proiectarea unui procesor constă și în apariția unor posibile condiții de hazard. Acestea apar din cauza dependențelor dintre instrucțiuni sau din întârzierile necesare pentru a obține datele corecte.

Apariția hazardurilor afectează arhitectura Pipeline în următoarele moduri:

- Scăderea performanței: Hazardurile pot forța pipeline-ul să introducă „no operations" (operații care nu fac nimic) pentru a aștepta ca datele să fie disponibile. Acest lucru poate reduce eficiența pipeline-ului, deoarece anumite cicluri de ceas sunt irosite.
- În cazul hazardurilor de control, dacă un branch este calculat greșit, toate instrucțiunile aflate deja în pipeline trebuie eliminate (flush), astfel că este necesar un timp suplimentar de execuție.
- Gestionarea hazardurilor necesită soluții suplimentare de proiectare, cum ar fi mecanisme de forwarding (pentru hazardurile de date) sau unități de predicție a salturilor (pentru hazardurile de control). Aceste soluții măresc complexitatea procesorului.

1.2 Obiective

Obiectivul principal a acestui proiect este acela de a se proiecta și implementa o unitate pentru detectarea si evitarea unor situații de hazard in arhitecturile pipeline. Se pornește de la arhitectura MIPS Pipeline studiată deja la Arhitectura Calculatoarelor. Acest procesor are rolul de a realiza un program unic, implementat de către student. Procesorul MIPS Pipeline este realizat pe 16 de biți. Se poate opta spre o implementare a MIPS-ului Pipeline pe 32 de biți.

Unitatea va fi proiectată utilizând limbajul VHDL și va fi integrată într-un proiect Xilinx Vivado care include deja un pipeline MIPS. Acest proiect conține un design MIPS existent, realizat la Arhitectura Calculatoarelor pe 16 de biți, iar unitatea de detectare a hazardurilor va fi adăugată la acest design pentru a îmbunătăți funcționalitatea procesorului. Instrucțiunile programului deja implementat la MIPS Pipeline se vor executa obișnuit, însă vor fi rezolvate hazardurile corespunzătoare.

Testarea și verificarea designului se realizează cu ajutorul introducerii unui testbench care permite simularea comportamentului unității și a pipeline-ului MIPS într-un mediu controlat.

2 Studiu bibliografic

2.1 Cum este realizată arhitectura MIPS Pipeline și cum au loc hazardurile în componența acesteia?

Arhitectura MIPS include etapele de pipeline (o instrucțiune respectă etapele de mai jos):

- `IF` – Extragerea instrucțiunii (Instruction Fetch);
- `ID/OF` – Decodificarea instrucțiunii / extragerea operanzilor (Instruction Decode / Operand Fetch);
- `UC` – Unitate de control (Unit Control);
- `EX` – Execuție (Execute);
- `MEM` – Memorie (Memory);
- `WB` – Scriere rezultat (Write-Back).

Un hazard apare atunci când o instrucțiune care se află în pipeline nu poate fi executată corect din cauza dependențelor sau a conflictelor cu alte instrucțiuni. În arhitectura de tip pipeline, hazardurile pot afecta performanța procesorului, cauzând întârzieri și chiar erori dacă nu sunt gestionate corespunzător.

Totuși de menționat este că, nu toate tipurile de hazarduri sunt la fel de importante, astfel că rezolvând anumite tipuri de hazarduri, se pot rezolva alte tipuri automat.

Tipurile de hazarduri sunt:

1. Hazardur structural: Apare atunci când două sau mai multe instrucțiuni din pipeline încearcă să acceseze aceeași resursă hardware în același timp, iar acea resursă nu poate fi utilizată simultan. De exemplu, dacă atât o instrucțiune de citire `load` cât și o instrucțiune de scriere `store` doresc să acceseze memoria într-un anumit ciclu de ceas și există o singură memorie partajată pentru date și instrucțiuni, va apărea un conflict.

   Exemplul 1: Structural Hazard

   ```
   lw $t0, 0($t1)          # Load word into $t0 from memory
   sw $t2, 4($t1)          # Store word from $t2 to memory
                           # at address $t1 + 4
   ```

2. Read-After-Write (hazard de date RAW): Apare atunci când o instrucțiune încearcă să citească o valoare dintr-un registru înainte ca o instrucțiune anterioară să termine de scris acea valoare. Acest tip de hazard este cauzat de dependențele dintre instrucțiuni, deoarece datele necesare de o instrucțiune ulterioară nu sunt încă disponibile.

   Exemplul 2: Read-After-Write Hazard

   ```
   add $t0, $t1, $t2       # Write to $t0
   sub $t3, $t0, $t4       # instruction already in pipeline
   ```

3. Load Data Hazard: Este un caz special de hazard de date care apare în timpul unei operații de încărcare din memorie. În acest caz, este necesar să se introducă întârzieri ("NoOp-uri") în pipeline pentru a permite terminarea încărcării datelor din memorie înainte ca instrucțiunea dependentă să încerce să le folosească.

Exemplul 3: Load Data Hazard

```
lw $t0, 0($t1)          # Load word from address in $t1
                        # into $t0
add $t2, $t0, $t3       # instruction already in pipeline
sub $t5, $t6, $t7       # instruction already in pipeline
```

4. Hazardurile de control (cunoscute și ca hazarduri de salt) apar în arhitectura procesorului, în special în pipeline, atunci când o instrucțiune de salt (sau branch) este executată. Salturile sunt instrucțiuni care pot schimba fluxul normal de execuție al programului, deoarece ele determină procesorul să sară la o altă adresă în memorie, în funcție de o anumită condiție (de exemplu, dacă o valoare este egală cu alta).

Exemplul 4: Control Hazard

```
beq $t0, $t1, pos     # if $t0 is equal to $t1, jump
                      # to pos
add $t2, $t3, $t4     # instruction already in pipeline
sub $t5, $t6, $t7     # instruction already in pipeline
```

În arhitectura de tip pipeline, rezolvarea hazardurilor este crucială pentru asigurarea funcționării corecte și eficiente a procesorului. Cu toate acestea, hazardurile structurale sunt, de obicei, mai puțin critice și mai ușor de gestionat în comparație cu hazardurile de date și de control. Pentru procesorul MIPS memoriile de date și instrucțiuni sunt separate, hazardurile structurale nu pot apărea.

2.2 Soluții în rezolvarea hazardurilor (studiate în anul precedent la Arhitectura Calculatoarelor)

Soluțiile care stau în rezolvarea hazardurilor se împart în funcție de tipul hazardurilor pe care dorim să le rezolvăm. Astfel că, pentru hazardul structural, o soluție suficientă ar fi aceea de a modifica scrierea în blocul de registre RF pe frontul descendent (în VHDL, se testează `falling_edge(clk)` sau `clk'event and clk = '0'`), astfel încât valoarea actualizată după scriere să poată fi citită asincron în partea a 2-a a perioadei de ceas, pentru a fi propagată spre etajele superioare. O altă metodă de rezolvarea a hazardului structural ar fi aceea de a adăuga un NoOp pe oricare dintre poziții pentru ca distanța să se extindă de la 3 la 4, se creează o întârziere astfel și se ajunge la ciclul corect de ceas pentru realizarea instrucțiunii respective. A doua soluție este mai costisitoare, astfel că se preferă schimbarea frontului de ceas.

Uneori un hazard structural poate fi considerat și un hazard de date, dar datorită utilizării simultane a blocului de registre, va fi considerat hazard structural. Hazardul de date apare fie la distanța 1 (d1), fie la distanța 2 (d2) și se adaugă NoOp-uri pentru rezolvarea lor. Uneori rezolvarea unor hazarduri de date, poate rezolva automat și hazardurile structurale, acolo unde ele există și este posibil acest lucru.

Apariția hazardului de control este indusă de prezența instrucțiunilor de salt, deoarece se întârzie modificarea registrului PC, până în etajul ID, pentru salturi necondiționate `jump`, respectiv până în etajul MEM, pentru salturi condiționate `branch`. La momentul saltului, prima instrucțiune, respectiv primele 3 instrucțiuni, care se succed celor de salt, vor fi de asemenea încărcate în pipeline, ceea ce poate să nu corespundă cu fluxul de execuție descris în cadrul programului. Soluția pentru rezolvarea hazardului de control constă în introducerea unui număr adecvat de NoOp, imediat după instrucțiunile de salt, încât execuția lor să asigure o întârziere suficientă, care nu afectează starea procesorului. Vor fi necesare: 1 instrucțiune NoOp după fiecare jump, respectiv 3 instrucțiuni NoOp după fiecare branch.

O optimizare în cazul unui salt necondiționat poate fi renunțarea la NoOp și interschimbarea instrucțiunii de jump cu cea situată înaintea sa, care oricum trebuie să se execute. Interschimbarea este posibilă doar dacă instrucțiunea anterioară îndeplinește simultan următoarele condiții:

1. Nu este de salt;
2. Nu se face salt la ea în cadrul programului;
3. Nu va genera hazard cu instrucțiunile la care se face salt

De remarcat este că, deși hazardurile structurale sunt o problemă de luat în considerare în proiectarea procesoarelor, ele nu sunt o provocare majoră în arhitectura MIPS clasică, datorită separării memoriilor și resurselor hardware bine segmentate în pipeline. Astfel că unitatea de detectare și evitare a unor situații de hazard se va proiecta doar pentru hazardurile de date (RAW și Load)și pentru hazardurile de control.

2.3 Rezolvarea hazardurilor de date

O metodă simplă, dar mai puțin eficientă este introducerea de stalls (bule) în pipeline, astfel încât instrucțiunile să aștepte până când datele necesare sunt disponibile.

Când se detectează că o instrucțiune necesită date care nu sunt încă disponibile, pipeline-ul este oprit temporar, iar instrucțiunile care urmează după instrucțiunea dependentă sunt întârziate. Apoi unitatea de detecție poate introduce o instrucțiune NoOp în pipeline pentru a se reuși finalizarea execuției instrucțiunii respective.

O variantă de rezolvare mai eficientă ar fi utilizarea unui buffer de registre (Register File Bypass), care permite accesul la rezultatele încă nescrise în fișierul de registre prin folosirea unui buffer între stadiile pipeline, în care rezultatele instrucțiunilor anterioare sunt stocate temporar. Un Register File Bypass este o unitate de redirecționare (forwarding unit) care să trimită rapid datele acolo unde sunt necesare imediat ce sunt disponibile. O astfel de unitate de redirecționare are nevoie de prezența unor semnale de control suplimentare care să permită redirecționarea datelor doar atunci când este nevoie.

Astfel că pentru implementarea acestei soluții, rezultatele din etapele `MEM` sau `WB` trebuie să fie stocate temporar în buffer, permițând accesul instrucțiunilor care au nevoie de ele înainte ca acestea să fie scrise în registre.

Se va opta spre implementarea cu Register File Bypass.

2.4 Rezolvarea hazardurilor de control

Hazardurile de control, care apar de obicei în cazul instrucțiunilor de salt (branch), sunt provocate de faptul că rezultatul condiției de salt nu este cunoscut până când instrucțiunea ajunge la stadiul de Execuție `EX` sau Memorie `MEM` în pipeline. Totodată, instrucțiunile adiționale intră în pipeline și există posibilitatea de a fi executate incorect dacă rezultatul saltului este adevărat, de unde este necesară operațiunea de flush. Procesul de flush aduce un cost de performanță, deoarece pipeline-ul trebuie să fie golit și apoi să fie încărcat din nou cu instrucțiunile corecte, ceea ce duce la întârzieri și la o utilizare ineficientă a resurselor procesorului.

