ARHITECTURA MICROPROCESOARELOR DLX

1. Scopul lucrarii

Lucrarea de fata prezinta arhitectura procesorului didactic virtual DLX, setul de instructiuni si modurile de adresare ale acestuia.

2. Memento teoretic

DLX-ul este un microprocesor didactic care a fost conceput pe baza unor teste executate pe o serie de microprocesoare comerciale cu o filosofie asemanatoare cu cea a DLX-ului.(de ex. MIPS, SPARC, etc.)

Setul de instructiuni al acestui microprocesor ca si al precursorilor lui se bazeaza pe o serie de consideratii generale privitoare la caracteristicile microprocesoarelor moderne actuale RISC cum ar fi:

- registrii de uz general cu o arhitectura LOAD-STORE
- suport pentru modurile de adresare :
    - indexat (cu un offset de adresa de 12 sau 16 biti)
    - imediat (operandul este o constanta pe 12 sau 16 biti)
    - registru (operandul se afla într-un regsitru general)
- suport pentru instructiuni simple de tip: load, store, add, substract, move registru, and, shift, compare equal, comapre not equal, branch (cu o adresa relativa la PC de dimensiune cel putin 8 biti), jump, call, return.
- suport pentru tipuri si date pe 8, 16, 32 pentru întregi si 64 biti conforme cu standardul IEEE 754 pentru virgula flotanta.

- în functie de optiune suport pentru codificare fixa a instructiunilor în caz ca se opteaza pentru performanta si codificare variabila daca se opteaza pentru cod redus.
- punerea la dispozitie a cel putin 16 registri de uz general (în general se folosesc pe 32 de biti) pe lînga o serie de registri separati pentru operatii în virgula flotanta precum si asigurarea ca toate modurile de adresare se pot aplica tuturor tipurilor de transfer de date
- urmarirea obtinerii unui set de instructiuni minimal.

În concordanta cu aceste criterii-cerinte, DLX-ul le adopta în felul urmator:

- set de instructiuni simplu de tip LOAD-STORE
- proiectat pentru o structura PIPELINE eficienta incluzând o codificare fixa a setului de instructiuni
- eficienta în rularea compilatoarelor

DLX ofera un model arhitectural bun pentru studiu nu doar datorita recentei popularitati al acestui tip de masina dar si datorita arhitecturii simple si inteligibile.

3. Desfasurarea lucrarii

Registrii microprocesorului DLX

DLX-ul are un set de 32 de registri generali (GPR) pe 32 de biti denumiti R0, R1,…R31. Aditional mai exista si un set de registri de virgula flotanta (FPR) care pot fi folositi registri pe 32 de biti în simpla precizie sau ca perechi (par-impar) pentru valori în dubla precizie. Registrii pentru virgula flotanta pe 64 de biti sunt denumiti F0, F1…F30. Sunt admise operatiile pe 32 si 64 de biti. Valoarea lui R0 este întotdeauna 0 (cablat la masa – caracteristica de altfel

comuna tuturor microprocesoarelor RISC). Exista câtiva registri speciali care pot fi convertiti în si din registrii de întregi (*Exemplu*: registrul de stare în virgula mobila folosit la pastrarea rezultatelor în virgula mobila). Exista de asemenea instructiuni pentru transferul între FPR si GPR.

Tipuri de date

Datele sunt pe 8 biti, 16 biti (semicuvânt) si pe 32 de biti (cuvânt) pentru operanzi întregi si 32 de biti simpla precizie si 64 de biti dubla precizie pentru operanzi în virgula mobila. Suportul pentru semicuvinte au fost adaugat pentru ca ele se regasesc în limbaje ca C si în sisteme de operare datorita faptului ca aici se urmareste o dimensionare cât mai economica a structurilor. Operanzi în virgula flotanta simpla precizie au fost adaugati din aceleasi ratiuni. Operatiile lui DLX se pot efectua pe operanzi întregi de 32 de biti si pe operanzi de 32 si 64 de biti în virgula flotanta. Operanzii pe octet sau semicuvânt sunt încarcati în registri urmati fiind de umplerea cu zerouri sau cu bitul de semn a locatiilor libere ramase din cei 32 de biti ai registrului. Odata încarcati acesti operanzi se vor comporta ca un operand pe 32 de biti.

