Universitatea Ştefan cel Mare Suceava
Facultatea de Inginerie Electrică şi Ştiinţa Calculatoarelor

CERCETĂRI PRIVIND PROIECTAREA PARTICULARIZATĂ A UNUI CPU PE BAZA UNUI PLANIFICATOR HARDWARE ŞI A MECANISMELOR SPECIFICE NUCLEELOR DE TIMP REAL

- raport de cercetare nr. 2 -

Coordonator ştiinţific,
prof. univ. dr. ing. Vasile-Gheorghiţă GĂITAN

Doctorand,
ing. MOISUC (CIOBANU) Elena-Eugenia

Suceava
- 2015 -

CUPRINS

ABREVIERI

ADDI - Addition Immediate
ALU - Arithmetic Logic Unit
BEQ - Branch if Equal
CAD - Computer Aided Design
CISC - Complex Instruction Set Computer
CLB - Configurable Logic Block
CPU - Central Processor Unit
DFF - D-Flip Flop
DIP - Dual Inline Package
EAB - Embedded Array Blocks
EX - Execute
FIFO - First-In First-Out
FPGA - Field Programmable Gate Array
GPR - General Purpose Register
HDL - Hardware Description Language
HSE - Hardware Scheduler Engine
ID - Instruction Decode
IF - Instruction Fetch
IR - Instruction Registers
ISA - Instruction Set Architecture
J - Jump
JTAG - Joint Test Action Group
LAB - Logic Array Block
LB - Load Byte
LE - Logic Elements
LUT - LookUp Table
MEM - MEMory
MIPS - Microprocessor without Interlocked Pipeline Stages
MPRA - Multi Pipeline Register Architecture
PC - Program Counter
PIA - Programmable Interconnect Array
PLD - Programmable Logic Device
RAM - Random Access Memory
RISC - Reduced Instruction Set Computer
ROM - Read Only Memory
RTL - Register Transfer Level
RTS - Real-Time System
SB - Store Byte
SRAM - Static Random Access Memory
VHDL - Very high speed integrated circuit Hardware Description Language
WB - Write Back

1. INTRODUCERE

În prezent, necesitatea optimizării modalităţii în care este folosit timpul poate fi observată cu uşurinţă în toate domeniile în care roboţii, de exemplu, au reuşit să degreveze activităţile umane. Apariţia procesoarelor şi a sistemelor de control bazate pe acestea, precum şi a algoritmilor de planificare software au permis eficientizarea organizării timpului în sistemele digitale cu rezoluţii de ordinul *ms* sau chiar mai mici. Ca şi în viaţa de zi cu zi, partajarea timpului rămâne astfel singura tehnică de execuţie pseudo-paralelă a task-urilor într-un sistem embedded de timp real (STR).

Cererea pentru procesoare de mare viteză în industria de semiconductori şi viaţa modernă este în continuă creştere. S-au făcut multe eforturi de cercetare pentru a optimiza viteza procesoarelor convenţionale, rezultând o integrare cu succes a modulelor de procesare a semnalelor complexe.

FPGA-urile sunt bine adaptate pentru reducerea căii combinaţionale, precum şi utilizarea operațiunilor paralele, care pot oferi o soluție mai bună pentru manipularea vitezei. În afară de aceasta, procesorul pipeline implementat în FPGA a început să depășească cele mai multe aplicații. Tehnica FPGA are capacitatea de a furniza un produs de nivel ridicat şi de a evita costul ridicat inițial şi ciclurile de dezvoltare lungi. Programabilitatea FPGA-ului permite upgrade-uri de design în domeniu fără a fi necesară înlocuirea hardware-ului. Acest lucru poate ajuta proiectantul pentru a efectua procesele de bază mai rapid.

În **Introducere** s-a prezentat **arhitectura MIPS** în general, pornind de la procesoarele **RISC**.

În **partea a 2-a** s-a considerat necesar un **scurt istoric** al procesoarelor MIPS.

În **partea a 3-a** se prezintă procesorul **MIPS pipeline**.

Partea a 4-a cuprinde selecţii din articolele doctorandului prezentate la conferinţe internaţionale şi publicate cu aceste prilejuri, conform referinţelor bibliografice.

Concluziile încheie acest raport, arătând importanţa în sistemele de timp-real a structurii hardware originală, bazată pe structura organizaţională a MIPS, utilizată pentru planificarea task-ului, static şi dinamic, şi care oferă managementul unitar al evenimentelor. Scopul a fost de a îmbunatăţi prin hardware performanţele RTOS pentru microcontrolere, pentru a comuta mai rapid între task-uri, pentru a îmbunătăți timpul de răspuns la evenimente externe, pentru a îmbunătăți comportamentul întreruperilor și pentru a oferi mai multe tipuri de bază de comunicare inter-task-uri.

1.1. ARHITECTURA RISC-MIPS

MIPS Technologies şi-a creat o bază puternică în electrocasnice şi media playere portabile. Sony PlayStation Portable foloseşte două procesoare bazate pe MIPS32 4K. În cadrul segmentului de networking, Cavium Networks şi Net logic Microsystems folosesc MIPS. MIPS este utilizat şi pentru a construi telefoane inteligente şi tablete Cruz de la Velocity Micro. TCL Corporation foloseşte procesoare MIPS pentru dezvoltarea de telefoane inteligente. Companiile pot obține o licență de arhitectură MIPS pentru proiectarea propriilor nuclee CPU utilizând setul de instrucțiuni MIPS. MIPS Technologies este utilizat predominant în combinație cu sistemele de operare Android și Linux.

Figure 1-1 Aplicaţii pentru MIPS

RISC (*Reduced Instruction Set Computer*) este o filosofie de design care a devenit populară în ultimii ani, dominând industria extrem de competitivă de calculatoare. Procesorul RISC operează cu foarte puține tipuri de date și face operații simple. El suportă câteva moduri de adresare și se bazează în special pe registre. Cele mai multe dintre instrucțiuni operează asupra datelor prezente în registrele interne. Design-ul RISC a dus la construirea computerelor ce execută mai rapid instrucțiunile. Într-o mașină RISC, doar instrucțiunile *load* și *store* accesează memoria. Instrucţiunea *load (l)* încarcă datele de la memorie şi instrucţiunea *store (s)* scrie datele în memorie. Aceasta este cheia pentru execuția într-un singur ciclu a instrucțiunilor. Comparativ cu CISC, procesoarele RISC au mai multe avantaje, cum ar fi viteza mai mare, o structură simplificată, mai ușor de implementat. Procesoarele RISC sunt folosite pe scară largă în sistemele integrate (embedded) [1].

Utilizarea VHDL pentru modelare este deosebit de atrăgătoare deoarece oferă o descriere formală a sistemului și permite utilizarea de stiluri de descriere specifice pentru a acoperi diferite niveluri de abstractizare (arhitecturală, transferul în registre și nivelul logic). MIPS este un procesor RISC. În particular, procesorul MIPS utilizează un set redus de instrucțiuni, prin urmare rezultă un minim efort uman. Pipeline-ul îmbunătățește rata de transfer a instrucţiunii. Există intenţia de a spori viteza procesorului și a simplifica hardware-ul, din motive de costuri. Dacă se implementează acest procesor folosind tehnologia FPGA, este posibil să se upgradeze sistemul cu noi caracteristici cerute de utilizatori.

2. SCURT ISTORIC MIPS

MIPS (*Micropocessor whithout Interlocked Pipeline Stages*) este o arhitectură de microprocesoare **RISC** (*Reduced Instruction Set Computer*) dezvoltată de *MIPS Technologies*.

La început, arhitecturile MIPS erau implementări pe 32 biţi. Setul de instrucţiuni a suferit şi el modificări, trecând prin MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V. Versiunile curente sunt MIPS32 pentru implementări pe 32 biţi şi MIPS64 pentru implementări pe 64 biţi, cu diverse extensii pentru operaţii speciale în virgulă mobilă, multithreading etc.

În 1981 la Universitatea Stanford, o echipă condusă de John L. Hennessy a început să lucreze la ceea ce va deveni primul procesor MIPS. Ideea ce stătea la baza proiectului era creşterea performanţelor microprocesorului cu ajutorul unei benzi de asamblare cu cât mai multe nivele, o tehnică deja cunoscută însă dificil de implementat. Principala barieră în calea

dezvoltării de arhitecturi cu bandă de asamblare era necesitatea implementării unui mecanism hardware care să blocheze banda la apariţia unei instrucţiuni ce durează mai multe cicluri astfel încât să nu se încarce alte instrucţiuni până la terminarea execuţei curente. Din această cauză, un aspect major al designului MIPS de la acea dată era cerinţa ca toate instrucţiunile să poată fi executate într-un singur ciclu, pentru a evita necesitatea folosirii mecanismului descris anterior. Ideea era că deşi astfel se eliminau o parte din instrucţiunile arhitecturii (cum ar fi înmulţirea şi împărţirea), performanţa sistemului va creşte deoarece procesorul ar lucra la frecvenţa mult mai mare.

În 1984 Hennessy era convins de potenţialul comercial al designului şi a părăsit universitatea pentru a înfiinţa compania MIPS Computer Systems. În 1985 compania a scos pe piaţa prima arhitectură, R2000, iar în 1988 au construit varianta îmbunătăţită, R3000. Aceste două procesoare au fost produsul de bază al companiei în timpul anilor `80, fiind folosite mai ales în staţiile de lucru produse de Silicon Graphics. Spre deosebire de designul didactic, procesoarele comercializate ofereau mecanisme pentru execuţia de instrucţiuni în mai multe cicluri, printre altele oferind instrucţiuni de înmulţire şi împărţire.

În 1991 MIPS a scos pe piaţă primul microprocesor pe 64 biţi, R4000, însă a avut dificultăţi financiare înainte de lansarea acestuia. Deoarece designul MIPS era important pentru Silicon Graphics, aceştia au cumpărat compania lui Hennessy, transformând-o în MIPS Technologies, subordonată SGI.

Tot la începutul anilor `90 MIPS a început să-şi vândă sub licentă design-urile, fapt care s-a dovedit o idee de succes întrucât simplitatea procesorului l-a făcut potrivit pentru o serie de aplicaţii ce folosiseră înainte procesoare CISC mai puţin performante.

Până la sfârşitul anilor `90 MIPS a devenit un lider în industria procesoarelor pentru sisteme embedded. Din profiturile de astăzi ale companiei, jumătate provin din licenţierea de design în timp ce o mare parte din cealaltă jumătate reprezintă muncă de design sub contract pentru alţi producători.

În 1999 MIPS şi-au formalizat sistemul de licenţiere în jurul a doua designuri de baza: MIPS32 şi MIPS64. NEC, Toshiba şi SiByte au cumpărat licenţe pentru MIPS64 încă de la lansarea acestuia, urmaţi de Philips, LSI Logic şi IDT. Drept urmare, procesoarele MIPS sunt astăzi unele dintre cele mai folosite nuclee de „categorie grea” în industria embedded.

Mai mult, există şi companii care folosesc arhitectura MIPS pentru a produce sisteme multicore. Deşi a câştigat pe piaţa embedded, MIPS, ca şi majoritatea procesoarelor de tip RISC, a pierdut pe piata staţiilor de lucru şi a calculatoarelor personale. La începutul anilor `90 se făceau speculaţii conform cărora MIPS şi alte procesoare RISC puternice vor depăşi arhitectura Intel IA32, fapt încurajat de apariţia de versiuni Windows NT pentru DEC Alpha, MIPS şi PowerPC. Cu toate acestea, odată cu lansarea de către Intel a procesoarelor de clasă Pentium în versiuni din ce în ce mai rapide, Microsoft Windows NT v4.0 a renunţat la orice efort de compatibilitate cu alte procesoare în afară de Intel. Iar cand SGI au decis să facă tranziţia spre arhitecturi Itanium şi IA32, dispariţia MIPS de pe piata staţiilor de lucru a devenit o realitate.

Modelul **R8000** (1994) a fost primul design MIPS superscalar, putând să execute 2 operaţii aritmetico-logice şi două operaţii cu memoria pe ciclu. Cu toate acestea, avea performanţe limitate de lucru cu întregi şi costul era destul de ridicat ceea ce l-au făcut să fi e folosit mai mult de utilizatori din domeniul ştiinţific (pentru performanţele bune în virgulă mobilă).

În 1995 s-a lansat modelul **R10000**, un procesor cu design pe un singur chip, viteză mai mare, cache-uri primare de instrucţiuni şi date de 32 kB şi de asemenea superscalar. Faţă de

modelul precedent, este mai ieftin, mai redus ca dimensiuni şi cu performanţe mai bune, mai ales în zona de lucru cu întregi, unde predecesorul avea probleme.

Următoarele implementări sunt bazate pe acest model şi sunt reprezentate de **R12000** (dimensiuni mai mici şi viteză mărită), **R14000** (viteze mai mari, posibilitatea de a folosi DDR şi SRAM, front side bus la 200 MHz pentru eficienţă crescută), **R16000** şi **R16000A** (din nou creştere a vitezei, cache de nivel 1 suplimentar, chip de dimensiuni mai mici).

Un produs MIPS din anul 1995 a fost MIPS T5 (redenumit apoi R1000), cu o arhitectură superscalară pe 64 de biţi nouă, compatibilă cu cipurile mai vechi Rxxx. Arhitectura scalară dispunea de 5 canale, 64 de registre interne şi o memorie cache internă de 32 KB, utilizânduse o tehnologie de fabricaţie de 0,35 de microni. Unele concepte deosebit de interesante cu privire la acest aspect au fost studiate la Universitatea Stanford cu MIPS-X, un produs derivat al arhitecturii MIPS ce avea o serie de caracteristici în plus. Multe dintre acestea au fost mai târziu introduse în procesorul comercial MIPS. Microprocesorul MIPS R2000 este un procesor pe 32 de biţi cu o memorie cache de nivel 2, diferenţiată pentru instrucţiuni şi date. O memorie tampon de scriere ajută la manipularea tuturor datelor stocate în memorie. Produsul R2000 foloseşte o magistrală comună pentru memoria cache externă – o arhitectură *non Harvard* (arhitectura *Harvard* presupune utilizarea de magistrale diferite pentru instrucţiuni şi pentru date). Construcţia acestui procesor înglobează o arhitectură radicală de coprocesor. Unitatea de control a întregilor din UCP este separată de aşa numitul „Coprocesor de control al sistemului” (*System Control Coprocessor*), care este, de fapt, un controlor de memorie cache integrat direct pe cip UCP şi unitatea de calcul în virgulă mobilă comunică prin intermediul memoriei. Microprocesorul înglobează 32 de regiştri generali şi 16 regiştri (pe 64 de biţi) separaţi pentru calcule în virgulă mobilă. Coprocesorul pentru calculul în virgulă mobilă conţine o unitate pentru adunare, una pentru împărţire şi una pentru înmulţire. Nu există biţi de testare a condiţiilor (indicatori de stare, sau flags, cum sunt denumiţi la Intel). Programarea regiştrilor este controlată software [2].

În concluzie, procesoarele MIPS au fost şi sunt un success comercial, fiind folosite astăzi în multe aplicaţii destinate consumului sau industriei. Nuclee MIPS se pot găsi în routere, modemuri pentru cablu, imprimante laser, roboţi, console de jocuri video şi PDA-uri.

3. MIPS32 PIPELINE

Consumul scăzut şi caracteristicile de încălzire ale sistemelor embedded ce reprezintă implementări MIPS, disponibilitatea utilitarelor pentru dezvoltare în domeniul embedded precum şi faptul că MIPS a devenit o arhitectură cunoscută, toate îi asigură procesorului de tip MIPS un rol important în industria embedded [3].

Procesorul MIPS pipeline este, în esenţă, un procesor într-un singur ciclu, având câteva caracteristici avansate adăugate, patru registre pipeline, o unitate de hazard şi o unitate de forward. O altă diferenţă majoră este poziţia branch-ului şi a jump-ului logic.

Avantajul de a avea aceste patru registre pipeline este acela că mai mult de o instrucţiune poate fi prezentă în acelaşi timp. Acest lucru permite următoarei instrucţiuni să intre în prima etapă de îndată ce instrucţiunea curentă o părăseşte. Ca diferenţă, la cele 5 cicluri de aşteptare pentru pentru a finaliza o instrucţiune, acum până la 5 instrucţiuni pot fi procesate în acelaşi timp.

Modelul de bază pentru un procesor MIPS cu 5 stadii pipeline:

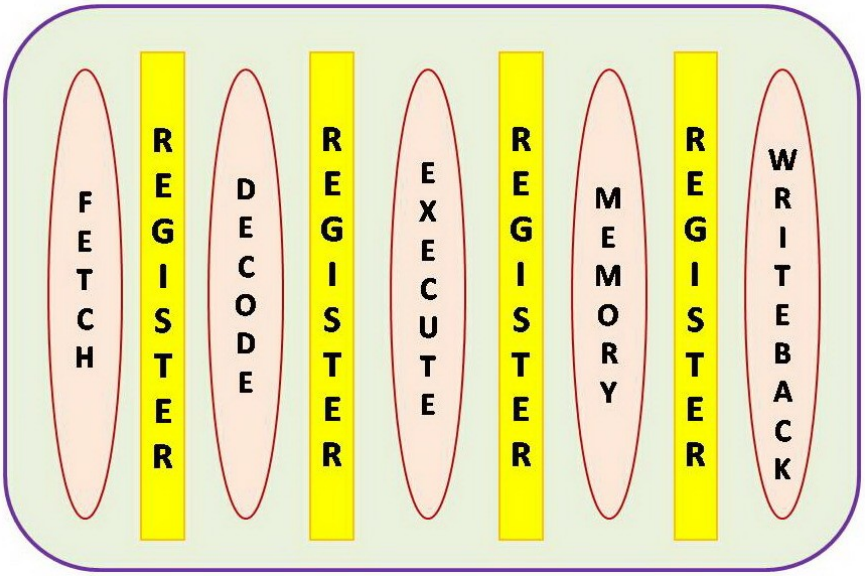


Figure 3-1 MIPS cu cinci stadii pipeline

Fig. 1 prezintă cele cinci etape şi cele patru registre care stochează date între fiecare etapă. Această figură arată mai bine cum pot exista simultan cinci instrucţiuni în interiorul procesorului, fără a fi nevoie să aştepte finalizarea pentru o singură instrucţiune în fiecare etapă. Fiecare registru stochează datele pentru o instrucţiune specifică, eliberând informaţiile necesare fiecărei etape, până când a trecut prin întreagul procesor.

Registrele inter-stadii sunt bistabili D master-slave (fig. 2). Masterul primeşte noile date de la etapa anterioară a instrucţiunii, în timp ce slave-ul furnizează datele următoarei etape.

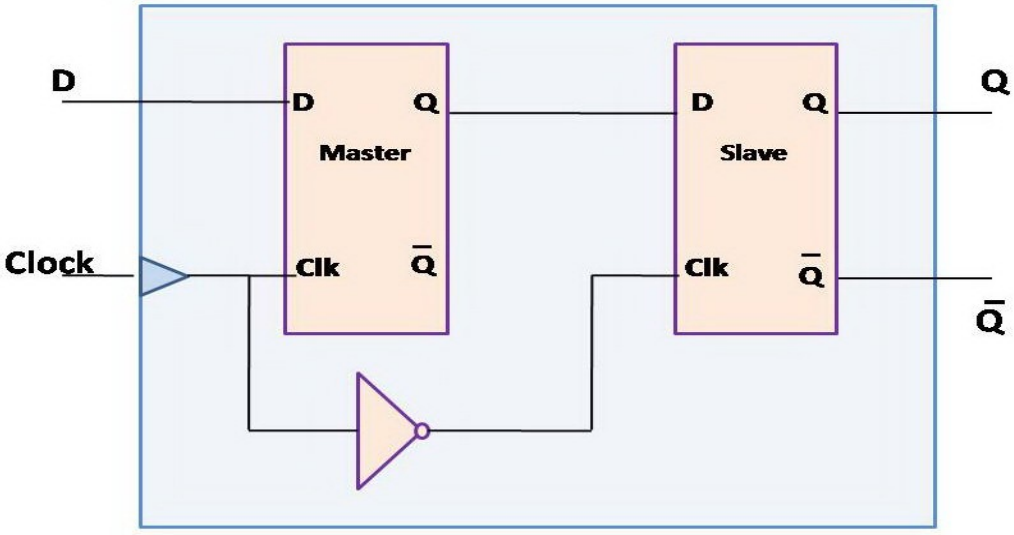


Figura 3-2 Registrele pipeline dintre stadii

3.1. ARHITECTURĂ VERSUS IMPLEMENTARE

După cum s-a putut observa din paragraful de mai sus, deoarece MIPS nu se referă neapărat la un procesor fizic ci mai mult la nişte specificaţii de design este importantă diferenţa dintre *arhitectura* MIPS şi o *implementare efectivă* a acestei arhitecturi:

- arhitectura se referă la setul de instrucţiuni, setul de regiştri, modelul excepţiilor, managementul memoriei, schema adreselor fizice şi virtuale precum şi alte caracteristici comune tipurilor de hardware din această categorie
- implementarea se referă la modul în care anumite procesoare aleg să realizeze fizic caracteristicile arhitecturii

De exemplu, o unitate de calcul în virgulă mobilă este o parte (deşi opţională) a arhitecturii MIPS. Diferite implementări ale unei astfel de unităţi pot avea pipeline-uri cu număr diferit de niveluri, folosind algoritmi diferiţi pentru a efectua înmulţiri sau împărţiri etc. Un alt exemplu bun este sistemul de cache-uri: majoritatea procesoarelor MIPS au cacheuri, însă nu sunt obligate să le implementeze la fel, după cum unele procesoare au mai multe cache-uri de niveluri diferite în timp ce altele au un singur cache.

Ideea de bază este că arhitectura MIPS este total decuplată de implementările hardware specifice, lăsând la latitudinea arhitecţilor de microprocesoare crearea de design-uri hardware proprii folosind ca schelet definiţia arhitecturală.

3.2. CONCEPTUL BENZII DE ASAMBLARE (PIPELINE)

Registrele pipeline sau registrele de stare sunt cele mai importante componente ale întregului procesor. Fără ele ar fi un procesor într-un singur ciclu, ceea ce ar duce la aplicaţii lente şi performanţă scăzută.

Aceste patru registre pipeline permit procesorului să lucreze pe 5 instrucţiuni la un moment dat (într-un singur ciclu). Creşterea vitezei şi a performanţei sunt direct legate de numărul de etape pipeline.

Microprocesorul serial:
- execută *n* instrucţiuni utilizând *s* etape în *ns* cicluri
- total timp de execuţie = *ns*

Microprocesorul pipeline:
- execută *n* instrucţiuni utilizând *s* etape
- total timp de execuţie = *s+(n-1)*

Exemplu:
n=30, s=5
- microprocesor serial: 150 cicluri
- microprocesor pipeline: 5+29=34 cicluri

Toate procesoarele recente încorporează benzi de asamblare ca tehnică-cheie de implementare. Cum toate stadiile sunt conectate, acţiunea ar trebui să fie gata în acelaşi timp. Timpul necesar pentru a muta o instrucţiune până la un alt stadiu din cele cinci etape este cunoscut sub numele de „ciclu de procesor”. Etapa cea mai lentă decide durata ciclului procesorului. Este responsabilitatea designer-ului de a echilibra durata ciclului de procesor pentru fiecare etapă. Întotdeauna trebuie luat în considerare faptul că pipeline-ul reduce timpul mediu de execuţie pe instrucţiune.

3.3. STRUCTURA PIPELINE

În cazul procesorului MIPS pipeline, la fiecare ciclu de ceas se iniţializează o nouă instrucţiune. Aici, fiecare ciclu de ceas înseamnă unul dintre stadiile de pipeline. Tabelul 1 reprezintă structura pipeline tipică, fiecărei instrucţiuni luându-i cinci clock-uri pentru a finaliza execuţia, iar hardware-ul va porni o nouă instrucţiune şi va executa câte o parte în fiecare etapă.

Tabel 3-1 Structura RISC pipeline

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB

În MIPS există trei tipuri diferite de instrucţiuni: tipul ***R***, tipul ***I*** şi tipul ***J***:

Tabel 3-2 Formatul instrucţiunilor MIPS

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

1. Instrucţiunile de tipul ***R***: prescurtarea pentru tipul ***register***. Acestea utilizează trei regiştri ca operanzi: doi ca sursă şi unul ca destinaţie.
2. Instrucţiunile de tipul ***I***: prescurtarea pentru tipul ***immediate***. Acestea utilizează doi operanzi regiştri şi un operand *16-bit immediate*.
3. Instrucţiunile de tipul ***J***: prescurtarea pentru tipul ***jump***. Se utilizează numai cu instrucţiunea *jump* şi foloseşte un singur operand *26-bit address*.

rs: *the first register source operand*.
rt: *the second register source operand*.
rd: *the register destination operand*, dă rezultatul operaţiunii.
shamt: *shift amount*, este folosit în instrucţiunea *shift* să menţină valoarea deplasată.
funct: *function*, selectează varianta specifică a operaţiunii în câmpul *op*.
imm: *the 16-bit address*, care este utilizat în instrucţiunile de transfer de date.
addr: *the 26-bit address*, care este folosit în instrucţiunile *jump*.

3.4. DATAPATH

Fig. 3 arată datapath-ul pentru un procesor MIPS pipeline. Regiştrii pipeline sunt între stadiile de pipeline. Aceste etape sunt denumite astfel încât să arate conexiunea de la un stadiu la următorul. Se cunoaşte că fiecare operaţiune trebuie să fie completă într-un ciclu de ceas. Registrele pipeline sunt necesare deoarece fiecare operaţiune ce trece de la o etapă la următoarea are nevoie să fie stocată temporar în registrul pipeline corespunzător.