Există mai multe metode de rezolvare a hazardurilor de control cum ar fi: Predicția dinamică a salturilor (Dynamic Branch Prediction). Aceasta se bazează pe estimarea direcției în care va continua programul, astfel încât procesorul poate decide să execute sau nu instrucțiunile care urmează după un salt, chiar înainte ca rezultatul condiției de salt să fie cunoscut. O altă variantă este Execuția Speculativă, metodă prin care procesorul continuă execuția instrucțiunilor care urmează după salt în mod speculativ, înainte să cunoască rezultatul saltului. Astel că, dacă saltul este confirmat, execuția va continua în mod normal. Altfel, predicția este greșită, urmând a se anula execuția speculativă ⇒golirea pipeline-ului (flush). În acest mod se poate îmbunătăți performanța procesorului, dar de remarcat este că în cazul unor predicții greșite, există penalizări.

O altă metodă ar fi proiectarea unui Buffer de Instrucțiuni pentru Hazarduri de Control.

Un Buffer de Instrucțiuni pentru Hazarduri de Control stochează temporar instrucțiuni care urmează imediat după o instrucțiune de tip salt (branch). Dacă predicția saltului este incorectă, aceste instrucțiuni pot fi anulate (flush), fără a afecta execuția corectă a programului. În cazul în care saltul este prezis corect, instrucțiunile sunt continuate fără întrerupere. Ca mecanism, se utilizează predicția dinamică, în care rezultatele anterioare ale salturilor sunt stocate într-un Branch Prediction Buffer (BPB). Dacă predicția spune că un salt va fi luat, se continuă execuția pe ramura respectivă. Dacă predicția este corectă, instrucțiunile stocate în buffer continuă să fie executate, permițând pipeline-ului să fie utilizat la capacitate maximă, fără blocaje. Dacă predicția este greșită, trebuie golit bufferul de respectivele instrucțiuni. În acest fel, instrucțiunile incorecte aflate deja în pipeline nu vor avea niciun efect asupra execuției corecte. ⇒ Se va utiliza această metodă, care se va îmbina cu bufferul de registre proiectat pentru hazardurile de date.

3 Analiza

3.1 Ce își propune proiectul?

La finalul implementării unității de predicție și rezolvare a hazardurilor, MIPS Pipeline realizat pe 16 de biți va avea următoarele caracteristici, fiind eficientizat:

a. Rezolvarea hazardurilor de tipul RAW și Load (hazarduri de date), indiferent de programul adăugat în Instruction Memory din IFetch;

b. Rezolvarea hazardurilor de control;

c. Realizarea unor programe de testare pentru rezolvarea hazardurilor;

d. Îmbunătățirea arhitecturii prin introducerea unor noi componente (etape).

3.2 Analiza proiectului

De menționat este că Mips 16 are 3 tipuri de instrucțiuni, menționate în imaginea atașată mai jos (R-type, I-type și J-type).

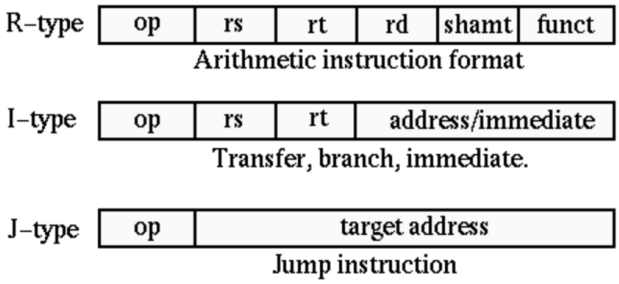


Figura 3.2.1.1: Formatul Instrucțiuniilor ISA 16

Conform standardului MIPS 16 ISA, câmpul opcode pe 3 biți are valoarea 0 pentru intrucțiuni de tip R (Register) și codifică unic instrucțiunile din celeltate 2 categorii I (Immediate), respectiv J (Jump). Instrucțiunile de tip R sunt codificate unic în câmpul function, pe 6 biți. S-au ales 10 instrucțiuni, care se pot observa în tabelul următor:

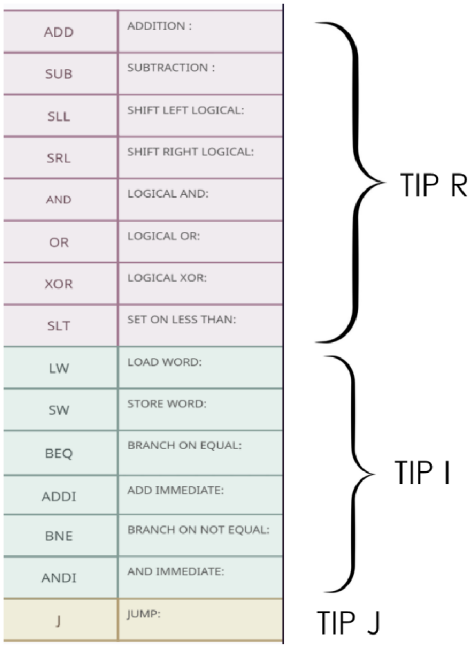


Figura 3.2.1.2: Instrucțiuni MIPS 16

3.2.1 Scenarii posibile pentru hazardul de date și soluții de rezolvare

Soluționarea hazardului de tip RAW (Read-After-Write) prin metoda *Register File Bypass* se bazează pe ideea de evitare a necesității de transfer direct între etape Pipeline-ului, folosind astfel o unitate de transfer. Valoarea necesară este redirecționată în mod intern, din fișierul de registre, pentru a permite utilizarea ei la momentul potrivit și totodată nu va mai fii necesară o întârziere *stall* în execuția instrucțiunilor.

Astfel că mai jos sunt remarcate etapele Pipeline-ului, cât și principiul ales **Register File**

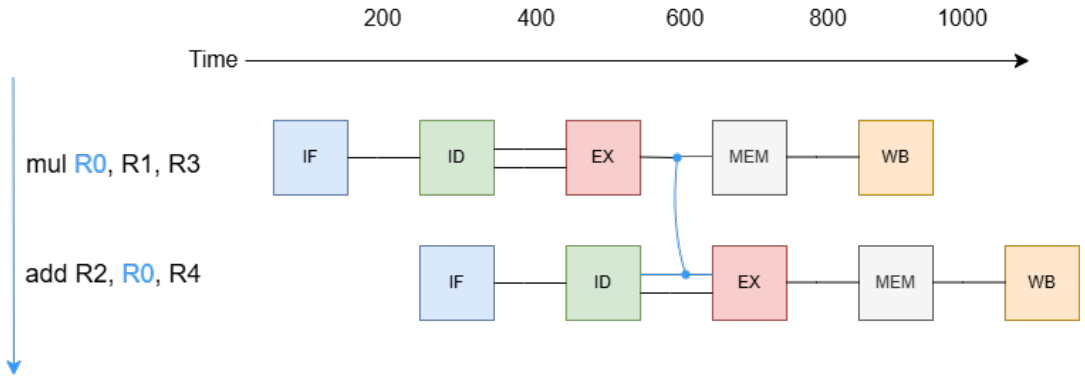
Bypass

Metoda **Register File Bypass** se asigură că fiecare instrucțiune poate citi ultima valoare scrisă în registre. Acest lucru este posibil chiar și în cazul în care valoarea este rezultatul instrucțiunii anterioare și încă nu a fost salvată în registrul final. În cazul acesta fiecare instrucțiune poate accesa valoarea corectă din registrul de ieșire al fișierului de registre.

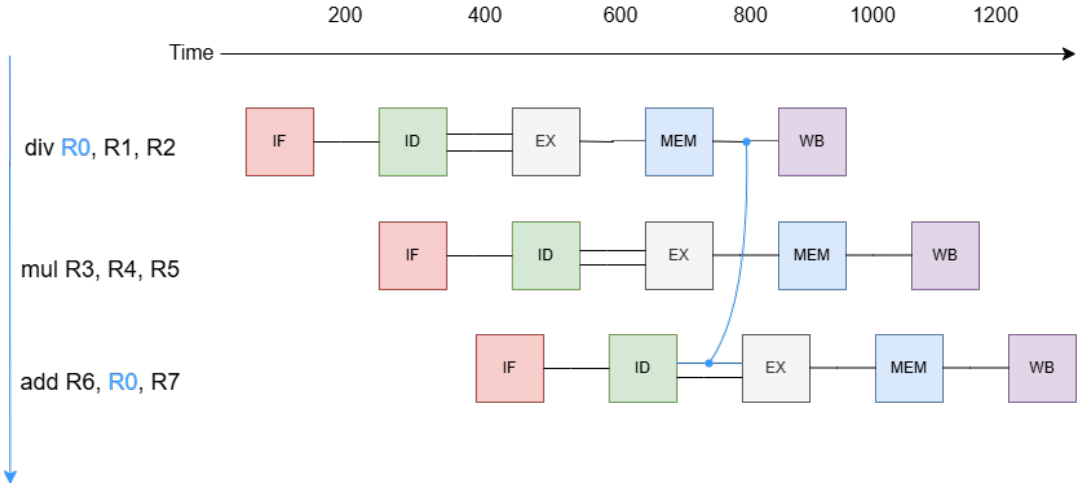
> Odată cu utilizarea Register File Bypass, majoritatea hazardurilor RAW pot fi eliminate fără a necesita transferuri suplimentare complexe între etapele pipeline-ului. Se vor utiliza așa-numitele registre temporare în care se vor salva valorile noi pentru a fi accesate. Unul dintre beneficii este evitarea utilizării întârzieriilor, care ar scădea eficiența.

Care sunt scenariile posibile în utilizarea metodei **Register File Bypass**?

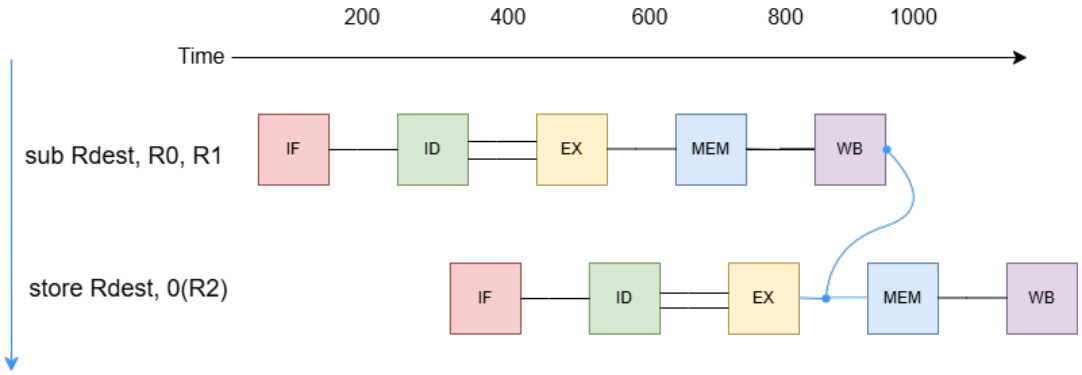
- Prima instrucțiune (I1) scrie într-un registru considerat registru destinație R_{dest}, iar a doua instrucțiune (I2) folosește R_{dest} ca operand:
  - Prima instrucțiune intră în etapa de execuție (Execution Unit) și se generează rezultatul.
  - A doua instrucțiune este deja în etapa ID (Instruction Decode) și are nevoie de valoarea lui R_{dest}.
  - I2 va citi direct valoarea din registrul considerat temporar de ieșire al fișierului de registre, unde este deja disponibilă valoarea calculată de I1. Acest lucru se întâmplă chiar dacă scrierea finală a lui I1 în R_{dest} nu s-a terminat încă. Nu mai este necesară adăugarea de stall-uri pentru citirea primei instrucțiuni.