Moduri de adresare la DLX

Singurile moduri de adresare suportate de catre DLX sunt cele imediat si respectiv indexat, ambele cu deplasamentul pe 16 biti. Modul de adresare direct registru se poate emula prin folosirea modului imediat cu valoarea zero în câmpul de deplasament pe 16 biti, iar modul de adresare absolut (direct) se poate emula prin folosirea modului indexat folosind registrul R0 ca registru de index. Astfel se pot folosi toate cele patru moduri de adresare majore desi fizic sunt suportate doar doua.

Memoria lui DLX este adresabila cu o adresa de 32 de biti. Fiind o arhitectura de tip LOAD-STORE toate accesele la / de la memorie se fac prin astfel de transferuri. Suportând tipurile de date descrise anterior, accesele la memorie care implica registri generali GPR se pot face pe octet, semicuvânt si cuvânt. Registrii pentru operanzii în virgula flotanta pot fi cititi sau scrisi din / în memorie folosind cuvinte în simpla si dubla precizie. Toate accesele la memorie trebuie aliniate.

Formatul instructiunii la DLX

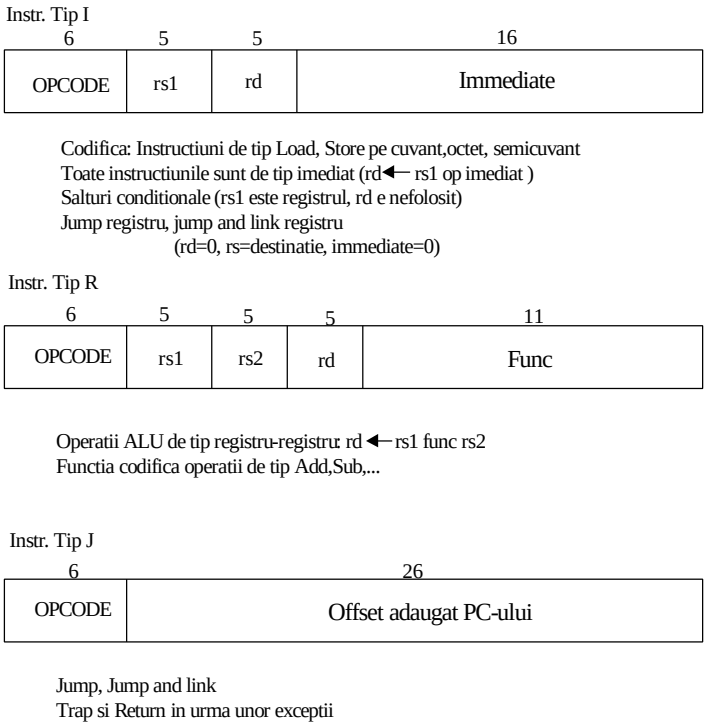

Figura 1. Formatul instructiunii

Formatul instructiunii este proiectat în asa fel încât sa asigure o decodificare optima si o functionare rapida a structurii pipeline. Setul de instructiuni este ortogonal pe 32 de biti cu un OPCODE primar de 6 biti. Acest format permite un

deplasament de 16 biti folosit ca index, constanta imediata sau adresa relativa (la PC) de salt.

Instructiunile DLX

DLX-ul suporta o lista de instructiuni simple mentionate anterior si în plus alte câteva. Exista patru clase de baza în care se pot împartii instructiunile: LOAD-STORE, ALU, salturi si instructiuni de comparare si instructiuni în virgula flotanta. Pentru toate aceste operatii oricare dintre cei 32 de registrii generali poate fi folosit cu specificarea faptului ca încarcarea lui R0 cu orice valoare ramâne fara efect. Valorile în virgula flotanta simpla precizie vor ocupa un singur registru în simpla precizie iar un operand în dubla precizie va ocupa o pereche de registri. Conversia între simpla si dubla precizie trebuie facuta în mod explicit.