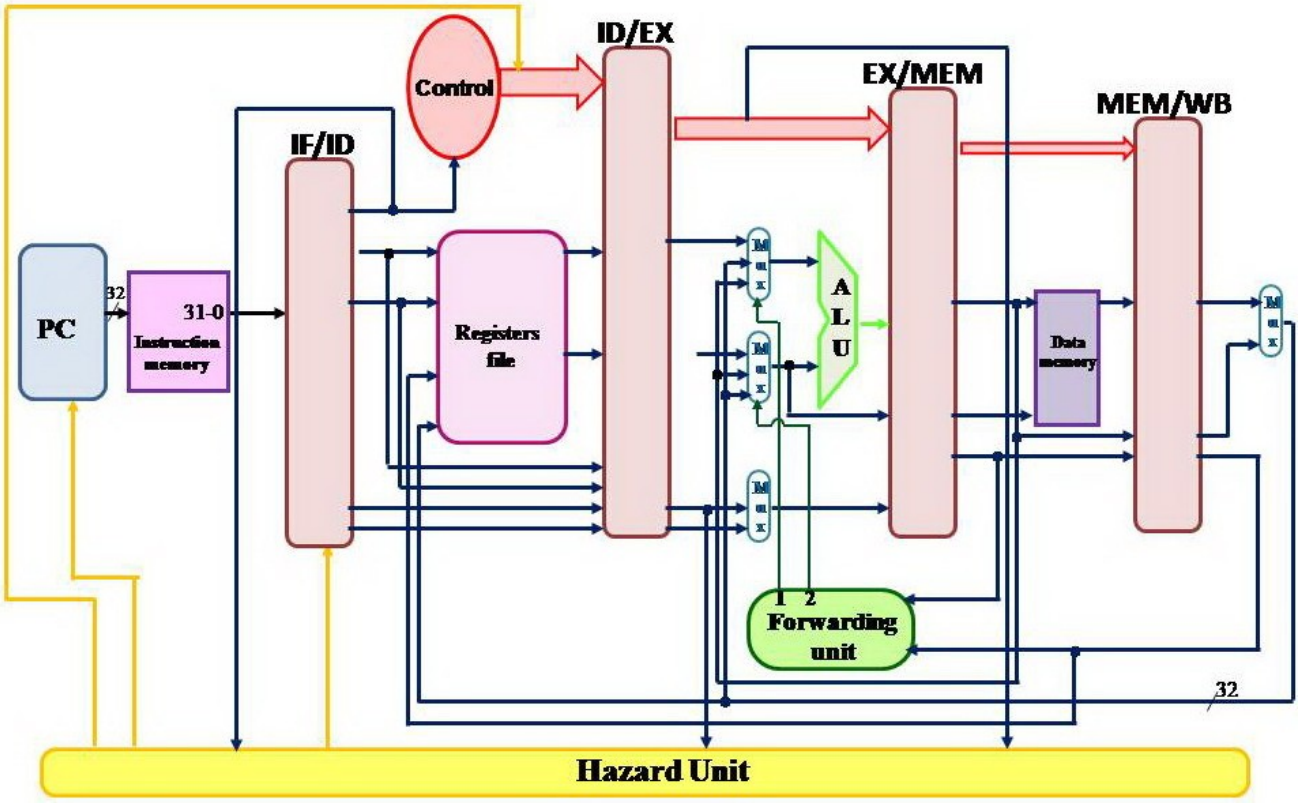


Figura 3-3 Datapath pipeline cu unitate de control, unitate de forward şi unitate de detecţie a hazardului

Etapele pipeline din MIPS pentru diferite tipuri de instrucţiuni sunt arătate în tabelul 3. Operaţiunile în fiecare etapă a structurii pipeline se exemplifică în tabelul 4. În tabelul 5 este exemplificat setul de instrucţiuni MIPS.

Tabel 3-3 Etapele pipeline din MIPS pentru diferite tipuri de instrucţiuni

***STORE* INSTRUCTION**				
IF	ID	EX	MEM	NOP
***LOAD* INSTRUCTION**				
IF	ID	EX	MEM	WB
***R* TYPE & ARITHMETIC *I* TYPE INSTRUCTION**				
IF	ID	EX	NOP	WB
***BRANCH* INSTRUCTION**				
IF	ID	EX	MEM	NOP
***JUMP* INSTRUCTION**				
IF	ID	NOP	NOP	NOP

Tabel 3-4 Operaţiile pe fiecare stadiu pipeline în arhitectura MIPS (*sursa:* [2])

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC] ; IF/ID.NPC, PC ← (1f ((EX/MEM.opcode = = branch) & EX/MEM.cond) {EX/MEM. ALUOutput} else {PC+4} ;		
ID	ID/EX.A ← Regs [IF/ID.IR [rs]] ; ID/EX.B ← Regs [IF/ID.IR [rt]] ; ID/EX.NPC ← IF/ID.NPC ; ID/EX.IR ← IF/ID.IR ; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]) ;		
	ALU instruction	***Load* or *Store* instruction**	***Branch* instruction**
EX	EX/MEM.IR ← ID/EX.IR ; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B ; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm ;	EX/MEM.IR to ID/EX.IR ; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm ; EX/MEM.B ← ID/EX.B ;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm « 2) ; EX/MEM.cond ← (ID/EX.A = = 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.ALIOutput] ; or Mem[EX/MEM.ALIOutput] ← EX/MEM.B ;	
WB	Regs[MEM/WB.IR [rd] ← MEM/WB.ALUOutput ; or Regs[MEM/WB.IR [rt]] ← MEM/WB.ALUOutput ;	For load only: Regs[MEM/WB.IR [rt]] ← MEM/WB.LMD ;	

Tabel 3-5 Setul de instrucţiuni MIPS

1.	**add**	Addition
2.	**addu**	Addition unsigned
3.	**addi**	Addition immediate
4.	**addiu**	Addition immediate unsigned
5.	**sub**	Subtraction
6.	**subu**	Subtraction unsigned
7.	**mul**	Multiply (without overflow)
8.	**and**	And
9.	**andi**	And immediate
10.	**or**	Or
11.	**ori**	Or immediate
12.	**xor**	Xor
13.	**xori**	Xor immediate
14.	**nor**	Nor

15.	**sll**	Shift left logical
16.	**sllv**	Shift left logical variable
17.	**sra**	Shift right arithmetic
18.	**srav**	Shift right arithmetic variable
19.	**srl**	Shift right logical
20.	**srlv**	Shift right logical variable
21.	**lb**	Load byte
22.	**lbu**	Load byte unsigned
23.	**lh**	Load halfword
24.	**lhu**	Load halfword unsigned
25.	**lw**	Load word
26.	**lui**	Load upper immediate
27.	**sb**	Store byte
28.	**sh**	Store halfword
29.	**sw**	Store word
30.	**beq**	Branch on equal
31.	**bne**	Branch on not equal
32.	**bgtz**	Branch on greater than zero
33.	**bgez**	Branch on greater than or equal to zero
34.	**bgezal**	Branch on greater than or equal to zero and link
35.	**blez**	Branch on less than or equal to zero
36.	**bltz**	Branch on less than zero
37.	**bltzal**	Branch on less than zero and link
38.	**slt**	Set less than
39.	**sltu**	Set less than unsigned
40.	**slti**	Set less than immediate
41.	**sltiu**	Set less than immediate unsigned
42.	**j**	Jump
43.	**jal**	Jump and link
44.	**jalr**	Jump and link register
45.	**jr**	Jump register

MIPS Instructions								
Bit #	[31..26]	[25..21]	[20..16]	[15..11]	[10..06]	[05..00]	Operations	
R-type	op	rs	rt	rd	sa	func		
add	000000	rs	rt	rd	00000	100000	rd <-- rs + rt;	PC <-- PC + 4
sub	000000	rs	rt	rd	00000	100010	rd <-- rs - rt;	PC <-- PC + 4
and	000000	rs	rt	rd	00000	100100	rd <-- rs & rt;	PC <-- PC + 4
or	000000	rs	rt	rd	00000	100101	rd <-- rs | rt;	PC <-- PC + 4
sll	000000	00000	rt	rd	sa	000000	rd <-- rt << sa;	PC <-- PC + 4
srl	000000	00000	rt	rd	sa	000010	rd <-- rt >> sa (logical);	PC <-- PC + 4
sra	000000	00000	rt	rd	sa	000011	rd <-- rt >> sa (arithmetic);	PC <-- PC + 4
I-type	op	rs	rt	immediate				
addi	001000	rs	rt	immediate			rt <-- rs + (sign_extend)immediate;	PC <-- PC + 4
andi	001100	rs	rt	immediate			rt <-- rs & (zero_extend)immediate;	PC <-- PC + 4
ori	001101	rs	rt	immediate			rt <-- rs | (zero_extend)immediate;	PC <-- PC + 4
lw	100011	rs	rt	immediate			rt <-- memory[rs + (sign_extend)immediate];	PC <-- PC + 4
sw	101011	rs	rt	immediate			memory[rs + (sign_extend)immediate] <-- rt;	PC <-- PC + 4
beq	000100	rs	rt	immediate			if (rs == rt) PC <-- PC + 4 + (sign_extend)immediate<<2; else	PC <-- PC + 4
bne	000101	rs	rt	immediate			if (rs != rt) PC <-- PC + 4 + (sign_extend)immediate<<2; else	PC <-- PC + 4
J-type	op	address						
j	000010	address					PC <-- (PC+4)[31..28],address<<2	

Figura 3-4 Instrucţiuni MIPS (*sursa:* [5])

Să aruncăm o privire asupra operațiunilor care au loc la fiecare etapă pipeline.

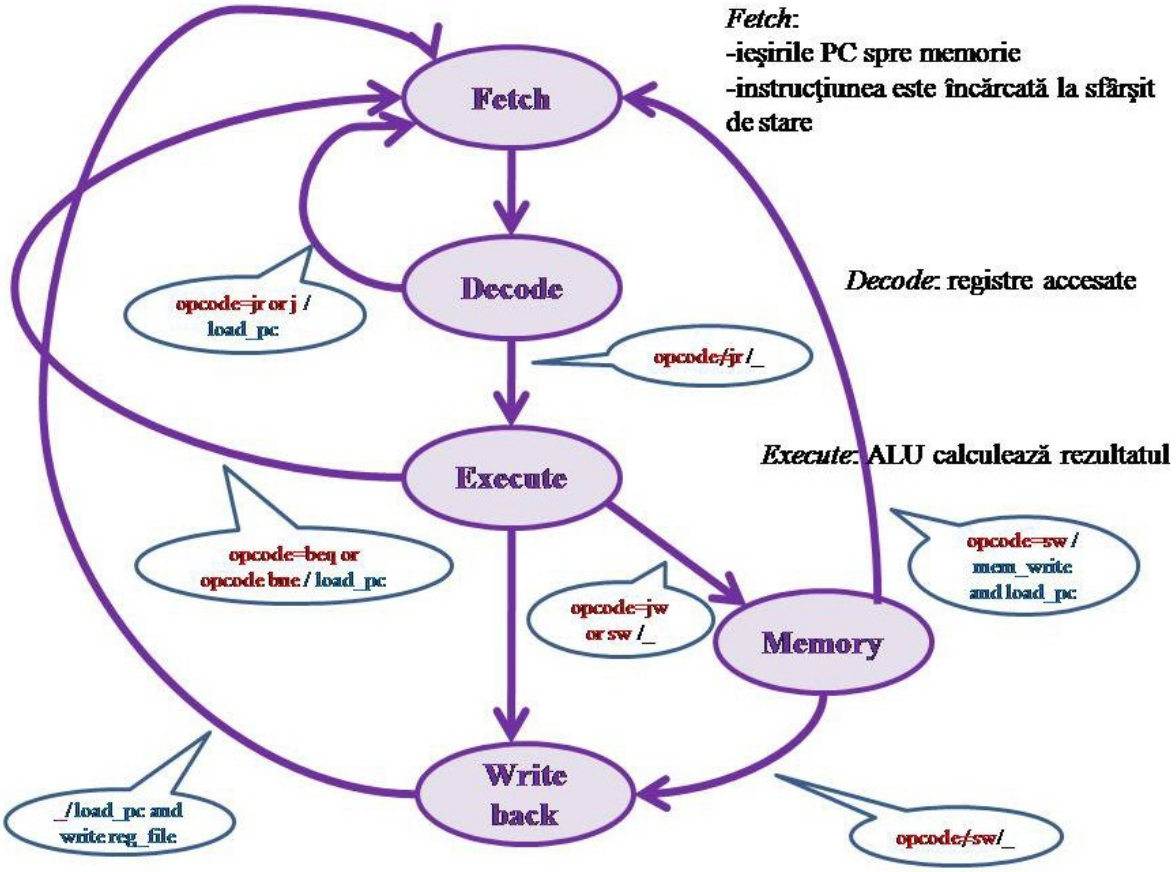


Figura 3-5 MIPS secvenţial

Faza *IF*

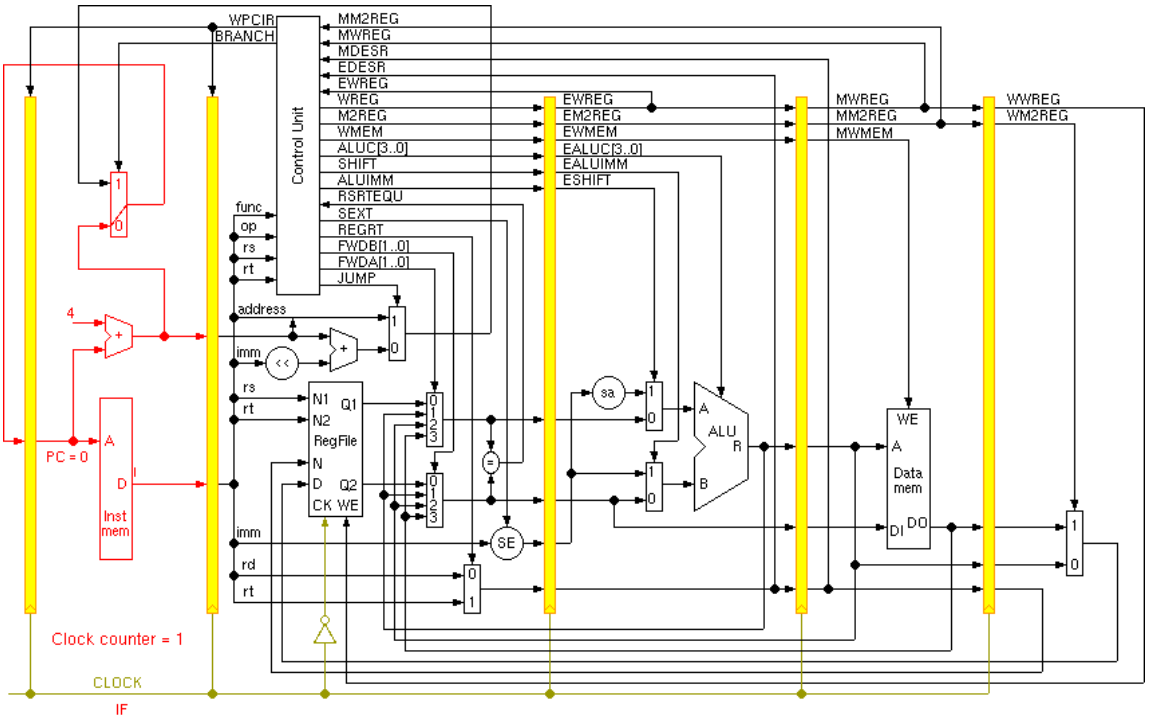


Figura 3-6 Faza *IF* (*sursa:* [5])

Faza ***IF*** trebuie să finalizeze următoarele operaţiuni:

1. Calculează adresa instrucţiunii următoare pentru registrul *Program Counter* (***PC***);
2. Citeşte instrucţiunea din memoria de instrucţiuni.

Faza *ID*

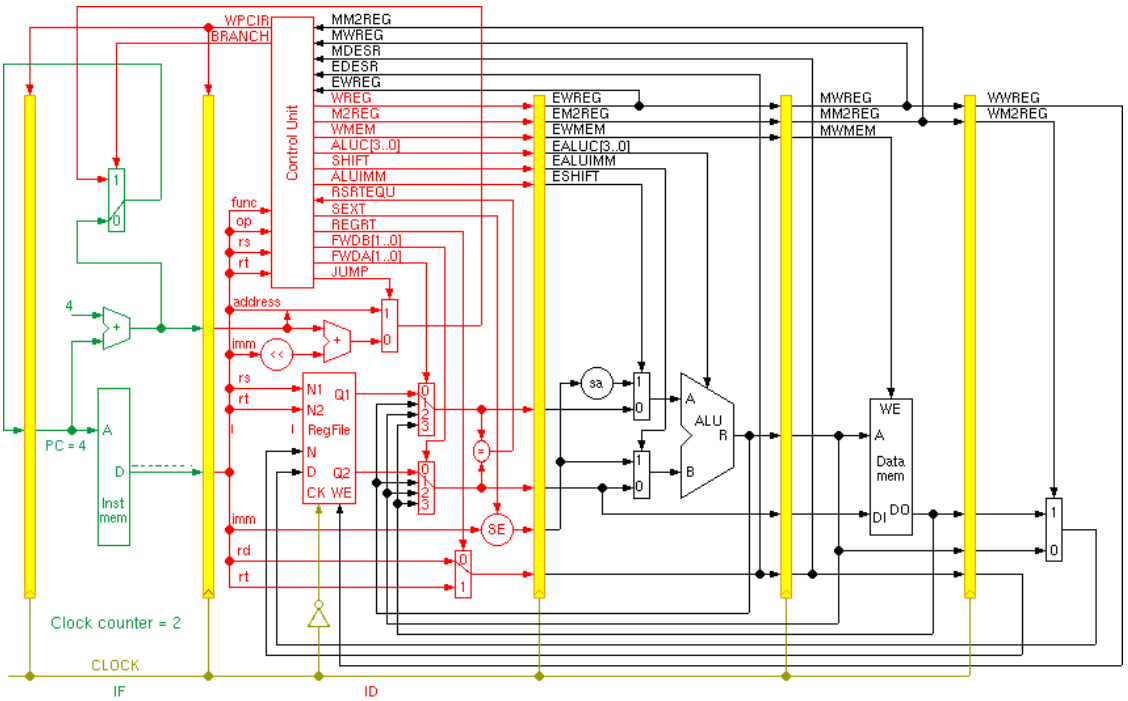


Figura 3-7 Faza *ID* (*sursa:* [5])

Faza ***ID*** este responsabilă pentru următoarele task-uri:
1. Decodarea instrucţiunii;
2. Gestionarea hazardului de date;
3. Aprecierea condiţiilor pentru o instrucţiune *branch*;
4. Extragerea operanzilor din fişierul de regiştri.

Faza *EX*

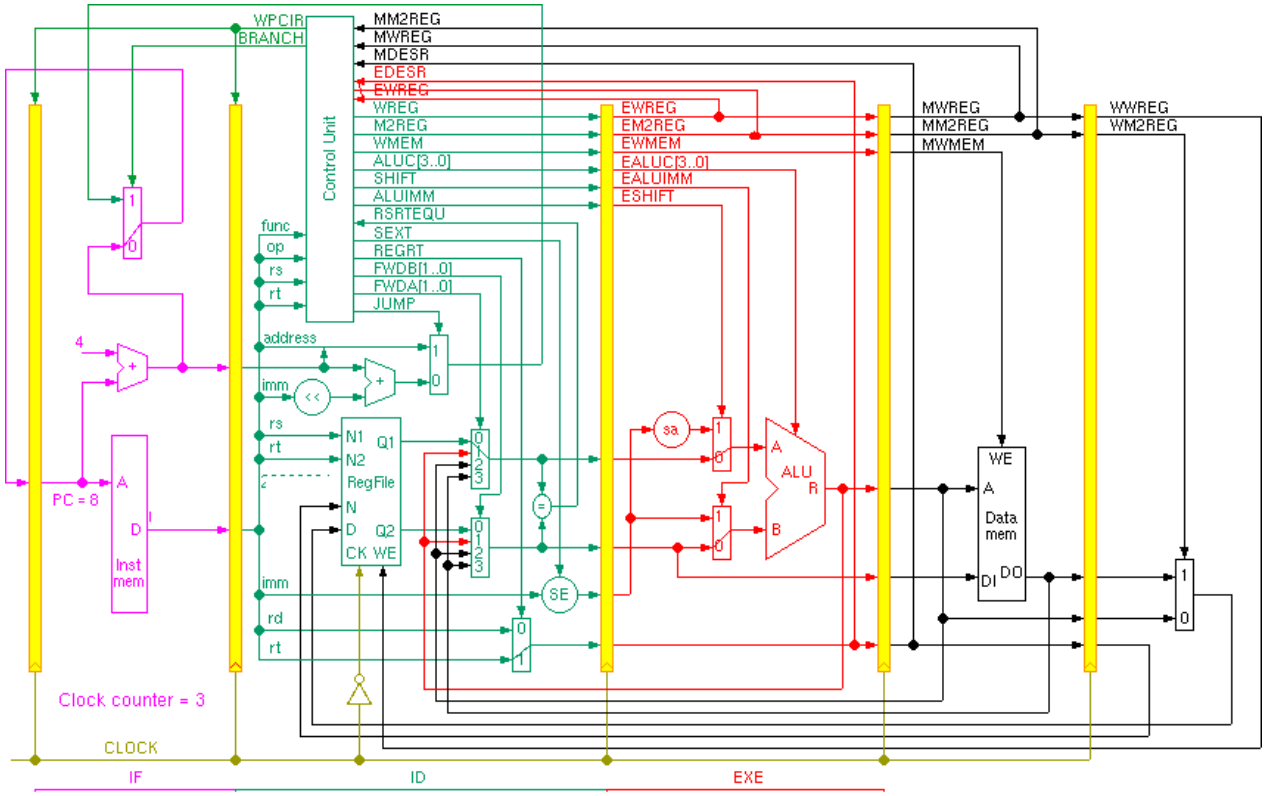


Figura 3-8 Faza *EX* (*sursa:* [5])

Faza ***EX*** urmează după faza ***ID***. Principalele task-uri pentru faza ***EX*** sunt:
1. Foloseşte ALU pentru a realiza operaţiile aritmetice/logice;
2. Salvează rezultatul calculelor în registrul *backup*;
3. Dă variate semnale de control.

Faza *MEM*

Faza ***MEM*** are următoarele task-uri principale:
1. Dă diverse semnale de control pentru accesarea memoriei şi finalizează task-ul de accesare a memoriei;
2. Salvează citirea datelor de la memorie la registrul *backup* astfel încât să poată fi folosite de faza ***WB***.

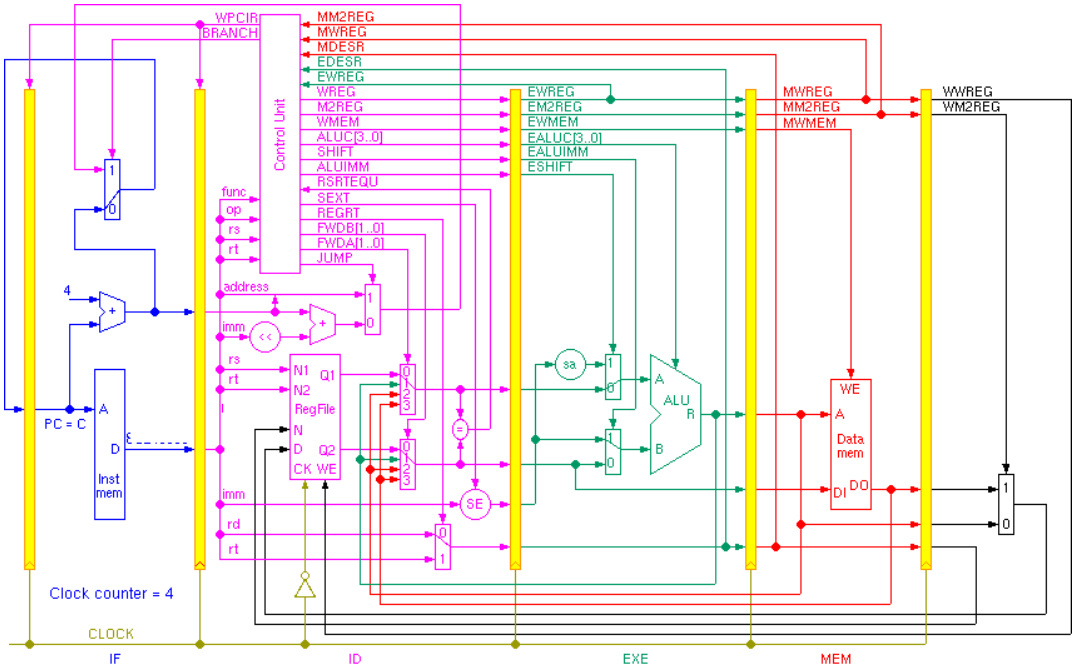


Figura 3-9 Faza *MEM* (*sursa:* [5])

Următoarele semnale sunt necesare la accesarea memoriei:
DATAO[31..0]: Date de ieşire din CPU.
DADDR[31..0]: Adresă pentru accesarea memoriei.
WRITEMEM: Controlul validării scrierii la memorie.

Faza *WB*

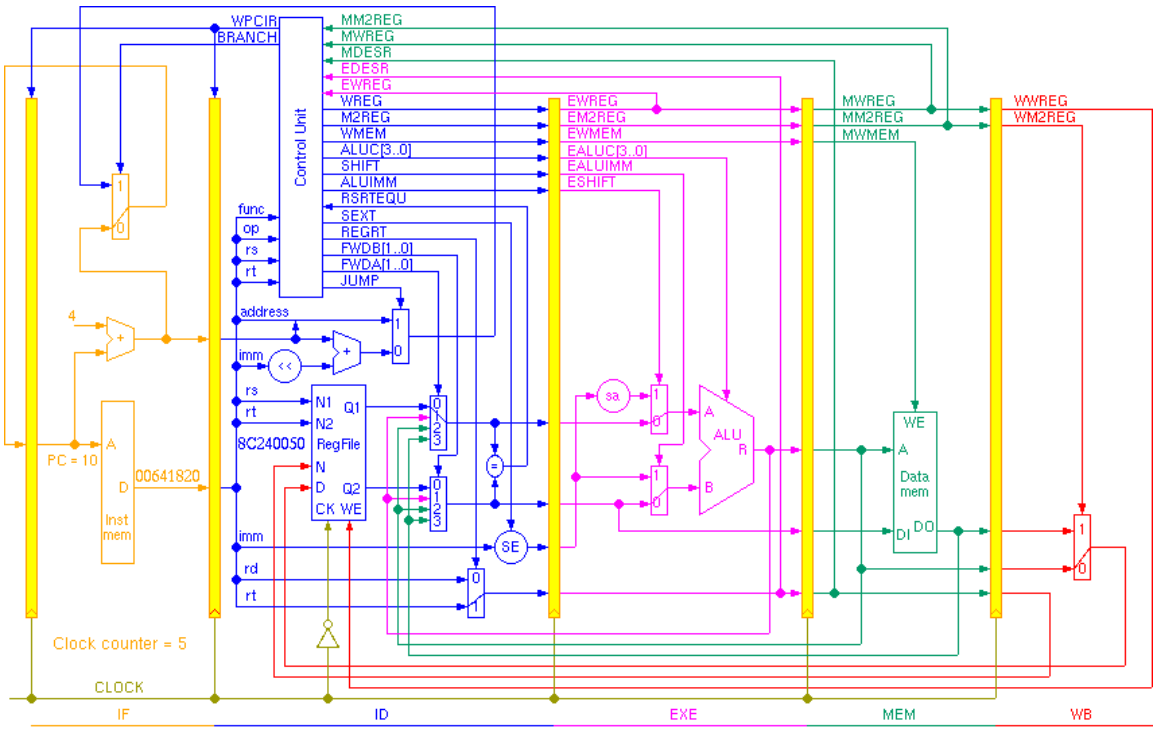


Figura 3-10 Faza *WB* (*sursa:* [5])

Numai task-ul pentru faza ***WB*** este pentru scrierea rezultatului calculului în registrul fişier. Pentru aceasta sunt necesare următoarele semnale:

1. *RESULT[31..0]*: Rezultatul final al execuţiei unei instrucţiuni;
2. *CONTROLW_MEM[13..9]*: Indexul registrului a fi scris;
3. *CONTROLW_MEM[8]*: Controlul permiterii scrierii de către registrul fişier.