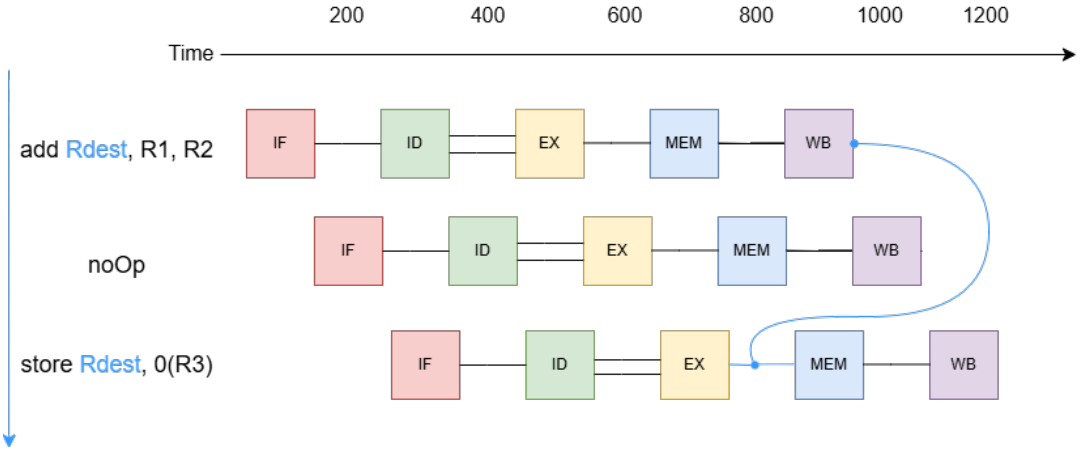
- Prima instrucțiune (I1) scrie într-un registru R_{dest}, urmată de o altă instrucțiune, iar a treia instrucțiune (I3) utilizează R_{dest} ca operand:
  - Instrucțiunea întâi duce la capăt etapa MEM, urmând scrierea valorii obținute în WB (Write-Back Unit - în memorie sau în registru).
  - Cea de-a doua instrucțiune va executa o operație independentă.
  - A treia instrucțiune se află în etapa ID (Instruction Decode) și are nevoie de valoarea lui R_{dest}. Se va folosi ultima valoare scrisă în Rdest de către I1.



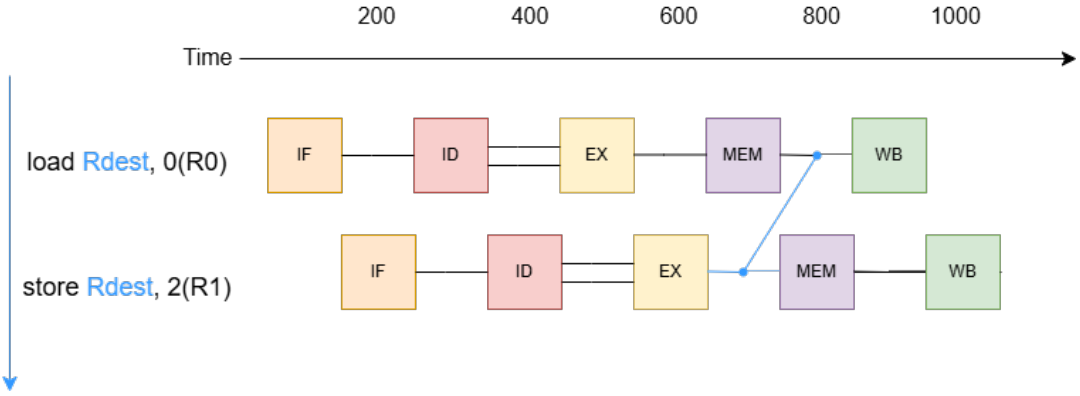
- Prima Instrucțiune (I1) scrie în R_{dest}, iar a doua Instrucțiune (I2) este o instrucțiune de store care introduce valoarea luiR_{dest} în memorie:
  - I1 este în etapa WB (scriere în registru) și urmează să scrie în R_{dest}.
  - I2 este în etapa MEM și are nevoie de valoarea lui R_{dest} pentru a o stoca în memorie. I2 o să poată să obțină valoarea direct din registrul temporar de ieșire a fișierului de registre prin bypass, astfel că nu va mai aștepta scrierea primei instrucțiuni în R_{dest}.



- Prima Instrucțiune (I1) scrie în R_{dest}, a doua instrucțiune este „sigură”, iar a treia este o instrucțiune store ce încearcă să stocheze valoarea dinR_{dest} în fișierul de registre:
  - I1 finalizează scrierea în registrul R_{dest}.
  - I3 ajunge în etapa MEM și are nevoie de valoarea din R_{dest}. Observație: Pentru că I1 a scris deja în R_{dest}, I3 poate folosi valoarea fără întârziere, citind-o direct prin bypass din fișierul de registre.



- Prima Instrucțiune este un load, iar următoarea este o instrucțiune store care stochează valoarea încărcată la o adresă diferită:
  - I1 se află în etapa MEM și încarcă o valoare în R_{dest}.
  - I2 are nevoie de valoarea din R_{dest} în MEM pentru a o stoca la adresa specificată. De remarcat că I1 este o operațiune de load, valoarea lui R_{dest} va fi disponibilă direct din registrul de ieșire al etapei MEM, iar I2 poate folosi această valoare fără întârziere.



3.2.2 Scenarii posibile pentru hazardul de control și soluții de rezolvare

Hazardurile de control apar în general din cauza instrucțiunilor de salt - *branch* sau de salt condiționat - *conditional branch*, deoarece acestea pot schimba fluxul de execuție în funcție de rezultate care nu sunt disponibile imediat.

Se remarcă următoarele scenarii:

- Instrucțiune de salt condiționat - executarea unei instrucțiuni de salt condiționat depinde de rezultatul evaluării unei condiții. Dacă valoarea condiției nu este încă calculată, procesorul nu știe care va fi următoarea instrucțiune și poate avea nevoie de un ciclu de așteptare.

Exemplu:

```
beq R0, R1, Instr ; Jump to 'Instr' if R0 == R1
add R2, R3, R4
mul R5, R6, R7
 Instr:
or R9, R10, R11
```

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7	Ciclu 8	Ciclu 9
BEQ	IF	ID	Stall	EX	MEM	WB			
ADD		IF	Stall	Stall	ID	EX	MEM	WB	
MUL			IF	Stall	Stall	ID	EX	MEM	WB
OR				IF	Stall	Stall	ID	EX	MEM

Tabela 1: Execuția pipeline pentru un salt condiționat

- În cazul unei instrucțiuni de salt direct, deși nu există o condiție de evaluat, procesorul trebuie totuși să determine adresa destinației. În timpul acestui proces, poate apărea o întârziere. Acest lucru se întâmplă în cazul salturilor necondiționate.

Exemplu:

```
MUL R1, R2, R3
JUMP R1
```

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7
JUMP	IF	ID	Stall	EX	MEM	WB	
Instr. următoare		IF	Stall	Stall	ID	EX	MEM

Tabela 2: Execuția pipeline pentru un salt necondiționat

- Cazul cu salt condiționat, dar care are loc în funcție de rezultatul unei operații efectuate de procesor. Atunci când saltul depinde de o valoare care este în calcul, decizia de salt poate întârzia din cauza faptului că rezultatul nu este disponibil.

Exemplu:

```
ADD R1, R2, R3
BEQ R1, R0, Instr
```

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7	Ciclu 8
BEQ	IF	ID	Stall	EX	MEM	WB		
Instr. depinde		IF	Stall	Stall	ID	EX	MEM	WB

Tabela 3: Execuția pipeline pentru un salt condiționat dependent de rezultat

Hazardurile de control vor fi rezolvate cu ajutorul unui Buffer de Instrucțiuni, care utilizează predicția dinamică a salturilor și un Branch Prediction Buffer - **BPB**. În această metodă, se folosește un buffer pentru a anticipa și a preîncărca instrucțiunile posibile în funcție de rezultatul predicției de ramificare. Prin utilizarea predicției dinamice prin bufferul de instrucțiuni BPB ⇒ se reduce numărul de întreruperi.

Predicția dinamică a ramificațiilor se bazează pe istoricul de execuție al instrucțiunilor de ramificare, în special asupra frecvenței cu care o ramificație a fost luată sau nu în trecut. Informațiile despre istoric sunt stocate într-un **BPB**, care reprezintă un tabel de predicție asociativ care memorează dacă ramificația a fost sau nu luată, respectiv adresa la care s-a ajuns dacă s-a întâmplat acest lucru.

BPB este verificat înainte ca o instrucțiune de ramificare să fie luată. Totodată, acest buffer de instrucțiuni anticipează dacă are sau nu loc saltul (ramificația). În caz afirmatic, se va prelua și adresa de salt. Altfel, se continuă execuția liniei de instrucțiuni secvențiale, accesând următoarea instrucțiune din adresă secvențială (PC + 1).

Bufferul de instrucțiuni joacă un rol esențial în anticiparea salturilor. Acesta permite încărcarea și stocarea instrucțiunilor posibile care ar putea fi executate după instrucțiunea de salt, indiferent de rezultatul predicției.

De remarcat este că instrucțiunile anticipate sunt stocate temporar în buffer până când se confirmă rezultatul saltului. Se găsesc 2 cazuri de funcționare:

1. Predicția corectă ⇒ instrucțiunile din buffer pot fi executate imediat și nu mai au loc pauze considerate semnificative.
2. Predicție greșită ⇒ bufferul poate anula instrucțiunile preîncărcate și începe preîncărcarea noilor instrucțiuni de la adresa corectă.

Etapele funcționării și rezolvării hazardurilor de control:

a) Consultarea BPB și decizia inițială: În timpul etapei IF, procesorul consultă BPB pentru instrucțiunea curentă. Dacă BPB indică faptul că saltul este probabil să fie efectuat, procesorul începe să preîncarce instrucțiunea de la adresa țintă prezisă în buffer. Dacă predicția sugerează că ramificația nu va fi luată, se continuă preluarea instrucțiunii următoare din secvența normală (PC + 1).

b) Un prim avantaj este minimizarea numărul de instrucțiuni care trebuie eliminate atunci când o predicție este greșită. Acest lucru se realizează datorită faptului că predicția are loc în etapa IF.

c) Verificarea predicției, respetiv actualizarea: În timpul etapei ID, procesorul determină dacă predicția a fost corectă comparând valoarea efectivă a condiției de salt. Dacă

predicția a fost greșită - *un exemplu este acela când se prezice un salt care nu se realizeaza*, instrucțiunile preîncărcate vor fi anulate din bufferul de instrucțiuni. În cazul în care ramificația a fost luată, se va incrementa predictorul din BPB la adresa specificată, iar adresa țintă este actualizată pentru a reflecta noua țintă corectă. Altfel, se decrementează, urmând ca adresa țintă să rămână neschimbată. De remarcat este faptul că predictorul de salt nu se va decrementa niciodată sub 0.

d) Cel de-al doilea avantaj este faptul ca instrucțiunile o să fie disponibile imediat după verificarea predicției și se reduce astfel numărul de pauze.

Folosind 2 biți în loc de unul singur, un salt care favorizează puternic direcția luată sau neluată va fi prezis greșit doar o singură dată. Cei 2 biți sunt folosiți pentru a codifica cele patru stări ale sistemului. Schema pe 2 biți este un exemplu general de predictor bazat pe contor, care se incrementează atunci când predicția este corectă și se decrementează altfel, folosind punctul de mijloc al intervalului său ca linie de separare între direcția luată și neluată.