În continuare se va arata un exemplu pentru instructiunile de tip load –store.

Example instruction	Instruction name	Meaning
LW R1,30(R2)	Load word	Regs[R1]$\leftarrow_{32}$ Mem[30+Regs[R2]]
LW R1,1000(R0)	Load word	Regs[R1]$\leftarrow_{32}$ Mem[1000+0]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{24} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{32}$ 0^{24} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{16} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
LF F0,50(R3)	Load float	Regs[F0]$\leftarrow_{32}$ Mem[50+Regs[R3]]
LD F0,50(R2)	Load double	Regs[F0]##Regs[F1]$\leftarrow_{64}$ Mem[50+Regs[R2]]
SW 500(R4),R3	Store word	Mem[500+Regs[R4]]$\leftarrow_{32}$ Regs[R3]
SF 40(R3),F0	Store float	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]
SD 40(R3),F0	Store double	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]; Mem[44+Regs[R3]]$\leftarrow_{32}$ Regs[F1]
SH 502(R2),R3	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$ Regs[R31]$_{16..31}$
SB 41(R3),R2	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$ Regs[R2]$_{24..31}$

Figura 2. Instructiunile de tip Load-Store la DLX

Pentru o întelegere mai buna a notatiilor din tabelul anterior sa consideram exemplul urmator:

$Regs[R10]_{16..31 \leftarrow 16} (Mem\ [Regs[R8]]_0)^8$ ## Mem[Regs[R8]]

Semnificatia acestuia este: octetul de la adresa de memorie specificata de continutul lui R8 cu o extensie de semn la 16 biti si este mai apoi încarcat în jumatatea inferioara a lui R10 (jumatatea superioara ramânând neschimbata).

De asemenea, toate instructiunile ALU sunt de tip registru-registru. Acestea includ adunari, scaderi, operatii logice de tip AND, OR, XOR precum si deplasari. De asemenea este posibila folosirea adresarii imediate a tuturor acestor forme împreuna cu o extensie de semn pe 16 biti. Operatiile de tip LHI încarca jumatatea superioara a registrului în timp ce pune la zero pe cea inferioara. Aceasta facilitate ofera posibilitatea construirii unei constante de 32 de biti prin doua instructiuni distincte. Încarcarea unei constante este o simpla adunare a constantei cu registrul R0 si de asemenea o instructiune de tip *move* registru-registru este tot o adunare unde una din surse este registrul R0. Exista si instructiuni de comparare: <, >,≤, ≥, ≠, =. Daca conditia este îndeplinita se seteaza 1 în registrul destinatie altfel se plaseaza 0.

Example instruction	Instruction name	Meaning
`ADD R1,R2,R3`	Add	`Regs[R1]←Regs[R2]+Regs[R3]`
`ADDI R1,R2,#3`	Add immediate	`Regs[R1]←Regs[R2]+3`
`LHI R1,#42`	Load high immediate	`Regs[R1]←42##0`^{16}
`SLLI R1,R2,#5`	Shift left logical immediate	`Regs[R1]←Regs[R2]<<5`
`SLT R1,R2,R3`	Set less than	`if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0`

Figura 3. Exemple de instrutiuni aritmetico-logice

Exista, de asemenea, instructiuni de ramificatie si salt. Din cele patru tipuri de instructiuni de salt, doua folosesc un offset de 26 de biti cu semn care se adauga PC-ului instructiunii urmatoare din secventa determinând astfel adresa destinatie. Salturile sunt tot de doua tipuri: **Plain jump** (salt simplu) si **Jump and**

link (salt cu legatura- folosit în cazul instructiunilor de tip CALL- apel de procedura- acesta din urma plaseaza adresa de revenire în registrul R31).