Etapa *Instruction Fetch* este acolo unde un contor de program va pune următoarea instrucţiune de la locaţia corectă în memoria de program. În plus, contorul de program a fost actualizat cu următoarea locaţie a instrucţiunii în mod secvenţial sau locaţia instrucţiunii ca determinată printr-un *branch*.

Etapa *Instruction Decode* este acolo unde unitatea de control determină ce valori trebuie setate pentru liniile de control în funcţie de instrucţiune. În plus, unitatea de detectare a hazardului este implementată în această etapă şi toate valorile necesare sunt aduse de la bank-urile de regiştri.

Etapa *Execute* este acolo unde instrucţiunea este de fapt trimis la **ALU** şi executată. Dacă este necesar, şi locaţiile *branch*-ului sunt calculate în acest stadiu. În plus, acesta este stadiul în care unitatea de forward va determina dacă ieşirea **ALU** sau a unităţii de memorie ar trebui să fie transmise la intrările **ALU**.

Etapa *Memory Access* este acolo unde, dacă este necesar, memoria sistemului este accesată pentru date. De asemenea, în cazul în care o scriere a memoriei de date este solicitată de instrucţiune, aceasta se face în acest stadiu. Pentru a evita complicaţiile suplimentare se presupune că o singură citire sau scriere este realizată într-un singur ciclu de ceas al CPU.

În cele din urmă, etapa *Write Back* este acolo unde toate valorile calculate sunt scrise înapoi în registrele lor corespunzătoare. Scrierea înapoi în bank-ul de registru are loc în prima jumătate a ciclului în scopul de a evita hazardele structurale şi de date.

CPU include o *unitate de detectare a hazardului* pentru a determina când trebuie să se adauge un blocaj de ciclu. Datorită avansului (*forwarding*) de date, acest lucru se va întâmpla doar atunci când o valoare este utilizată imediat după ce a fost încărcată din memorie sau atunci când se produce un *branch*. Unitatea de detectare a hazardului prezintă Program Counter-ul din actualizarea cu următoarea sa valoare calculată, şterge registrele *Instruction Fetch* şi transmite un *NOP* prin restul benzii de asamblare.

Tabel 3-6 Intrările unităţii de forward (*sursa:* [5])

if_id_instr	vector(31 downto 0)	full 32 bit instruction in the id stage
id_ex_instr	vector(31 downto 0)	full 32 bit instruction in the ex stage
if_id_Rs	vector(4 downto 0)	first parameter in id stage
if_id_Rt	vector(4 downto 0)	second parameter in id stage
id_ex_Rs	vector(4 downto 0)	first parameter in the ex stage
id_ex_Rt	vector(4 downto 0)	second parameter in the ex stage
ex_mem_Rd	vector(4 downto 0)	result register in the mem stage
ex_mem_memtoReg	std_logic	mux control for register file in mem stage
ex_mem_regWr	std_logic	write enable for register file in mem stage
mem_wb_Rd	vector(4 downto 0)	result register in wb stage
mem_wb_memtoReg	std_logic	mux control for register file in wb stage
mem_wb_regWr	std_logic	write enable for register file in wb stage
mem_wb_readMem	std_logic	load signal in wb stage

Tabel 3-7 Ieşirile unităţii de forward (*sursa:* [5])

ctrl_A	vector (1 downto 0)	mux control for parameter A in ex stage
ctrl_B	vector (1 downto 0)	mux control for parameter B in ex stage
mem_to_mem	std_logic	mux control for memory data input
branch_ctrl_A	std_logic	mux control for parameter A in branch logic
branch_ctrl_B	std_logic	mux control for parameter B in branch logic

Un exemplu de cod VHDL [5] pentru unitatea de forward este prezentat în fig. 11 din Anexa 1.

Tabel 3-8 Intrările unităţii de detecţie a hazardului (*sursa:* [5])

Signals	*Size*	*Description*
if_id_instr	vector(31 downto 0)	Instruction in the ID stage
id_ex_instr	vector(31 downto 0)	Instruction in the EX
branch_taken	std_logic	Signal notifying if branch was taken or not
id_ex_memRead	std_logic	Indicates if lw is executing in ID/EX stage
mem_lw	std_logic	Lw indicator in MEM stage
ex_lw	std_logic	Lw indicator in EX stage
mem_lw_Rt	vector(4 downto 0)	Second parameter in lw instr in MEM stage
ex_lw_Rt	vector(4 downto 0)	Second parameter in lw instr in EX stage
id_ex_Rt	vector(4 downto 0)	Second parameter in ID/EX stage

Tabel 3-9 Ieşirile unităţii de detecţie a hazardului (*sursa:* [5])

Signals	*Size*	*Description*
id_ex_mux	std_logic	Control for ID/EX nop insert
if_id_flush	std_logic	Control for ID/ID flush
pc_stall	std_logic	Control for PC stall
id_stall	std_logic	Control for ID/EX stall
ex_stall	std_logic	Control for ex/MEM stall

Un exemplu de cod VHDL [5] pentru unitatea de detecţie a hazardului este prezentat în fig. 12 din Anexa 1.

Unitatea de control este necesară pentru a comuta operaţiunile **în** şi **între stadii**. Informațiile de control trebuie să fie parte a instrucțiunii deoarece aceste informații sunt necesare în diferite etape ale benzii de asamblare. Acest lucru poate fi realizat prin adăugarea inter-etape a mai multor biți registrului de stocare pentru a transmite datele de control care încă trebuie utilizate.

Un exemplu unitate de control [4] este prezentat în fig. 13 din Anexa 2.

Tabel 3-10 Tabelul de adevăr pentru funcţiile unităţii de control (*sursa:* [2])

Control	Signal name	R-format	lw	sw	beq
Inputs	op5	0	1	1	0
	op4	0	0	0	0
	op3	0	0	1	0
	op2	0	0	0	1
	op1	0	1	1	0
	op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Tabel 3-11 Tabelul de adevăr pentru funcţiile unităţii de control cu instrucţiunea *jump* (*sursa:* [2])

Control	Signal name	R-format	lw	sw	beq	Jump
Inputs	op5	0	1	1	0	0
	op4	0	0	0	0	0
	op3	0	0	1	0	0
	op2	0	0	0	1	0
	op1	0	1	1	0	1
	op0	0	1	1	0	0
Outputs	RegDst	1	0	x	x	x
	ALUSrc	0	1	1	0	x
	MemtoReg	0	1	x	x	x
	RegWrite	1	1	0	0	x
	MemRead	0	1	0	0	x
	MemWrite	0	0	1	0	x
	Branch	0	0	0	1	x
	ALUOp1	1	0	0	0	x
	ALUOp0	0	0	0	1	x
	Jump	0	0	0	0	1

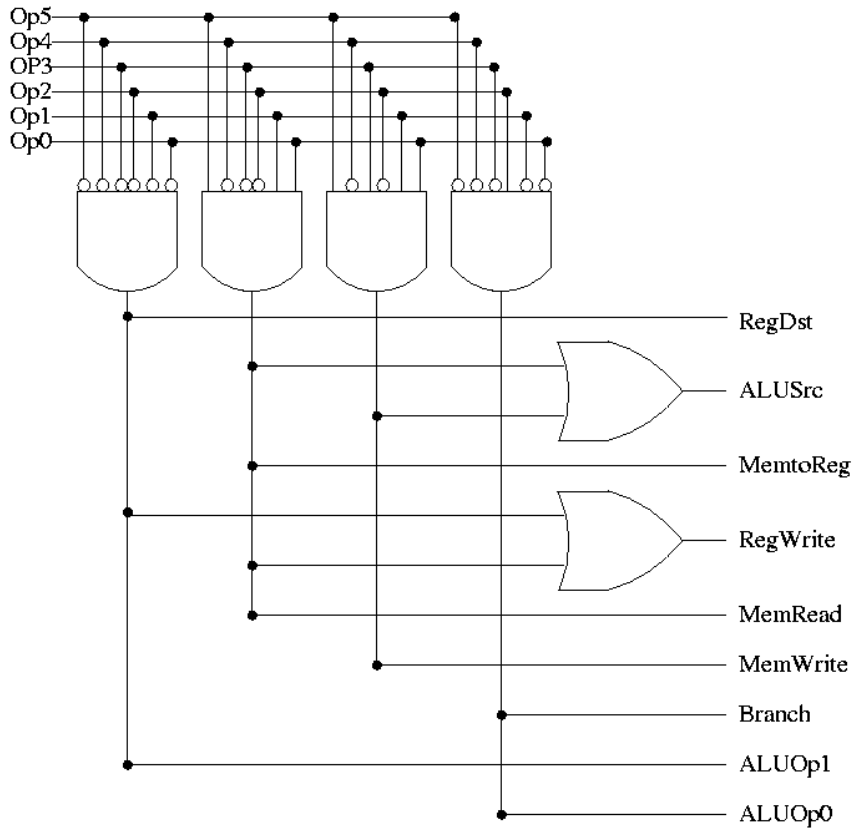


Figura 3-11 Implementare PLA (*sursa:* [2])

Un studiu detaliat al *unităţii de control principale* va fi prezentat în următorul raport de cercetare. Pentru simplificare se împarte unitatea de control principal în două părţi:

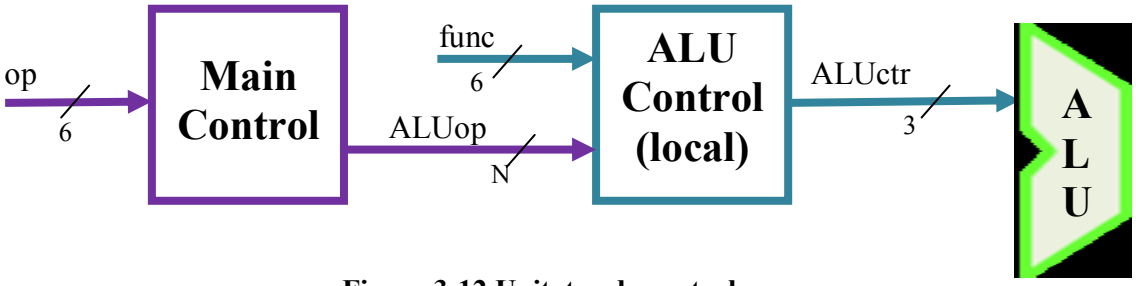
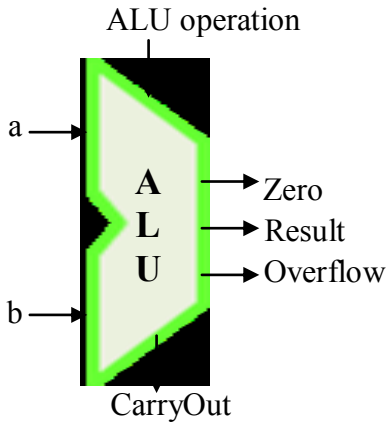


Figura 3-12 Unitatea de control

3.5. ALU şi ALU CONTROL

Unitatea logică aritmetică (ALU) este o parte esențială a procesorului. Aceasta efectuează operațiile aritmetice (adunare și scădere) și operații logice (AND, OR, XOR etc). Deplasarea și rotația task-urilor sunt efectuate, de asemenea, prin ALU. Designul conține două intrări de date pe 32 de biți, *alu_in1* și *alu_in2*, și o ieșire de transport, *carryout*. Cele cinci intrări de control ALUOP0, ALUOP1, ALUOP2, ALUOP3 și ALUOP4 decid care operație ar trebui să fie executată. Ieșirile includ un rezultat logic aritmetic pe 32 de biți și unul dintre cele trei *flag*-uri de 1-bit (*alu_out_pos*: pozitiv; *alu_out_neg*: negativ; *alu_out_zero*: zero). Cele mai multe operațiuni consumatoare de timp din ALU sunt adunarea și scăderea.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

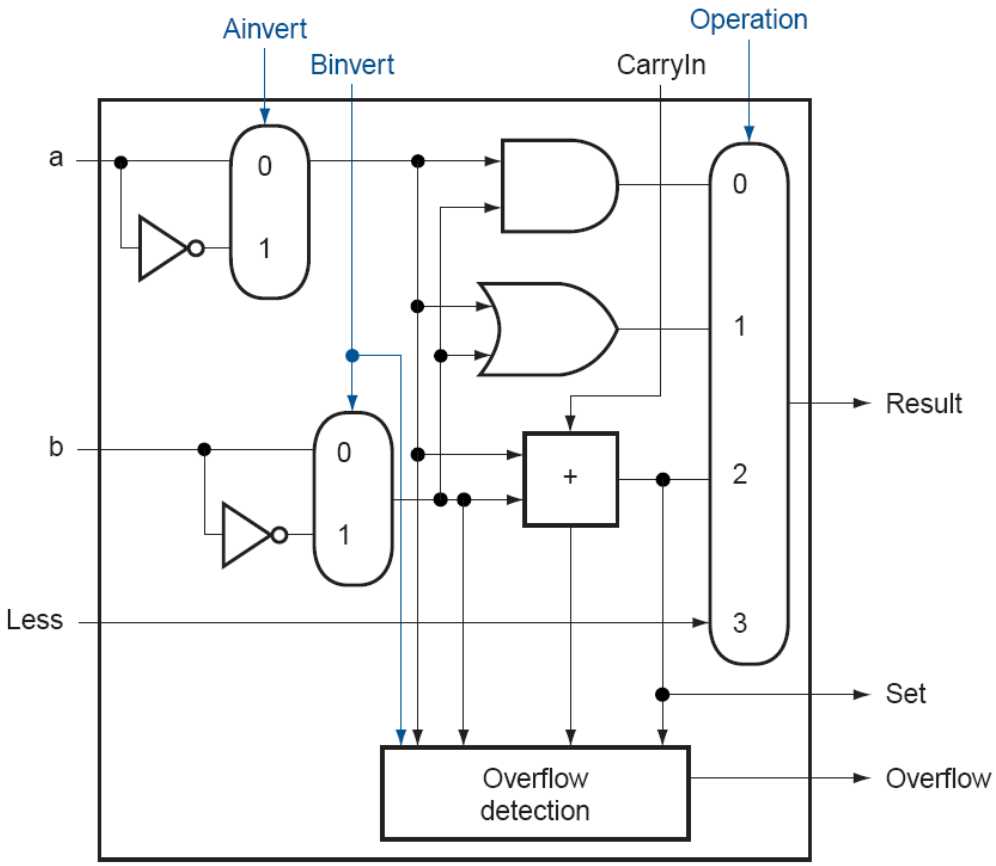


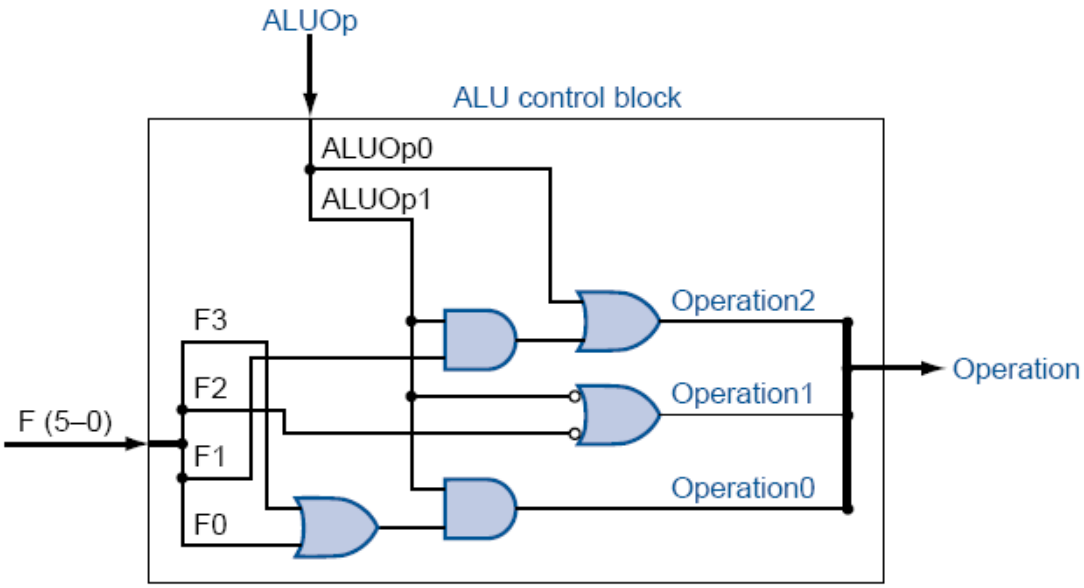
Figura 3-13 Unitatea ALU (*sursa:* [4])

Pentru o înţelegere aprofundată şi practică a procesorului MIPS pipeline, s-au implementat pe *Quartus II* (fig. 17) mai multe elemente şi instanţe având ca sursă [6]. Au fost analizate, corectate pentru a fi simulate şi, în continuare, vor fi exemplificate, pentru ALU – Anexa 3 **(**fig. 18), **ALU Out** – Anexa 4 (fig. 19 a-g) şi **ALU Control** – Anexa 5 (fig. 20, 21, 22, 23, 24, 25 a-f), secveţele VHDL şi desenele logice RTL (*Register Transfer Level*) corespunzătoare.

Figura 3-14 Blocul ALU Control

Instruction opcode	ALUOp	Instruction operation	Function field	Desired ALU action	ALU control input
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
branch equal	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set or less than	101010	set or less than	0111

(a)



(b)

Figura 3-15 ALU Control (*sursa:* [4])

4. STUDIU ASUPRA UNEI ARHITECTURI DE MICROCONTROLER ÎN SISTEME HARDWARE DE TIMP-REAL

4.1. INTRODUCERE

Un aspect important legat de Sistemele de Operare de Timp-Real (SOTR) este gestionarea întreruperilor, timer-elor, mutexes, watchdog timers, sincronizărilor şi comunicarii ca evenimente unitare.

Pentru a obține un timp de răspuns predictibil pentru diferite tipuri de evenimente care au loc simultan, este necesar să existe un mecanism de prioritizare pentru ele. Prin urmare, acest studiu propune un mecanism hardware pentru gestionarea evenimentelor enumerate mai sus, în scopul de a îmbunătăți soluția software, care nu este eficientă deoarece generează întârzieri. Metoda este implementată în arhitectura *n*-task-uri *MPRA* (*Multi Pipeline Register Architecture*), care este o arhitectură de microcontroler cu capacități de real-time implementată în hardware, care permite comutarea între task-uri în timp de un ciclu-procesor și un timp de răspuns la evenimente de până la 1,5 cicluri-procesor.

În cele mai multe sisteme de operare de timp-real (RTOS), tratarea întreruperii este implementată în software și astfel poate mări timpul de răspuns la evenimente externe și supraîncărca procesorul. Prin urmare, cele mai noi sisteme în timp real implementează în hardware tratarea întreruperilor pentru a elimina aceste două probleme. Prin analiza modelelor tradiționale de gestionare a întreruperilor, putem sublinia incapacitatea lor de a furniza determinismul temporal necesar în sistemele în timp real. În acest studiu se prezintă o metodă de gestionare a întreruperilor implementată în hardware, bazată pe o metodă care utilizează un spațiu unificat de priorități pentru task-uri și întreruperi, astfel încât nu există un controler de întreruperi special. Diferența majoră față de alte arhitecturi cu planificator hardware este că MPRA este o arhitectură multipipeline, ceea ce înseamnă că fiecare task are propriul set de registre pipeline.

Una dintre cerințele fundamentale ale RTS (Real-Time Systems) este determinismul task-urilor critice de timp-real. Rata de execuție a task-urilor, supratask-ul generat de sistemul de operare, precum și timpul pentru operațiunile de comutare ale contextului task-urilor sunt doar câţiva parametri care pot genera jitters şi ratarea deadline din RTS bazate pe planificatoare software.

Această parte a 4-a a lucrării de față prezintă câteva rezultate sintetizate ale cercetărilor efectuate pentru doctorat şi publicate în procedeeing-urile conferinţelor internaţionale la care a participat doctorandul.

În prima parte, ne propunem un sistem de întrerupere pentru un planificator în timp real implementat în hardware, care elimină supratask-ul generată de sistemul de operare și operațiile de comutare de context. Noutatea este metoda utilizată pentru selectarea task-urilor și gestionarea sistemului de întrerupere care nu are nevoie de un controler de întreruperi dedicat.

Cercetarea prezentată în această lucrare se bazează pe o arhitectură funcțională de procesor cu regiştri multipipeline (MPRA), prezentată în [7] şi [8], care prevede un timp foarte redus pentru operaţiunile de comutare de context ca urmare a arhitecturii multipipeline. Procesorul implementează o structură cu resurse multiplexate pentru regiştrii pipeline şi regiştrii fişier şi se bazează pe un planificator hardware integrat, care poate rula proprii algoritmi de planificare, cum ar fi EDF (*Earliest Deadline First*) sau RMA (*Rate Monotonic Algorithm*).

Planificatorul hardware este integrat în procesor, iar operațiile de comutare de context impun remaparea regiştrilor fişier şi a regiştrilor pipeline, ambele acțiuni fiind conduse de către unitatea HSE (Hardware Scheduler Engine) [9].

MPRA oferă un mecanism dinamic de gestionare a întreruperilor. În MPRA, întreruperile sunt tratate ca task-uri şi, pentru a garanta un sistem robust de planificare şi de a elimina inversiunile de prioritate [10], este necesară o asignare corespunzătoare a priorităților şi o evaluare corectă a problemei. În cele mai multe aplicații bazate pe microcontroler, o regulă prioritară permite ca toate evenimentele sa fie capturate şi tratate. Din cauza acestei reguli fixe, care nu se schimbă în timpul executării cererii, se poate întâmpla ca unele întreruperi cu prioritate scăzută atașate la task-uri cu prioritate scăzută să întrerupă task-uri cu prioritate mare, provocând apariţia jitter-elor lor. Pentru a elimina această problemă, MPRA atribuie priorități pentru întreruperi dintr-un grup comun de task-uri.

Dispozitivele FPGA (*Field Programmable Gate Array*) oferă elemente integrate cu complexitatea aplicaţiei de circuite integrate orientate (ASIC - *Application Specific Integrated Circuit*), cu avantajul de programabilitate sau, mai bine zis, de configurare [11]. Dispozitivele FPGA permit proiectarea de arhitecturi hardware specializate cu avantajul flexibilităţii mediului programabil ce poate fi implementat [12]. Pe baza acestor dispozitive, suporturile hardware pentru RTOS pot fi uşor puse în aplicare [13], [14]. În această lucrare, ne propunem un suport hardware pentru gestionarea predictibilă a întreruperilor.

Sistemele de timp-real (RTOS) existente, comerciale şi „free”, pentru sisteme embedded, nu pot permite unui task să se sincronizeze simultan cu mai multe evenimente utilizate pentru partajarea resurselor, sincronizare şi comunicare între task-uri, cum ar fi semafoare, „flag”-uri, mutex, semnale, evenimente, mesaje. Această problemă a fost identificată în RTOS care rulează pe arhitecturi de microcontrolere care nu au memorie cache şi unități de management de memorie virtuală. Astfel RTOS includ următoarele: FreeRTOS, uC-OS / II, KeilOS, µTKernel, µITRON, Hartik, XMK OS, EmbOS, SharcOS, Ecos, Erika şi Portos [20], [21], [22]. Procesoarele de uz general utilizate pentru sisteme embedded pot crea diferite probleme, în principal datorită consumului lor ineficient de energie şi performanţelor non-deterministe. Aceasta poate determina utilizarea unei platforme supradimensionată pentru a asigura comportamentul adecvat al sistemului în cea mai rea situaţie. Ca urmare, aceste procesoare nu sunt potrivite pentru sisteme embedded cu cerințe de consum scăzut de energie şi de capacitate de timp real. În prezent, dispozitivele FPGA [23], [24], [25] sunt larg răspândite, sunt mult mai ieftine şi au capacități mai mari, echivalentul a milioane de porți logice [26], [27], [28]. Din acest motiv, această lucrare propune un suport implementat hardware pentru prioritizarea şi tratarea evenimentelor. *Multi Pipeline Register Architecture* (*MPRA*) prezentată în [29] şi [30] a fost modificată în [31] şi transformat în *n*-task *MPRA* (*nMPRA*). Ideea de bază a fost de a replica registrele pipeline şi de a crea mai multe instanțe ale procesorului, numite *semi CPU*, pentru a avea un semi-procesor pentru fiecare task *i* ($sCPU_i$).

nMPRA constă dintr-o structură hardware originală utilizată pentru planificarea task-ului, static şi dinamic şi oferă managementul unitar al evenimentelor. Scopul a fost de a îmbunatăţi prin hardware performanţele RTOS pentru microcontrolere, pentru a comuta mai rapid între task-uri, pentru a îmbunătăți timpul de răspuns la evenimente externe, pentru a îmbunătăți comportamentul întreruperilor, care sunt tratate ca evenimente în acest caz şi pentru a oferi mai multe tipuri de bază de comunicare inter-task-uri (mesaje, mutex-uri etc.).