Mai jos se poate observa automatul de stări al unui predictor dinamic realizat pe 2 biți:

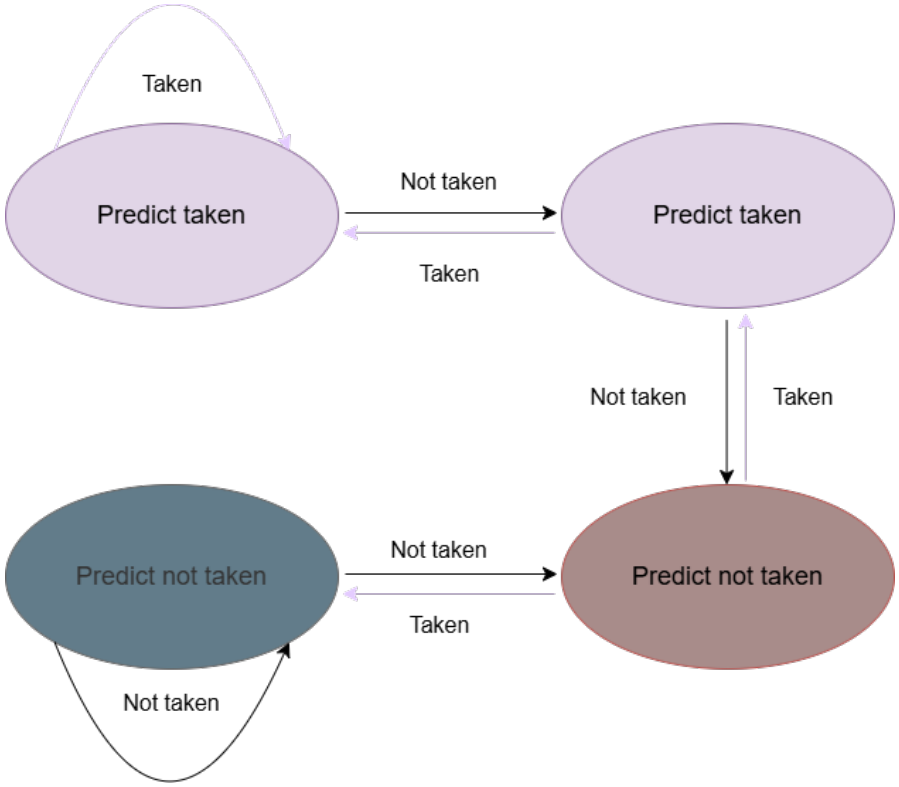


Figura 3.2.2.1: Starea unui 2-bit predictor

3.2.3 Îmbinarea soluțiilor de rezolvare a hazardurilor de date și control

Se va combina rezolvare hazardurilor de date cu rezolvarea hazardurilor de control: **Register File Bypass** și **Branch Prediction Buffer**. Prin Register File Bypass, datele sunt transmise direct din etapele intermediare (de ex., din etapa MEM sau WB) către instrucțiunile care au nevoie de ele, fără să aștepte ca datele să fie scrise complet în fișierul de registre. În cazul hazardurilor de control, predicția dinamică a salturilor cu ajutorul unui BPB ajută procesorul să ia o decizie anticipată despre următoarea instrucțiune,iar acest lucru se realizează chiar

înainte ca evaluarea condiției de salt să fie finalizată. În acest fel, procesorul poate sa continue execuția instrucțiunii fără să introducă întreruperi.

Prin utilizarea Register File Bypass împreună cu Branch Prediction Buffer, procesorul poate continua execuția instrucțiunilor în mod optim, reducând latența atât pentru hazardurile de date, cât și pentru cele de control. Acest lucru duce la rezolvarea atât a hazardurilor de date, cât și a celor de control.

Instrucțiunile care depind de datele generate de alte instrucțiuni aflate mai sus în pipeline pot beneficia de Register File Bypass. Această tehnică elimină pauzele și permite accesarea datelor corecte chiar și atunci când instrucțiunea anterioară nu și-a finalizat complet scrierea în registrul destinație R_{dest}. În timpul execuției condiției de salt în etapa ID, Register File Bypass poate oferi acces direct la valorile necesare pentru verificarea condiției de salt, chiar dacă ele provin de la o instrucțiune anterioară aflată în pipeline.

Odată ce instrucțiunea de ramificare ajunge în etapa de decodificare ID, procesorul poate verifica dacă predicția a fost corectă. Dacă predicția a fost greșită, procesorul elimină instrucțiunile preîncărcate și începe executarea de la adresa corectă, conform rezultatelor evaluării condiției de salt. BPB este actualizat pentru a reflecta noua decizie de ramificare.

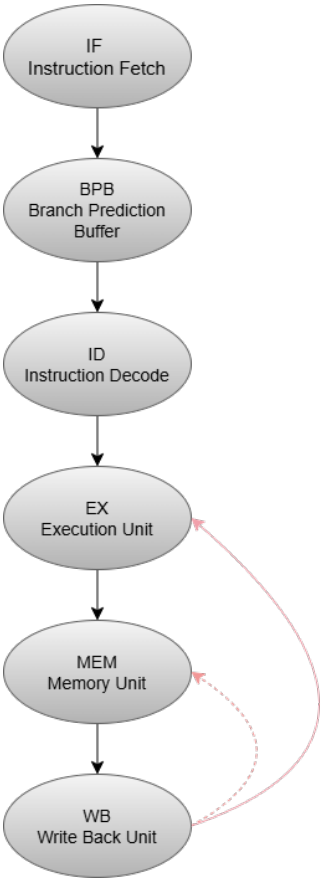


Figura 3.2.2.2: Etapele Pipeline actualizate cu Register File Bypass si BPB

4 Design

Notații Instrucțiuni:

- `r_ins $dest, $op1, $op2` - pentru instrucțiuni de tip R
- `i_ins $dest, $op1, imm` - pentru instrucțiuni de tip I
- `branch $op1, $op2, imm` - pentru instrucțiuni de tip ramificație (branch)
- `jmp imm` - pentru instrucțiuni de tip jump

Instrucțiune	Tip	Descriere RTL Abstractă
`add $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs] + RF[rt]`
`sub $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs] - RF[rt]`
`sll $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs]` $\ll$ `RF[rt]`
`srl $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs]` $\gg$ `RF[rt]`
`and $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs] & RF[rt]`
`or $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs]` $\|\|$ `RF[rt]`
`xor $rd, $rs, $rt`	R	`RF[rd]` $\leftarrow$ `RF[rs]` $\oplus$ `RF[rt]`
`addi $rt, $rs, imm`	I	`RF[rt]` $\leftarrow$ `RF[rs] + SExt(imm)`
`subi $rt, $rs, imm`	I	`RF[rt]` $\leftarrow$ `RF[rs] - SExt(imm)`
`lw $rt, $rs, imm`	I	`RF[rt]` $\leftarrow$ `Mem[RF[rs] + SExt(imm)]`
`sw $rt, $rs, imm`	I	`Mem[RF[rs] + SExt(imm)]` $\leftarrow$ `RF[rt]`
`beq $rt, $rs, imm`	I	`if RF[rs] == RF[rt] then PC` $\leftarrow$ `PC + 4 else PC` $\leftarrow$ `PC + 4 + SExt(imm)`
`bneq $rt, $rs, imm`	I	`if RF[rs]` $\neq$ `RF[rt] then PC` $\leftarrow$ `PC + 4 else PC` $\leftarrow$ `PC + 4 + SExt(imm)`
`jmp imm`	J	`PC` $\leftarrow$ `Ext(imm)`

Tabela 4: Instrucțiuni suportate de procesor

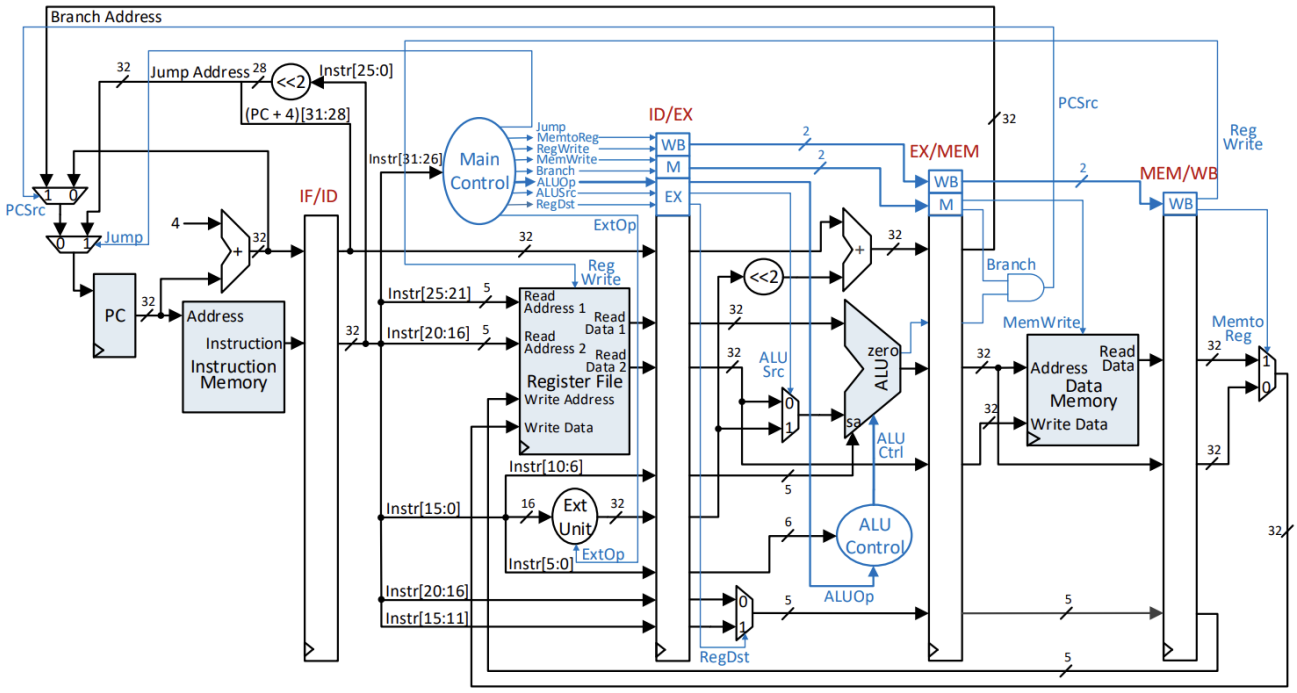


Figura 4.0.1: Mips Pipeline 16 biți înainte de rezolvarea hazardurilor

Schema procesorului Mips Pipeline de mai sus este realizată pe 32 de biți, pentru 16 biți există modificări precum PC + 1, în loc de PC + 4, respectiv numărul de biți pentru intrările și ieșirile în componente, care se reduce la jumătate.

4.1 Proiectarea Unității de Forwarding introducând Register File Bypass

Implementarea cu Register File Bypass și BPB are ca scop reducerea latenței și a penalităților de predicție greșită, îmbunătățind astfel performanța pipeline-ului.

Observație : Notații pentru semnale

- **Registrul Rs** se referă la adresa primului operand pentru instrucțiunile de tip R și I.
- **Registrul Rt** se referă la adresa celui de-al doilea operand al instrucțiunii de tip R și la adresa destinației instrucțiunii de tip I.
- **Semnalul RegWrite** este semnalul de control generat de unitatea de control care determină scrierea în registrul de date.

4.1.1 Register File Bypass către Etapa EX

Pentru a rezolva hazardurile de date din pipeline, **Register File Bypass** permite accesul direct la datele recent scrise în **Register File** fără a aștepta propagarea completă a datelor prin pipeline.