Instructiunile de ramificatie sunt toate conditionale. Conditia de ramificatie este specificata de catre instructiunea care testeaza registrul sursa daca e sau nu zero; registrul poate contine o valoare data sau rezultatul unei operatii de comparare. Adresa de ramificatie este specificata ca un offset pe 16 biti cu semn care este adaugat PC-ului instructiunii urmatoare din secventa de program. În figura 4 se exemplifica acest tip de instructiuni.

Example instruction	Instruction name	Meaning
J name	Jump	PC←name; ((PC+4)-2^{25}) ≤ name < ((PC+4)+2^{25})
JAL name	Jump and link	R31←PC+4; PC←name; ((PC+4)-2^{25}) ≤ name < ((PC+4)+2^{25})
JALR R2	Jump and link register	Regs[R31]←PC+4; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)-2^{15}) ≤ name < ((PC+4)+2^{15})
BNEZ R4,name	Branch not equal zero	if (Regs[R4]!=0) PC←name; ((PC+4)-2^{15}) ≤ name < ((PC+4)+2^{15})

Figura 4. Instructiuni de salt si ramificatie

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte
LH,LHU,SH	Load half word, load half word unsigned, store half word
LW,SW	Load word, store word (to/from integer registers)
LF,LD,SF,SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one FP register or a DP pair to another register or pair
MOVFP2I,MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD,ADDI,ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB,SUBI,SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT,MULTU,DIV,DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__) ; shifts are shift left logical, right logical, right arithmetic
S__,S__I	Set conditional: "__" may be LT,GT,LE,GE,EQ,NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT,BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
ADDD,ADDF	Add DP, SP numbers
SUBD,SUBF	Subtract DP, SP numbers
MULTD,MULTF	Multiply DP, SP floating point
DIVD,DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
__D,__F	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Figura 5. Setul complet de instructiuni DLX

Implementarea DLX

Sa pornim la început cu o schema a DLX-ului care *nu* implementeaza conceptul de pipeline. Ea este menita a ne face sa întelegem mai bine toate etapele prin care trece o instructiune. În schema de mai jos se prezinta o implementare simpla unde orice instructiune se proceseaza în cel mult patru ciclii. Nu este un caz optim de implementare dar este proiectat astfel încât sa realizeze un pas cât mai natural spre cazul *pipeline*. Pentru implementarea de fata s-au

folosit o serie de registri temporari care nu apartin arhitecturii originale; ei au menirea de a simplifica doar trecerea spre o structura pipeline.

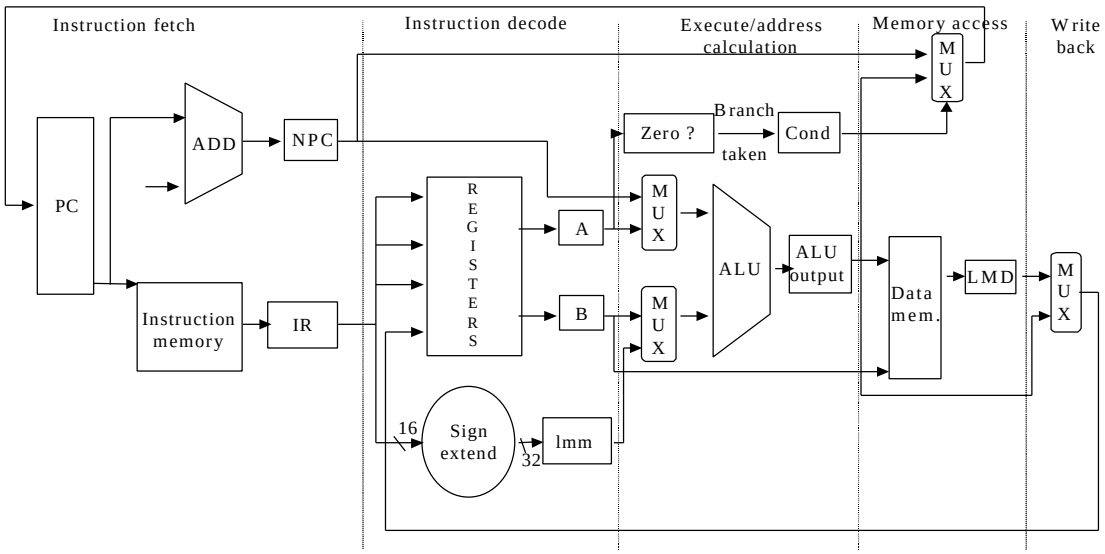

Figura 6. Implementarea simpla DLX

Conform schemei de fata orice instructiune DLX se poate implementa în cel mult cinci cicli. Cei cinci ciclii sunt:

- **Instruction Fetch (IF)**

IR ← Mem[PC]

NextPC ← PC+4

Obtine PC-ul si aduce instructiunea respectiva din memorie în registrul IR; incrementeaza PC-ul cu 4, acesta continând astfel adresa urmatoarei instructiuni din secventa de program. Registrul IR contine instructiunea curenta iar NPC contine PC-ul urmator.

- **Instruction Decode (ID)**

A ← Regs[$IR_{6..10}$];

B ← Regs[$IR_{11..15}$];

Imm ← ((IR_{16})^{16}## $IR_{16..31}$)

Decodifica instructiunea si acceseaza setul de registri pentru a citi continutul acestora. Acestia sunt cititi temporar într-un grup de doi registrii temporari A si B pentru a putea fi folositi ulterior în cadrul ciclilor ulteriori. Celor mai putin semnificativi 16 biti ai registrului IR li se extinde semnul si vor fi stocati în registrul temporar Imm pentru folosirea în urmatorul ciclu. Decodificarea se face în paralel cu citirea registrilor deoarece câmpurile acestora în formatul instructiunii se afla la locatii fixe. Tehnica poarta numele de *fixed-field decoding*. În acest stadiu se calculeaza si extensia de semn în caz ca aceasta este necesara deoarece portiunea imediata dintr-o instructiune se afla localizata mereu în acelasi loc în orice format DLX.

- **Execution (EX)**

În functie de tipul instructiunii DLX avem urmatoarele patru situatii:

- Referinta la memorie:

ALUOutput ← A+Imm;

Se efectueaza adunarea în ALU si se obtine adresa efectiva de memorie care se plaseaza în ALUOutput

- Instructiune ALU de tip Registru-Registru:

ALUOutput ← A op B;

Se efectueaza operatia specificata de opcode între valorile continute în cei doi registri si rezultatul se depune în registrul temporar ALUOutput.

- Instructiune ALU de tip Registru-Imediat:

ALUOutput ← A op Imm;

Se efectueaza operatia ALU specificata de opcode între registru si operandul imediat iar rezultatul se plaseaza în registrul temporar ALUOuput.

- Branch:

ALUOuput ← NPC+Imm;
Cond ← (A op 0)

Pentru a se calcula adresa de salt, ALU aduna NPC cu valoarea din IMM care în prealabil a suferit o extensie de semn. Registrul A care a fost citit în ciclul anterior este verificat pentru a se vedea daca saltul a fost facut sau nu. Op este operatorul relational determinat de opcode-ul branch-ului; de exemplu, pentru instructiunea BEQZ acesta este ”==”

Arhitectura Load-Store a DLX-ului se bazeaza pe faptul ca adresele efective si ciclii de executie pot fi combinati într-un singur ciclu deoarece nici o instructiune nu necesita sa calculeze simultan adresa unei date, adresa destinatie a unei instructiuni si sa execute operatia asupra datei. În aceeasi categorie sunt incluse si instructiunile de tip Jump.

- **Memory access/branch completion (MEM)**

În acest caz avem trei tipuri de instructiuni:

- Referinte la memorie:

LMD ← Mem[ALUOutput] sau
Mem[ALUOutput] ← B;

Daca instructiunea este un load data este citita din memorie în registrul LMD (Load Memory Register) iar daca este o instructiune de tip store atunci data din registrul B este scrisa în memorie.