Astăzi, tot mai multe şi mai complexe sisteme se bazează, în întregime sau în termeni absoluți, pe controlul de către procesor. Domeniile de utilizare pot implica siguranța umană, impunând condiții stricte de timp. Prin urmare, dezvoltarea de mecanisme de timp-real şi

planificare a proceselor cu cerinţe de condiţii de timp este o provocare [32], [33]. În acest context, sistemele de operare trebuie să asigure mecanisme de planificare a task-urilor de timp-real pentru a se asigura îndeplinirea condiţiilor stricte de timp.

În sistemele în care resursele sunt limitate, situaţie găsită în cele mai multe sisteme de control digitale (sisteme integrate), o punere în aplicare eficientă a planificatoarelor este crucială [34]. Sistemele de operare de timp-real (RTOS) au fost proiectate pentru sisteme fizice de timp real, a căror funcţionare depinde nu numai de rezultatul logic al prelucrării, ci şi de momentul în care sunt produse rezultatele [35]. Astfel, timpul devine o coordonată esenţială, cu impact în toate fazele de dezvoltare şi exploatare a întregului sistem [36].

Un Real-Time System (RTS) este un sistem suficient de rapid pentru a garanta deadline-urile pentru cazul cel mai rău de operare [33]. Planificarea task-urilor se referă la găsirea de soluţii fiabile pentru alocarea procesorului pentru fiecare task, astfel încât să nu există nici o suprapunere în executarea lor în timpul funcţionării sistemului [33]. Un parametru care poate afecta performanţa unui RTS este o suprapunere generată de sistemul de operare. Operaţiunile de planificare şi de comutare de context pot influenţa în mod semnificativ deadline-ul în RTS critice. Acesta este motivul pentru implementarea în hardware a algoritmilor de planificare în loc de a folosi planificatorul software. O caracteristică importantă a sistemelor real-time este determinismul şi predictibilitatea task-urilor critice de timp-real. Suprapunerea generată de operaţiunile de comutare de context a task-ului şi rata de execuţie a task-ului sunt doar doi factori care pot cauza jitter-e şi deadline-uri lipsă în RTS cu planificatoare software.

Gaitan şi colab. a propus o arhitectură hardware de planificator [36] integrat în structura unui procesor cu resurse multiplicate, numite registre pipeline. Ei au propus, de asemenea, câteva arhitecturi de planificatoare hardware originale, statice şi dinamice, cu management unificat pentru diferite tipuri de evenimente, acces la resurse partajate, generarea şi transferul de mesaje între task-uri care asigură sincronizarea şi comunicarea şi o metodă de ataşare a întreruperii la task-uri, ceea ce permite controlul comportamentului lor.

Arhitectura *Multi Pipeline Register* (*MPRA*) foloseşte un planificator hardware care este o parte componentă a procesorului, iar controlul acestuia este prin instrucţiuni dedicate, care sunt transmise prin pipeline. Salvarea de context a task-ului în arhitectura clasică de procesor implică salvarea registrelor procesorului în memorie sau pe stivă şi se poate produce jitter şi astfel poate fi afectat timpul de răspuns al RTS critice. În procesorul *MPRA*, registrul *Program Counter* (*PC*), regiştrii pipeline şi registrele generale sunt resurse multiplicate, iar memoria necesară pentru punerea în aplicare a acestor registre este direct proporţională cu numărul de task-uri din sistem. În scopul de a asigura predictibilitatea RTS, *MPRA* pune în aplicare un sistem de planificare rigidă, care se bazează pe codarea adresei de prioritate în *Hardware Scheduler Engine* (*HSE*). Cu toate acestea, lasă suficientă libertate utilizatorului de a pune în aplicare algoritmi doriţi de planificare, ca *Earliest Deadline First* (*EDF*) sau *Rate Monotone Algorithm* (*RMA*), printr-o ordonare corectă a task-urilor în funcţie de frecvenţa de execuţie sau deadline-uri. Registrul fişier al *MPRA* are o implementare specială care, în plus de remaparea contextului sub controlul direct al HSE, permite izolarea apelurilor de procedură, astfel încât utilizatorul nu trebuie să salveze contextele.

Arhitectura descrisă utilizează numai structura organizatorică a arhitecturii *MIPS* (*Microprocessor without Interlocked Pipeline Stages*). Arhitectura include un planificator funcţional într-un bloc funcţional al procesorului şi oferă posibilitatea de a schimba contextul task-urilor în doar o jumătate de ciclu de ceas, eliminând astfel dezavantajele planificatoarelor software. Performanţa *MPRA* nu constă în puterea de procesare, ci în viteza de comutare a contextelor task-urilor şi viteza de execuţie a algoritmului de planificare. Studiile extind ideile

de bază exprimate în [37], [38], definind o nouă funcționalitate originală pentru *nHSE* și *nMPRA*.

Scopul acestei lucrări este de a extinde în continuare soluția prezentată în [36] prin îmbunătățirea performanțelor și predictibilitatea sistemului de întreruperi și de tratare a evenimentelor. Noutatea prezentată în această lucrare se referă la implementarea hardware a mecanismului de selectare a evenimentelor folosind registre-capcană, la arhitectura registrului *PC* și introducerea instrucțiunii *ret_esr* (*return from the event service routine*), pentru a permite gestionarea automată a evenimentelor. Soluția pentru sistemul de întreruperi prezentată în această lucrare are un grad ridicat de flexibilitate și, spre deosebire de soluțiile software, furnizează același timp de răspuns pentru toate întreruperile și evenimentele. Soluția este aplicabilă pentru microcontrolere mici.

Partea a 4-a a lucrării este structurată după cum urmează: secțiunea II prezintă *nMPRA*, secțiunea III prezintă arhitectura *nHSE* şi tratarea întreruperilor, în secţiunea IV este prezentată tratarea hardware a evenimentelor, prioritizarea globală a evenimentelor pe categorii, folosind un sistem de prioritizare hardware, secțiunea V prezintă analiza evenimentelor *syn* şi *mutex*, în secţiunea VI este prezentată arhitectura PC_i modificată.

4.2. *nMPRA*

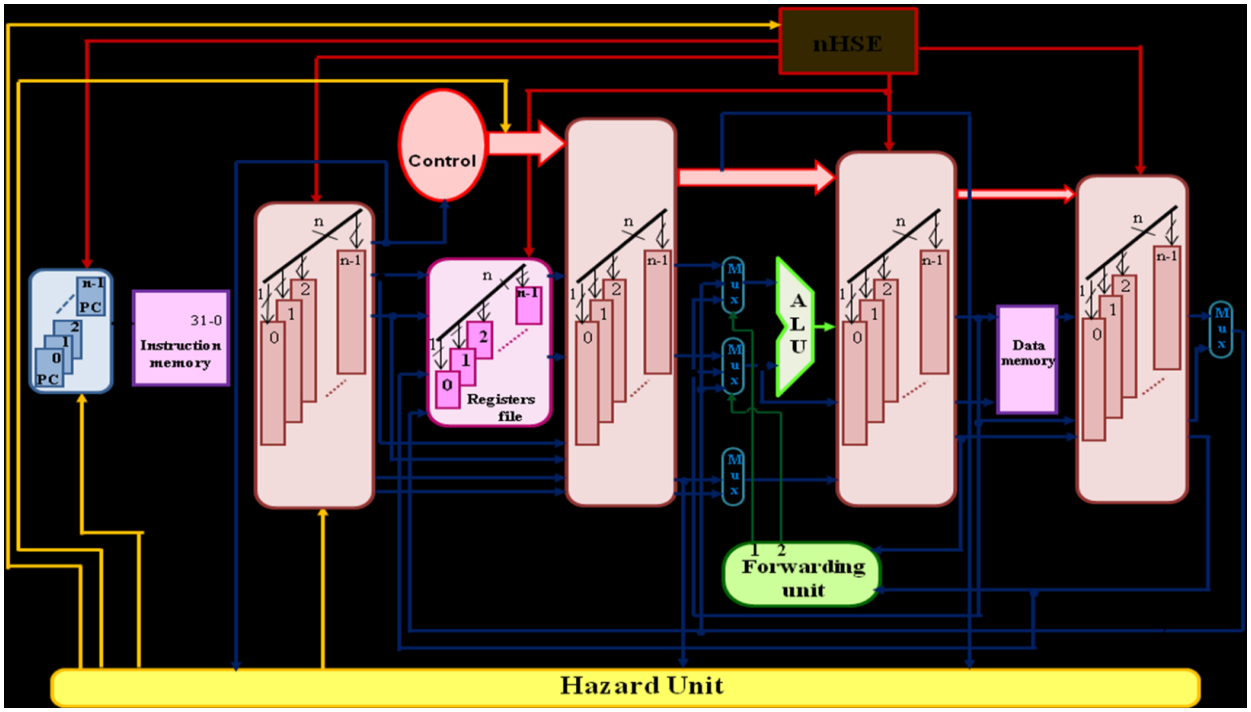


Figura 4-1 *nMPRA* (*sursa*: [41])

Arhitectura *nMPRA* este prezentată în Fig. 1. *MPRA* a fost transformată în *nMPRA*, adică a fost multiplicată de *n* ori; pentru fiecare task avem câte un set de regiştrii pipeline (IFID, IDEX, EXMEM, MEMWB), un Program Counter (PC) și un Banked Register File. Datorită faptului că fiecare task are propriul set de registre pipeline și propriul set de registre generale comutarea de context se poate face într-un ciclu procesor, iar răspunsul la un eveniment extern poate fi întârziat maximum 1,5 cicli procesor. Din aceste motive arhitectura este foarte rapidă. Celelalte resurse sunt folosite în comun de către toate taskurile. O instanță a acestui procesor o

vom numi „semi CPU” ($sCPU_i$) pentru task-ul i ($i=0, ..., n-1$) şi va executa un singur task ($task_i$).

Toate $sCPU_i$ sunt identice cu excepția $sCPU_0$ care va fi singura activă după reset, singura care va executa instrucțiuni supervizor și singura care va avea acces la *registrele de monitorizare* ale *nMPRA*. $sCPU_0$ are întotdeauna prioritatea 0 și este cea mai prioritară *sCPU* [31].

Noua arhitectură are un planificator static şi unul dinamic. Planificatorul static este preemptiv și atribuie priorități statice pentru toate task-uri. Planificatorul poate efectua task-ul de comutație rapidă la apariția unui eveniment. Toate evenimentele asociate unui task vor avea prioritatea task-ul, chiar dacă se schimbă dinamic.

Nu este un controler specializat pentru întreruperi, dar *nMPRA* oferă o structură distribuită care permite de a atribui o întrerupere la un task RTOS. Programatorii pot modifica prioritatea întreruperilor prin atribuirea lor către un alt task [9].

Planificatorul în *nMPRA* monitorizează constant toate evenimentele care se adresează la $sCPU_i$. Lista posibilelor evenimente aşteptate constă în: evenimente timer (TEv_i), evenimente watchdog timer ($WDEv_i$), două evenimente deadline ($D1Ev_i$ şi $D2Ev_i$, primul fiind echivalent cu o alarmă și al doilea fiind echivalent cu o avarie), întreruperi atașate task-ului i ($IntEv_i$), mutex-uri utilizate pentru tratarea resurselor partajate ($MutexEv_i$), evenimente de sincronizare și de comunicare între task-uri ($SynEv_i$) şi semnalul de execuţie auto-susţinere pentru $sCPU_i$ curent ($lr\_runs\_CPU_i$).

Ori de câte ori un eveniment este tratat și sursa sa este ștearsă, actualul $sCPU_i$ poate pierde controlul asupra procesorului. Evenimentele enumerate mai sus pot fi validate cu ajutorul următoarelor semnale: lr_enT_i, lr_enWD_i, lr_enD1_i, lr_enD2_i, lr_enInt_i, $lr\_enMutex_i$ şi lr_enSyn_i, care sunt grupate într-un registru special, numit Task Register (TR_i). Singura excepție este $lr\_run\_sCPU_i$. Semnalele rezultate lr_TEv_i, lr_WDEv_i, lr_D1Ev_i, lr_D2Ev_i, lr_IntEv_i, $lr\_MutexEv_i$, lr_SynEv_i şi $lr\_run\_sCPU_i$ sunt grupate într-un registru numit Event Status Task Register ($ESTR_i$), care poate fi accesat pentru a vedea ce evenimente aşteptate de task-ul i au apărut.

4.3. ARHITECTURA nHSE ŞI TRATAREA ÎNTRERUPERILOR

În arhitectura *MPRA*, planificatorul hardware *HSE [36]* este inclus în procesor și, prin urmare, nu necesită timp suplimentar pentru arbitrarea magistralelor, nici întârzieri ale rezultatelor din cauza transferului de date între planificator și procesor și poate fi controlat direct de instrucțiunile transmise pipeline. Pentru că avem mai multe registre pipeline, se poate provoca o izolare a contextelor hardware. Arhitectura a fost proiectată ținând cont de posibilitățile aplicabilității practice, astfel că ea poate fi foarte ușor integrată în microcontrolere. Întreruperile sunt considerate evenimente care pot fi atașate task-urilor sau la sistemele de operare în timp-real și sunt tratate ca thread-uri [39], nu ca întreruperi în maniera clasică.

nHSE (fig. 2) are la intrare evenimente (întreruperi, deadline, timer-e watchdog, timer-e, mutex-uri, evenimente mesaj, ca şi semnale de validare pentru planificatoarele statice și dinamice și inhibarea executării instrucțiunilor de încărcare și de memorare) utilizate pentru a genera semnalele de activare a $sCPU_i$ [39].

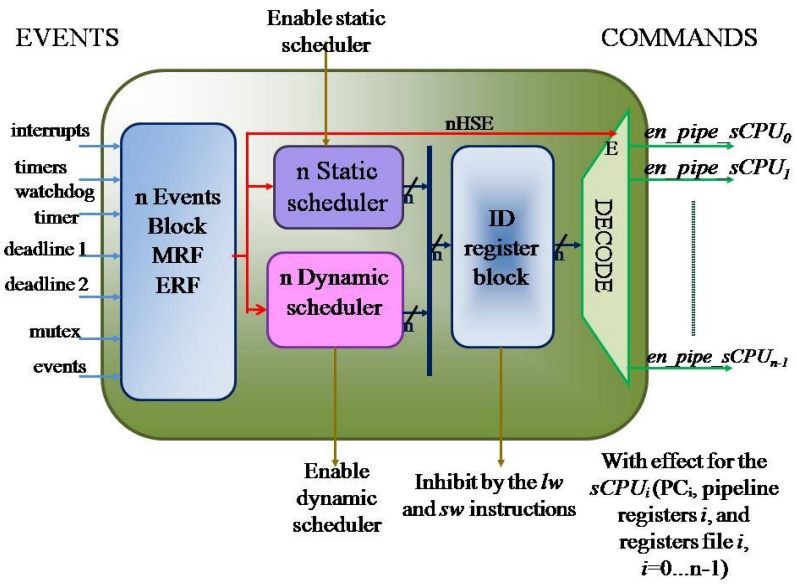


Figura 4-2 Arhitectura *nHSE* (*sursa*: [39])

Arhitectura *nHSE* este ilustrată în fig. 2. În soluția propusă, întreruperile sunt evenimente care pot fi atribuite nucleului de timp real sau task-urilor. Nucleul trebuie să fie un monitor în timp real. După ce a intrat în monitor, nu mai poate fi întrerupt (întreruperile sunt dezactivate). Ca urmare, funcțiile de monitorizare trebuie să fie scurte, fără interacțiuni și pauze. Întreruperile atașate unui task pot întrerupe numai task-uri cu prioritate strict mai mică, care sunt în stare de rulare.

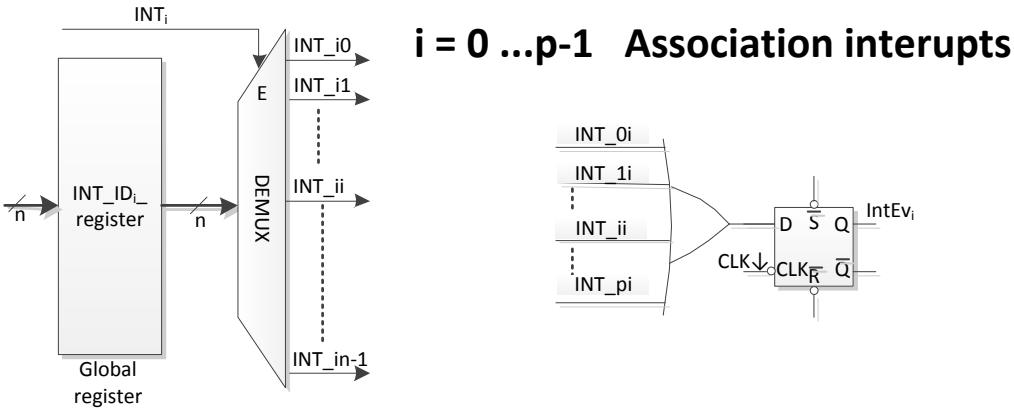


Figura 4-3 Asocierea întreruperilor la *sCPU*$_i$ (task *i*) (*sursa*: [9])

Să presupunem că există *p* întreruperi în sistem. Pentru fiecare întrerupere a sistemului, există un registru global, numit *INT_ID*$_i$*_register*, cu *n* biți utili care memorează ID-ul task-ul asociat întreruperii. Întreruperea activată *INT*$_i$ (fig. 3) validează demultiplexorul *DEMUX* care, la rândul său, va activa unul dintre semnalele *INT_i*$_0$ ... *INT_i*$_{n-1}$. Poarta *SAU* poate colecta toate întreruperile din sistem. Toate întreruperile pot fi atașate la *sCPU*$_i$ dacă toți *p* regiștrii *INT ID*$_i$ *register* (*i = 0, ..., p-1*) sunt setați cu valoarea *i*. De asemenea, nici o întrerupere nu poate fi atașată la *sCPU*$_i$ dacă niciunul dintre cei *p* regiștrii *INT_ID*$_i$*_register* (*i = 0, ..., p-1*) nu este scris cu valoarea *i*.

Bistabilul *D* sincronizează apariția aleatorie a evenimentelor, cum ar fi întreruperea *INT*$_i$ produce evenimentul *IntEv*$_i$ (fig. 3) și este înregistrată pe frontul descrescător al ceasului de sistem.

Această schemă propusă are unele caracteristici puternice și interesante: nu există un controler special de întreruperi; întreruperile iau prioritatea task-urilor ($sCPU_i$); unui task îi poate fi atașat nici unul, unul, mai multe sau toate cele *p* întreruperi; toate întreruperile atașate la aceeași task au aceeași prioritate; o întrerupere atașată la un task poate întrerupe doar task-urile cu cea mai mică prioritate; o întrerupere poate fi atribuită unui singur task; o întrerupere poate fi văzută ca un task; toate întreruperile pot fi atribuite unui singur task; o întrerupere nu resetează pipeline-ul altui $sCPU_i$; nu necesită salvarea și restaurarea de context; întreruperile pot fi *nested* (imbricate).

Presupunem că dispozitivele care generează întreruperi au un bit pentru a semnala starea de întrerupere, un bit de validare a întreruperii și un bit de ștergere a întreruperii. Analizând schema de întreruperi, am observat următoarea problemă: ce se întâmplă dacă toate întreruperile sunt atașate la același $sCPU_i$ și au loc simultan? În această parte vom prezenta două soluții la această problemă:

1. Soluția software este prezentată în fig. 4. Este simplă (nu are nevoie de module hardware suplimentare) și versatilă deoarece prioritățile întreruperilor pot fi schimbate cu ușurință. Unul dintre dezavantaje este întârzierile introduse de blocurile de test și de rutinele de tratare a întreruperilor în cazul în care mai multe întreruperi sunt atașate aceluiași $sCPU_i$ și au loc simultan. În plus, întârzierea generată de blocurile de test depinde de poziția întreruperii în blocul de test.

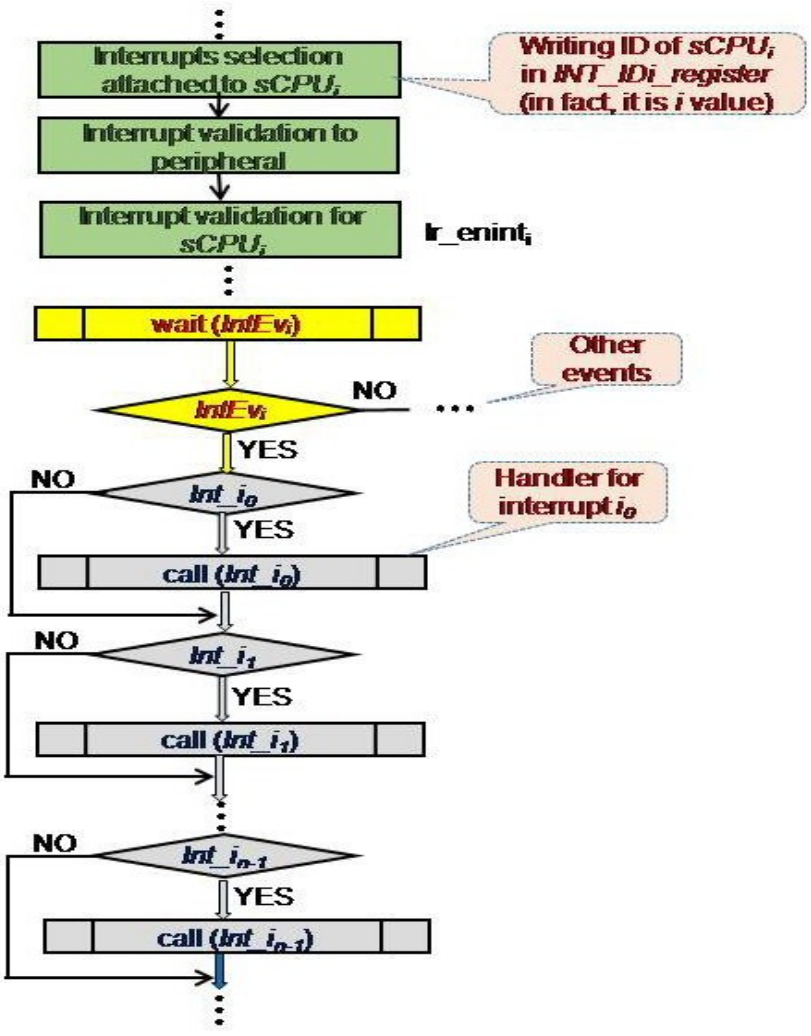


Figura 4-4 Soluția software (*sursa*: [39])

2. O altă soluție implică un bloc hardware suplimentar așa cum se arată în fig. 5.

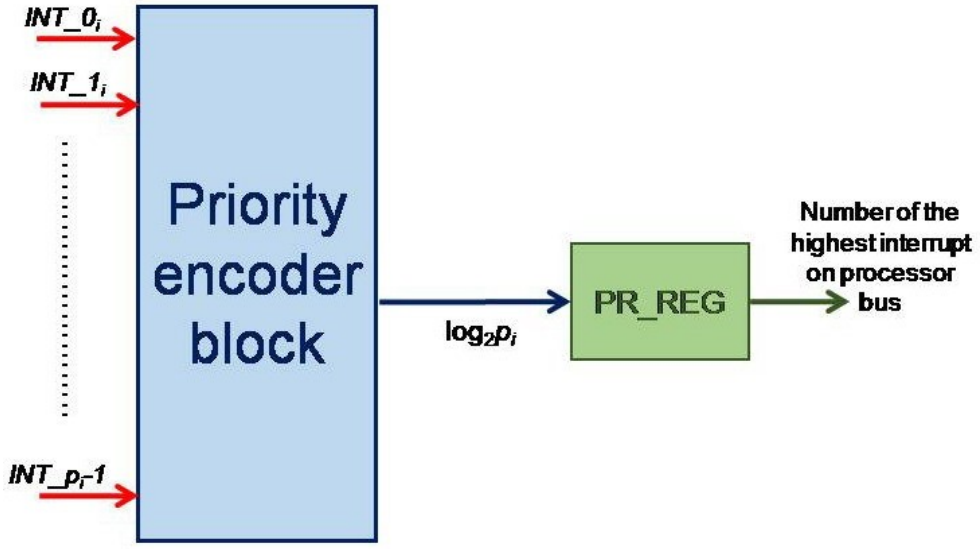


Figura 4-5 Blocul hardware adiţional (*sursa*: [39])

La apariția uneia sau a mai multor întreruperi, blocul codificator de prioritate va genera un număr corespunzător întreruperii de cea mai înaltă prioritate. Acest număr se înmulțește cu 4 pentru a calcula deplasamentul într-un tabel cu celule-capcană pentru întreruperi (fig. 6). Se citeşte de acolo adresa handler-ului pentru întreruperi şi i se transferă acestuia controlul. De data aceasta, pentru fiecare întrerupere, întârzierea dată de blocul de decizie va fi aceeaşi. Soluţia este mai rapidă (fig. 7), dar necesită un bloc hardware suplimentar [39], a cărui complexitate este dată de numărul total de întreruperi din procesor; ca exemplu, este prezentată în fig. 8 tabela de adevăr, ecuaţiile de funcţionare şi codul VHDL pentru un priority encoder pentru $p_i=4$ (a) şi pentru $p_i=8$ (b) (INT_0_i este cea mai mare întrerupere).