Pentru a rezolva primele două tipuri de hazarduri de date în pipeline-ul MIPS pe 16 de biți, se adaugă o unitate de *forwarding* în etapa de Execuție (EX). Această unitate va permite

redirecționarea datelor necesare pentru a evita întârzierile (stall-uri) atunci când o instrucțiune aflată în EX are nevoie de un rezultat produs recent de o instrucțiune anterioară aflată în etapele MEM sau WB.

- Datele redirecționate sunt extrase din registrele de pipeline EX/MEM sau MEM/WB și sunt trimise direct ca intrări pentru ALU. Aceasta permite operandului din registrul `Rs` (Intrare I1) și/sau `Rt` (Intrare I2) să primească valoarea corectă fără a aștepta ca instrucțiunea anterioară să finalizeze scrierea în registrul destinat.
- se utilizează multiplexoare la intrările ALU pentru a selecta sursa corectă de date. Unitatea de forwarding controlează aceste multiplexoare pentru a determina sursa corespunzătoare. De exemplu, dacă operandul `Rs` din instrucțiunea curentă necesită valoarea rezultată în MEM, multiplexorul va selecta valoarea din EX/MEM și o va trimite către Intrarea I1 a ALU.

Modificările hardware necesare includ:

1. **Multiplexoare** la intrările ALU, care sunt controlate de semnalele unității de forwarding pentru a asigura selectarea corectă a datelor redirecționate.
2. **Forwarding Unit** în etapa EX pentru a identifica sursa corectă a datelor necesare și a controla demultiplexoarele pentru Intrările I1 și I2 ale ALU.

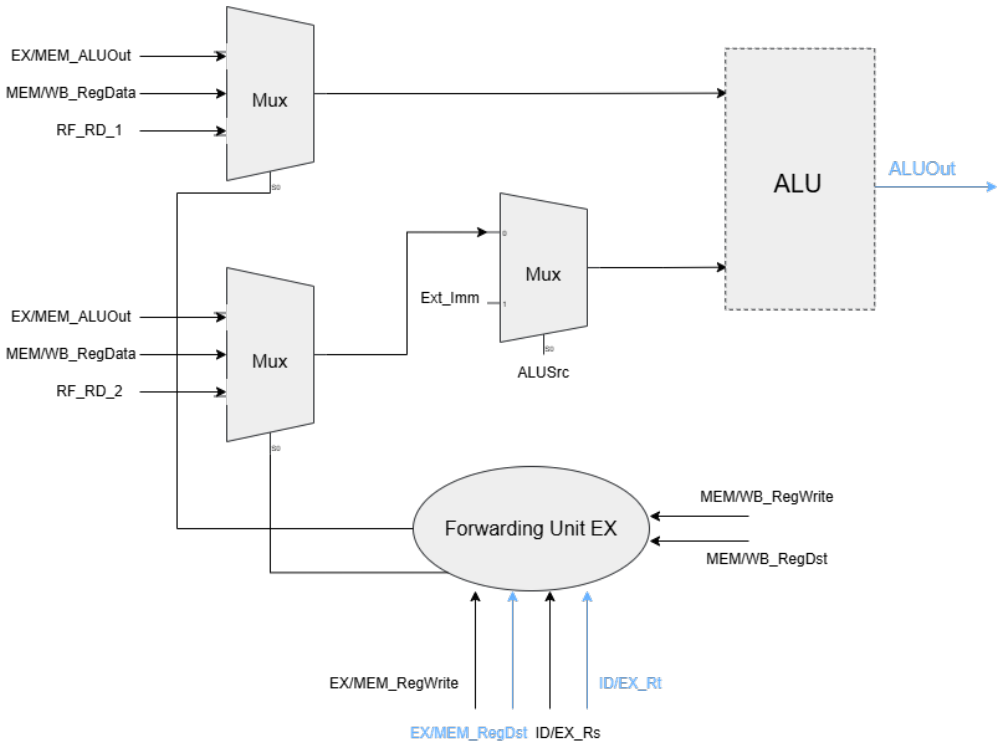


Figura 4.1.1.1: Actualizare Execution Unit

4.1.2 Forwarding în Etapa MEM

În această etapă, datele trebuie redirecționate către intrarea de date a memoriei de date. Pentru a realiza acest lucru, adăugăm o unitate de forwarding în etapa `MEM`, necesară pentru datele din instrucțiunea anterioară și cea de dinaintea acesteia.

Un caz special apare atunci când datele necesare nu mai sunt disponibile în pipeline, deoarece au fost deja scrise în Register File. O soluție de rezolvare este adăgarea unui un buffer pentru că informațiile despre adresa din Register File și datele asociate sunt menținute pentru un ciclu suplimentar. Acest buffer permite ca datele scrise anterior în Register File să fie accesibile la etapa `MEM`, astfel rezolvând hazardurile de date apărute atunci când instrucțiunile ulterioare au nevoie de aceleași date dar nu le pot obține imediat din pipeline.

Bufferul `WB_BUF` funcționează ca un *buffer de trecere* și este responsabil să asigure disponibilitatea datelor în etapa `MEM`, chiar și atunci când acestea au fost deja scrise în registrul de destinație, astfel că permite accesarea lipsită de întârzieri a instrucțiunilor aflate în execuție.

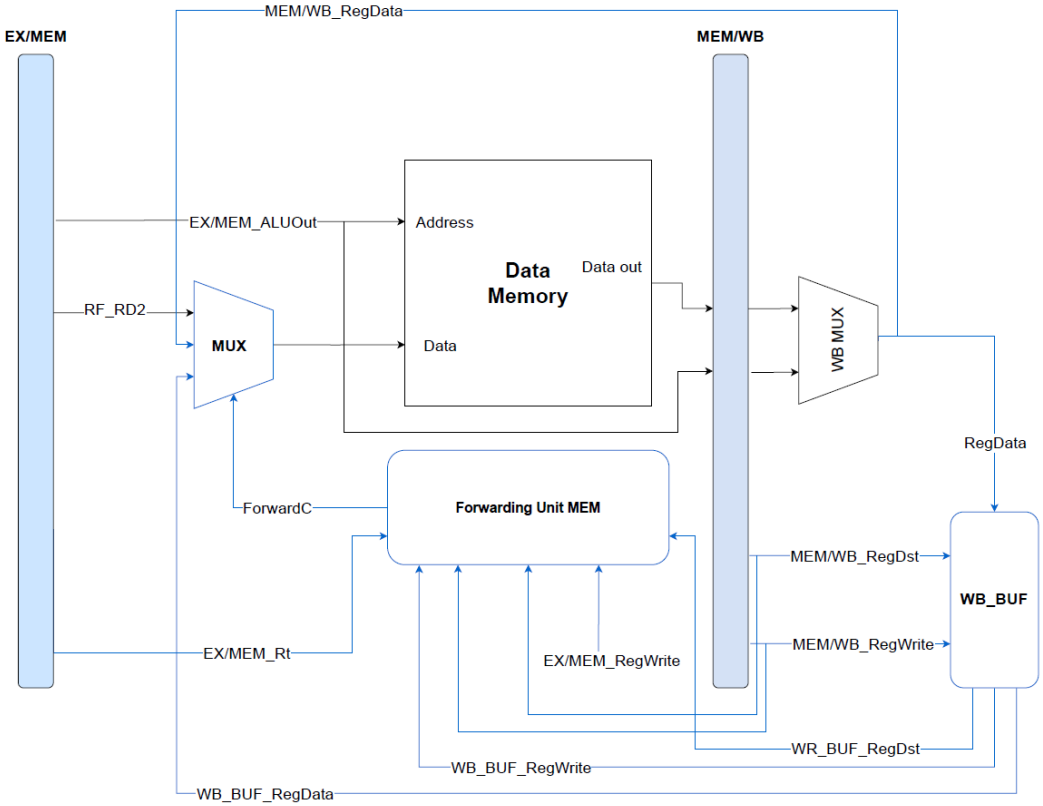


Figura 4.1.2.1: Actualizare Memory Unit

4.2 Integrarea Branch Prediction Buffer pentru Hazarduri de Control

Se utilizează predicția dinamiă pentru realizarea și funcționarea BPB. Predicția dinamică se bazează pe analiza comportamentului anterior al ramificațiilor (salturilor) pentru a anticipa cu mai multă precizie dacă o ramificație va fi luată sau nu în execuțiile viitoare. Este necesar un registru suplimentar care să păstreze istoricul predicțiilor pentru fiecare adresă de instrucțiune.

Un circuit de comparație verifică dacă predicția BPB corespunde cu condiția efectivă de salt după ce aceasta este evaluată în etapa EX sau MEM. Dacă predicția este incorectă, pipeline-ul este resetat pentru a reîncărca instrucțiunile corecte. BPB utilizează o tehnică de predicție dinamică bazată pe un contor de 2 biți (saturating counter) pentru a urmări comportamentul unui salt. Acesta memorează predicțiile de „luat” sau „neluat” pentru salturile, dar se

utilizează și pentru predicțiile ulterioare (se pot considera presupuneri despre ramificații). BPB este o structură de memorie care stochează istoricul ramificațiilor anterioare, precum și adresele țintă ale acestora.

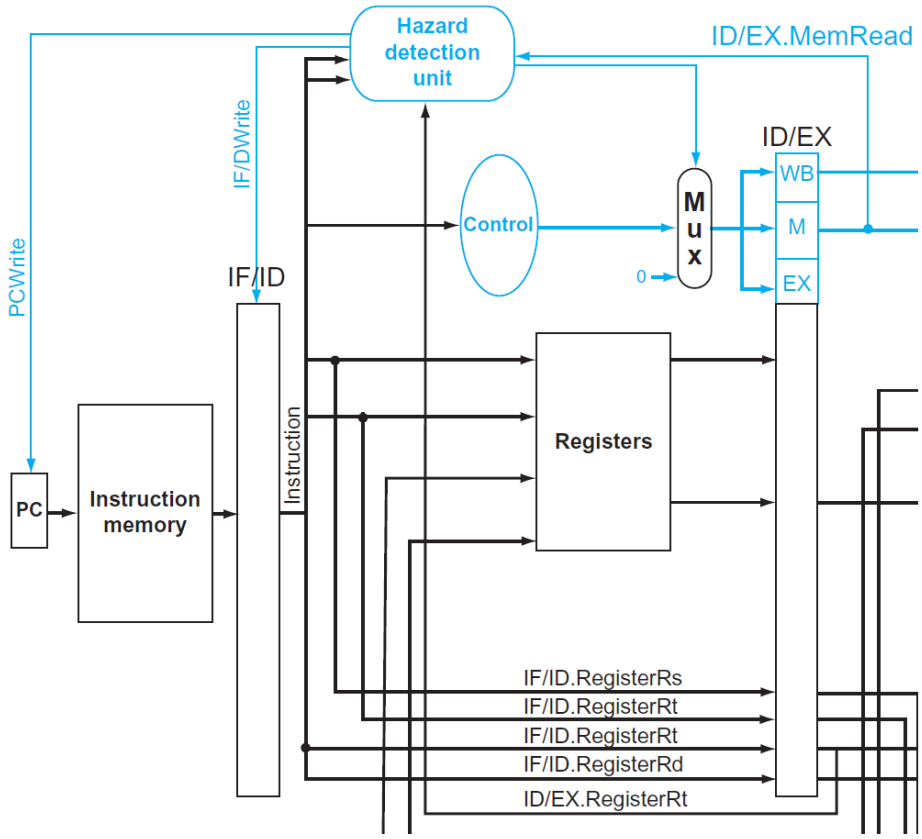


Figura 4.2.1: Actualizare Mips Pipeline cu Branch Prediction

Pe lângă BPB, este necesare și adăugarea unor componente hardware suplimenatare pentru ca să se poată realiza atât predicția dinamică, cât și rezolvarea hazardurilor de control. Acestea sunt:

- **BTB (Branch Target Buffer):** folosit pentru a stoca adresele de salt ale ramificațiilor. Atunci când procesorul întâlnește un salt, acesta verifică BTB pentru a vedea dacă există o adresă de salt asociată cu saltul respectiv.
- **Unitatea de Forwarding:** este implementată pentru rezolvarea hazardurilor de date cu Register File Bypass și va lucra în paralel cu BTB și BPB, pentru a se asigura că datele și ramificațiile sunt corect gestionate. Nu este necesare o altă implementare a ei.