- Branch:

If (cond)
    PC ← ALUOutput
else
    PC ← NPC

Daca saltul se face PC-ul este înlocuit cu adresa de salt aflata în registrul ALUOutput iar daca saltul nu se face el este înlocuit cu PC-ul incrementat aflat în registrul NPC.

- **Write Back (WB)**

- Instructiune ALU registru-registru:

$Regs[IR_{16..20}] \leftarrow ALUOutput;$

- Instructiune ALU registru –imediat:

$Regs[IR_{11.15}] \leftarrow ALUOutput;$

- Instructiuni de tip Load:

$$Regs[IR_{11.15}] \leftarrow LMD;$$

Scrie rezultatul în setul de registri fie ca acesta provine din memorie sau din iesirile ALU. Se observa din figura 6 ca la sfârsitul fiecarui ciclu, orice valoare calculata si necesara într-un ciclu ulterior, se memoreaza într-o locatie care poate fii de memorie, registru general, registrul PC sau unul din registrii temporari (LMD,Imm, A, B, IR, NPC, ALUOutput, sau Cond).

În aceasta implementare instructiunile de salt (branch) sunt singurele care reclama doar patru ciclii, celelalte desfasurându-se de-a lungul acinci ciclii masina.

Implementarea Pipeline la DLX

Daca studiem mai cu atentie schema precedenta a DLX-ului se observa ca implementarea conceptului de pipeline se poate face aproape fara nici o modificare la schema de baza. În aceasta varianta fiecare ciclu masina va deveni o etapa în structura pipeline. Aceasta duce la o executie a instructiunilor ca în secventa de mai jos:

Numarul ciclului									
Nr. Instructiune	1	2	3	4	5	6	7	8	9
Instructiunea I	IF	ID	EX	MEM	WB				
Instructiunea i+1		IF	ID	EX	MEM	WB			
Instructiunea i+2			IF	ID	EX	MEM	WB		
Instructiunea i+3				IF	ID	EX	MEM	WB	

Instructiunea i+4					IF	ID	EX	MEM	WB

În realitate însa lucrurile sunt un pic mai complexe. Pentru început trebuie sa ne asiguram de faptul ca nu încercam sa utilizam o resursa a sistemului, una din unitatile sale functionale, de mai multe ori în acelasi ciclu în scopuri diferite. De exemplu, o singura unitate de adunare nu poate fi folosita concomitent la calculul unei adrese efective si la efectuarea unei scaderi oarecare. Ca si în structura anterioara vom folosi o memorie de instructiuni diferita fata de cea de date care va fi implementata prin cache-uri separate pe instructiuni si date (arhitectura Harvard). Se elimina astfel conflictul care ar putea apare între un fetch instructiune si un acces concomitent la memoria de date. Setul de registri se va folosi de doua ori în acest caz: odata pentru a citi (ID-ul instructiuni) si apoi pentru a scrie rezultatul (WB). Problema scrierii si citirii simultane se va ignora pentru moment.

Datorita structurii pipeline e necesar ca valorile transmise dintr-o etapa în alta sa fie stocate în registrii intermediari numiti registrii pipeline (pipeline latches). Astfel ajungem la structura urmatoare:

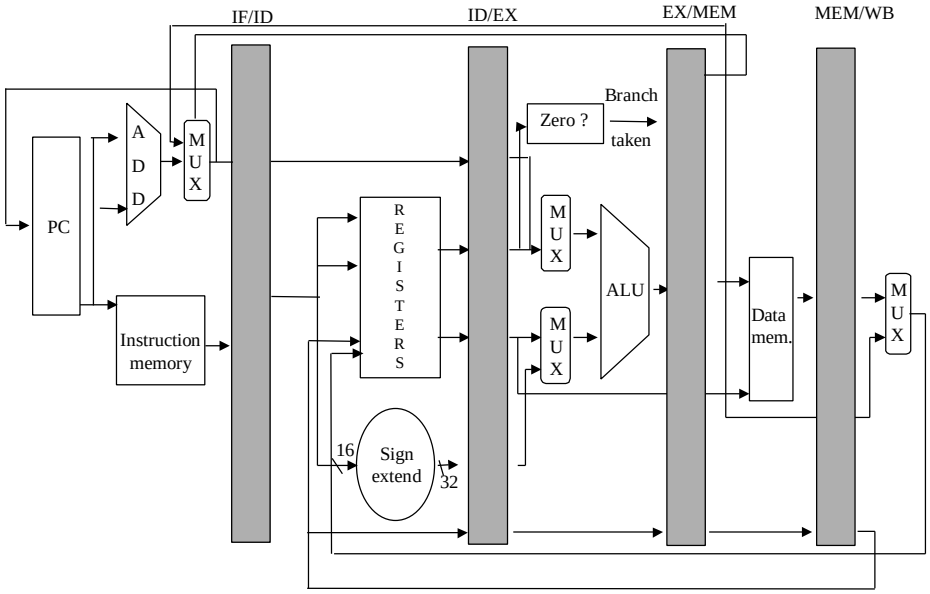

Figura 7. Implementarea DLX cu structura pipeline

Prin registrii pipeline trec atât date cât si informatii de control al fluxului structurii pipeline. Aceste informatii trebuie sa fie transferate de la un nivel la altul al structurii pipeline pâna în momentul în care vor fi folositi. Nu se pot folosi doar registrii temporari datorita faptului ca se pot suprascrie valori care nu au fost folosite înca. De exemplu, câmpul unui registru necesar unei scrieri pe parcursul unei operatii LOAD sau ALU, este luat din latch-ul MEM/WB si nu din IF/ID. Aceasta deoarece se doreste ca operatia respectiva sa scrie registrul destinatie desemnat de operatia respectiva si nu unul din registrii care tranziteaza de la IF la ID. Acesta din urma e copiat dintr-un latch în altul pâna în faza WB unde e folosit. De remarcat este faptul ca primele doua etape din structura pipeline sunt independente de tipul de instructiune deoarece orice instructiune este decodificata doar la finele ciclului ID. Activitatea ciclului IF depinde daca instructiunea este un salt care se face sau care nu se face. Daca saltul se face atunci adresa de salt este folosita ca PC si pentru a calcula adresa urmatoare. Pentru a controla fluxul datelor în schema de mai sus e necesar sa cunoastem functionarea celor patru MUX-uri din figura 7.

Cele doua MUX-uri de la intrarile ALU sunt utilizate în functie de tipul instructiunii care este dat de câmpul IR din registrul ID/EX. MUX-ul de la iesirea sumatorului, înaintea registrului IF/ID este utilizat daca instructiunea este un salt. Astfel daca instructiunea este un salt, PC-ul este încarcat cu noua valoare a adresei de salt în cazul ca acesta se face. Noua adresa se obtine pe baza fazei EX/MEM. Multiplexorul alege daca sa foloseasca PC-ul curent sau valoarea din EX/MEM NPC(vezi registrul **NPC** figura 6). Acest multiplexor este controlat de catre registrul EX/MEM cond (registrul ***Cond*** în figura 6). Cel de-al patrulea multiplexor este setat în functie de tipul instructiunii din ciclul WB: ALU sau LOAD.

Se observa de asemenea necesitatea unui al cincilea multiplexor care sa aleaga portiunea corecta a registrului IR în latch-ul MEM/WB pentru a specifica

registrul destinatie. Acesta poate avea doua variante: registru-registru ALU respectiv ALU-imediat sau LOAD.

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IR$_{16}$)^{16}##IR$_{16..31}$;		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;	 EX/MEM.ALUOutput ← ID/EX.NPC+ID.EX.Imm; EX/MEM.cond ← (ID/EX.A *op* 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUOutput;	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD;	

Figura 8. Nivelele pipeline DLX

Bibliografie

[1] **Hennessy J., Patterson D.** - *Computer Organization and Design*: *The Hardware / Software Interface*, Morgan Kaufmann Publishers, San Francisco, California, 1994

[2] **Gründbacher H.** - *Windows Deluxe Simulator - Tutorial*, University of Technology, Viena, 1992