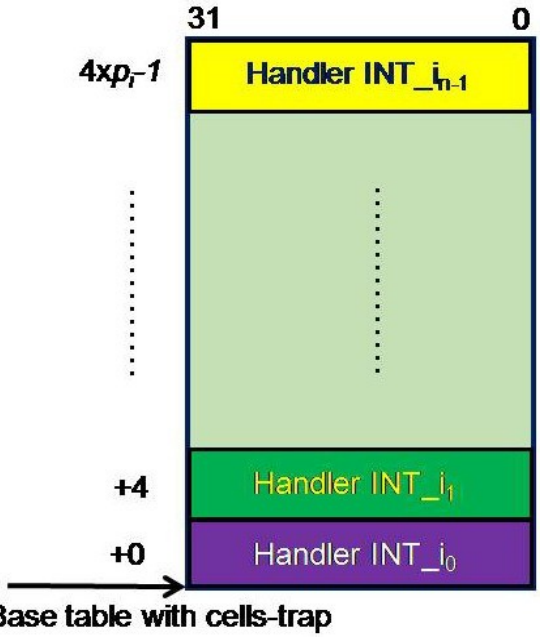


Figura 4-6 Tabel cu celule capcană (*sursa*: [39])

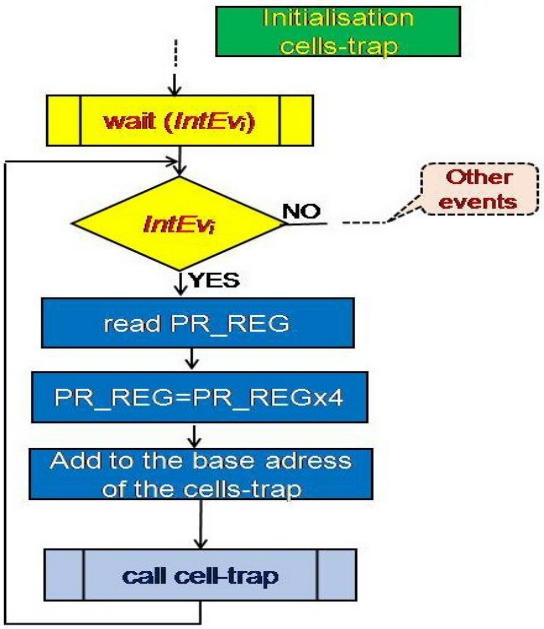


Figura 4-7 Soluţia hardware (*sursa*: [39])

INT_0_i	INT_1_i	INT_2_i	INT_3_i	pr_1	pr_0	lr_enint_i
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

$$pr\_1 = INT\_0_i + INT\_1_i$$
$$pr\_0 = INT\_0_i + INT\_2_i \cdot \overline{INT\_1_i}$$
$$lr\_enint_i = INT\_0_i + INT\_1_i + INT\_2_i + INT\_3_i$$

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY priority_encoder_4_2 IS
PORT (INT_ : IN  STD_LOGIC VECTOR (3 DOWNTO 0);
pr_ : OUT        STD_LOGIC VECTOR (1 DOWNTO 0);
              lr_enint_i : OUT    STD_LOGIC);
END priority_encoder_4_2;

ARCHITECTURE arch OF priority_encoder_4_2 IS BEGIN
pr_ <= "11" WHEN INT_(0) = '1' ELSE
"10" WHEN INT_(1) = '1' ELSE
"01" WHEN INT_(2) = '1' ELSE
        "00";
lr_enint_i <= '0' WHEN INT_ = "0000" ELSE '1';
END arch;
```

(a)

INT_0_i	INT_1_i	INT_2_i	INT_3_i	INT_4_i	INT_5_i	INT_6_i	INT_7_i	pr_2	pr_1	pr_0	lr_enint_i
0	0	0	0	0	0	0	0	x	x	x	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	x	0	0	1	1
0	0	0	0	0	1	x	x	0	1	0	1
0	0	0	0	1	x	x	x	0	1	1	1
0	0	0	1	x	x	x	x	1	0	0	1
0	0	1	x	x	x	x	x	1	0	1	1
0	1	x	x	x	x	x	x	1	1	0	1
1	x	x	x	x	x	x	x	1	1	1	1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY priority_encoder_8_3 IS
PORT (INT_ : IN    STD_LOGIC VECTOR (7 DOWNTO 0);
pr_ : OUT          STD_LOGIC VECTOR (2 DOWNTO 0);
              lr_enint_i :    OUT    STD_LOGIC);
END priority_encoder_8_3;

ARCHITECTURE arch OF priority_encoder_8_3 IS BEGIN
pr_ <= "111" WHEN INT_(0) = '1' ELSE
"110" WHEN INT_(1) = '1' ELSE
"101" WHEN INT_(2) = '1' ELSE
"100" WHEN INT_(3) = '1' ELSE
"011" WHEN INT_(4) = '1' ELSE
"010" WHEN INT_(5) = '1' ELSE
"001" WHEN INT_(6) = '1' ELSE
        "000";
lr_enint_i <= '0' WHEN INT_ = "00000000" ELSE '1';
      END arch;
```

$$lr\_enint_i = INT\_0_i + INT\_1_i + INT\_2_i + INT\_3_i + INT\_4_i + INT\_5_i + INT\_6_i + INT\_7_i$$

$$pr\_0 = \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot \overline{INT\_2_i} \cdot \overline{INT\_3_i} \cdot \overline{INT\_4_i} \cdot \overline{INT\_5_i} \cdot \overline{INT\_6_i} \cdot INT\_7_i +$$
$$+ \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot \overline{INT\_2_i} \cdot \overline{INT\_3_i} \cdot INT\_4_i + \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot INT\_2_i + INT\_0_i$$

$$pr\_1 = \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot \overline{INT\_2_i} \cdot \overline{INT\_3_i} \cdot \overline{INT\_4_i} \cdot INT\_5_i$$
$$+ \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot \overline{INT\_2_i} \cdot \overline{INT\_3_i} \cdot INT\_4_i + \overline{INT\_0_i} \cdot INT\_1_i + INT\_0_i$$

$$pr\_2 = \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot \overline{INT\_2_i} \cdot INT\_3_i$$
$$+ \overline{INT\_0_i} \cdot \overline{INT\_1_i} \cdot INT\_2_i + \overline{INT\_0_i} \cdot INT\_1_i + INT\_0_i$$

(b)

Figura 4-8 Codificator de prioritate (a) $p_i = 4$, (b) $p_i = 8$

Soluția propusă în această lucrare este oarecum bazată pe „întreruperi ca threads", concept prezentat în [15]. În [16] autorii prezintă soluții software și hardware pentru a preveni supraîncărcarea cauzată de întreruperi. Modelul integrat propus în această lucrare poate gestiona această supraîncarcare prin diverse tehnici de planificare, cum ar fi utilizarea de servere sporadice [16]. Unele *RTOSs* dezactivează toate întreruperile externe și le tratează printr-un mecanism „polling" pe timer [17]. În [18], autorii au propus o metodă în care întreruperile sunt tratate ca „threads". Propunerea are ca scop creşterea scalabilităţii arhitecturilor de sistem multiprocesor orientate spre sistemele de operare a serverelor de reţea şi „firele" de întrerupere cu nivele de prioritate specifice. În [19], prioritățile întreruperilor pot fi dinamice (prin reatașarea la alte task-uri sau prin schimbarea priorităţii task-ului la care este atașată).

Arhitectura propusă folosește un spațiu unificat de priorități pentru task-uri și întreruperi și se bazează pe mecanismul de activare hardware a task-urilor. În arhitectura prezentată, ca o continuare a cercetării de la [7] și [8], întreruperile sunt tratate ca task-uri. De obicei, există situații de inversare a priorităţilor când task-urile de prioritate ridicată sunt suspendate de întreruperile atribuite task-urilor de prioritate joasă. Ordonarea task-urilor și priorităților în același spațiu de adrese are rolul de a elimina acest dezavantaj.

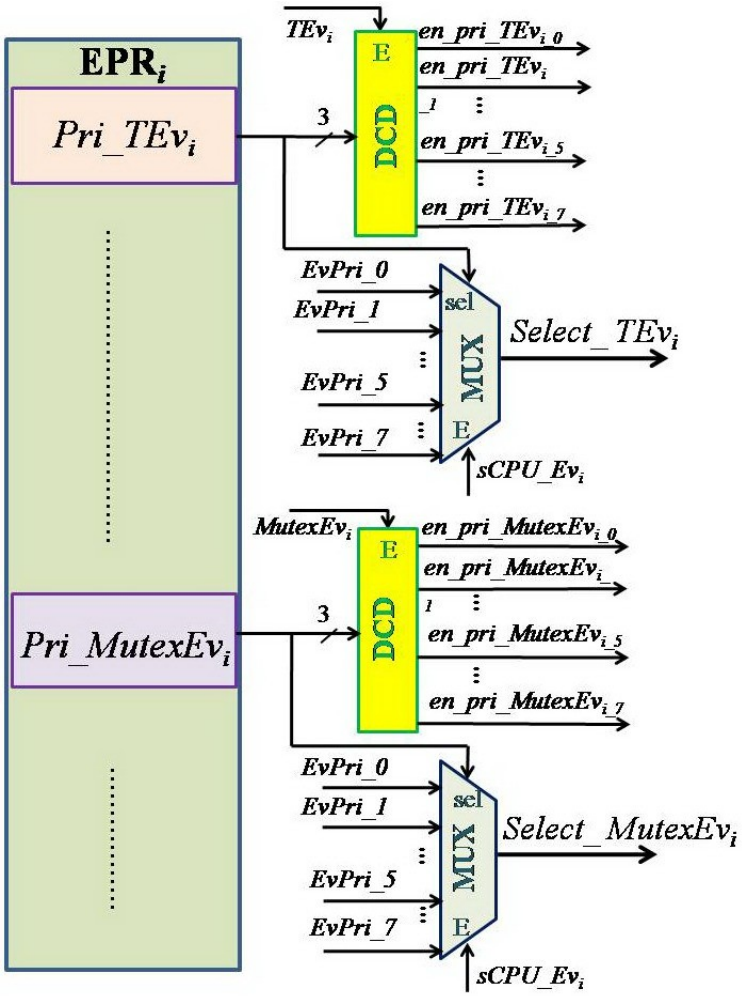
nHSE utilizează un spațiu unificat de priorități pentru întreruperi și task-uri și o regulă de planificare în care un task de mare prioritate nu poate fi întrerupt de întreruperile atribuite task-urilor mai puţin prioritare. Această regulă vine în sprijinul necesităţii de a asigura termenele limită de finalizare a execuţiei în task-urile care trebuie să ofere răspuns de timp real stimulilor externi. *HSE* oferă posibilitatea activării sau dezactivării sistemului de întreruperi. Deoarece întreruperile respectă acelaşi regim de execuţie ca şi task-urile, activarea sau dezactivarea execuţiei lor se face utilizând aceleaşi instrucţiuni care sunt adresate şi task-urilor. În *nMPRA*, fiecare timer (cronometru) asociat unui task poate fi configurat pentru a genera o întrerupere atunci când timpul alocat task-ului se apropie de finalizare. În arhitectura *MPRA* utilizatorul are opţiunea de a selecta modul în care întreruperile sunt tratate: externe task-urilor (situaţie în care întreruperile sunt văzute ca şi task-uri) sau ataşate acestora.

Cele mai moderne RTOS au implementate blocuri software separate de partajare a resurselor, de sincronizare și comunicare între task-uri, care nu pot fi evaluate decât secvenţial, dar nu şi simultan. De exemplu, nu poţi aştepta în acelaşi timp o întrerupere, eliberarea unui semafor şi primirea unui mesaj. Soluția propusă în acest studiu permite acest mod de funcționare prin implementarea în hardware, care permite o evaluare paralelă a evenimentelor din sistem.

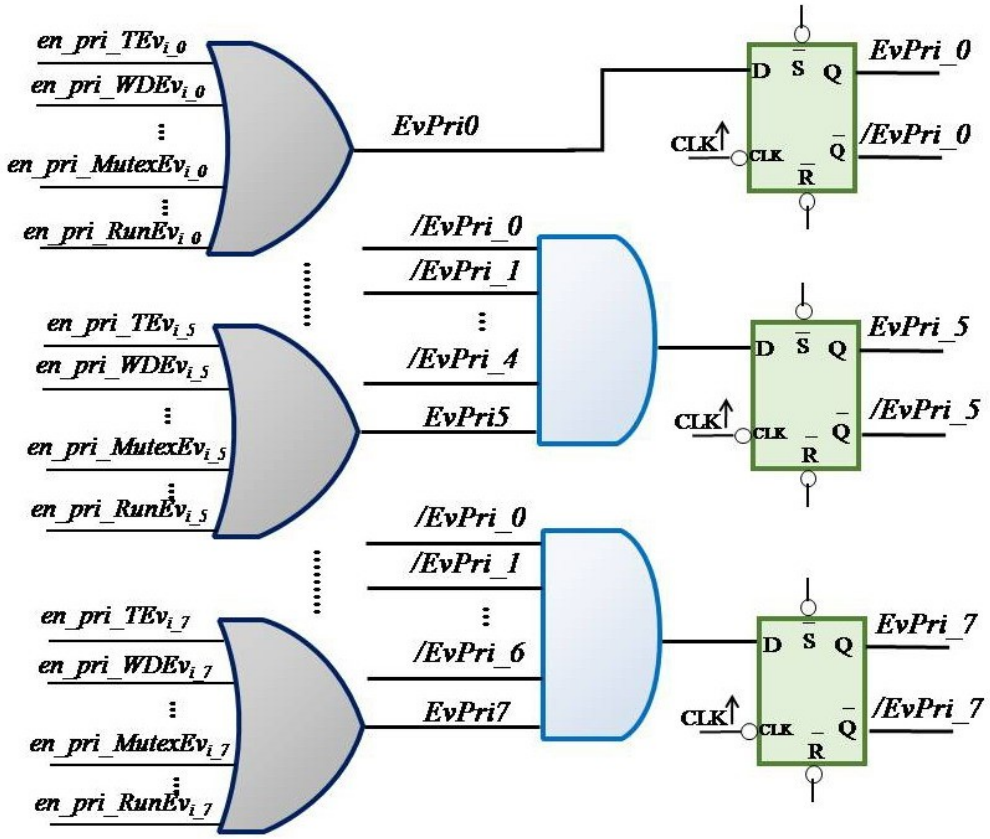
4.4. TRATAREA HARDWARE A EVENIMENTELOR

În caz de activare simultană a mai multor evenimente asociate cu un task ($sCPU_i$), trebuie să existe o modalitate de a selecta ordinea în care evenimentele sunt tratate. Pentru aceasta, fiecare $sCPU_i$ are atașat un *Event Priority Register* (EPR_i), care conține nivelul de prioritate al fiecărui tip de eveniment asociat cu acel $sCPU_i$. Prioritatea este diferită pentru fiecare tip de evenimente, de la 0 la 7 (există 8 tipuri de evenimente). După resetare, la pornire, *nMPRA* activează $sCPU_0$, care va executa tot software-ul de inițializare și secvențele de pornire pentru toate celelalte $sCPU_i$ ($i = 1 \ldots n-1$). $sCPU_0$ stabilește nivelul de prioritate pentru fiecare tip de eveniment asociat cu fiecare $sCPU_i$ conform cerințelor utilizatorului.

Fig. 9 prezintă o schemă de prioritizare care selectează tipul de eveniment activ curent cu cea mai mare prioritate, în scopul de a fi servit [41]. În funcție de cerințele utilizatorului, nivelul de prioritate al fiecărui tip de eveniment poate fi static și stabilit la începutul aplicaţiei (offline) sau poate fi modificat dinamic în timpul executării aplicaţiei (online). Dacă tipul selectat de eveniment conține mai multe evenimente active, trebuie să facem o altă selecție a evenimentului care va fi tratat în primul rând, în funcție de tipul evenimentului.



(a)



(b)

Figura 4-9 Schema globală de prioritizare a evenimentelor (*sursa*: [41])

Fig. 9 prezintă schema globală de prioritizare a evenimentelor. Având în vedere că în *nMPRA* avem opt tipuri de evenimente, vom avea nevoie de opt scheme de decodare şi de selectare a evenimentelor (fig. 9a). Priorităţile categoriilor de evenimente sunt grupate în registrul EPR_i (*Event Priority Register*) ce poate stoca priorităţile următoarelor tipuri de evenimente [34]: Pri_TEv_i, Pri_WDEv_i, Pri_D1Ev_i, Pri_D2Ev_i, Pri_IntEv_i, $Pri\_MutexEv_i$, Pri_SynEv_i, şi Pri_RunEv_i. Semnalul de activare a fiecărui tip de eveniment activează un decodor care generează prioritatea tipului de eveniment conform nivelul de prioritate memorat în EPR_i. Ieşirea câmpului de prioritate este utilizată şi pentru selectarea intrării active a multiplexoarelor *MUX*, care colectează rezultatul schemei de prioritizare prezentată în fig. 9b. Porţile *SAU* permit fiecărui tip de evenimente de a-şi selecta prioritatea, porţile *ŞI* validează o anumită prioritate, iar bistabilul *D* are rolul de a realiza sincronizarea cu ceasul sistemului.

Atunci când mai multe evenimente atribuite unui task ($sCPU_i$) sunt activate simultan, trebuie să găsim o metodă de a selecta ordinea de tratare (fig. 10, 11). După ce am selectat această ordine se obţine adresa rutinei de serviciu pentru eveniment asociată cu evenimentul selectat (rutina de serviciu pentru eveniment este executată de task-ul $sCPU_i$).

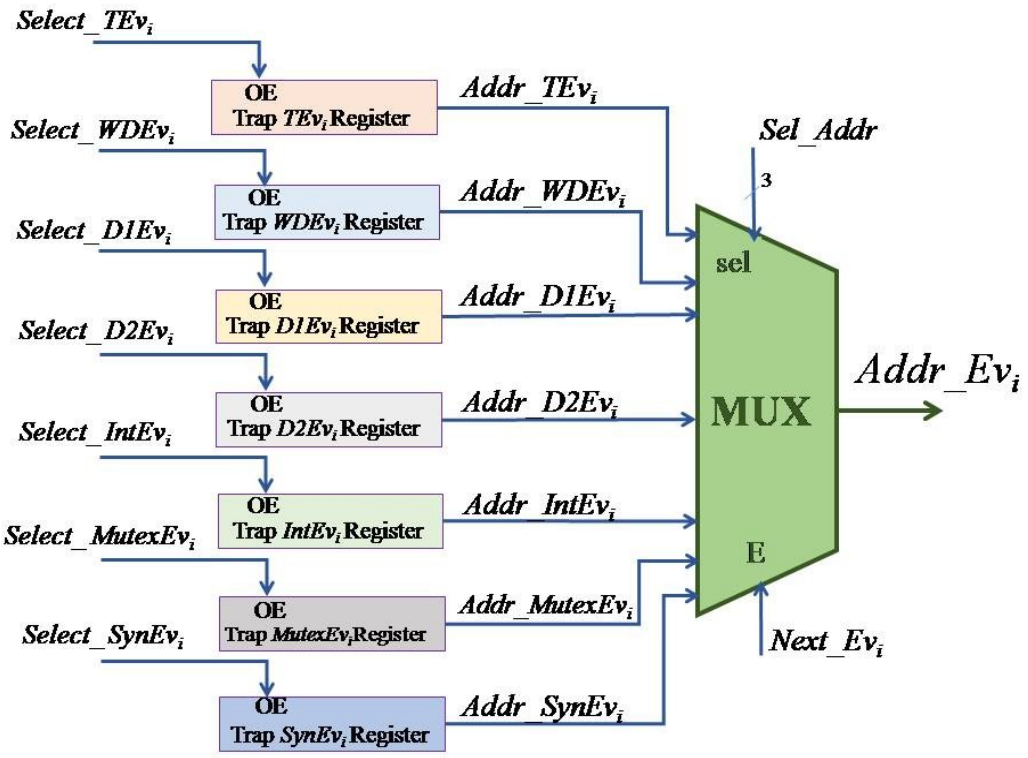
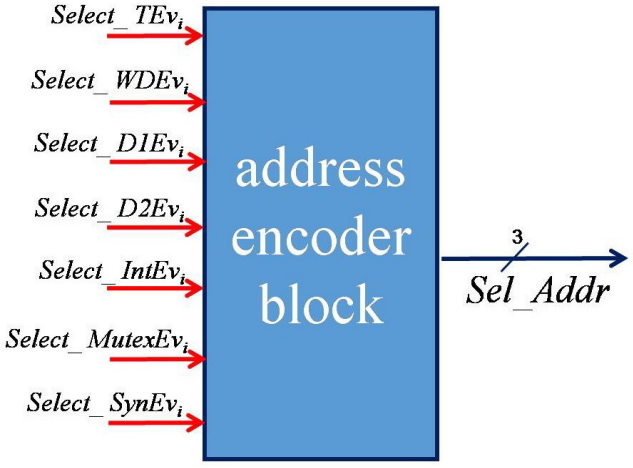


Figura 4-10 Selection of the highest priority event handler address (*sursa*: [41])



(a)

Select_TEv$_i$	Select_WDEv$_i$	Select_D1Ev$_i$	Select_D2Ev$_i$	Select_IntEv$_i$	Select_MutexEv$_i$	Select_SynEv$_i$	A_2	A_1	A_0	
1	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	1	0	2
0	0	1	0	0	0	0	0	1	1	3
0	0	0	1	0	0	0	1	0	0	4
0	0	0	0	1	0	0	1	0	1	5
0	0	0	0	0	1	0	1	1	0	6
0	0	0	0	0	0	1	1	1	1	7

(b)

$A_0 = Select\_TEv_i + Select\_D1Ev_i + Select\_IntEv_i + Select\_SynEv_i$
$A_1 = Select\_WDEv_i + Select\_D1Ev_i + Select\_MutexEv_i + Select\_SynEv_i$
$A_2 = Select\_D2Ev_i + Select\_IntEv_i + Select\_MutexEv_i + Select\_SynEv_i$

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY encoder_8_3 IS
PORT (Select_ : IN STD_LOGIC VECTOR (7
DOWNTO 0);
A : OUT  STD_LOGIC VECTOR (2 DOWNTO 0);
        END encoder_8_3;