  Se cunoaște faptul că, hazardurile de date și de control se pot suprapune. Dacă o instrucțiune de salt se consideră că va fi luată și procesorul începe să execute instrucțiuni pe baza acestei predicții, iar instrucțiunile anterioare nu au terminat încă, forwarding unit trebuie să asigure că datele care sunt necesare vor fi accesibile, fără a se produce conflicte de date: *ex.: accesarea unui registru în timp ce se află încă în proces de scriere.*
- **Unitatea de „Flush”:** resetează pipeline-ul atunci când predicția saltului corespunzător este incorectă. Dacă predicția din BPB este incorectă, pipeline trebuie golit (se realizează operațiunea numită - flush). Apoi se încarcă instrucțiunile considerate corecte.

Modul în care se execută predicția dinamică:

1. Etapa IF: Procesorul verifică BPB și BTB pentru a obține predicțiile și adresele țintă ale salturilor. Dacă predicția sugerează că ramificația va fi luată, procesorul începe să *fetch-uiască* instrucțiuni din adresa de salt.
2. Etapa ID: Dacă predicția este incorectă (ex.: ramificația nu este luată, dar a fost prezisă ca fiind luată), procesorul se ocupă de operațiunea de flush explicată mai sus.
3. Etapa EX: Procesorul execută instrucțiunea curentă și folosește unitatea de forwarding pentru a redirecționa datele corecte către ALU. În acest fel se evită apariția hazardurilor de date.

4.3 Hazard Detection Unit (HDU)

Unitatea de detecție a hazardurilor are rolul de a detecta hazardurile de tip `load`, adică acele situații în care o instrucțiune `load` care aduce date din memorie înregistrează datele într-un registru, iar următoarea instrucțiune le folosește imediat.

Această unitate verifică dacă instrucțiunea load din etapa MEM are ca destinație același registru ca instrucțiunea aflată în etapa ID. Dacă este așa, introduce un ciclu de întârziere (stall) în pipeline. Modul în care se va integra componenta hardware HDU se poate observa în Figura 4.1.2 (Actualizare Mips Pipeline cu Branch Prediction). Fluxul de control al HDU este următorul:

- HDU verifică dacă instrucțiunea aflată în etapa ID depinde de instrucțiunea `load` aflată în pipeline.
- **Introducerea întârzierii (stall)**:
  - Instrucțiunea din `IF/ID` este blocată temporar pentru un ciclu.
  - Instrucțiunea din `ID` își păstrează valorile semnalelor de control pe 0, prevenind modificări în registre.
- **Menținerea valorilor**:
  - HDU blochează avansul contorului de program (PC) pentru a permite retragerea unei instrucțiuni din `IF`.
  - `IF/ID` menține instrucțiunea curentă până când hazardul este rezolvat.
- De remarcat este că, după un ciclu de întârziere, pipeline-ul este deblocat, iar execuția continuă.

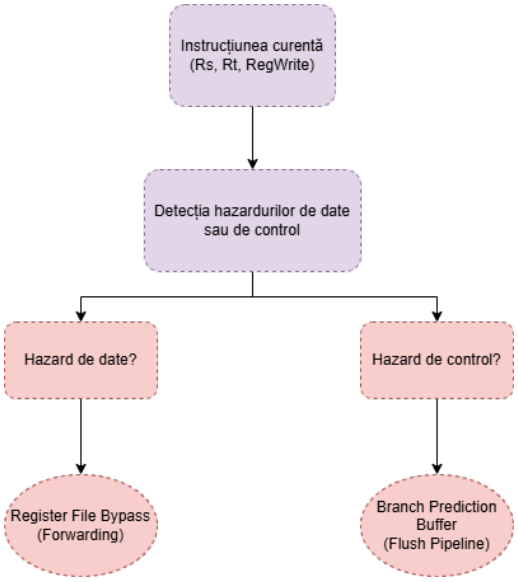


Figura 4.3.1: Principiul de funcționare după introducerea componentei HDU

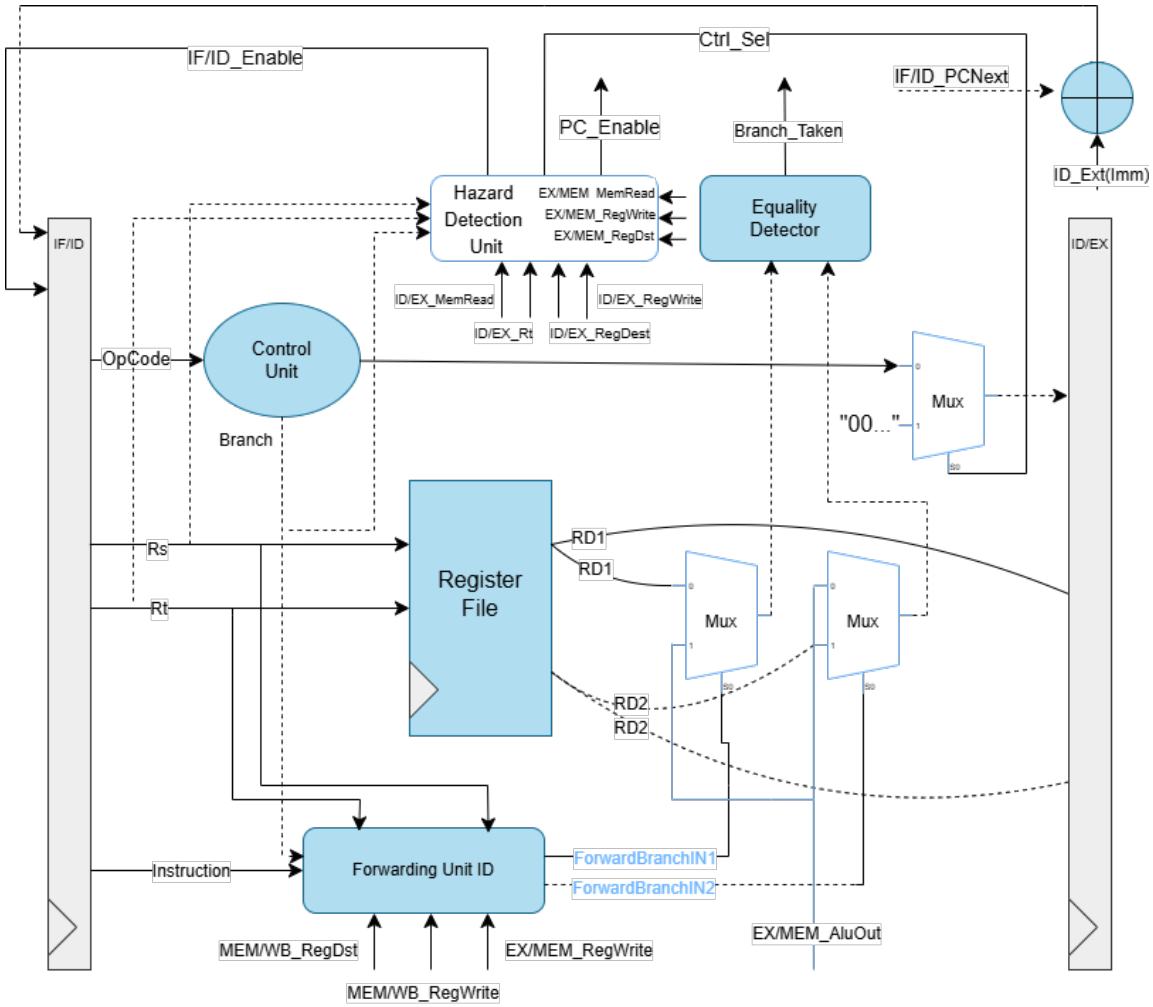


Figura 4.3.2: Actualizare Mips Pipeline cu HDU

5 Implementare

Se realizează modificări asupra implementării unui Mips Pipeline pe 32 de biți care nu ține cont de prezența și rezolvarea hazardurilor structurale, de date și de control. De remarcat este faptul ca Mips Pipeline este implementat ținând cont de 5 etape: `IF`, `ID/OF`, `UC`, `EX`, `MEM`, `WB`.

Se vor actualiza aceste etape prin adăugarea unor noi componente pentru rezolvarea hazardurilor, care sunt menționate mai sus:

- O unitate de Forwarding introducând Register File Bypass (se implementează Register File Bypass) - se va urmări implementarea prezentată în *Design*;
- Branch Prediction Buffer (implementarea predicției dinamice a salturilor) - necesită adăugarea unui BTB și a unei unități de *flush*, respectiv reutilizarea unității de forwarding menționată mai sus;
- Hazard Detection Unit.

5.1 Forwarding Unit

Forwarding Unit detectează dacă o valoare necesară pentru o instrucțiune curentă (în etapa de `EX`) a fost deja calculată în etapele anterioare și permite utilizarea directă a acestei valori fără a aștepta finalizarea scrierii în registru.

5.1.1 Instruction Fetch Forwarding Unit

Într-un procesor cu pipeline, etapa de fetch (preluarea instrucțiunilor din memorie) poate fi afectată de hazarduri, mai ales în cazul salturilor (branches) sau în cazul unei instrucțiuni care depinde de un rezultat încă necalculat.
Forwarding-ul în această etapă permite procesorului să redirecționeze instrucțiunile corecte înainte ca rezultatele anterioare să fie complet procesate.

Implementare Forwarding Unit

```
forward_A <= '0';
forward_B <= '0';

if (ID_Branch = '1') and (EX_MEM_RegWrite = '1') and
    EX_MEM_RegDst = ID_Rs then
    forward_A <= '1';
end if;

if (ID_Branch = '1') and (EX_MEM_RegWrite = '1') and
    EX_MEM_RegDst = ID_Rt then
    forward_B <= '1';
end if;
```

5.1.2 Execution Forwarding Unit

Pentru o unitate de execuție corespunzătoare implementării unui MIPS Pipeline extins, care include mecanisme de forwarding, trebuie să se realizeze câteva modificări. Acestea trebuie să includă posibilitatea de bypassing pentru valorile `ALURes` recente, folosind semnale de forwarding primite dintr-o unitate de forwarding. Modificările constau în:

- Semnale adiționale pentru forwarding: Se introduc două semnale `ForwardA` și `ForwardB` pentru a selecta corect sursele ALU, în funcție de bypass. Aceste semnale sunt generate de o Forwarding Unit externă.
- Multiplexoare suplimentare pentru intrările ALU:

  Acestea permit selectarea între valorile din registre (`RD1`, `RD2`), valorile din stadiile `MEM/WB`, sau extinderea imediată.
- Integrarea mecanismului de bypass în logica existentă: Valorile sunt selectate corect în funcție de hazardurile detectate.