ARCHITECTURE arch OF encoder_8_3  IS BEGIN
A <= "000" WHEN Select _(0) = "10000000" ELSE
"001" WHEN Select _(1) = "10000000" ELSE
"010" WHEN Select _(2) = "00100000" ELSE
"011" WHEN Select _(3) = "00010000" ELSE
"100" WHEN Select _(4) = "00001000" ELSE
"101" WHEN Select _(5) = "00000100" ELSE
"110" WHEN Select _(6) = "00000010" ELSE
            "111";
      END arch;
```

(c)

Figura 4-11 Schema de selecţie a adresei semnalelor

Toate evenimentele care sunt singurele din categoria lor, cum ar fi cele de timp (TEv_i, $WDEv_i$, $D1Ev_i$ and $D2Ev_i$) şi cele care sunt tratate la nivel global prin intermediul software-ului, au asociat un registru-capcană, care indică spre rutina lor de service a evenimentului. Adresele de rutină de serviciu a evenimentelor sunt încărcate pentru fiecare task într-un registru-vector de eveniment (fig. 12) la pornire, prin $sCPU_0$, după resetare.

address of the *Timer* event servicing routine	Trap TEv_i Register
address of the *WatchDog* event servicing routine	Trap $WDEv_i$ Register
address of the *Deadline1* event servicing routine	Trap $D1Ev_i$ Register
address of the *Deadline2* event servicing routine	Trap $D2Ev_i$ Register
base address of the trap-cells table for interrupts	Trap $IntEv_i$ Register
address of the *Mutex* events servicing routine	Trap $MutexEv_i$ Register
address of the *Syn* events servicing routine	Trap $SynEv_i$ Register

Figura 4-12 Registru-vector de eveniment (*sursa*: [41])

În cazul în care unul dintre aceste evenimente a avut loc, task-ul cu care acesta este asociat devine activ şi are cea mai mare prioritate printre evenimentele active, registrul Program Counter corespunzător $sCPU_i$ (PC_i) este încărcat automat cu conţinutul registrului-pointer, determinând executarea rutinei asociate cu evenimentul. Adresa de întoarcere, care a fost valoarea din registrul PC_i înainte de încărcarea adresei de rutină, se salvează automat într-un registru de rezervă în interiorul PC_i. După terminarea execuţiei rutinei de serviciu a evenimentului, dacă există alte evenimente active pentru task-ul *i*, adresa de rutină pentru evenimentul cu cea mai mare prioritate printre cele rămase va fi încărcată automat în PC_i. După servirea tuturor evenimentelor active, PC_i este încărcat cu adresa de revenire din registrul „backup".

Procedura descrisă mai sus este valabilă pentru fiecare eveniment din sistem. În secţiunea următoare vom analiza cazul evenimentelor *syn* şi *mutex*, care pot fi mai mult de unul din categoria lor.

4.5. ANALIZA EVENIMENTELOR *SYN* ŞI *MUTEX*

Evenimentele de sincronizare şi de comunicare între $sCPU_i$ ($SynEv_i$) nu pot fi tratate individual, din cauza metodei alese pentru punerea lor în aplicare, care utilizează un set de registre globale cu acces rapid.

Registrul *ERF* (Event Register File) este format din *s* registrer de $2n + k + 1$ biţi, memorând evenimentul însuşi pe cel mai semnificativ bit. Următorii *n* biţi reţin ID-ul task-ului sursă care a activat evenimentul, următoarii *n* biţi reţin ID-ul task-ului destinaţie pentru căruia i se adresează evenimentul, iar ultimii *k* biţi sunt utilizaţi ca un domeniu mesaj care poate fi utilizat pentru orice scop. Scrierea unui eveniment în *ERF* se va face la prima adresă liberă determinată de schema hardware. Când toate locaţiile sunt scrise, se activează semnalul *gr_en_mem_full*, indicând faptul că sunt utilizate toate evenimentele. Aceste evenimente sunt tratate în acelaşi mod ca evenimentele de timp, având asociat doar un singur registru-capcană care deţine adresa rutinei de servire.

Activarea semnalului $Select\_SynEv_i$ determină încărcarea automată a adresei de rutină în PC_i, salvând adresa de întoarce în registrul backup din interiorul PC_i.

În rutina de serviciu a evenimentelor *syn* va trebui căutat *ERF* şi citite toate evenimentele asociate cu task-ul *i* şi apoi să fie şterse. Prioritatea evenimentelor *syn* este dată de poziţia lor în *ERF*. Problema cu evenimentele *syn* este că acestea sunt scrise în *ERF* la prima adresaă liberă şi nu putem prezice ce eveniment va fi scris la o anumită adresă. Pentru a citi un eveniment *syn* din *ERF*, trebuie căutat folosind principiul *CAM* (*Content Addressable Memory*). Căutarea începe de la adresa *0* şi se termină la prima adresă pentru care există o concordanţă între ID-ul task-ului destinaţie şi ID-ul task-ului curent. Utilizând o instrucţiune de citire pentru a citi conţinutul registrului *ERF*, putem găsi, în cazul în care există o potrivire, emitentul evenimentului şi ce mesaj a fost trimis la task-ul curent [31]. Excluderea reciprocă este folosită ori de câte ori software-ul este utilizează resursele partajate.

În *nMPRA*, *mutexes* sunt implementate cu ajutorul unui set de registre globale cu acces rapid (se poate face în faza de execuţie - EX). Registrul *MRF* (*Mutex Register File*) este compus din *m* registre de lungime $n + 1$ biţi care conţin bit-ul *mutex* în poziţia cea mai înaltă şi ID-ul task-ului proprietar, care deţine *mutex* în cea mai mică prioritate a celor *n* biţi [31].

Registrele *MRF* pot fi accesate de orice $sCPU_i$, iar acest lucru înseamnă că sunt o resursă partajată pentru toate $sCPU_i$ din sistem. Odată ce un mutex blocat se eliberează, se activează semnalul $MutexEv_i$. Pentru fiecare $sCPU_i$ se poate decide ce *mutex* este luat în considerare cu ajutorul semnalelor $lr\_en\_M_0$, $\ldots lr\_en\_M_{m-1}$. Aceste semnale pot fi stocate în registrele locale, numite *Enable Mutex Register* (EMR_i). Pot fi unul sau mai multe EMR_i, în

funcție de numărul de biți *mutex* implementați în *MRF*. În general, vom folosi doar un singur resursă partajată la un moment dat. Din acest motiv, este ușor de a gestiona *mutexes* în software, fără a fi nevoie de a avea un sistem de prioritizare hardware implementat pentru selectarea *mutex*-ului curent.

Folosind o instrucțiune de test și setare pentru accesarea unui *mutex* se poate bloca task-ul curent dacă este luat *mutex*-ul, determinându-l să aștepte eliberarea *mutex*-ului. Din moment ce știm ce mutex ne așteptăm să fie eliberat, este ușor a selecta și a trata în software evenimentul *mutex* după producerea acestuia. Deci, semnalul $Select\_MutexEv_i$ are doar scopul pentru a indica faptul că *mutex* așteptat nu mai este luat.

Semnalul este folosit pentru a selecta registrul-pointer asociat cu *mutexes* și pentru a încărca conținutul acestora în registrul PC_i, determinând executarea rutinei de serviciu pentru *mutex*.

4.6. ARHITECTURA *PROGRAM COUNTER*

Atunci când are loc un eveniment, task-ul asociat devine gata pentru execuție și, dacă are o prioritate mai mare decât task-ul în starea de execuție, atunci registrul *Program Counter* corespunzător $sCPU_i$ (PC_i) (vezi fig. 13) este setat cu conținutul registrului-capcană (adresa rutinei asociate pentru evenimentul produs). Pentru a putea implementa redirectarea automată către rutinele de tratare a evenimentelor, PC_i trebuie modificat astfel încât să salveze intern adresa de revenire dintr-o rutină de tratare a unui eveniment [41].

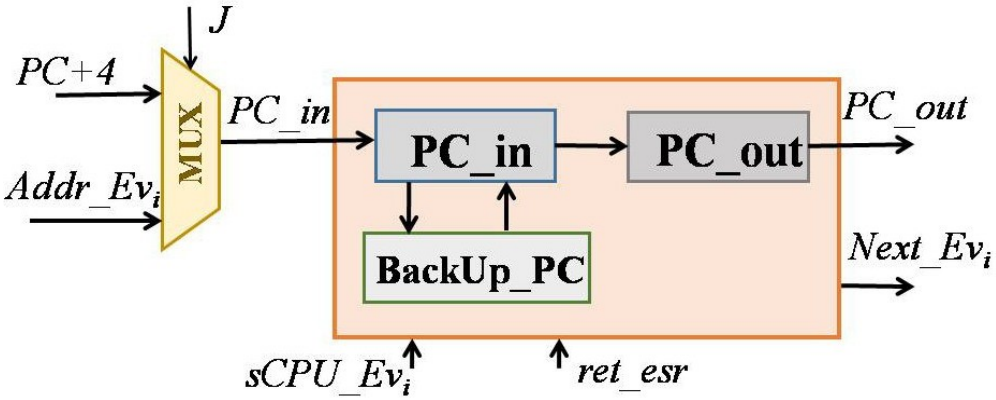


Figura 4-13 Arhitectura *Program Counter* (*sursa*: [41])

În interiorul PC_i vom avea un registru, numit *BackUp_PC*, în care se salvează adresa curentă din PC_i în momentul apariției unui eveniment ce trebuie tratat, semnalizat de semnalul $sCPU\_Ev_i$. În PC_i se va încărca automat, folosind registrul capcană corespunzător, adresa rutinei de tratare a evenimentului activ cel mai prioritar. Revenirea din rutina de tratare a evenimentului este semnalizată prin execuția instrucțiunii *retesr* (*return from the event service routine*), care detemină activarea semnalului *ret_esr* ce semnalizează PC_i să încarce adresa de revenire din registrul *BackUp_PC*, continuând astfel execuția normală a programului. Salvarea adresei de revenire în registrul *BackUp_PC* determină dezactivarea semnalului $NextEv_i$, indicând astfel că se tratează un eveniment și niciun alt eveniment nu mai poate fi tratat până când nu se termină tratarea evenimentului curent. După revenirea din rutina de tratare a evenimentului curent, semnalul $NextEv_i$ va fi reactivat pentru a semnala că se poate trece la tratarea evenimentului următor [41]. În fig. 13 este prezentată structura PC_i.

5. CONCLUZII

Arhitectura propusă constituie o metodă eficientă de optimizare a procesorului, având numeroase avantaje: reacția la un eveniment nu depășește 1,5 cicli de ceas procesor dacă evenimentul apărut este atașat unui task mai prioritar decât taskul curent; nu resetează banda de asamblare, nu necesită salvare/restaurare de context, accelerează execuția prin apeluri de subrutine cu copierea automată a parametrilor și comutarea setului de regiștrii, stivă locală de mare viteză; instrucțini puternice pentru partajarea resurselor, sincronizarea și comunicația între taskuri.

Rolul specific al STR este acela de a asigura controlul predictibil şi determinist al unui proces. STR sunt acele sisteme care oferă un răspuns corect într-un interval de timp prestabilit. Rapiditatea nu este o caracteristică specifică a STR, aceasta fiind mai mult un termen abstract. Viteza de răspuns la evenimente este însă un concept propriu STR diferit de cel menţionat anterior. Din acest motiv, şi probabil datorită unei imagini neclare asupra subiectului, unii ingineri au considerat că cercetarea în domeniul STR nu este una de viitor deoarece creşterea continuă a vitezei de lucru a procesoarelor va produce echipamente suficient de rapide pentru a satisface cerinţele celor mai exigente aplicaţii. În realitate situaţia este diferită deoarece viteza de execuţie a task-urilor nu implică şi garantarea unei scheme de planificare pentru toate seturile de task-uri din sistem.

STR joacă un rol important în societatea zilelor noastre deoarece majoritatea sistemelor utilizate pentru a uşura munca omului sunt controlate de microprocesoare.

Importanţa sistemelor de timp real creşte cu atât mai mult cu cât ele sunt folosite în controlul aplicaţiilor critice în care o funcţionare incorectă a sistemelor de control ar putea produce pagube materiale sau chiar pierderi de vieţi omeneşti. Se pot da ca exemplu aici sistemele de control din industria atomo-electrică, constructoare de maşini, aero-spaţială, auto etc. Domeniul de aplicare a STR este extins şi include o gamă largă de alte dispozitive în care răspunsul sistemului de control este valid şi după depăşirea unui termen limită. Majoritatea acestor aplicaţii care utilizează STR sunt implementate pe o diversitate largă de echipamente hardware în care sistemele embedded deţin cea mai ridicată pondere datorită dimensiunilor şi a scalabilităţii.

În arhitectura MPRA dimensiunea memoriei consumate pentru implementarea fişierului de regiştri este direct proporţională cu numărul de task-uri. Ar fi util ca această memorie să fie utilizată în schimb ca o memorie de uz general cu acces rapid.

6. CONTRIBUŢIILE DOCTORANDULUI

Prin cercetările prezentate în această lucrare am îmbunătăţit arhitectura CPU prezentată în [9] printr-o soluţie inovatoare pentru prioritizarea întreruperilor ataşate la aceeaşi task. Am propus o soluţie inovativă de ataşare a întreruperilor la task-uri (*sCPU*). În acest context, întreruperea împrumută prioritatea şi comportamentul task-ului. Astfel, comportamentul întreruperilor este mult mai predictibil în contextul unei aplicaţii de timp real (un task nu poate fi întrerupt decât de întreruperile ataşate unui task mai prioritar). Schema propusă are unele caracteristici puternice şi interesante, cum ar fi: nu trebuie să existe un controler specializat pentru întreruperi; întreruperile moştenesc prioritatea task-ului (*sCPU*); un task poate ataşa niciuna, una, mai multe, sau chiar toate cele *p* întreruperi din sistem; întreruperea ataşată unui task poate întrerupe un task mai puţin prioritar dar nu poate întrerupe execuţia taskului căruia îi este ataşată, sau un task mai prioritar; întreruperea poate fi ataşată unui singur task; întreruperea poate fi un task; toate întreruperile pot fi ataşate unui singur task;

întreruperea nu resetează banda de asamblare a altor $sCPU_i$; nu implică salvări şi restaurări de context; întreruperile pot fi nested; priorităţile întreruperilor pot fi dinamice (prin reataşarea la un task, sau prin schimbarea priorităţii taskului la care este ataşată). Spre deosebire de soluția de testare în buclă, soluția propusă asigură un timp de răspuns uniform pentru orice întrerupere. Mai mult decât atât, soluția propusă poate oferi priorități statice pentru întreruperi. Putem spune că soluția prezentată conține un management unitar al întreruperilor și o soluție hardware pentru a atașa întreruperile la task-urile sistemelor de timp-real implementate în hardware.

Schema de prioritizare globală [40] termină implementarea schemei de tratare hardware a evenimentelor [41] pentru arhitectura *nMPRA*. Se prezintă registrele-capcană asociate categoriilor de evenimente și structura registrului *Program Counter* (PC_i) [41].

Pentru sincronizarea și comunicația între taskuri s-a prevăzut un set de evenimente, implementat sub forma unui ERF, de asemenea ca resursă globală pentru toate $sCPU_i$. Accesul pentru setarea unui eveniment este fără adresă (nu necesită căutări în tabele pentru a depista un loc liber). Hardware-ul depistează automat prima adresă liberă și setează evenimentul și informația atașată acestuia, sau sesizează și semnalizează că nu există nici un eveniment liber. Task-ul destinație nu trebuie decât să citească din ERF pentru a afla dacă un eveniment i se adresează sau nu. Achitarea evenimentului se face automat.

Schema de prioritizare prezentată permite tratarea hardware a evenimentelor. Avantajul este reducerea timpului de detectare a sursei evenimentului și de începere a rutinei corespunzătoare de service a evenimentelor. Chiar se permite introducerea de noi evenimente în sistem prin simpla adăugare a câmpurilor necesare în *Task Register* (TR_i), *Event Status Task Register* ($ESTR_i$) şi *Event Priority Register* (EPR_i), actualizarea schemei de prioritizare globală a evenimentelor și introducerea unui nou registru-capcană pentru fiecare categorie nouă de eveniment. Schema este simplă și poate fi aplicată tuturor evenimentelor [41].

În raportul de cercetare următor va fi prezentat un studiu pentru o memorie pentru microcontrolerul prezentat şi o implementare pe FPGA a unităţii de control.

MULŢUMIRI

Această lucrare a fost susținută de proiectul „Performanță durabilă în cercetare doctorală și post-doctorală PERFORM-Contract nr. POSDRU / 159 / 1.5 / S / 138963”, proiect co-finanțat din Fondul Social European prin Programul Operațional Sectorial Resurse Umane 2007-2013.

Listă figuri

Listă tabele

BIBLIOGRAFIE

[1] J. Hennessy, D. Patterson, "*Computer Architecture: A Quantitative Approach",* Morgan Kaufmann, San Francisco, USA, 2007.

[2] J. Hennessy, D. Patterson, "*Computer Organization and Design: The Hardware/Software Interface",* Morgan Kaufmann, Waltham, USA, 2012.

[3] Rupali S. Balpande, Rashmi S. Keote, *Design of FPGA based Instruction Fetch & Decode Module of 32-bit RISC (MIPS) Processor*, 2011 International Conference on Communication Systems and Network Technologies, 978-0-7695-4437-3/11, 2011 IEEE.

[4] http://www.ecs.umass.edu/ece/ece232/

[5] http://cobweb.ecn.purdue.edu/~ece437l/materials

[6] http://opencores.org/project,mips_enhanced

[7] E. Dodiu, V.G. Gaitan, A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics, Croatia, May 2012.

[8] E. Dodiu and V.G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation", 2012 IEEE EIT International Conference on Electro-Information Technology, Indianapolis, IN, USA, 6-8 May 2012, ISBN: 978-1-4673-0818-2, ISSN: 2154-0373.

[9] V.G. Gaitan, N.C. Gaitan, I. Ungurean, "CPU Architecture based on a Hardware Scheduler and Independent Pipeline Registers", submitted in IEEE Transactions on VLSI System, 2014.

[10] L.E. Leyva-del-Foyo, and P. Mejia-Alvarez, "Custom Interrupt Management for Real-Time and Embedded System Kernels", in Proceedings of the 10th IEEE ECOOP Workshop on Exception Handling in Object Oriented Systems Development, 2005.

[11] Shi-Hai Zhu, "Hardware Implementation Based on FPGA of Interrupt Management in a Real-time Operating System", Information Technology Journal, 2013.

[12] B.C . Alecsa, "FPGA implementation of a matrix structure for integer division", Proceedings of the 3rd International Symposium on Electrical and Electronics Engineering, Galati, Romania, 2010.

[13] I. Liu, J. Reineke, E.A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties", in Signals, Systems and Computers, 2010, Conference Record of the Fortz Fourth Asilomar Conference on (pp. 2111/2115), IEEE.

[14] M. Shahbazi, P. Poure, S. Saadate, M.R. Zolghadri, "Fault-Tolerant Five-Leg Converter Topology with FPGA-Based Reconfigurable Control", IEEE Transactions on, vol. 60, no. 6, June 2013.

[15] W. Hofer, D. Lohmann, F. Schele, W. Schroder-Preikschat, "SLOTH: Threads as Interrupts", 30th IEEE Real-Time Systems Symposium, ISBN: 978-0-7695-3875-4, 204-213, 2009.
[16] J. Mäki-Turja, G. Fohler , K. Sandström, "Towards Efficient Analysis of Interrupts in Real-Time Systems". 11th EUROMICRO Conference on Real-Time Systems, York, England, May 1999.
[17] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. González Harbour, "A practitioner's handbook for real-time analysis", Kluwer Academic Publishers, 1993.
[18] K. Jeffay, D. L. Stone, "Accounting for Interrupt Handling Cost in Dynamic Priority Task Systems", Proc. of the IEEE Real-Time Systems Symposium, pp. 212-221, December 1993.
[19] L. Cheng-Min, "Nested interrupt analysis of low cost and high performance embedded systems using GSPN framework", IEICE Trans. Inform. Syst., E93-D: 2509-2519, 2010.
[20] Tan, S.L.; Bao Anh, T.N., "Real-time operating system (RTOS) for small (16-bit) microcontroller", Consumer Electronics, ISCE '09, IEEE 13th International Symposium on, pp.1007-1011, 25-28 May 2009.
[21] Bao Anh, T.N.; Tan, S.L., "Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers", Micro, IEEE Computer Society, ISSN: 0272-1732, 21 August 2009.
[22] Gaitan N.C., "Real-time acquisition of the distributed data by using an intelligent system", Electronics And Electrical Engineering, vol. 8, no. 104, pp. 13-18, 2010, ISSN: 1392,1215.
[23] Shawash, J.; Selviah, D.R., "Real-Time Nonlinear Parameter Estimation Using the Levenberg–Marquardt Algorithm on Field Programmable Gate Arrays", Industrial Electronics, IEEE Trans. on, vol.60, no.1, pp.170-176, Jan. 2013.
[24] Shahbazi, M.; Poure, P.; Saadate, S.; Zolghadri, M.R., "FPGA-Based Reconfigurable Control for Fault-Tolerant Back-to-Back Converter Without Redundancy", Industrial Electronics, IEEE Trans. on, vol.60, no.8, pp.3360-3371, Aug. 2013.
[25] Shi-Hai Zhu, "Hardware Implementation Based on FPGA of Interrupt Management in a Real-time Operating System", Information Technology Journal, 2013.
[26] Shahbazi, M.; Poure, P.; Saadate, S.; Zolghadri, M.R., "Fault-Tolerant Five-Leg Converter Topology With FPGA-Based Reconfigurable Control", Industrial Electronics, IEEE Trans. on, vol.60, no.6, pp.2284-2294, June 2013.
[27] Tran, T.; Ohishi, K.; Yokokura, Y.; Mitsantisuk, C., "FPGA-based High-Performance Force Control System with Friction-Free and Noise-Free Force Observation", Industrial Electronics, IEEE Trans. on, 2013.
[28] Alecsa, B.C., "FPGA implementation of a matrix structure for integer division", Proceedings of the 3rd International Symposium on Electrical and Electronics Engineering, Galati, Romania, 2010.
[29] Dodiu, E.; Gaitan, V.G.; Graur, A., "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics, Croatia, 24 May 2012, ISSN: 1847-3946.

[30] Dodiu, E.; Gaitan, V.G., "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation", 2012 IEEE EIT International Conference on Electro-Information Technology, Indianapolis, IN, USA, 6-8 May 2012, ISBN: 978-1-4673-0818-2, ISSN: 2154-0373.
[31] V.G. Gaitan, N.C. Gaitan, I. Ungurean, "CPU Arhitecture based on a Hardware Scheduler and Independent Pipeline Registers", unpublished, IEEE Trans. on VLSI System, 2014.
[32] G. Butazzo, "Hard Real-Time Computing Systems", Pavia-Italz: Spinger, 2005.
[33] N.C. Gaitan, " Real-time Acquisition of the Distributed Data by using an Intelligent System", Electronics and Electrical Engineering, Kaunas: Technologija, No. 8, Issue 104, Octomber 2010, ISSN 1392-1215.
[34] M.V. Micea, C.S. Certejan, V. Stangaciu, R. Cioarga, V. Cretu, and E. Petriu, "Inter-Task Communication and Synchronization in the Hard Real-Time Compact", ROSE 2008 – IEEE International Workshop on Robotic and Sensors Environments, Ottawa – Canada, 2008.
[35] G. Butazzo, P. Gai, "Efficient EDF Implementation for Small Embedded Systems", OSPERT 2006- Workshop on Operating Systems Platforms for Embedded Real-Time applications, Dresden- Germany, 2006.
[36] V.G. Gaitan, N.C. Gaitan, I. Ungurean, "CPU Architecture based on a Hardware Scheduler and Independent Pipeline Registers", IEEE Transactions on VLSI System, 2014.
[37] E. Dodiu, V.G. Gaitan, A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics, Croatia, May 2012.
[38] E. Dodiu and V.G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation", 2012 IEEE EIT International Conference on Electro-Information Technology, Indianapolis, USA, 6-8 May 2012, ISBN: 978-1-4673-0818-2, ISSN: 2154-0373.
[39] N.C. Gaitan, V.G. Gaitan, E.E. (Ciobanu) Moisuc, "Improving Interrupt Handling in the nMPRA", 12th International Conference on Development and Application Systems, Suceava, Romania, May 15-17, 2014, ISBN: 978-1-4799-5094-2/14.
[40] E.E. (Ciobanu) Moisuc, Al. B. Larionescu, V. G. Gaitan, "Hardware Event Treating in nMPRA", 12th International Conference on Development and Application Systems, Suceava, Romania, May 15-17, 2014, ISBN: 978-1-4799-5094-2/14.
[41] E.E. (Ciobanu) Moisuc, Al. B. Larionescu, I. Ungurean, "Hardware Event Handling in the Hardware Real-Time", 18th International Conference on System Theory, Control and Computing to be held in Sinaia, Romania, October 17-19, 2014, ISBN: 978-1-4799-4602-0 ©2014 IEEE.

ANEXE

Anexa 1

```
    begin
       --check ex for ctrl B

       if(ex_mem_regWr = '1' and (ex_mem_Rd /= b"00000") and (ex_mem_Rd = id_ex_Rt)) then
         ctrl_B <= b"10";
       --check mem for ctrl B

         elsif(mem_wb_regWr = '1' and (mem_wb_Rd /= b"00000") and

            not (ex_mem_regWr = '1' and (ex_mem_Rd /= b"00000") and (ex_mem_Rd = id_ex_Rt))
            and (mem_wb_Rd = id_ex_Rt)) then
         ctrl_B <= b"01";
       else
         ctrl_B <= b"00";
      end if;
    end process Foward2_Process;
```

Figura 0-1 Exemplu de VHDL pentru unitatea de forward (*sursa:* [5])

```
--checks for lw or some other instr followed by a branch
lw_branch_hazzard_Process1: process(mem_lw, ex_lw, if_id_instr, id_ex_instr, mem_lw_Rt)
--check for lw then something then branch
  begin
    if(((if_id_instr(31 downto 26) = "000100") or (if_id_instr(31 downto 26) = "000001") or (if_id_instr(31 downto 26) = "000111") or
      (if_id_instr(31 downto 26) = "000110") or (if_id_instr(31 downto 26) = "000101")) and (mem_lw = '1') and
      ((if_id_instr(25 downto 21) = mem_lw_Rt) or (if_id_instr(20 downto 16) = mem_lw_Rt))) then

         id_stall <= '1';
         pc_stall <= '1';
    else
         id_stall <= '0';
         pc_stall <= '0';
   end if;
 end process lw_branch_hazzard_Process1;
```

Figura 0-2 Exemplu de VHDL pentru unitatea de detecţie hazard (*sursa:* [5])

Anexa 2

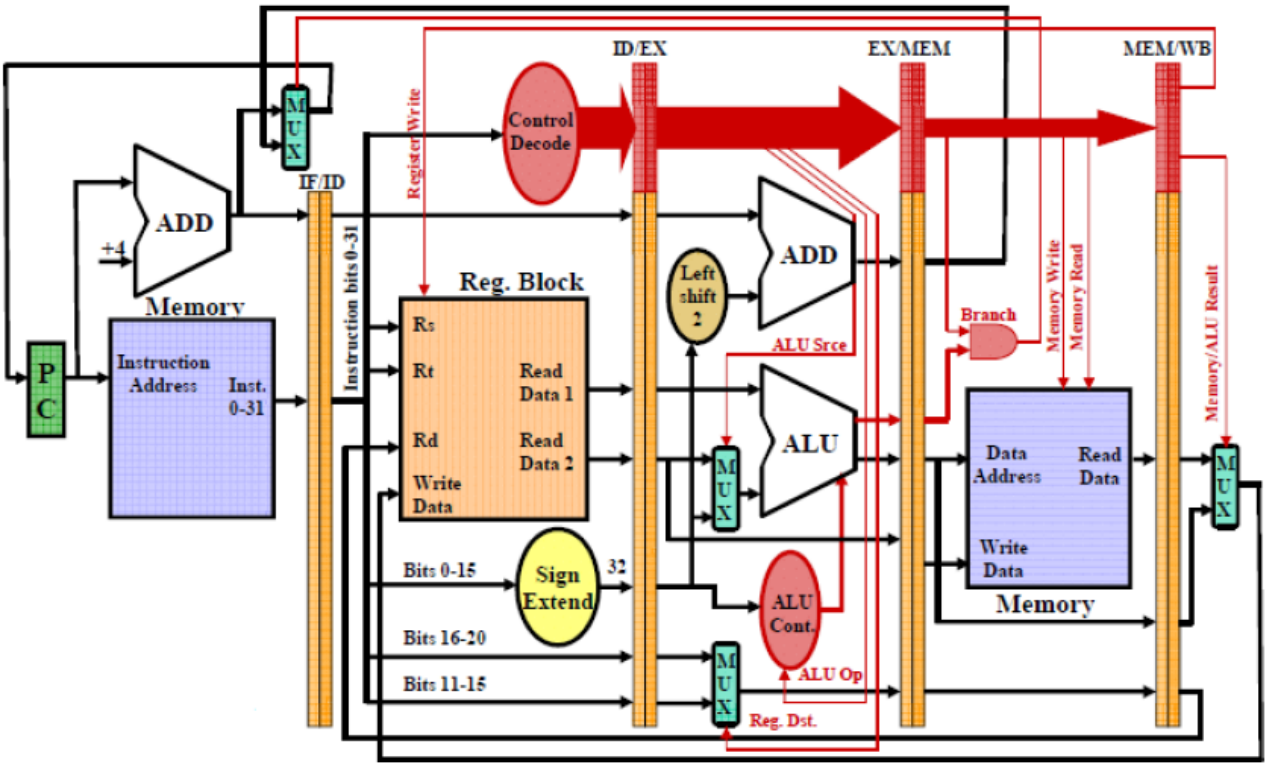


Figura 0-3 Exemplu unitate de control (*sursa:* [4])

Anexa 3

Flow Status	Successful - Wed Aug 12 20:05:09 2015
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	ALU
Top-level Entity Name	ALU
Family	Cyclone III
Device	EP3C25F324C6
Timing Models	Final
Total logic elements	1,244
Total combinational functions	1,244
Dedicated logic registers	264
Total registers	264
Total pins	291
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	8
Total PLLs	0

Figure 0-4 Implementare ALU pe Quartus II

```
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  entity ALU is
24      Port
25            (
26             clk : in  std_logic;
27             rst,Mult_en,sel_top : in  STD_LOGIC;
28             A_in : in  std_logic_vector(31 downto 0);
29             B_in : in  std_logic_vector(31 downto 0);
30             I : in  std_logic_vector(31 downto 0);
31             immed_addr : std_logic_vector(15 downto 0);
32             ALUOp : in  std_logic_vector(2 downto 0);
33             ALUmux : in  std_logic_vector(1 downto 0);
34             From_i_op : IN std_logic_vector(1 downto 0);
35             From_i_mux : IN std_logic_vector(1 downto 0);
36             lui : in  STD_LOGIC;
37             ALUSrcA : in std_logic;
38             ALUSrcB : in  STD_LOGIC_VECTOR(1 downto 0);
39             N  : in std_logic_vector(31 downto 0);
40             M : out std_logic_vector(31 downto 0);
41             Alu_out_exit : out  std_logic_vector(31 downto 0);
42             Hi_out : out std_logic_vector(31 downto 0);
43             Lo_out : out std_logic_vector(31 downto 0);
44             Zero,pass : out std_logic
45  );
46  end ALU;
47
48  architecture Behavioral of ALU is
49
50  component Logic is
51  Port
52  (
53             A : in  STD_LOGIC_VECTOR (31 downto 0);
54             B : in  STD_LOGIC_VECTOR (31 downto 0);
55             ALUop : in  STD_LOGIC_VECTOR (1 downto 0);
56             S : out  STD_LOGIC_VECTOR  (31 downto 0)
57  );
58  end component;
59  component adder is
60  Port
61  (
62             A : in  STD_LOGIC_VECTOR (31 downto 0);
```

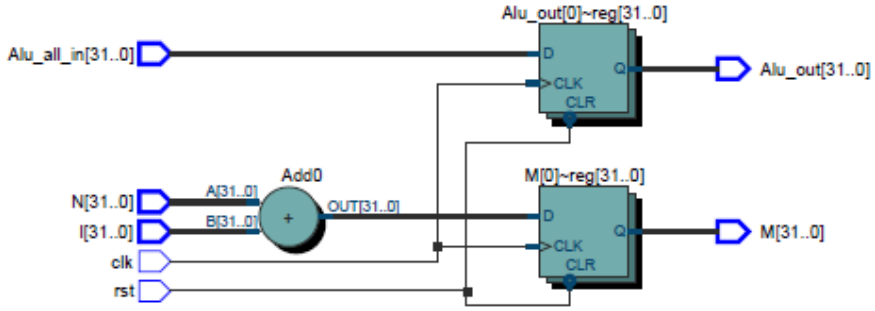
```
            B : in  STD_LOGIC_VECTOR (31 downto 0);
            ALUop : in  STD_LOGIC_VECTOR (1 downto 0);
            --ov : out std_logic;
            S : out  STD_LOGIC_VECTOR (31 downto 0)
);
end component;
component Mux_4_1 is
Port
(
            Logic_out : in  std_logic_vector(31 downto 0);
            Adder_out : in  std_logic_vector(31 downto 0);
            Shift_out : in  std_logic_vector(31 downto 0);
            Slt_out : in  std_logic_vector(31 downto 0);
            ALUmux : in  std_logic_vector(1 downto 0);
            Mux_out : out std_logic_vector(31 downto 0)
);
end component;
Component Shift_mux is
Port
(
            A : in  STD_LOGIC_VECTOR (4 downto 0);
            Shamt : in  STD_LOGIC_VECTOR (4 downto 0);
            sv : in  STD_LOGIC;
            lui : in  STD_LOGIC;
            Shamt_out : out  STD_LOGIC_VECTOR (4 downto 0)
);
end Component;
component Shift is
Port
(
            rst : in  STD_LOGIC;
            B : in  STD_LOGIC_VECTOR (31 downto 0);
            ALUop : in  STD_LOGIC_VECTOR (1 downto 0);
            Shamt_in : in  STD_LOGIC_VECTOR (4 downto 0);
            S : out  STD_LOGIC_VECTOR (31 downto 0)
);
end component;
component SLT is
Port
(
            Adder_out : in  STD_LOGIC_VECTOR (0 downto 0);
            Slt_out : out  STD_LOGIC_VECTOR (31 downto 0)
);
end component;
 component Or_tree is
 Port (     Mux_out : in  STD_LOGIC_VECTOR (31 downto 0);
             Zero : out  STD_LOGIC
       );

 end component;

 component In_mux is
 Generic (
          busw : integer := 31
 );
 Port (
             A_in : in  STD_LOGIC_VECTOR (busw downto 0);
             B_in : in  STD_LOGIC_VECTOR (busw downto 0);
             I : in  STD_LOGIC_VECTOR (busw downto 0);
             ALUSrcA : in  STD_LOGIC;
             ALUSrcB : in  STD_LOGIC_VECTOR(1 downto 0);
             A : out  STD_LOGIC_VECTOR (busw downto 0);
```

```
125             B : out  STD_LOGIC_VECTOR (busw downto 0)
126             );
127   end component;
128   component Alu_out is
129   port (
130         clk : in  STD_LOGIC;
131         rst : in  STD_LOGIC;
132         I   : in std_logic_vector(31 downto 0);
133         N   : in std_logic_vector(31 downto 0);
134         Alu_all_in : in std_logic_vector(31 downto 0);
135         M : out std_logic_vector(31 downto 0);
136         Alu_out : out std_logic_vector(31 downto 0)
137
138   );
139   end component;
140   component ALu_control is
141   PORT
142   (
143             instr_15_0 : IN std_logic_vector(15 downto 0);
144             ALUOp : IN std_logic_vector(2 downto 0);
145             ALUmux : IN std_logic_vector(1 downto 0);
146             From_i_op : IN std_logic_vector(1 downto 0);
147             From_i_mux : IN std_logic_vector(1 downto 0);
148             sv : out  STD_LOGIC;
149             Mul_out_c : out  STD_LOGIC_VECTOR(1 downto 0);
150             ALUmux_c : OUT std_logic_vector(1 downto 0);
151             ALUopcode : OUT std_logic_vector(1 downto 0)
152   );
153   end component;
154   component Mult_out is
155   port (
156              clk : in std_logic;
157              rst,Mult_en : in  STD_LOGIC;
158              Mul_out_c : in STD_LOGIC_VECTOR (1 downto 0);
159              A_in : in STD_LOGIC_VECTOR (31 downto 0);
160              Hi_to_out : in STD_LOGIC_VECTOR (31 downto 0);
161              Lo_to_out : in STD_LOGIC_VECTOR (31 downto 0);
162              Hi_out : out STD_LOGIC_VECTOR (31 downto 0);
163              Lo_out : out STD_LOGIC_VECTOR (31 downto 0)
164
165      );
166   end component;
167   component mult_lfsr is
168   port (
169        clock_top : in std_logic;
170        reset_top : in std_logic;
171        --seed_top  : in std_logic_vector(63 downto 0);
172        sel_top   : in std_logic;
173        X_top     : in std_logic_vector(31 DOWNTO 0);
174        Y_top     : in std_logic_vector(31 DOWNTO 0);
175        -- P_top     : out std_logic_vector(63 downto 0)
176        Hi_to_out : out STD_LOGIC_VECTOR (31 downto 0);
177        Lo_to_out : out STD_LOGIC_VECTOR (31 downto 0);
178        pass      : out std_logic
179      );
180   end component mult_lfsr;
181   signal Logic_out,Adder_out,Shift_out,Slt_out,Mux_out,A,B,Hi_to_out,Lo_to_out :
      std_logic_vector(31 downto 0);
182   signal Shamt_m_out: std_logic_vector(4 downto 0);
183   signal ALUopcode,ALUmux_c,Mul_out_c : std_logic_vector(1 downto 0);
184   signal sv : std_logic;
185
```

```
186   begin
187   Logic_a:Logic port map(A=>A,B=>B,ALUop=>ALUopcode,S=>Logic_out);
188   Adder_a:adder port map(A=>A,B=>B,ALUop=>ALUopcode,S=>Adder_out);
189   Shift_a:Shift port map(rst=>rst,B=>B,ALUop=>ALUopcode,Shamt_in=>Shamt_m_out,S=>Shift_out);
190   Shift_mux_a:Shift_mux port map(A=>A(4 downto 0),Shamt=>immed_addr(10 downto 6),sv=>sv,lui=>
      lui,
191                                  Shamt_out=>Shamt_m_out);
192   SLT_a:SLT port map(Adder_out=>Adder_out(31 downto 31),Slt_out=>Slt_out);
193   Mux_4_1_a:Mux_4_1 port map(Logic_out=>Logic_out,Adder_out=>Adder_out,
194                              Shift_out=>Shift_out,Slt_out=>Slt_out,ALUmux=>ALUmux_c,Mux_out=>
      Mux_out);
195   Or_tree_a:Or_tree port map(Mux_out=>Mux_out,Zero=>Zero);
196   --Mult_a:Mult port map(A=>A,B=>B,Hi_to_out=>Hi_to_out,Lo_to_out=>Lo_to_out);
197   In_mux_a:In_mux port map(A_in=>A_in,B_in=>B_in,I=>I,ALUSrcA=>ALUSrcA,ALUSrcB=>ALUSrcB,A=>A,B
      =>B);
198   Alu_out_a:Alu_out port map(clk=>clk,rst=>rst,I=>I,N=>N,M=>M,Alu_all_in=>Mux_out,Alu_out=>
      Alu_out_exit);
199   ALu_control_a:ALu_control port map(instr_15_0=>immed_addr,ALUOp=>ALUOp,ALUmux=>ALUmux,
      From_i_op=>From_i_op,
200                                      From_i_mux=>From_i_mux,sv=>sv,
201                                      ALUmux_c=>ALUmux_c,Mul_out_c=>Mul_out_c,ALUopcode=>
      ALUopcode);
202   Mult_out_a:Mult_out port map(clk=>clk,rst=>rst,Mult_en=>Mult_en,Mul_out_c=>Mul_out_c,A_in=>
      A_in,
203                                Hi_to_out=>Hi_to_out,Lo_to_out=>Lo_to_out,
204                                Hi_out=>Hi_out,Lo_out=>Lo_out);
205   Mult_lfsr_a:mult_lfsr port map(clock_top=>clk,reset_top=>rst,sel_top=>sel_top,X_top=>A,Y_top
      =>B,
206                                  Hi_to_out=>Hi_to_out,Lo_to_out=>Lo_to_out,pass=>pass);
207   end Behavioral;
208
209
```

Figura 0-5 Codul VHDL pentru ALU

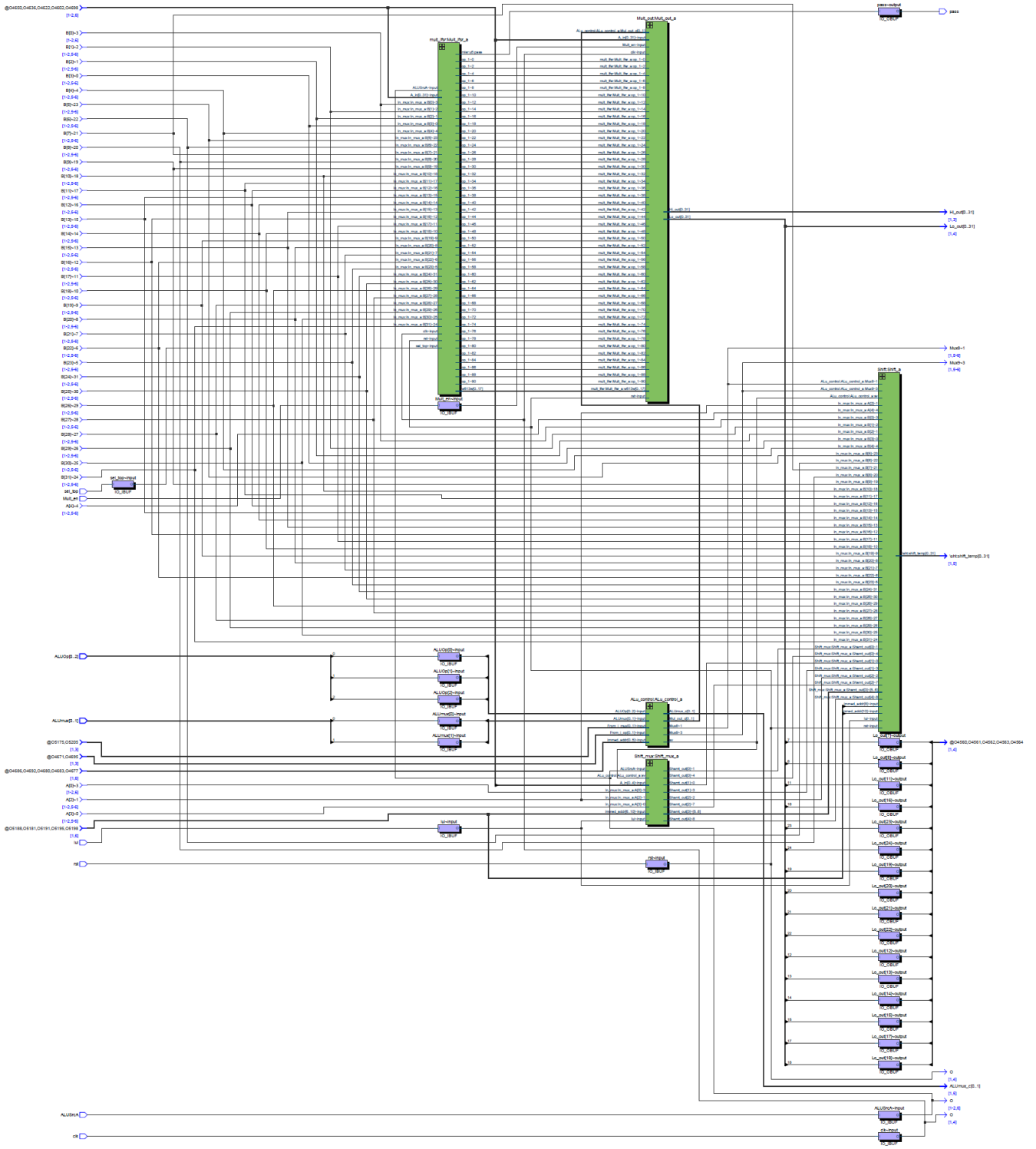
Anexa 4



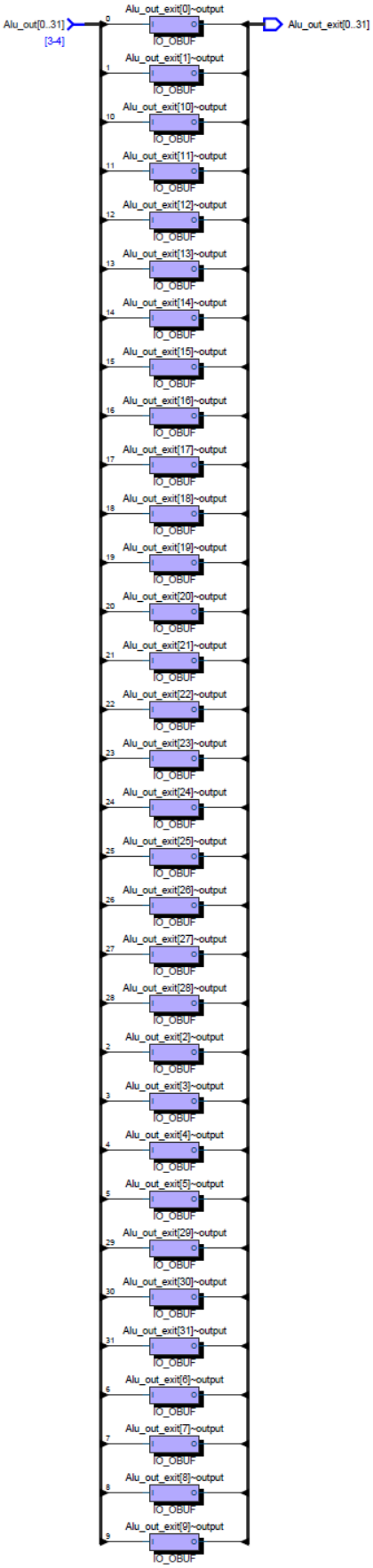
(a)

```
15   -- Revision:
16   -- Revision 0.01 - File Created
17   -- Additional Comments:
18   --
19   ----------------------------------------------------------------------------
20   library IEEE;
21   use IEEE.STD_LOGIC_1164.ALL;
22
23   use IEEE.NUMERIC_STD.ALL;
24   use IEEE.STD_LOGIC_UNSIGNED.ALL;
25
26   entity Alu_out is
27   port (
28         clk : in  STD_LOGIC;
29         rst : in  STD_LOGIC;
30         I   : in std_logic_vector(31 downto 0);
31         N   : in std_logic_vector(31 downto 0);
32         Alu_all_in : in std_logic_vector(31 downto 0);
33         M : out std_logic_vector(31 downto 0);
34         Alu_out : out std_logic_vector(31 downto 0)
35
36   );
37
38   end Alu_out;
39
40   architecture Behavioral of Alu_out is
41
42   begin
43         process(clk,rst,Alu_all_in)
44         begin
45                        if rst = '0' then
46                                 Alu_out <= (others => '0');
47                         elsif (RISING_EDGE(Clk))then
48                                Alu_out <= Alu_all_in;
49
50                        end if;
51         end process;
52
53         process(clk,I,N,rst)
54         variable M_var,N_var,I_var : signed(31 downto 0);
55         variable shift_temp : std_logic_vector(31 downto 0);
56         begin
57              N_var := signed(N);
58              shift_temp :=  to_stdlogicvector(to_bitvector(I) sla 2);
59              I_var := signed(shift_temp);
60              M_var := N_var + I_var;
61
62              if rst = '0' then
63                       M <= (others => '0');
64               elsif (RISING_EDGE(Clk))then
65                        M <= std_logic_vector(M_var);
66               end if;
67        end process;
68
69   end Behavioral;
70
71
```

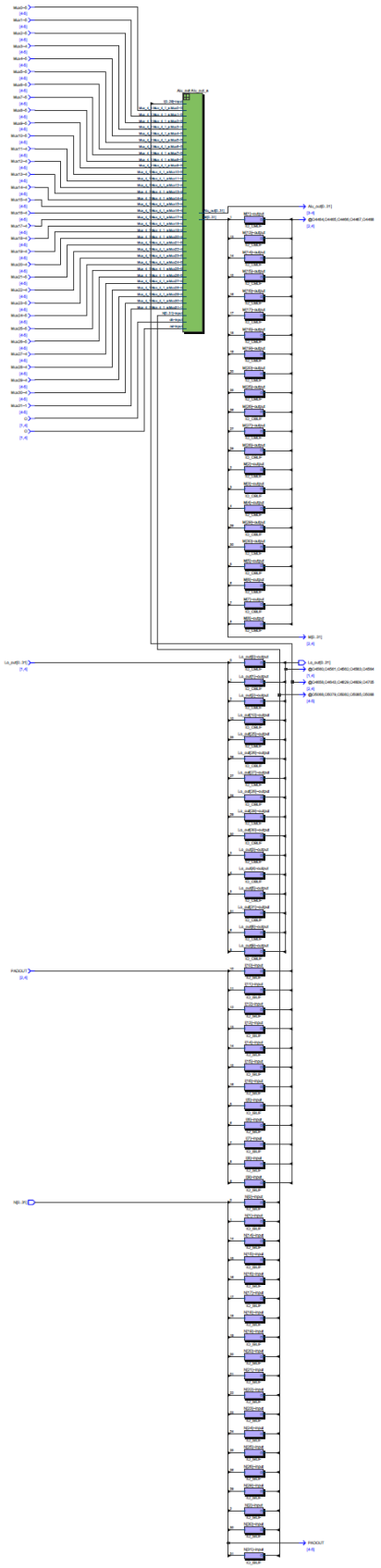
(b)



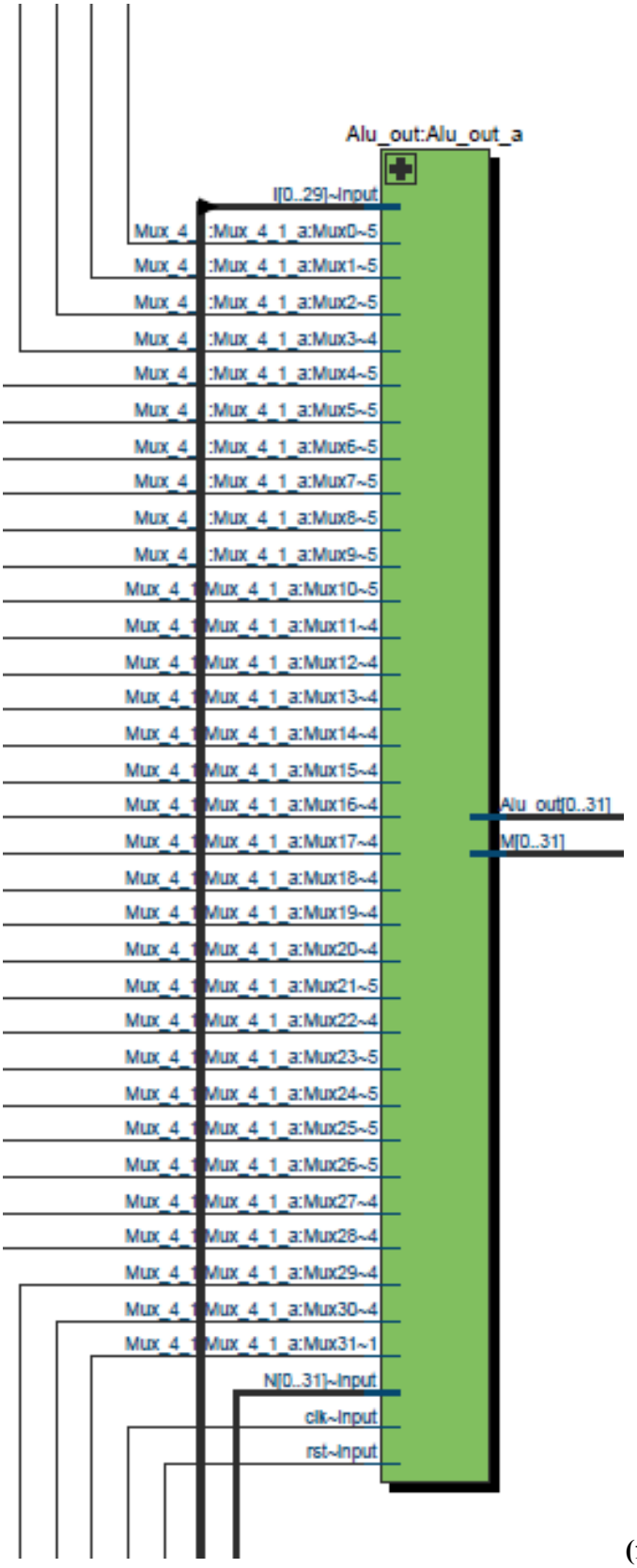
(c)


Alu_out[0..31]
[3-4]
Alu_out_exit[0]~output
Alu_out_exit[1]~output
Alu_out_exit[10]~output
Alu_out_exit[11]~output
Alu_out_exit[12]~output
Alu_out_exit[13]~output
Alu_out_exit[14]~output
Alu_out_exit[15]~output
Alu_out_exit[16]~output
Alu_out_exit[17]~output
Alu_out_exit[18]~output
Alu_out_exit[19]~output
Alu_out_exit[20]~output
Alu_out_exit[21]~output
Alu_out_exit[22]~output
Alu_out_exit[23]~output
Alu_out_exit[24]~output
Alu_out_exit[25]~output
Alu_out_exit[26]~output
Alu_out_exit[27]~output
Alu_out_exit[28]~output
Alu_out_exit[2]~output
Alu_out_exit[3]~output
Alu_out_exit[4]~output
Alu_out_exit[5]~output
Alu_out_exit[29]~output
Alu_out_exit[30]~output
Alu_out_exit[31]~output
Alu_out_exit[6]~output
Alu_out_exit[7]~output
Alu_out_exit[8]~output
Alu_out_exit[9]~output
IO_OBUF
Alu_out_exit[0..31]

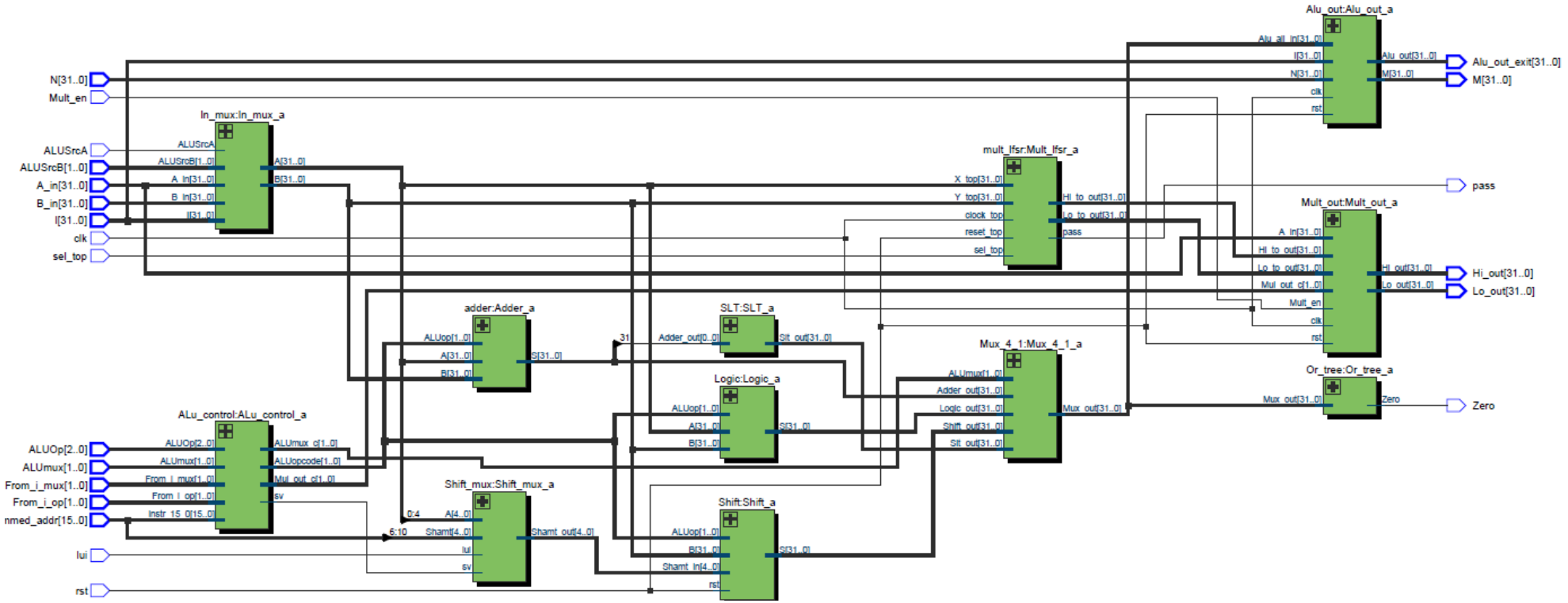
(d)



(e)


Alu_out:Alu_out_a
I[0..29]~input
Mux_4_:Mux_4_1_a:Mux0~5
Mux_4_:Mux_4_1_a:Mux1~5
Mux_4_:Mux_4_1_a:Mux2~5
Mux_4_:Mux_4_1_a:Mux3~4
Mux_4_:Mux_4_1_a:Mux4~5
Mux_4_:Mux_4_1_a:Mux5~5
Mux_4_:Mux_4_1_a:Mux6~5
Mux_4_:Mux_4_1_a:Mux7~5
Mux_4_:Mux_4_1_a:Mux8~5
Mux_4_:Mux_4_1_a:Mux9~5
Mux_4_1Mux_4_1_a:Mux10~5
Mux_4_1Mux_4_1_a:Mux11~4
Mux_4_1Mux_4_1_a:Mux12~4
Mux_4_1Mux_4_1_a:Mux13~4
Mux_4_1Mux_4_1_a:Mux14~4
Mux_4_1Mux_4_1_a:Mux15~4
Mux_4_1Mux_4_1_a:Mux16~4
Mux_4_1Mux_4_1_a:Mux17~4
Mux_4_1Mux_4_1_a:Mux18~4
Mux_4_1Mux_4_1_a:Mux19~4
Mux_4_1Mux_4_1_a:Mux20~4
Mux_4_1Mux_4_1_a:Mux21~5
Mux_4_1Mux_4_1_a:Mux22~4
Mux_4_1Mux_4_1_a:Mux23~5
Mux_4_1Mux_4_1_a:Mux24~5
Mux_4_1Mux_4_1_a:Mux25~5
Mux_4_1Mux_4_1_a:Mux26~5
Mux_4_1Mux_4_1_a:Mux27~4
Mux_4_1Mux_4_1_a:Mux28~4
Mux_4_1Mux_4_1_a:Mux29~4
Mux_4_1Mux_4_1_a:Mux30~4
Mux_4_1Mux_4_1_a:Mux31~1
N[0..31]~input
clk~input
rst~input
Alu_out[0..31]
M[0..31]

(f)



(g)

Figura 0-6 ALU Out

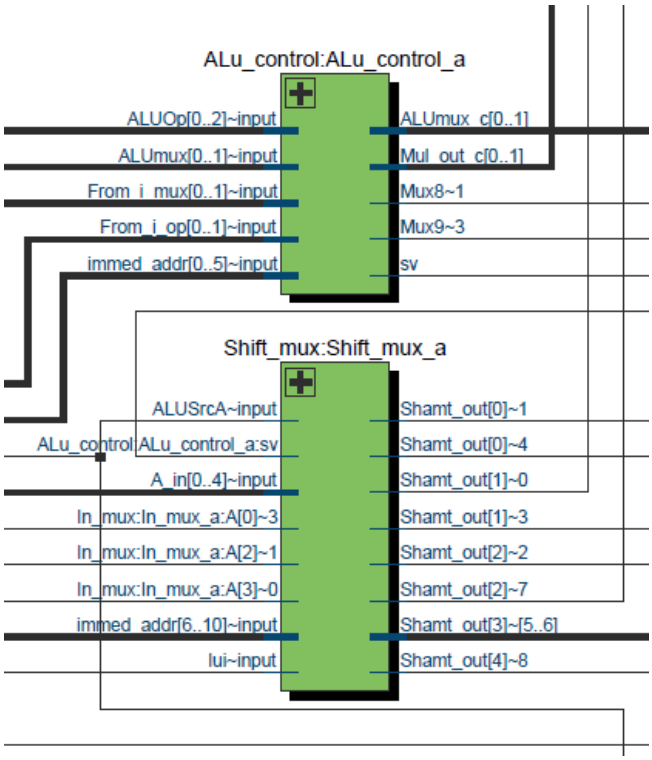
Anexa 5