Execution Unit: Adăugare Semnale Adiționale

```
if MEM_WB_RegWrite = '1' then
    if MEM_WB_RegDst = ID_EX_Rs then forward_A <= "
       10";
    elsif MEM_WB_RegDst = ID_EX_Rt then forward_B <=
        "10";
    end if;
end if;

if EX_MEM_RegWrite = '1' then
    if EX_MEM_RegDst = ID_EX_Rs then forward_A <= "
       01";
    elsif EX_MEM_RegDst = ID_EX_Rt then forward_B <=
        "01";
    end if;
end if;
```

5.1.3 Memory Forwarding Unit

Evită latențele cauzate de dependențele de date între instrucțiuni consecutive prin redirecționarea (forwarding) valorilor calculate către alte etape ale pipeline-ului înainte ca acestea să fie scrise în memorie sau registre.
Unit-ul identifică dependențele între instrucțiuni, cum ar fi cazul în care o instrucțiune depinde de rezultatul unei instrucțiuni anterioare aflate încă în pipeline.
În loc să aștepte finalizarea scrierii într-un registru sau memorie, datele necesare sunt preluate direct din unitatea de execuție sau din memoria intermediară a procesorului.
Prin acest mecanism, se minimizează necesitatea de a opri pipeline-ul (stall) și se îmbunătățește performanța.

Execution Unit: Adăugare Semnale Adiționale

```
ForwardC <= "00";
if MEM_WB_RegWrite = '1' then
    if EX_MEM_Rt = MEM_WB_RegDst and EX_MEM_MemWrite
        = '1' then forwardC <= "01";
    end if;
end if;
if WB_BUF_RegWrite = '1' then
    if EX_MEM_Rt = WB_BUF_RegDst and EX_MEM_MemWrite
        = '1' then forwardC <= "10";
    end if;
end if;
```

5.2 Branch Prediction Table

Pentru a implementa o actualizare a pipeline-ului MIPS cu Branch Prediction se dezvoltă un model hardware care să includă:

- **Branch Prediction Buffer (BPB)** pentru predicția dinamică a salturilor.
- O **unitate de control** care gestionează instrucțiunile ramură și poate detecta hazardurile de control.
- Adăugarea unui **mux** și logica aferentă pentru selectarea PC-ului pe baza rezultatului predicției.

Planul Implementării în VHDL

Pentru implementarea acestui pipeline cu Branch Prediction, se utilizează modulele hardware explicate mai jos:

- **Hazard Detection Unit (HDU)** – Detectează hazardurile cauzate de conflictele de date.
- **Branch Prediction Buffer (BPB)** – Realizează predicția direcției ramurii.
- **Control Logic** – Selectează valoarea PC-ului pe baza predicției sau recalculării.
- **MUX și registre pipeline** – Conectează etapele IF, ID, și EX.

5.2.1 Hazard Detection Unit (HDU)

Implementarea pentru HDU va fi detaliată în secțiunea următoare și poate fi reutilizată pentru detectarea conflictelor `load-use`.

5.2.2 Branch Prediction Buffer (BPB)

Se utilizează un buffer cu două stări pentru predicția direcției ramurii. Acest buffer va stoca informații despre fiecare instrucțiune de ramură și va permite predicții pe baza istoricului salturilor anterioare.
Pentru ramificație luată (inc_predictor = '1'): incrementează contorul daca nu e maxim.
Pentru ramificație neluată (inc_predictor = '0'): decrementează contorul daca nu e minim.

Implementare VHDL pentru BPB

Implementare HDU

```
if rising_edge(clk) and pc_enable = '1' then
    if instr_branch = '1' then
        -- branch luat
        if inc_predictor = '1' then
            if curr_predictor < "11" then curr_predictor :=
               curr_predictor + 1;
            end if;
        -- branch neluat
            else
            if curr_predictor > "00" then curr_predictor :=
               curr_predictor - 1;
            end if;
        end if;
    end if;
```

5.2.3 Logic Control Unit

Logica de control într-un Branch Prediction Buffer (BPB) este responsabilă pentru următoarele operațiuni:

- **Selecția corectă a predicției:** BPB folosește un mecanism de predicție bazat pe istoricul ramurilor. De obicei, acest mecanism se bazează pe o tehnică de tipul *2-bit saturating counter* pentru a decide dacă o ramură va fi luată sau nu. Logica de control se asigură că, pe baza semnalelor de intrare (instrucțiuni de ramură și istoricul acestora), BPB face predicția corectă a direcției ramurii (de exemplu, ramura luată sau nu).
- **Actualizarea predicției în urma execuției ramurii:** După ce instrucțiunea de ramură a fost executată și rezultatul este disponibil (în etapa EX), logica de control se asigură că predicția din BPB este actualizată conform rezultatului real al ramurii. Aceasta poate implica actualizarea valorii din 2-bit counter pe baza rezultatului efectiv (ramură luată sau nu).
- **Gestionarea și corectarea fluxului execuțional al pipeline-ului:** Logica de control se asigură că fluxul de execuție din pipeline este corect, în special atunci când predicția

a fost incorectă. În cazul unei predicții greșite, pipeline-ul este "curățat" prin semnalul de *flush*, iar instrucțiunile incorecte sunt eliminate pentru a preveni erori suplimentare.

Arhitectura MIPS și Adresele PC

În arhitectura MIPS:

- Adresa PC este reprezentată pe 16 biți.
- Instrucțiunile MIPS sunt aliniate pe 16 biți (2 octeți), astfel încât adresele lor sunt multipli de 4: 0x0000, 0x0004, 0x0008....
- Ultimii 2 biți ai adresei PC (PC[1:0]) sunt întotdeauna 00 datorită alinierii la 4 octeți:
- PC[1:0] nu sunt relevanți pentru indexare.
- PC[15:2] conțin adresa efectivă pentru instrucțiuni.

De ce selectăm biți specifici?

Pentru a indexa o structură hardware precum un Branch Prediction Buffer (BPB), trebuie să:

- Selectăm un subset de biți din PC pentru a genera un index.
- Numărul de biți necesari depinde de dimensiunea bufferului:
  - Dacă BPB are 16 intrări, avem nevoie de 4 biți pentru index ($2^4 = 16$).
  - Dacă BPB are 32 intrări, avem nevoie de 5 biți pentru index ($2^5 = 32$).
  - Dacă BPB are 64 intrări, avem nevoie de 6 biți pentru index ($2^6 = 64$).

Cum alegem biții din PC?

Pentru a face selecția biților:

- Ignorăm PC[1:0] (deoarece sunt mereu 00).
- Începem să folosim biții începând de la PC[2] în sus. Apoi selectăm câți biți avem nevoie în funcție de dimensiunea bufferului.

Exemplu: (în acest fel s-au ales biții pentru index în implementarea BP)

Buffer de 16 intrări (4 biți pentru index):

- Selectăm PC[5:2] (4 biți de la bitul 2 până la bitul 5).

6 Testare

Etapa de Testare și Validare implică rularea simulărilor pentru nivelul de vârf al procesorului MIPS pentru toate hazardurile menționate înb capitolele anterioare. Structura generală a

unui test include programul de intrare, diagrama *pipeline*-ului cu dependențele, rezultatele așteptate (cu și fără rezolvarea hazardurilor) și rezultatele efective.

Conținutul fișierului de registre și al memoriei de date este prezentat în listarea de mai jos.

6.1 Conținutul Fișierului de Registre și al Memoriei de Date

Exemplu pentru Register File:

```
signal reg_fileSignal: reg_file := (
                x"0000",
                x"0000",
                x"0001",
                x"0000",
                x"0003",
                x"0002",
                x"0002",
                x"ABCD",
                others => x"1111");
```

Exemplu pentru salvarea programului în memorie, tot în Hexa:

```
type RAM is array (0 to 63) of std_logic_vector(31 downto 0);
signal ram_memory : RAM := (
            x"0000",
                x"0A03",
                x"0001",
                x"0000",
                x"0005",
                x"0002",
others => X"0000");
```

6.2 Testarea Hazardurilor de Date

6.2.1 Exemplul 1: Forwarding to EX Stage (Hazard de Date)

Instrucțiuni

```
B "0000_0001_0010_0100", -- 0: mul $4 = $1 * $2
B "0010_0100_0101_0000", -- 1: addi $5 = $4 + 4
B "0000_0101_0110_0111", -- 2: sub $7 = $5 - $6
others => x"0000"
```

Diagramă Pipeline

Rezultate Așteptate

Fără transmitere (forwarding):

- CC_5: $4 = 0x12$

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7	Ciclu 8	Ciclu 9
MUL	IF	ID	EX	MEM	WB				
ADDI		IF	ID	EX	MEM	WB			
SUB			IF	ID	EX	MEM	WB		

Tabela 5: Pipeline pentru forwarding la etapa EX

- CC_7: $5 = 0x16$ (întârziat)
- CC_9: $7 = 0x10$

Cu transmitere (forwarding):

- CC_3: ForwardA = 0b01
- CC_4: ForwardB = 0b10
- CC_5: $4 = 0x12$
- CC_6: $5 = 0x16$
- CC_7: $7 = 0x10$

6.2.2 Exemplul 3: Load Data Hazard

Instrucțiuni

```
B "10000_0001_0110_0010", -- 0: lw $6, 2($1)
B "00000_0110_0011_0111", -- 1: add $7 = $6 + $3
B "00100_0111_1000_0101", -- 2: addi $8 = $7 + 5
others => x"0000"
```

Diagramă Pipeline

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7	Ciclu 8	Ciclu 9
LW	IF	ID	EX	MEM	WB				
ADD		IF	Stall	ID	EX	MEM	WB		
ADDI				IF	ID	EX	MEM	WB	

Tabela 6: Pipeline pentru hazard de tip load data

Rezultate Așteptate

Fără transmitere (forwarding):

- CC_5: $6 = 0x2$
- CC_6: $7 = 0x5$
- CC_7: $8 = 0xA$

Cu transmitere (forwarding):

- CC_4: Stall = 1 (pentru sincronizare)
- CC_6: $6 = 0x2$
- CC_6: $7 = 0x5$
- CC_7: $8 = 0xA$

6.2.3 Exemplul 4: Control Hazard

Instrucțiuni

```
B "00001_0001_0010_1100", -- 0: beq $1, $2, -4
B "00000_0011_0100_0101", -- 1: and $5 = $3 & $4
B "00000_0101_0110_0111", -- 2: or $7 = $5 | $6
others => x"0000"
```

Diagramă Pipeline

Instrucțiune	Ciclu 1	Ciclu 2	Ciclu 3	Ciclu 4	Ciclu 5	Ciclu 6	Ciclu 7	Ciclu 8	Ciclu 9
BEQ	IF	ID	Stall	EX	MEM	WB			
AND		IF	Stall	Stall	ID	EX	MEM	WB	
OR			IF	Stall	Stall	ID	EX	MEM	WB

Tabela 7: Pipeline pentru hazard de control

Rezultate Așteptate

Fără mecanisme de predicție:

- CC_3: Se detectează ramificarea.
- CC_4: Instrucțiunile se corectează.

Cu predicție de ramificare:

- CC_2: Predicția corectă evită întârzierea.

Observație: Opcode-urile s-ar putea să difere în implementare pentru că aceasta s-a mai modificat ulterior. Atunci când dorim să testăm acest caz spre ex trebuie să ne uităm în fișerele `Control Component`, respectiv `Execution Unit` pentru instrucțiuniile de tip R.

6.2.4 Program care îmbină toate tipurile de hazarduri (exemplu)

Cerința:

Să se înlocuiască toate elementele dintr-un sir:

- Dacă un element este mai mic decât X, se va împărți la 2.
- Dacă un element este între X și Y (inclusiv), se va dubla.
- Dacă un element este mai mare decât Y, se va înlocui cu 1.