```
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23
24  entity ALu_control is
25
26  PORT
27  (          --clk : IN std_logic;
28             instr_15_0 : IN std_logic_vector(15 downto 0);
29             ALUOp : IN std_logic_vector(2 downto 0);
30             ALUmux : IN std_logic_vector(1 downto 0);
31             From_i_op : IN std_logic_vector(1 downto 0);
32             From_i_mux : IN std_logic_vector(1 downto 0);
33             sv : out  STD_LOGIC := '0';
34             Mul_out_c : out  STD_LOGIC_VECTOR(1 downto 0);
35             ALUmux_c : OUT std_logic_vector(1 downto 0);
36             ALUopcode : OUT std_logic_vector(1 downto 0)
37  );
38
39
40  end ALu_control;
41
42  architecture Behavioral of ALu_control is
43
44  begin
45  Alu_Control : PROCESS(instr_15_0, ALUOp,ALUmux,From_i_op,From_i_mux)
46  CONSTANT ADDr : std_logic_vector(5 downto 0) := "100000";
47  CONSTANT ADDr_u : std_logic_vector(5 downto 0) := "100001";
48  CONSTANT SUBr : std_logic_vector(5 downto 0) := "100010";
49  CONSTANT SUBr_u : std_logic_vector(5 downto 0) := "100011";
50  CONSTANT ANDr : std_logic_vector(5 downto 0) := "100100";
51  CONSTANT ORr : std_logic_vector(5 downto 0) := "100101";
52  CONSTANT XORr : std_logic_vector(5 downto 0) := "100110";
53  CONSTANT NORr : std_logic_vector(5 downto 0) := "100111";
54  CONSTANT SLTr : std_logic_vector(5 downto 0) := "101010";
55  CONSTANT SLTUr : std_logic_vector(5 downto 0) := "101011";
56  CONSTANT MULTr : std_logic_vector(5 downto 0) := "011000";
57  CONSTANT Mtlor : std_logic_vector(5 downto 0) := "010011";
58  CONSTANT Mthir : std_logic_vector(5 downto 0) := "010001";
59  CONSTANT Mflor : std_logic_vector(5 downto 0) := "010010";
60  CONSTANT Mfhir : std_logic_vector(5 downto 0) := "010000";
61  CONSTANT Sllr : std_logic_vector(5 downto 0)  := "000000";
62  CONSTANT SLLVr : std_logic_vector(5 downto 0) := "000100";
```

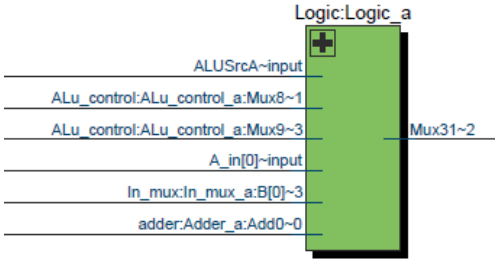
```
63    CONSTANT SRLr : std_logic_vector(5 downto 0)  := "000010";
64    CONSTANT SR?r : std_logic_vector(5 downto 0)  := "000011";
65    CONSTANT SRLVr : std_logic_vector(5 downto 0) := "000110";
66    CONSTANT SRAVr : std_logic_vector(5 downto 0) := "000111";
67    CONSTANT JRr  : std_logic_vector(5 downto 0)  := "001000";
68
69    BEGIN
70      --if FALLING_EDGE(clk) then
71      case ALUOp is
72          when "000" =>
73                          ALUopcode <= "00";  -- add  I types
74                          ALUmux_c <= ALUmux;
75          when "001" => ALUopcode <= "01";  -- add unsigned
76                          ALUmux_c <= ALUmux;
77          when "010" => ALUopcode <= "01";  -- subtract signed
78                          ALUmux_c <= ALUmux;
79          when "011" => ALUopcode <= "11";  -- subtract unsigned
80                          ALUmux_c <= ALUmux;
81          when "110" => -- operation depends on function field  r types
82                    case instr_15_0(5 downto 0) is    -- r types
83                    when ADDr => ALUopcode <= "00"; -- add signed
84                                 ALUmux_c <= "10";
85                    when ADDr_u => ALUopcode <= "01";  -- add unsigned
86                                 ALUmux_c <= "10";
87                    when SUBr => ALUopcode <= "10"; -- subtract
88                                  ALUmux_c <= "10";
89                    when SUBr_u => ALUopcode <= "11"; --subtract unsigned
90                                  ALUmux_c <= "10";
91                    when ANDr => ALUopcode <= "00"; -- AND
92                                  ALUmux_c <= "11";
93                    when ORr => ALUopcode <= "01"; -- OR
94                                  ALUmux_c <= "11";
95                    when XORr => ALUopcode <= "10"; -- XOR
96                                  ALUmux_c <= "11";
97                    when NORr => ALUopcode <=  "11"; -- NOR
98                                  ALUmux_c <= "11";
99                    when SLTr => ALUopcode  <=  "10";    -- SLT
100                                 ALUmux_c <= "01";
101                   when SLTUr => ALUopcode  <=  "11";    -- SLTU
102                                 ALUmux_c <= "01";
103                   when MULTr => ALUopcode <=  "00"; -- MULT
104                                 ALUmux_c <= "00";
105                                 Mul_out_c <= "00";
106                   when Mtlor => ALUopcode <=  "00"; -- Mtlo
107                                 ALUmux_c <= "00";
108                                 Mul_out_c <= "01";
109                   when Mthir => ALUopcode <=  "00"; -- Mthi
110                                 ALUmux_c <= "00";
111                                   Mul_out_c <= "10";
112                    when Mflor => ALUopcode <=  "00"; -- Mflo
113                                   ALUmux_c <= "00";
114                                --  Mul_out_c <= "00";
115                    when Mfhir => ALUopcode <=  "00"; -- Mfhi
116                                   ALUmux_c <= "00";
117                                --  Mul_out_c <= "00";
118                    when Sllr =>   ALUopcode <= "00";  --sll
119                                   ALUmux_c <= "00";
120                                   sv <= '0';
121                    when SRLr =>   ALUopcode <= "10";  --srl
122                                   ALUmux_c <= "00";
123                                   sv <= '0';
124                    when SR?r =>   ALUopcode <= "11";  --sra
```

```
125                                     ALUmux_c <= "00";
126                                     sv <= '0';
127                         when SLLVr =>  ALUopcode <= "00";  -- sllv
128                                     ALUmux_c <= "00";
129                                     sv <= '1';
130                         when SRLVr =>  ALUopcode <= "10";  --srlv
131                                     ALUmux_c <= "00";
132                                     sv <= '1';
133                         when SRAVr =>  ALUopcode <= "11";      --srav
134                                     ALUmux_c <= "00";
135                                     sv <= '1';
136                         when others => ALUopcode <= "00";
137                                     ALUmux_c <= "00";
138                                     Mul_out_c <= "00";
139                                     sv <= '0';
140                         end case;
141                 when "111" =>    -- I types Aluop from ir
142                           ALUopcode <= From_i_op;
143                           ALUmux_c  <= From_i_mux;
144             when others => ALUopcode <= "00";
145           end case;
146          -- end if;
147        END PROCESS;
148
149
150
151
152        end Behavioral;
153
154
```

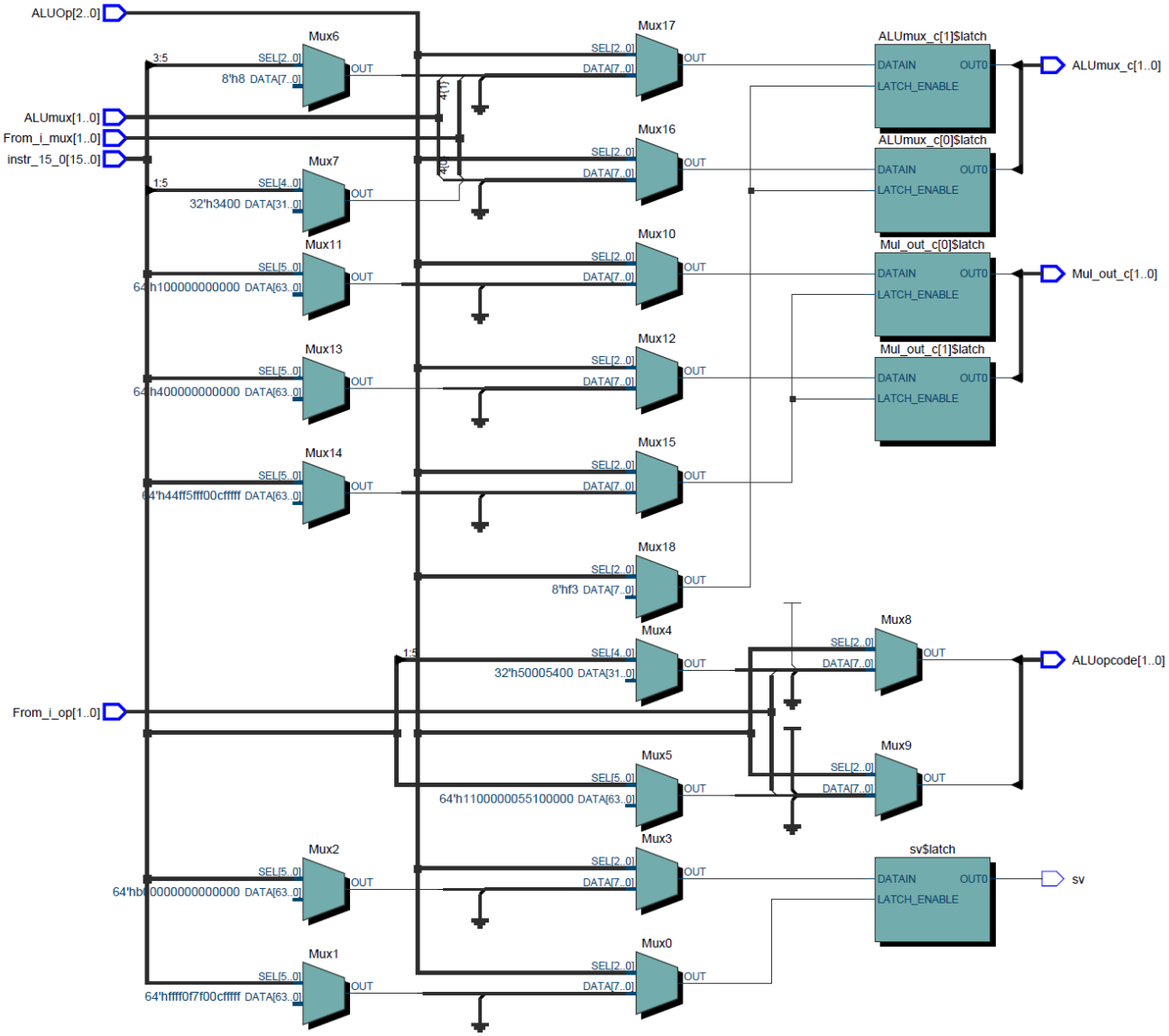
(a)



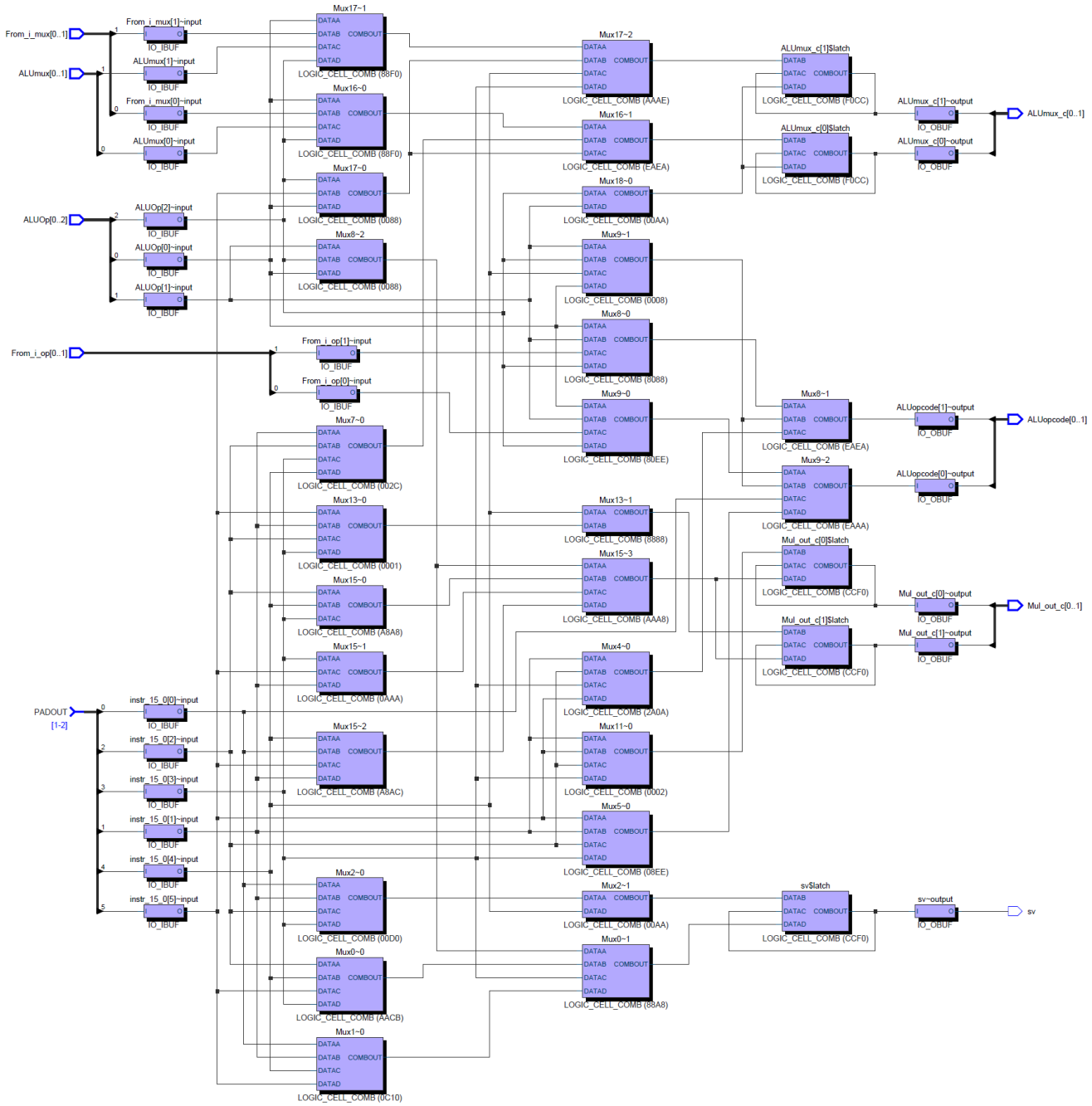
(b)



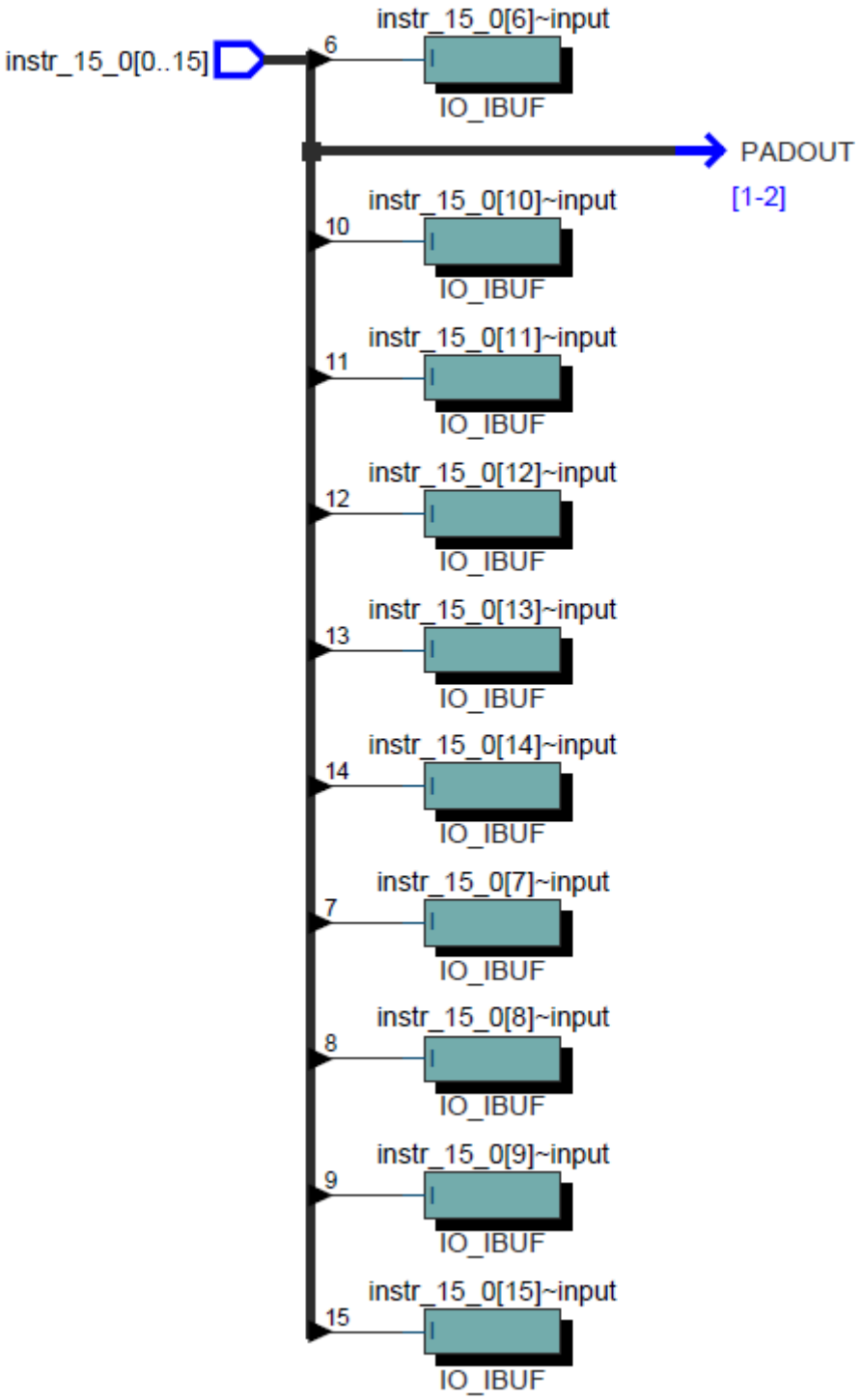
(c)


ALUOp[2..0]
ALUmux[1..0]
From_i_mux[1..0]
instr_15_0[15..0]
From_i_op[1..0]
Mux6
Mux7
Mux11
Mux13
Mux14
Mux17
Mux16
Mux10
Mux12
Mux15
Mux18
Mux4
Mux5
Mux2
Mux1
Mux8
Mux9
Mux3
Mux0
SEL[2..0]
SEL[4..0]
SEL[5..0]
DATA[7..0]
OUT
8'h8 DATA[7..0]
32'h3400 DATA[31..0]
64'h100000000000 DATA[63..0]
64'h400000000000 DATA[63..0]
64'h44ff5fff00cfffff DATA[63..0]
8'hf3 DATA[7..0]
32'h50005400 DATA[31..0]
64'h1100000055100000 DATA[63..0]
64'hffff0f7f00cfffff DATA[63..0]
ALUmux_c[1]$latch
ALUmux_c[0]$latch
Mul_out_c[0]$latch
Mul_out_c[1]$latch
sv$latch
DATAIN
LATCH_ENABLE
OUT0
ALUmux_c[1..0]
Mul_out_c[1..0]
ALUopcode[1..0]
sv

(d)


From_i_mux[0..1]
From_i_mux[1]~input
ALUmux[0..1]
ALUmux[1]~input
From_i_mux[0]~input
ALUmux[0]~input
IO_IBUF
ALUOp[0..2]
ALUOp[2]~input
ALUOp[0]~input
ALUOp[1]~input
From_i_op[0..1]
From_i_op[1]~input
From_i_op[0]~input
PADOUT
[1-2]
instr_15_0[0]~input
instr_15_0[2]~input
instr_15_0[3]~input
instr_15_0[1]~input
instr_15_0[4]~input
instr_15_0[5]~input
Mux17~1
Mux16~0
Mux17~0
Mux8~2
Mux7~0
Mux13~0
Mux15~0
Mux15~1
Mux15~2
Mux2~0
Mux0~0
Mux1~0
Mux17~2
Mux16~1
Mux18~0
Mux9~1
Mux8~0
Mux9~0
Mux13~1
Mux15~3
Mux4~0
Mux11~0
Mux5~0
Mux2~1
Mux0~1
ALUmux_c[1]$latch
ALUmux_c[0]$latch
ALUmux_c[1]~output
ALUmux_c[0]~output
ALUmux_c[0..1]
Mux8~1
Mux9~2
ALUopcode[1]~output
ALUopcode[0]~output
ALUopcode[0..1]
Mul_out_c[0]$latch
Mul_out_c[1]$latch
Mul_out_c[0]~output
Mul_out_c[1]~output
Mul_out_c[0..1]
sv$latch
sv~output
sv
LOGIC_CELL_COMB (88F0)
LOGIC_CELL_COMB (0088)
LOGIC_CELL_COMB (002C)
LOGIC_CELL_COMB (0001)
LOGIC_CELL_COMB (A8A8)
LOGIC_CELL_COMB (0AAA)
LOGIC_CELL_COMB (A8AC)
LOGIC_CELL_COMB (00D0)
LOGIC_CELL_COMB (AACB)
LOGIC_CELL_COMB (0C10)
LOGIC_CELL_COMB (AAAE)
LOGIC_CELL_COMB (EAEA)
LOGIC_CELL_COMB (00AA)
LOGIC_CELL_COMB (0008)
LOGIC_CELL_COMB (8088)
LOGIC_CELL_COMB (80EE)
LOGIC_CELL_COMB (8888)
LOGIC_CELL_COMB (AAA8)
LOGIC_CELL_COMB (2A0A)
LOGIC_CELL_COMB (0002)
LOGIC_CELL_COMB (08EE)
LOGIC_CELL_COMB (00AA)
LOGIC_CELL_COMB (88A8)
LOGIC_CELL_COMB (F0CC)
LOGIC_CELL_COMB (EAEA)
LOGIC_CELL_COMB (EAAA)
LOGIC_CELL_COMB (CCF0)

(e)



(f)

Figura 0-7 ALU Control