Șirul se află în memorie începând cu adresa A ($A \geq 4$) și are N elemente. A, N, X, Y se citesc din memorie de la adresele 0, 1, 2, respectiv 3.

Constrângere: $X \leq Y$

Pentru verificare, se poate adăuga o bucla de citire a elementelor șirului, la final.

Implementarea în limbaj Assembly este următoarea:

```
 Rezolvare in instructiuni assembly pentru MIPS
# initializare registrii utilizati
xor $1, $0, 0           # initializeaza $1 (A, adresa de inceput
    a sirului)
xor $2, $0, 0           # initializeaza $2 (N, numarul de
   elemente)
xor $3, $0, 0           # initializeaza $3 (X, limita inferioara
   )
xor $4, $0, 0           # initializeaza $4 (Y, limita superioara
   )
xor $10, $0, 0         # initializeaza $10 (contor pentru bucla)

# Citire A, N, X, Y din memorie
lw $1, 0($1)             # A = mem[0]
lw $2, 1($2)             # N = mem[1]
lw $3, 2($3)             # X = mem[2]
lw $4, 3($4)             # Y = mem[3]

loop:
# se verifica daca s-au parcurs toate elementele
addi $10, $10, 1         # $10 = i++
beq $10, $2, done        # Daca $10 == n se iese din bucla, n -
   nr de elememente

 # Citire element din sir (adresa curenta este $1)
lw $5, 0($1)             # $5 este elem curent sir

# Verifica daca $5 < X
slt $6, $5, $3           # $6 = 1 daca $5 < X, altfel 0
beq $6, $zero, verif     # instr slt s-a modific cu sub si srl
   peste tot
# Daca $5 > = X, se va trece la verificarea intervalului
# Daca $5 < X, imparte la 2
srl $5, $5, 1      # impartire la 2 realizata cu shiftarea la
   dreapta cu 1 bit
```

```
sw $5, 0($1)        # Salveaza rezultatul inapoi in memorie
j loop              # Revine la urmatorul element

verif:
# Verifica daca X <= $5 <= Y
slt $6, $3, $5        # $6 = 1 daca X <= $5, altfel 0
slt $7, $5, $4        # $7 = 1 daca $5 <= Y, altfel 0
and $8, $6, $7        # $8 = 1 daca X <= $5 <= Y, altfel 0
beq $8, $zero, greater_than_Y # Daca nu este in interval, trece
   la verificarea > Y
# Daca X <= $5 <= Y, se va dubla elementul curent
sll $5, $5, 1       # inmultire realizat prin shiftare la stanga
   cu 1 bit
 sw $5, 0($1)       # Salveaza rezultatul inapoi in memorie
j loop              # Revine la urmatorul element

mai_mare_ca_Y:
# Daca $5 > Y, seteaza-l la 1
li $5, 1                # $5 = 1, s-a inlocuit in cod cu addi
sw $5, 0($1)            # se salveaza rezultatul inapoi in
   memorie
j loop                  # se revine la elem urmator

-- final:
```

Instrucțiuni

```
    B"000_001_000_000_0_111"  -- xor $1, $0, $0
    B"000_010_000_000_0_111"  -- xor $2, $0, $0
    B"000_011_000_000_0_111"  -- xor $3, $0, $0
    B"000_100_000_000_0_111"  -- xor $4, $0, $0
    B"000_101_000_000_0_111"  -- xor $10, $0, $0
    B"010_001_000_0000000"   -- lw $1, 0($1)
    B"010_010_001_0000001"   -- lw $2, 1($2)
    B"010_011_010_0000010"   -- lw $3, 2($3)
    B"010_100_011_0000011"   -- lw $4, 3($4) -- loop: de aici in jos
    B"001_010_010_0000001"   -- addi $10, $10, 1
    B"100_010_101_1111010"   -- beq $10, $2, FINAL
    B"010_101_010_0000000"   -- lw $5, 0($1)
    B"000_101_011_110_0_010"  -- sub $6, $5, $3
    B"000_110_110_110_0_101"  -- and $6, $6, $6
    B"100_110_110_1111100"   -- beq $6, $zero, verif
    B"000_101_101_000_0_100" -- srl $5, $5, 1
    B"011_101_010_0000000"   -- sw $5, 0($1)
    B"111_0000000000010"     -- j loop
    -- OBS: adress target se numara de la inceput pana unde incepe loop-ul
```

```
B"000_011_011_110_0_010"  -- sub $6, $3, $5
B"000_110_110_110_00100"  -- srl $6, $6, $6
B"000_101_100_111_0_010"  -- sub $7, $5, $4
B"000_110_111_111_0_100"   -- srl $7, $7, $7
B"001_110_110_111_0_101"  -- and $8, $6, $7
B"100_111_110_1111000"   -- beq $8, $zero, mai mare ca Y
B"000_101_101_000_0_011" -- sll $5, $5, 1
B"011_101_010_0000000"   -- sw $5, 0($1)
B"111_0000000000010"    -- j loop
B"001_000_010_0000001"  --  addi $5, $0, 1
B"011_101_010_0000000"   -- sw $5, 0($1)
B"111_0000000000010"     -- j loop
B"111_0000000000000"
```

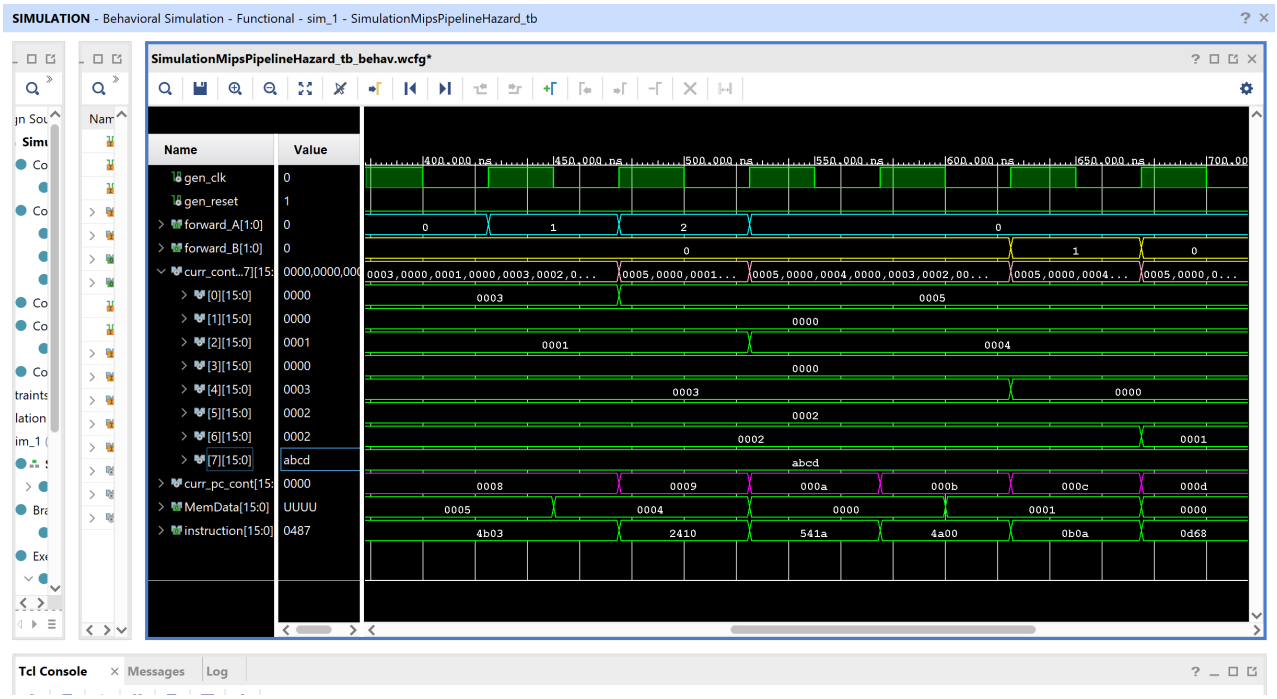


Figura 6.2.5.1: Rezultatele simulării în Vivado asupra programului de mai sus

7 Concluzii

Pentru început, pe cât de simplu ar părea rezolvarea hazardurilor în Pipeline, pe atât de complicată este odată cu avansarea în analiză, respetiv implementare și proiectare. Fiecare optimizare poate introduce noi complexități sau probleme neprevăzute care trebuie gestionate cu atenție. Cu cât se explorează mai mult complexitățile sistemului, cu atât devine mai evidentă creșterea complexității. Totuși, obiectivul de a obține o soluție bine optimizată și eficientă a fost realizat. Atât preocupările legate de performanță, cât și cele de fiabilitate au fost rezolvate eficient, permițând sistemului să funcționeze fără intervenții constante. Proiectul poate fi testat cu o plăcuță FPGA (exemplu fiind o plăcuță Basys3, deoarece Mips-ul implementat este pe 16 de biți.

Optimizări posibile

Rezolvarea Hazardurilor prin Tehnica Tomasulo

Hazarduri de date

Tomasulo rezolvă hazardurile de date prin utilizarea unui mecanism similar forwarding-ului, dar într-un mod mai complex și mai automatizat. Instrucțiunile care sunt gata să își scrie rezultatele într-un registru sau într-o locație de memorie pot transmite aceste rezultate altor instrucțiuni care așteaptă aceste date, fără a aștepta finalizarea întregii etape de execuție.

Tomasulo utilizează stații de rezervare care permit instrucțiunilor să aștepte pentru datele de care au nevoie, chiar înainte ca aceste date să fie disponibile din punct de vedere hardware. Aceste stații de rezervare permit ca instrucțiunile să fie emise și să aștepte fără a bloca pipeline-ul, ceea ce ajută la eliminarea hazardurilor de date.

Fiecare instrucțiune este emisă într-o stație de rezervare și începe execuția atunci când toate datele necesare devin disponibile. Aceasta înseamnă că hazardurile de date (cum ar fi dependențele de registre) sunt gestionate eficient, fără a fi nevoie de pauze (stalls).

Hazarduri de control

Deși Tomasulo nu se concentrează în mod direct pe hazardurile de control (precum predicțiile ramurilor), implementarea unui sistem de execuție speculativă poate ajuta indirect. Tomasulo poate continua să proceseze instrucțiuni care nu depind de rezultatul unui salt.

Hazarduri structurale

Tomasulo rezolvă hazardurile structurale prin alocarea dinamică a resurselor hardware, în funcție de necesitățile instruirii și de disponibilitatea acestora. Astfel, în loc să existe blocaje din cauza resurselor limitate (de exemplu, unități aritmetico-logice sau registre), Tomasulo poate distribui eficient resursele pentru a maximiza utilizarea acestora.

Bibliografie

[1] Documentație proprie realizată la Arhitectura Calculatoarelor pentru implementarea proiectului compus din Mips32 și Mips32 Pipeline, anul universitar 2023-2024.

[2] D. A. Patterson și J. L. Hennessy, *Computer Organization and Design: The Hardware/-Software Interface*, 5th ed. Morgan Kaufmann, 2014.

[3] C. Vancea *Arhitectura Calculatoarelor - suport curs și laborator*. Disponibil la: `https://users.utcluj.ro/~vcristian/AC.html` (accesat: 12 octombrie 2024).

[4] `https://users.utcluj.ro/~vcristian/Indrumator_laborator_2024_Draft.pdf#page=41` (accesat: 2 noiembrie 2024).

[5] `https://www.cse.iitd.ac.in/~rijurekha/col216/edition5.pdf` (accesat: 15 noiembrie 2024).